

Programming in Java – Part 02
Behavior parameterization



Paolo Vercesi
ESTECO SpA

© 2019 ESTECO SpA

Agenda



- Behavior parameterization
- Lambda expressions
- The `java.util.function` package
- Method references

© 2019 ESTECO SpA

Behavior parameterization

© 2019 ESTECO SpA

Parameterization

So far, you have seen three types of parameterization, each one serves a different purpose but all of them are associated with some reuse

“Data” parameterization

Data is used as a parameter in method or constructor invocation

Parametrized type or generic

A type parameter is used to specify the type of data upon which classes, interfaces, and methods operate

Behavior parameterization

An object implementing some behavior is used as a parameter in method or constructor invocation



© 2019 ESTECO SpA

“Data” parameterization

```
public class Accumulator {

  private int value;

  public void incrementByOne() {
    value += 1;
  }

  public void incrementByTwo() {
    value += 2;
  }

  public void incrementByThree() {
    value += 3;
  }
  …
}
```



```
public class Accumulator {

  private int value;

  public void incrementBy(int delta) {
    value += delta;
  }
}
```

© 2019 ESTECO SpA

Parametrized type

```
public interface List {

    void add(Object item);

    void set(int index, Object item);

    Object get(int index);
}
```

```
public interface StringList {

    void add(String item);

    void set(int index, String item);

    String get(int index);
}
```



```
public interface List<T> {

    void add(T item);

    void set(int index, T item);

    T get(int index);
}
```



© 2019 ESTECO SpA

Behavior parameterization

```
static void main() {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    TreeSet<String> lengthOrder = new TreeSet<>(new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() - o2.length();
        }
    });
    lengthOrder.addAll(citiesOfFvg);

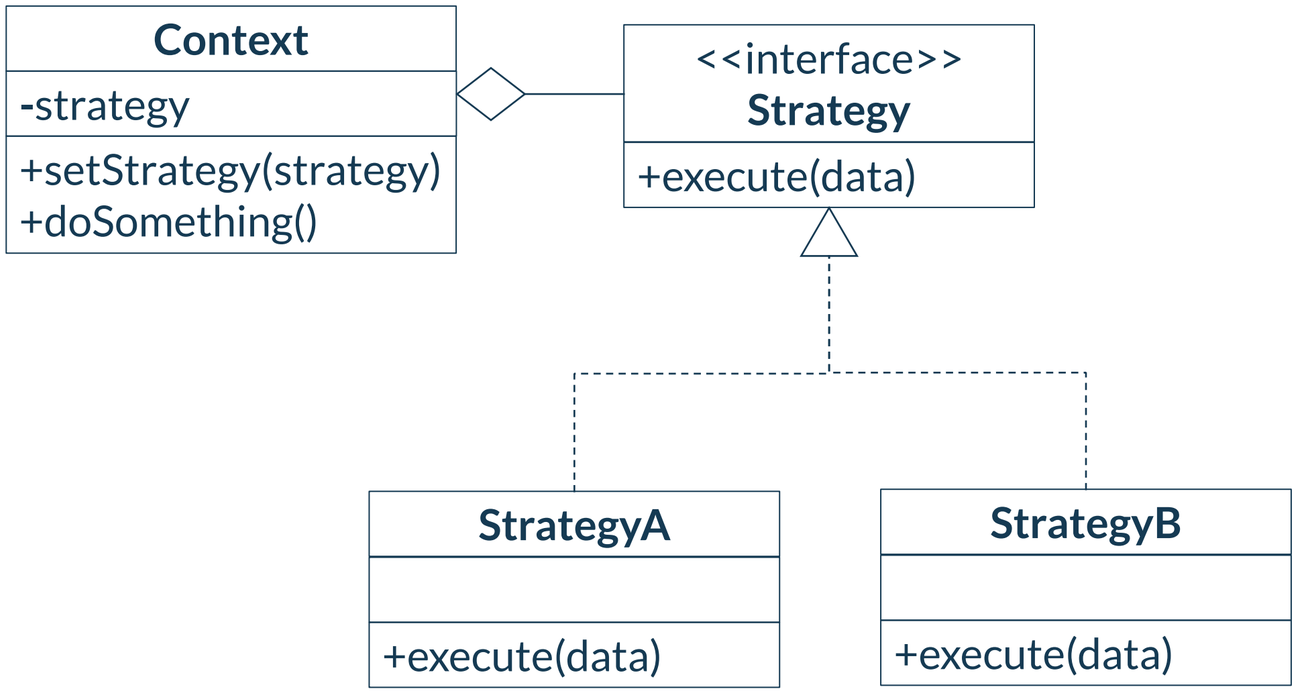
    IO.println(lengthOrder);
}
```

The `TreeSet` class defines a set in which the elements are sorted by a criteria, but it doesn't define what's the criteria, that criteria is a parameter



© 2019 ESTECO SpA

Digression – Strategy pattern



© 2019 ESTECO SpA

(Digression) - Abstract method and template methods

com.esteco.sdm.Shop

```
public abstract class Shop {

    private String name;
    private boolean open;

    protected Shop(String name) {
        this.name = name;
    }

    public final void open() {
        open = true;
        opened();
    }

    protected abstract void opened();
}
```



© 2019 ESTECO SpA

Digression – The open-closed principle

"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"

Abstract methods (inheritance)

Strategy pattern (composition)

Meyer, Bertrand (1988). Object-Oriented Software Construction. Prentice Hall

© 2019 ESTECO SpA

Example List.sort()

```
public interface List<E> extends Collection<E> {

  …

  default void sort(Comparator<? super E> c) {
    …
  }
}
```

```
public interface Comparator<T> {

  int compare(T o1, T o2);

  …
}
```

```
static void main() {
    List<String> list = new ArrayList<>();
    list.add("Trieste");
    list.add("Muggia");
    list.add("Duino-Aurisina");
    list.add("Sgonico");
    list.add("Monrupino");
    list.add("San Dorligo della Valle");

    list.sort(new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.compareTo(o2);
        }
    });

    IO.println(list);
}
```



© 2019 ESTECO SpA

Example Thread

```
public class Thread implements Runnable {

  public Thread()

  public Thread(Runnable target)

  public Thread(String name)

  public Thread(Runnable target, String name)

  …
}
```

```
public interface Runnable {

    public abstract void run();
}
```

```
static void main() {
  var thread = new Thread(new Runnable() {
    @Override
    public void run() {
      while (true) {
        IO.println("Running");
        try {
           Thread.sleep(5000);
        } catch (Exception ex) {
           ex.printStackTrace();
        }
      }
    }
  });
  thread.start();
  IO.println("End of main");
}
```



© 2019 ESTECO SpA

Example JButton action

```
public class JButton extends AbstractButton {

  …
  public void addActionListener(ActionListener l) {
    listenerList.add(ActionListener.class, l);
  }
}
```

```
public interface ActionListener extends EventListener {

  public void actionPerformed(ActionEvent e);

}
```

```
JButton button = new JButton("Click here!");
button.addActionListener(new ActionListener() {
  Override
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(button, "Hello, World!");
  }
});
```



© 2019 ESTECO SpA

Example with logging

```
public class Logger {
  …
  public void log(Level level, String msg) {
    if (!isLoggable(level)) {
      return;
    }
    LogRecord lr = new LogRecord(level, msg);
    doLog(lr);
  }

  public void log(Level level, Supplier<String> msgSupplier) {
    if (!isLoggable(level)) {
      return;
    }
    LogRecord lr = new LogRecord(level, msgSupplier.get());
    doLog(lr);
  }
}
```

```
public interface Supplier<T> {
    T get();
}
```

```
static void main(String[] args) {
  Logger logger = Logger.getAnonymousLogger();

  logger.log(Level.INFO, Arrays.toString(args));

  logger.log(Level.INFO, new Supplier<String>() {
    @Override
    public String get() {
      return Arrays.toString(args);
    }
  });
}
```

The string is lazily created only when needed



© 2019 ESTECO SpA

What do they have in common?

```
public interface Comparator<T> {

  int compare(T o1, T o2);

}
```

```
public interface Runnable {

    public abstract void run();

}
```

```
public interface ActionListener extends EventListener {

    public void actionPerformed(ActionEvent e);

}
```

```
public interface Supplier<T> {

    T get();

}
```



© 2019 ESTECO SpA

Functional interfaces

A functional interface is an interface
that contains only one abstract method

The abstract methos is called function descriptor

Is behavior parameterization limited to functional interfaces? Not necessarily!

Functional interfaces are just a very common case of behavior parameterization

Functional interfaces can be implemented by "regular" classes, anonymous classes, lambda expressions, and method references



© 2019 ESTECO SpA



Lambda expressions

© 2019 ESTECO SpA

A lambda expression is an implementation of a functional interface (or of the function descriptor)



© 2019 ESTECO SpA

Example JButton action

```
public class JButton extends AbstractButton {

  …
  public void addActionListener(ActionListener l) {
    listenerList.add(ActionListener.class, l);
  }
}
```

```
public interface ActionListener extends EventListener {

  public void actionPerformed(ActionEvent e);

}
```

```
JButton button = new JButton("Click here!");
button.addActionListener(e -> JOptionPane.showMessageDialog(button, "Hello, World!"));
```

© 2019 ESTECO SpA

Example with logging

```
static void main(String[] args) {
  Logger logger = Logger.getAnonymousLogger();

  logger.log(Level.INFO, Arrays.toString(args));

  logger.log(Level.INFO, () -> Arrays.toString(args));
}
```

```
public class Logger {
  …
  public void log(Level level, String msg) {
    if (!isLoggable(level)) {
      return;
    }
    LogRecord lr = new LogRecord(level, msg);
    doLog(lr);
  }

  public void log(Level level, Supplier<String> msgSupplier) {
    if (!isLoggable(level)) {
      return;
    }
    LogRecord lr = new LogRecord(level, msgSupplier.get());
    doLog(lr);
  }
}
```

```
public interface Supplier<T> {
    T get();
}
```



© 2019 ESTECO SpA

Example List.sort()

```
public interface List<E> extends Collection<E> {

  …

  default void sort(Comparator<? super E> c) {
    …
  }
}
```

```
public interface Comparator<T> {

  int compare(T o1, T o2);

  …
}
```

```
static void main() {
    List<String> list = new ArrayList<>();
    list.add("Trieste");
    list.add("Muggia");
    list.add(“Duino-Aurisina");
    list.add("Sgonico");
    list.add("Monrupino");
    list.add("San Dorligo della Valle");

    list.sort((o1, o2) -> o1.compareTo(o2));
}

System.out.println(list);
```

© 2019 ESTECO SpA

Example Thread

```
public class Thread implements Runnable {

  public Thread()

  public Thread(Runnable target)

  public Thread(String name)

  public Thread(Runnable target, String name)

  …
}
```

```
public interface Runnable {

    public abstract void run();
}
```

```
static void main() throws Exception {
  var thread = new Thread(() -> {
    IO.println("Running");
    try {
      Thread.sleep(1000);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  });
  thread.start();
  IO.println("End of main");
}
```



© 2019 ESTECO SpA

Lambda syntax

```
(x1, x2)    ->    x1 + x2
```

```
lambda      ->    lambda
parameters        body
```

parameters required by the lambda expression

arrow operator, reads as “becomes” or “goes to”

the actions of the lambda expression

Are we summing up two Integers, two Doubles,
one Integer and one Double, or concatenating two Strings?



© 2019 ESTECO SpA

Summary of lambda syntax

Lambda parameters	
`()`	empty parameters list
`param` or `(param)`	single parameter
`(param1, param2, …, paramN)`	multiple parameters are enclosed in brackets

`->`

Lambda body	
`expression`	single expression style
`{ … return expression}`	block style returning a value
`{ … }`	block style with no return value

A lambda must be compatible with the method defined by the functional interface

- parameters
- return value
- thrown exceptions



© 2019 ESTECO SpA

Examples

`ActionListener l = e -> showMessageDialog("Hello, World!");`	one parameter ActionEvent no return value
`Comparator<Comparable> c = (o1, o2) -> o1.compareTo(o2);`	two parameters Comparable int return value
`Comparator<String> c = (o1, o2) -> o1.compareTo(o2);`	two parameters String int return value
`Supplier s = () -> Arrays.toString(args)`	no parameters Object return value
`Supplier<String> s = () -> Arrays.toString(args)`	no parameters String return value
`Runnable r = () -> { try { IO.println("running"); sleep(1000); } catch (Exception ex) { ex.printStackTrace(); } }`	no parameters no return value

Parameter types and return value are inferred from the implemented functional interface

To increase readability, we can specify the parameter types

The same lambda expression can be assigned to different functional interfaces



© 2019 ESTECO SpA

Capturing lambdas

Can lambda expressions use variables outside the scope of the lambda?

Yes, with the same restrictions of anonymous classes

CapturingLambda.java

```
public class CapturingLambda {

  private double a = 3.14;
  private Runnable runnable;

  public CapturingLambda() {
    double b = 0.1;
    runnable = () -> IO.println(a + b);
  }

  public void apply() {
    runnable.run();
  }

  public void update() {
    a = 9.81;
  }

  static void main() {
    CapturingLambda cl = new CapturingLambda();
    cl.apply();
    cl.update();
    cl.apply();
  }
}
```

Local variables must be final or effectively final

No restrictions on instance variables



© 2019 ESTECO SpA

Take aways

- ❑ Lambda expressions are implementations of functional interfaces and function descriptors
- ❑ Lambda expressions are both objects and functions at the same time
- ❑ Parameters and return types can be inferred from the implemented functional interface
- ❑ Local variables can be used only if final or effectively final
- ❑ Instance variables can be used without restrictions



© 2019 ESTECO SpA



The `java.util.function` package

© 2019 ESTECO SpA

Basic functional interfaces in `java.util.function`

Interface	Function	Purpose
`Function<T, R>`	`R apply(T t)`	Applies an operation to an object of type T and returns the result as an object of type R
`Consumer<T>`	`void accept(T t)`	Applies an operation on an object of type T
`Supplier<T>`	`T get()`	Returns an object of type T
`Predicate<T>`	`boolean test(T t)`	Determines if an object of type T fulfills some constraint. Returns a boolean value that indicates the outcome



© 2019 ESTECO SpA

Function<T, R>

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

Function<Integer, Integer> f1 = x -> x * 2;
Function<String, String> f2 = x -> x + x;

IO.println(p.compose(f1).andThen(f2).apply(3));
```

The `Function` interface is close to the idea of “function” we have from mathematics

In general, a function is not expected to produce side-effects

Default methods	Description
`<V> Function<T,V> andThen(Function<? super R,? extends V> after)`	Returns a composed function that first applies this function to its input, and then applies the after function to the result
`<V> Function<V,R> compose(Function<? super V,? extends T> before)`	Returns a composed function that first applies the before function to its input, and then applies this function to the result



© 2019 ESTECO SpA

Consumer<T>

```
Consumer<String> c = x -> IO.println(x);

Consumer<String> c2 = c.andThen(y -> System.err.println(y));

c2.accept("Software Development Methods");
```

A `Consumer` performs
an action given an item

A consumer is not a “function” in mathematical sense, and it is expected to produce side-effects

Default methods	Description
`Consumer<T> andThen(Consumer<? super T> after)`	Returns a composed Consumer that performs, in sequence, this operation followed by the after operation



© 2019 ESTECO SpA

Supplier<T>

```
Supplier<Long> s = () -> System.currentTimeMillis();

IO.println(s.get());

Supplier<String> m = () -> Arrays.toString(args);

Logger.getAnonymousLogger().log(Level.INFO, m);
```

A `Supplier` provides a value

A supplier doesn't take any argument, it can be easily associated with lazy evaluation

The returned value is not evaluated in advance but only when it is needed



© 2019 ESTECO SpA

Predicate<T>

```
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();

IO.println(notBetweenZeroAndFive.test(6));
```

A `Predicate` is a function that returns a boolean value

Default methods	Description
`Predicate<T> and(Predicate<? super T> other)`	Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another
`Predicate<T> negate()`	Returns a predicate that represents the logical negation of this predicate
`Predicate<T> or(Predicate<? super T> other)`	Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another



© 2019 ESTECO SpA

Variations

- Functions have a natural arity based on how they are most used.
- The basic shapes can be modified by an arity prefix to indicate a different arity, such as BiFunction (binary function from T and U to R).
- There are additional derived function shapes which extend the basic function shapes, including UnaryOperator (extends Function) and BinaryOperator (extends BiFunction).

Interface	Function	Purpose
`BiFunction<T, U, R>`	`R apply(T t, U u)`	Accepts two arguments and produces a result, this is the two-arity specialization of Function
`UnaryOperator<T>`	`T apply(T t)`	Operation on a single operand that produces a result of the same type as its operand, this is a specialization of Function where the operand and the result are of the same type
`BinaryOperator<T>`	`T apply(T t0, T t1)`	Operation upon two operands of the same type, producing a result of the same type as the operands. This is a specialization of BiFunction for the case where the operands and the result are all the same type

© 2019 ESTECO SpA

Specializations

- Type parameters of functional interfaces can be specialized to primitives with additional type prefixes
- To specialize the return type for a type that has both generic return type and generic arguments, we prefix ToXxx, as in ToIntFunction
- Otherwise, type arguments are specialized left-to-right, as in DoubleConsumer or ObjIntConsumer
- These schemes can be combined, as in IntToDoubleFunction
- If there are specialization prefixes for all arguments, the arity prefix may be left out

Interface	Function	Purpose
`ToIntFunction<T>`	`int applyAsInt(T value)`	Produce an int-valued result
`DoubleConsumer`	`accept(double value)`	Perform an operation on a Double value
`ObjIntConsumer<T>`	`accept(T t, int value)`	Operation that accepts an object-valued and an int-valued argument, returns no result
`IntToDoubleFunction`	`double applyAsDouble(int value)`	Accepts an int-valued argument and produces a double-valued result

© 2019 ESTECO SpA

Take aways

- The `java.util.function` package defines functional interfaces for common operations, such as `Function`, `Consumer`, `Supplier`, and `Predicate`
- Variations on arity or derived extensions are available
- Specialized versions exist to work with `int`, `long`, and `double` primitive types
- Provided default methods allow the combination of such operations



© 2019 ESTECO SpA

Method references

© 2019 ESTECO SpA

Method references

The implementation of a functional interface require the specification of a compatible method

A functional interface can be implemented by a compatible
- Java class (possibly anonymous)
- lambda expression
- method reference

```
interface IntFunction<T> {
    int apply(T t);
}

static void main() {
    IntFunction<String> f1 = x -> x.length();
    IntFunction<String> f2 = String::length;

    IO.println(f1.apply("Software Development Method"));
    IO.println(f2.apply("Java is great!"));
}
```

Any method that applied to a String returns an int

Reference to the length method of the String class, invoked on the parameter



© 2019 ESTECO SpA

Method references

class name :: method name

```
interface IntBiFunction<T, U> {
    int apply(T t, U u);
}

static void main() {
    IntBiFunction<String, Character> b1 = (s, c) -> s.indexOf(c);
    IntBiFunction<String, Character> b2 = String::indexOf;

    IO.println(b1.apply("Software Development Methods", 't'));
    IO.println(b2.apply("Java is great!", 't'));
}
```

The first parameter of the function descriptor is the object upon which we invoke the method

The next parameters of the function descriptor become the actual parameters of the method



© 2019 ESTECO SpA

Static method references

class name :: static method name

Any method taking 0 parameters and returning a long value

```
interface LongSupplier {
    long get();
}

static void main() {
    LongSupplier s1 = () -> System.currentTimeMillis();
    LongSupplier s2 = System::currentTimeMillis;

    IO.println(s1.get());
    IO.println(s2.get());
}
```

Reference to the currentTimeMillis static method of the System class

© 2019 ESTECO SpA

Class constructor references

class name :: new

```
interface ListSupplier {
    List get();
}

static void main() {
    ListSupplier s1 = () -> new ArrayList();
    ListSupplier s2 = ArrayList::new;

    IO.println(s1.get());
    IO.println(s2.get());
}
```

© 2019 ESTECO SpA

Instance method references

object reference :: method name

```
interface RandomGenerator {
    int get(int scale);
}

static void main() {
    Random random = new Random();
    RandomGenerator g1 = s -> random.nextInt(s);
    RandomGenerator g2 = random::nextInt;

    IO.println(g1.get(10));
    IO.println(g2.get(10));
}
```

The lambda expressions uses a method defined in an object captured by the lambda



© 2019 ESTECO SpA

Class constructor references

```
interface ListSupplier {
    List get();
}

interface ListSupplier2 {
    List get(int capacity);
}

static void main() {
    ListSupplier s1 = () -> new ArrayList();
    ListSupplier s2 = ArrayList::new;

    IO.println(s1.get());
    IO.println(s2.get());

    ListSupplier2 s3 = c -> new ArrayList(c);
    ListSupplier2 s4 = ArrayList::new;

    IO.println(s3.get(25));
    IO.println(s4.get(25));
}
```

Is this code legal?

Are they references to the same constructor?

The, sad, answer is NO!

© 2019 ESTECO SpA

Take aways

- In most cases method references can replace lambda expressions
- It takes a while to get used to method references



© 2019 ESTECO SpA

Exercise

© 2019 ESTECO SpA

Exercise

Reimplement the `TermFrequency` exercise by using lambda expressions/method references.
And by allowing the user to print the table following different sorting strategies.



© 2019 ESTECO SpA

Self assessment

© 2019 ESTECO SpA

Quiz 1: Behavior parameterization

1. **What is the primary purpose of behavior parameterization?**
   A. To allow a method to operate on different data types using generics
   B. To make a class closed for modification, but open for extension (Open-Closed Principle)
   C. To allow an algorithm to be defined by a class or method while deferring the execution of specific steps to a parameter (e.g., an object implementing a required interface)
   D. To reuse code by passing simple data values like integers or strings into a method

2. **In the provided examples (`List.sort()`, `Thread`, `TFBuilder`), how is the behavior typically passed as a parameter before the introduction of lambda expressions?**
   A. By passing a simple data type like an `int` or `String`
   B. By passing an object that implements a single-method interface (like `Comparator` or `Runnable`), often using an anonymous inner class
   C. By passing a method reference directly to the invocation
   D. By defining the behavior as an abstract method in the base class



© 2019 ESTECO SpA

Quiz 1: Behavior parameterization

3. **The use of `Supplier<String>` in the `Logger` class for message creation (`log(Level, Supplier<String>)`) is an example of which concept?**
   A. Thread synchronization
   B. Eager evaluation
   C. Lazy evaluation, where the expensive string creation only occurs if the logging level is enabled
   D. The Strategy pattern



© 2019 ESTECO SpA

Quiz 2: Lambda expressions

1. **What is the strict definition of a functional interface in Java?**
   A. An interface that has at least two abstract methods
   B. An interface that is part of the `java.util.function` package
   C. An interface that contains only one abstract method (known as the function descriptor)
   D. Any interface that is implemented by a lambda expression
2. **What does the arrow operator (`->`) represent in the syntax of a lambda expression?**
   A. It denotes a block of code with a return value
   B. It separates the lambda body from the return type
   C. It is used to declare that the lambda implements a functional interface
   D. It separates the lambda's parameters from its body, read as "becomes" or "goes to“



© 2019 ESTECO SpA

Quiz 2: Lambda expressions

3. **Which rule applies to the use of a local variable (defined outside the lambda) within a lambda expression's body?**
   A. Local variables must be declared final or be effectively final (their value cannot change after initialization)
   B. There are no restrictions on local variables, only on instance variables
   C. The variable can be modified freely within the lambda's body, but only if it's a primitive type
   D. Local variables must be static
4. **In a lambda expression like `(o1, o2) -> o1.compareTo(o2)`, how does the compiler determine the type of the parameters (o1 and o2) and the return value?**
   A. The compiler uses runtime reflection to check the argument types
   B. The types must always be explicitly declared in the parameter list
   C. The types are inferred from the abstract method (function descriptor) of the functional interface it is assigned to
   D. The types are determined by Java's built-in `Comparator` class



© 2019 ESTECO SpA

Quiz 3: The `java.util.function` package

1. **Which functional interface from `java.util.function` is designed to accept an input of type T, perform an operation, and return a result of type R?**
   A. `Function<T, R>`
   B. `Consumer<T>`
   C. `Supplier<T>`
   D. `Predicate<T>`
2. **The `Predicate<T>` interface is primarily used to perform what type of operation?**
   A. Apply an operation to an object and produce a side-effect
   B. Apply an operation to an object and return a new object of the same type
   C. Determine if an object of type T fulfills some constraint, returning a boolean value
   D. Return an object of type T without taking any arguments



© 2019 ESTECO SpA

Quiz 3: The `java.util.function` package

3. **Which variation of the basic functional interfaces is a specialization of `Function` where the operand and the result are required to be the same type?**
   A. `BiFunction<T, U, R>`
   B. `UnaryOperator<T>`
   C. `BinaryOperator<T>`
   D. `ToXxxFunction<T>`

4. **The default methods `and()`, `negate()`, and `or()` are part of which core functional interface?**
   A. `Function<T, R>`
   B. `Consumer<T>`
   C. `Supplier<T>`
   D. `Predicate<T>`



© 2019 ESTECO SpA

Quiz 4: Method references

1. **A method reference is a shorthand for a lambda expression that simply calls an existing method. Which method reference syntax is used to create an implementation of a functional interface by referring to a constructor (e.g., `Supplier<ArrayList>`)?**
   A. `objectReference::new`
   B. `ClassName::constructor`
   C. `ClassName::new`
   D. `ClassName::create`

2. **The method reference `String::length` is an example of what type of method reference?**
   A. A reference to a constructor
   B. A reference to an instance method of an object captured by the lambda
   C. A reference to an instance method of an arbitrary object of a particular type (where the first lambda parameter is the object instance)
   D. A reference to a static method



© 2019 ESTECO SpA

Quiz 4: Method references

3. **What is the structure of a method reference used to refer to a static method (e.g., for a `LongSupplier` implementing `System.currentTimeMillis()`)?**
   A. `objectReference::methodName`
   B. `objectReference::staticMethodName`
   C. `ClassName::staticMethodName`
   D. `static::methodName`



© 2019 ESTECO SpA

Correct answers

Quiz 1:

1. B
2. B
3. C

Quiz 2:

1. C
2. D
3. A
4. C

Quiz 3:

1. A
2. C
3. B
4. D

Quiz 4:

1. C
2. B
3. C

© 2019 ESTECO SpA



Thank you!

esteco.com

Read the ESTECO Copyright Policy © 2019 ESTECO SpA