Programming in Java – Lambda expressions

Paolo Vercesi
Technical Program Manager

Agenda

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

Behavior parameterization

Parameterization

So far, we 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

“Data” parameterization

```java
public class Accumulator {

  private int value;

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

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

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

```java
public class Accumulator {

  private int value;

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

Parametrized type

```java
public interface List {

    void add(Object item);

    void set(int index, Object item);

    Object get(int index);
}
```

```java
public interface StringList {

    void add(String item);

    void set(int index, String item);

    String get(int index);
}
```

```java
public interface List<T> {

    void add(T item);

    void set(int index, T item);

    T get(int index);
}
```

Behavior parameterization

```
public class TFBuilder {
  …
  TF build(String text) {
    Collection<String> tokens = tokenizer.tokenize(text);
    Map<String, Integer> map = new HashMap<>();
    for (String term : tokens) {
      term = normalizer.normalize(term);
      if (filter.accept(term)) {
        map.merge(term, 1, new BiFunction<>() {
          @Override
          public Integer apply(Integer v, Integer d) {
            return v + 1;
          }
        });
      }
    }
    return new TF(map);
  }
}
```

The TFBuilder class defines what is the algorithm to follow to build a TermFrequency object, but is doesn't define how we tokenize a string nor how we normalize and filter the terms

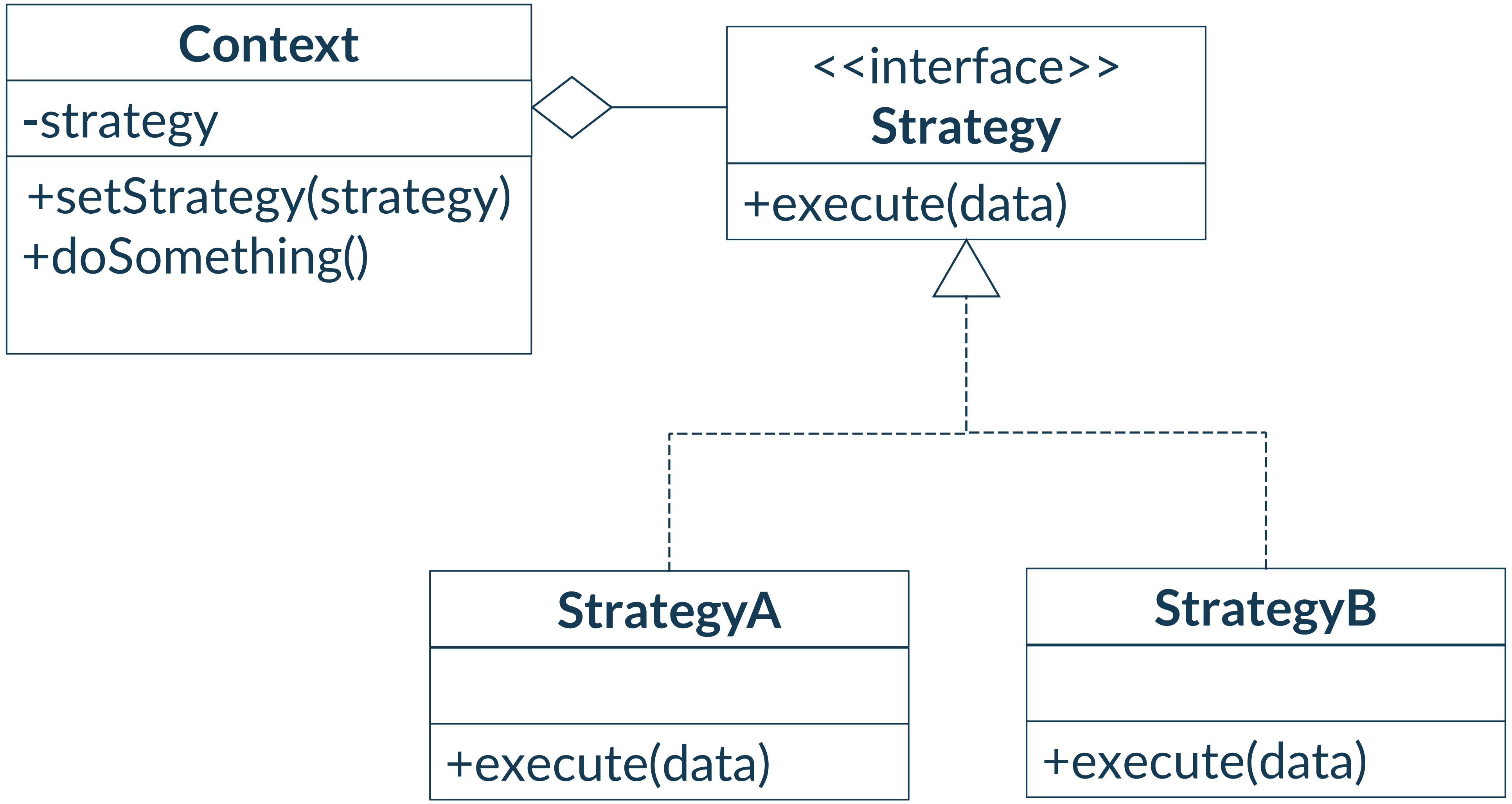
Tokenization, normalization, and filtration are parametrized

```
public interface Tokenizer {
  Collection<String> tokenize(String text);
}

public interface Normalizer {
  String normalize(String token);
}

public interface Filter {
  boolean accept(String token);
}
```

Digression – Strategy pattern



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

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);

  …
}
```

```
public static void main(String[] args) {
    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);
        }
    });
}

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

Example Thread

```java
public class Thread implements Runnable {

  public Thread()

  public Thread(Runnable target)

  public Thread(String name)

  public Thread(Runnable target, String name)

  …
}
```

```java
public interface Runnable {

    public abstract void run();
}
```

```java
public static void main(String[] args) throws Exception {
  var thread = new Thread(new Runnable() {
    @Override
    public void run() {
      while (true) {
        System.out.println("Running");
        try {
           Thread.sleep(1000);
        } catch (Exception ex) {
           ex.printStackTrace();
        }
      }
    }
  });
  thread.start();
  System.out.println("End of main");
}
```

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!");
  }
});
```

Example with logging

```java
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);
  }
}
```

```java
public 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

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

What do they have in common?

```java
public interface Comparator<T> {

  int compare(T o1, T o2);

  …
}
```

```java
public interface Runnable {

    public abstract void run();
}
```

```java
public interface ActionListener extends EventListener {

    public void actionPerformed(ActionEvent e);

}
```

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

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

Lambda expressions

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

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!"));
```

Example with logging

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

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

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

```java
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);
  }
}
```

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

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);

  …
}
```

```
public static void main(String[] args) {
    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);
```

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();
}
```

```
public static void main(String[] args) throws Exception {
  var thread = new Thread(() -> {
    System.out.println("Running");
    try {
      Thread.sleep(1000);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  });
  thread.start();
  System.out.println("End of main");
}
```

Lambda syntax

(x1, x2) -> x1 + x2

lambda parameters -> lambda 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?

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

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 {`
`System.out.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

Capturing lambdas

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

Yes, with the same restrictions of anonymous classes

```
public class CapturingLambda {

  private double a = 3.14;

  public CapturingLambda() {
    double b = 0.1;
    Runnable lambda = () -> System.out.println(a + b);
    lambda.run();
    a = 6;
    lambda.run();
  }

  public static void main(String[] args) {
    CapturingLambda capturingLambda = new CapturingLambda();
  }
}
```

Local variables must be final or effectively final

No restrictions on instance variables

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

Method references

Method references

The implementation of a function 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);
}

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

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

Any method that applied to a String return an int

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

Method references

class name :: method name

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

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

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

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

The next parameters of the lambda become the actual parameters of the method

Static method references

class name :: static method name

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

public static void main(String[] args) {
    LongSupplier s1 = () -> System.currentTimeMillis();
    LongSupplier s2 = System::currentTimeMillis;

    System.out.println(s1.get());
    System.out.println(s2.get());
}
```

Any method taking 0 arguments and returning a long value

All lambda parameters becomes actual parameters when invoking the method

Reference to the currentTimeMillis static method of the System class

Class constructor references

class name :: new

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

public static void main(String[] args) {
    ListSupplier s1 = () -> new ArrayList();
    ListSupplier s2 = ArrayList::new;

    System.out.println(s1.get());
    System.out.println(s2.get());
}
```

All lambda parameters becomes actual parameters when invoking the constructor

Instance method references

object reference :: method name

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

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

    System.out.println(g1.get(10));
    System.out.println(g2.get(10));
}
```

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

All lambda parameters becomes actual parameters when invoking the method

Class constructor references

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

interface ListSupplier2 {
    List get(int capacity);
}

public static void main(String[] args) {
    ListSupplier s1 = () -> new ArrayList();
    ListSupplier s2 = ArrayList::new;

    System.out.println(s1.get());
    System.out.println(s2.get());

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

    System.out.println(s3.get(25));
    System.out.println(s4.get(25));
}
```

Is this code legal?

Are they references to the same constructor?

The, sad, answer is NO!

Take aways

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

The java.util.function package

Basic functional interfaces in java.util.function

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

Function<T, R>

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

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

System.out.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

Consumer<T>

```
Consumer<String> c = x -> System.out.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

Supplier<T>

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

System.out.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

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();

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

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.

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

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

Take aways

- The `java.util.function` package defines functional interfaces for common operations, such as Function, Consumer, Supplier, 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

Assignment

Assignment

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

Thank you!

esteco.com