Java Functional Programming (Functions)

Functional vs imperative programming

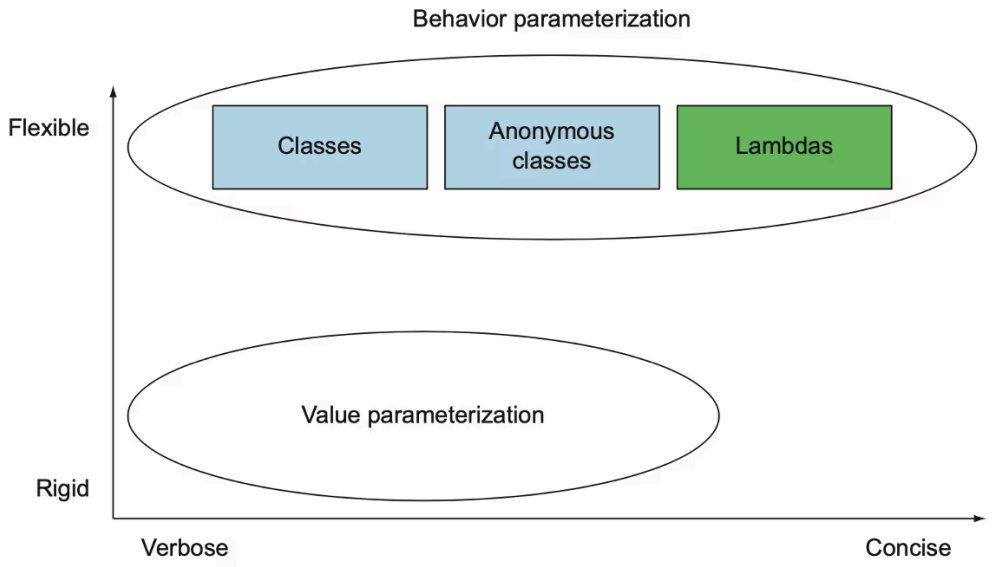
Because of changing needs, a number of languages (Java, Python, Scala) are introducing ways for supporting functional programming.

The functional programming paradigm describes a program by applying and composing functions.

- Passing functions to functions (**behaviour parametrization**)
- Functional programs can be very terse and elegant, packing a lot of behaviour into very few lines of code. Functional programmers make the case that in a multicore world, you need to avoid mutable state in order to scale out your programs.

The imperative programming paradigm allows you to describe a program in terms of a sequence of statements that mutate state.

- Passing objects to functions (**value parametrization**)
- Object-oriented programmers retort that in actual business environments object-oriented programming scales out well in terms of developers and, as an industry, we know how to do it.



Lambda expressions

By **lambda expression** (or just "a lambda"), we mean a **function** that isn't bound to its name (an anonymous function) but can be assigned to a variable.

The most general form of a lambda expression looks like this:

`(parameters) -> { body };`.

The brackets `{ }` are required only for multi-line lambda expressions:

`(parameters) -> expression`

The part before `->` is the list of parameters (like in methods), and the part after that is the body that can return a value.

Lambda expressions examples

A boolean expression: `(List<String> list) -> list.isEmpty()`

Creating objects: `() -> new Apple(45)`

Consuming from an object: `(Apple a) -> { System.out.println(a.getWeight()); }`

Select/extract a field from an object: `(String s) -> s.length()`

Multiply two integers: `(int a, int b) -> a * b`

Compare two objects: `(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`

A case study: filtering students

We need to select all students with a given average, and also print a string representation of the selected students.

```
public static List<Student> filterStudentsByGrade(List<Student> stu
    List<Student> result = new ArrayList<>();
    for (Student s : students) {
        if (s.getAverage() == average) {
            String str = String.format("%s_%s_%f", s.getLastname(),
            System.out.println(str);
            result.add(s);
        }
    }
    return result;
}
```

After a while, requirements change, and you need to select all the students with an average comprised within a given range. You can add an alternative method.

```
public static List<Student> filterStudentsByGradeRange(List<Student:
    List<Student> result = new ArrayList<>();
    for (Student s : students) {
        if (s.getAverage() >= low && s.getAverage() <= high) {
            String str = String.format("%s_%s_%f", s.getLastname(),
            System.out.println(str);
            result.add(s);
        }
    }
    return result;
}
```

However, this approach breaks the DRY (Don't Repeat Yourself) principle. The two methods vary only in one line!

Strategy Pattern

We can define a set of interfaces and implement the Strategy Pattern. Each interface can have multiple implementations for different strategies.

```
public interface StudentPredicate {
    boolean test(Student s);
}
```

```
public interface StudentFunction {
    String apply(Student s);
}
```

```
public interface StudentConsumer {
    void accept(String s);
}
```

```
public static List<Student> filterStudents(List<Student> students, S
    List<Student> result = new ArrayList<>();
    for (Student s : students) {
        if (sp.test(s)) {
            String str = sf.apply(s);
            sc.accept(str);
            result.add(s);
        }
    }
    return result;
}
```

This code is much more flexible than our first attempt. We can pass to the *filterStudents* method actual behaviour instead of parameters! However, when you want to pass behaviour, you're forced to one of these options:

(a) provide classes implementing the interfaces and then instantiate the needed objects (verbose)!

```
class StudentGoodPredicate implements StudentPredicate {
    public boolean test(Student p) {
        return s.getAverage() >= 26 && s.getAverage() <= 30;
    }
}
```

```
class StudentSoAndSoPredicate implements StudentPredicate {
    public boolean test(Student p) {
        return s.getAverage() >= 20 && s.getAverage < 24;
    }
}
```

(b) provide anonymous implementations of the interfaces (better but still verbose!).

```
public static void main(String[] args) {
    List<Student> result = filterStudents(students, new StudentPred:
        @Override
        public boolean test(Student s) {
            return s.getAverage() >= 26 && s.getAverage() <= 30;
        }
    }, new StudentFunction() {
        @Override
        public String apply(Student s) {
            return String.format("%s_%s_%f", s.getLastname(), s.getI
        }
    }, new StudentConsumer() {
        @Override
        public void accept(String s) {
            System.out.println(s);
        }
    });

}
```

(c) provide anonymous implementations of the interfaces using **lambda expressions**. Indeed, functional interfaces or **interfaces defining only one method** are ideal candidates for making use of lambda expressions. **Lambda expressions can be used for providing the implementation of their single method**!

```
public static void main(String[] args) {
    List<Student> result = filterStudents(students,
            s -> s.getAverage() >= 26 && s.getAverage() <= 30,
            s -> String.format("%s_%s_%f", s.getLastname(), s.getNar
            s -> System.out.println(s));
}
```

Strategy Pattern + Generics

Instead of using custom interfaces designed only for students, we can use generic functional interfaces included in the Java API such as: `Function<T,R>`, `Predicate<T>`, `Consumer<T>` (see package java.util.function).

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T s);
}
```

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T s);
}
```

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T s);
}
```

By making use of generics we can further generalize the method we're studying. It can now receive a generic `Predicate<T>`, `Function<T,R>`, `Consumer<R>` and a list of type T. This allows another remarkable improvement in expressiveness with the same amount of code. In fact, it can receive a generic list (not only a student list), filter it based on a predicate, and print the result of transformation!

```
public static <T, R> List<T> filter(List<T> l, Predicate<T> sp, Fun
    List<T> result = new ArrayList<>();
    for (T s : l) {
        if (sp.test(s)) {
            R x = sf.apply(s);
            sc.accept(x);
            result.add(s);
        }
    }
    return result;
}


public static void main(String[] args) {
    result = filterStudents(students,
            s -> s.getAverage() >= 26 && s.getAverage() <= 30,
            s -> String.format("%s_%s_%f", s.getLastname(), s.getNa
            s -> System.out.println(s));
}
```

Functional interfaces

Function

As seen above, the most general case of a lambda is a functional interface with a method that receives one value and returns another. This function is represented by the *Function* interface, which is parameterized by the types of its argument and a return value:

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

```
public static void main(String[] args) {
    // if it has only one argument "()" are optional
    Function<Integer, Integer> adder1 = x -> x + 1;

    // with type inference
    Function<Integer, Integer> mult2 = (Integer x) -> x * 2;

    // with multiple statements
    Function<Integer, Integer> adder5 = (x) -> {
        x += 2;
        x += 3;
        return x;
    };

    // with two different types
    Function<String, Integer> length = s -> s.length();
}
```

Predicate

The Predicate functional interface is a specialization of a Function that receives a generified value and returns a boolean.

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

A typical use case of the Predicate lambda is to filter a collection of values:

```
public static void main(String[] args) {
    List<String> names = new ArrayList<>(List.of("Angela", "Aaron",
    names.removeIf(s -> s.startsWith("B"));
}
```

```
interface List<E> {
    // ...
    boolean removeIf(Predicate<? super E> filter);
    // ...
}
```

In the code above, we remove from a list the names that start with the letter "B". The Predicate implementation encapsulates the filtering logic.

Consumer

The Consumer accepts a generified argument and returns nothing. It is a function that is representing side effects.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

For instance, let's greet everybody in a list of names by printing the greeting in the console. The lambda passed to the *List.forEach()* method implements the Consumer functional interface:

```
public static void main(String[] args) {
    List<String> names = new ArrayList<>(List.of("Angela", "Aaron",
    names.forEach(name -> System.out.println("Hello, " + name));
}
```

```
interface Iterable<T> {
    // ...
    void forEach(Consumer<? super T> action);
    // ...
}
```

Supplier

The Supplier accepts nothing and returns a generified results. There is no requirement that a new or distinct result be returned each time the supplier is invoked.

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

For instance, we can consume items coming from different suppliers.

```
public static <T> void personalizedConsumer(Consumer<T> consumer, Su
    consumer.accept(supplier.get());
}

public static void main(String[] args) {
    Supplier<String> supplierBasic = () -> "Hello World!";

    Supplier<String> supplierRandom = () -> {
        RandomGenerator rnd = RandomGenerator.getDefault();
        return "Hello World!-" + Integer.valueOf(rnd.nextInt(100));
    };

    // prints "Hello World!"
    personalizedConsumer(
            str -> System.out.println(str),
            supplierBasic);

    // prints "Hello World!-random number"
    personalizedConsumer(
            str -> System.out.println(str),
            supplierRandom);
}
```

Primitive Function Specializations

Since a primitive type can't be a generic type argument, there are versions of the Function interface for the most used primitive types and their combinations in argument and return types:

- IntFunction, LongFunction, DoubleFunction: arguments are of specified type, return type is parameterized

```
@FunctionalInterface
public interface IntFunction<R> {
    R apply(int value);
}
```

- ToIntFunction, ToLongFunction, ToDoubleFunction: return type is of specified type, arguments are parameterized

```
@FunctionalInterface
public interface ToIntFunction<T> {
    int apply(T value);
}
```

- DoubleToIntFunction, DoubleToLongFunction, IntToDoubleFunction, IntToLongFunction, LongToIntFunction, LongToDoubleFunction: having both argument and return type defined as primitive types, as specified by their names

```
@FunctionalInterface
public interface DoubleToIntFunction {
    int apply(double value);
}
```

Two-Arity Function Specializations

To define lambdas with two arguments, we have to use additional interfaces that contain "Bi" keyword in their names: BiFunction, ToDoubleBiFunction, ToIntBiFunction, and ToLongBiFunction.

BiFunction has both arguments and a return type generified, while ToDoubleBiFunction and others allow us to return a primitive value.

```
@FunctionalInterface
public interface BiFunction<T,U,R> {
    R apply(T t, U u);
}
```

One of the typical examples of using this interface is in the *Map.replaceAll()* method, which allows replacing all values in a `Map<K,V>` with a BiFunction:

```
interface Map<K,V> {
    // ...
    default void replaceAll(BiFunction<? super K,? super V,? extend
    // ...
}
```

```
public static void main(String[] args) {
    Map<String, Integer> salaries = new HashMap<>(Map.of(
            "John", 40000,
            "Freddy", 30000,
            "Samuel", 50000
    ));

    salaries.replaceAll((name, salary) ->
            name.startsWith("F") ? salary * 2 : salary + 2);

    // prints: {Samuel=50002, John=40002, Freddy=60000}
    System.out.println(salaries);
}
```

BiConsumer has both arguments generified and returns void. One of its use cases is iterating through the entries of a map.

```
@FunctionalInterface
public interface BiConsumer<T,U> {
    void accept(T t, U u);
}
```

One of the typical examples of using this interface in the standard API is in the *Map.forEach()* method, which performs the given action for each entry.

```
interface Map<K,V> {
    // ...
    default void forEach(BiConsumer<? super K,? super V> action);
    // ...
}
```

```
public static void main(String[] args) {
    Map<String, Integer> salaries = new HashMap<>(Map.of(
            "John", 40000,
            "Freddy", 30000,
            "Samuel", 50000
    ));

    salaries.forEach((name, salary) -> System.out.println(name + " e

    // prints:
    // Samuel earns 50000 dollars/year!
    // John earns 40000 dollars/year!
    // Freddy earns 30000 dollars/year!
}
```

Operators

Operator interfaces are special cases of a function that receive and return the same value type. The UnaryOperator interface receives a single argument.

```
@FunctionalInterface
public interface UnaryOperator<T> {
    T apply(T t);
}
```

One of its use cases in the Collections API is the *List.replaceAll()* method replacing all items in a list with some computed values of the same type:

```
interface List<E> {
    // ...
    default void replaceAll(UnaryOperator<E> operator);
    // ...
}
```

```
public static void main(String[] args) {
    List<String> names = new ArrayList<>(List.of("Angela", "Aaron",
    names.replaceAll(name -> name.toUpperCase());
}
```

Method references

By method reference, we mean a function that refers to a particular method via its name and can be invoked any time we need it. The base syntax of a method reference

looks like this:

```
objectOrClass :: methodName
```

where `objectOrClass` can be a **class name** or a **particular instance** of a class.

Here is an example, we create a reference to the standard static method `max` of the `Integer` class.

```
// lambda expression
BiFunction<Integer, Integer, Integer> max = (x, y) -> Integer.max(x,


// method reference
BiFunction<Integer, Integer, Integer> max = Integer::max;
```

Here, `Integer::max` is a method reference to a static method.

This code works because the definition of the method `int max(int a, int b)` fits the type `BiFunction<Integer, Integer, Integer>`: they both mean taking two integer arguments and returning an integer value.

Now we have the `max` object that can be used as a function by invoking the `apply` method. Let's invoke it!

```
System.out.println(max.apply(50, 70)); // 70
```

It is recommended to use method references rather than lambda expressions if you just need to invoke a standard method without other operations. Your code will be shorter, more readable, and easier to test.

Kinds of method references

It's possible to write method references to both static and instance (non-static) methods.

In general, there are four kinds of method references:

- reference to a static method;
- reference to an instance method of an existing object;
- reference to an instance method of an object of a particular type;
- reference to a constructor.

Reference to a static method

The general form is the following:

```
ClassName :: staticMethodName
```

Let's take a look at the reference to the static method `sqrt`of the class `Math`:

```
Function<Double, Double> sqrt = Math::sqrt;
```

Now we can invoke the `sqrt` method for double values:

```
sqrt.apply(100.0d); // the result is 10.0d
```

The `sqrt` method can be also written using the following lambda expression:

```
Function<Double, Double> sqrt = x -> Math.sqrt(x);
```

Reference to an instance method of an object

The general form looks like this:

```
objectName :: instanceMethodName
```

Let's check out the example of a reference to the `indexOf` method of a particular string.

```
String whatsGoingOnText = "What's going on here?";

Function<String, Integer> indexWithinWhatsGoingOnText = whatsGoingOn
```

Here is the result of applying it to different arguments:

```
System.out.println(indexWithinWhatsGoingOnText.apply("going")); // 7
System.out.println(indexWithinWhatsGoingOnText.apply("Hi"));    // -
```

As you can see, actually we always work with the `whatsGoingOnText` object captured from the context.

The following example of a lambda expression is a full equivalent of the reference above and can make your understanding of the situation better:

```
Function<String, Integer> indexWithinWhatsGoingOnText = string -> wh
```

Reference to an instance method of an object of a particular type

Here is a general form of a reference:

```
ClassName :: instanceMethodName
```

In that case, you need to pass an instance of the class as the function argument.

Let's focus on the following reference to an instance of the method `doubleValue` of the class `Long`:

```
Function<Long, Double> converter = Long::doubleValue;
```

Now we can invoke the `converter` for long values:

```
converter.apply(100L); // the result is 100.0d
converter.apply(200L); // the result is 200.0d
```

Also, we can write the same converter using the following lambda expression:

```
Function<Long, Double> converter = val -> val.doubleValue();
```

Reference to a constructor

This reference has the following declaration:

```
ClassName :: new
```

For example, let's consider our custom class `Person` with a single field `name`.

```
class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Here is a reference to the constructor of this class:

```
Function<String, Person> personGenerator = Person::new;
```

This function produces new `Person` objects based on their names.

```
Person johnFoster = personGenerator.apply("John Foster"); // we hav
```

Here is the corresponding lambda expression that does the same.

```
Function<String, Person> personGenerator = name -> new Person(name)
```

Summarizing: implementing functional interfaces

There are several ways to implement a functional interface.

Anonymous classes

A functional interface can be implemented by using an anonymous class or regular inheritance, as shown below:

```
Function<Long, Long> square = new Function<Long, Long>() {
    @Override
    public Long apply(Long value) {
        return value * value;
    }
};

long val = square.apply(10L); // the result is 100L
```

In this example, we model a math function that squares a given value. This code works perfectly, but it is a bit unclear since it contains a lot of extra characters to perform a single line of useful code.

Lambda expressions

A functional interface can also be implemented and instantiated by using a lambda expression.

```
Func<Long, Long> square = value -> value * value;

long val = square.apply(10L); // the result is 100L
```

The type of the functional interface (left) and the type of the lambda (right) are the same from a semantic perspective. Parameters and the result of a lambda expression correspond to the parameters and the result of **a single abstract method of the functional interface**.

Method references

Another way to implement a functional interface is by using method references. In this case, the number and type of arguments and the return type should match the number and types of arguments and the return type of the single abstract method of a functional interface.

Suppose, there is a method `square` that takes and returns a `long` value:

```
class Functions {

    public static long square(long value) {
        return value * value;
    }
}
```

The argument and the return type of this method fits the `Function<Long, Long>` functional interface. This means we can create a method reference and assign it to an object of the `Function<Long, Long>` type:

```
Function<Long, Long> square = Functions::square;
```

Optional values

Like many programming languages, Java uses `null` to represent the absence of a value. Sometimes this approach leads to exceptions like **NPEs** since non-null checks make code less readable. The British computer scientist Tony Hoare, the inventor of the `null` concept, even describes introducing `null` as a **"billion-dollar mistake"** since it has led to innumerable errors, vulnerabilities, and system crashes.

The `Optional<T>` class represents the presence or absence of a value of the specified type `T`. An object of this class can be either **empty** or **non-empty**. `Optional` is like a box that contains either some value or nothing. It wraps a value or `null` keeping the possibility to check it by using special methods.

Creating an Optional

In the following code, we create two Optional objects. The first object represents an empty value (such as `null`), and the second one keeps a string value. The `isPresent` method checks whether an object is empty or not.

```
public static void main(String[] args) {
    Optional<String> absent = Optional.empty();
    Optional<String> present = Optional.of("Hello");

    System.out.println(absent.isPresent()); // false
    System.out.println(present.isPresent()); // true
}
```

In a situation when you don't know whether a variable is `null` or not, you should pass it to the `ofNullable` method instead of the `of` method. It creates an empty Optional if the passed value is `null`.

```
public static void main(String[] args) {
    Optional<String> optMessage1 = Optional.ofNullable(null);
    Optional<String> optMessage2 = Optional.ofNullable("Hello");

    System.out.println(optMessage1.isPresent()); // false
    System.out.println(optMessage2.isPresent()); // true

}
```

Getting the value from an Optional

The most obvious thing to do with an Optional is to get its value.

- `get` returns the value if it's present, otherwise throws an exception;
- `orElseThrow` behaves as `get`, but the name describes it better;
- `orElse(T other)` returns the value if it's present, otherwise returns `other`;
- `orElseGet(Supplier<? extends T> other)` returns the value if it's present, otherwise invokes `other` and returns its result.

Let's see how they work. First, we use the `get` method to obtain the present value:

```
public static void main(String[] args) {
    Optional<String> optName = Optional.ofNullable("John");
    Optional<String> optNull = Optional.ofNullable(null);

    System.out.println(optName.get());           // "John"
    System.out.println(optName.orElseThrow());   // "John"
    System.out.println(optName.orElse("Default name")); // "John"

    System.out.println(optNull.get());           // throws NoSuchEl
    System.out.println(optNull.orElseThrow());   // throws NoSuchEl
    System.out.println(optNull.orElse("Default name"));   // "Defau
}
```

Conditional actions

There are also convenient methods that take functions as arguments and perform some actions on values wrapped inside Optional:

- `ifPresent(Consumer<? super T> action)` performs the given action with the value, otherwise does nothing;
- `ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)` performs the given action with the value, otherwise performs another action.

The `ifPresent` method allows us to run some code on the value if the Optional is not empty. The method takes a **consumer function** that can process the value.

```
public static void main(String[] args) {
    Optional<String> optName = Optional.ofNullable("John");
    Optional<String> optNull = Optional.ofNullable(null);

    optName.ifPresent((name) -> System.out.println(name.length()));
    optNull.ifPresent((name) -> System.out.println(name.length()));
}
```

The "classic" equivalent of these two code snippets looks like the following. This code is more error-prone because it is possible to forget to perform the `null` check explicitly and then get the **NPE**.

```
public static void printName(String name) {
    if (name != null) {
        System.out.println(name.length());
    }
}
```

The `ifPresentOrElse` method is a safer alternative to the whole `if-else` statement. It executes one of two functions depending on whether the value is present in the `Optional`.

```
public static void main(String[] args) {
    Optional<String> optName = Optional.ofNullable("John");
    Optional<String> optNull = Optional.ofNullable(null);

    // prints 4
    optName.ifPresentOrElse(
            (name) -> System.out.println(name.length()),
            () -> System.out.println(0)
    );

    // prints 0
    optNull.ifPresentOrElse(
            (name) -> System.out.println(name.length()),
            () -> System.out.println(0)
    );
}
```

References

- https://www.baeldung.com/java-functional-programming
- https://www.baeldung.com/java-8-functional-interfaces
- https://www.baeldung.com/java-8-lambda-expressions-tips