Function Structure

In this section we discuss issues related to the form of functions: Their number of arguments etc.

Function arguments

The fewer arguments a function has, the simpler it is to know what will happen when we call the function. For example, consider a comparison function:

```
isGreater(a, b);
```

Is this function supposed to return `true` if `a` is greater than `b`, or if `b` is greater than `a`? Even though we might guess that it is the former, it is not possible to know for sure without reading the code. On the other hand, if the method was structured instead as follows, there is no doubt:

```
a.isGreaterThan(b);
```

Another way to resolve the case of multiple arguments is to perform an "Extract Parameter Object" refactoring; if those parameters tend to often be used together, maybe their common usage can be codified in this new class.

Here is another example. Imagine a class for an HVAC system managing a heating unit, a cooling unit, and a fan. What would the following function do with such a unit?

```
hvac.set(true, false, true);
```

Instead, the following 3 lines are much more clear, if a bit longer:

```
hvac.turnHeaterOn();
hvac.turnCoolerOff();
hvac.turnFanOff();
```

We will see an alternative that does it in one line in a more readable form in a bit.

Another example. Consider the following line:

```
person.setScoreIfHigher(42, 35, true);
```

This method is supposed to set a person's score to a specific value if it was exceeding another specific value. But which value is which? And oh by the way, that `true` means that we also set it to this value if the score was missing. How are we supposed to know that however?

Imagine if we instead had a "filter" class, and we could do something like this:

```
person.setScore(35, Filter.ifValueIsHigherThan(42), true);
```

Still not perfect, but it's start. We could also use a boolean variable name to help out:

```
boolean shouldSetNull = true;
person.setScore(35, Filter.ifValueIsHigherThan(42), shouldSetNull);
```

Boolean arguments

The two last examples showcase a problem with boolean arguments. When those arguments are present, it is very unclear what each value is meant to represent. Furthermore, it is almost certain that the function will need to do at least two different things in order to respond to the two possibilities, violating one of the function rules.

There are typically two effective ways of dealing with boolean arguments:

1. Create two separate functions. In the above example, we could have one function named `setScoreIncludingNullCase` and one function named `setScoreExcludingNullCase`.
2. Use a custom enum type to represent the boolean cases. In the example above we could do:

   ```
   person.setScore(35, Filter.ifValueIsHigherThan(42), NullCase.SET);
   person.setScore(35, Filter.ifValueIsHigherThan(42), NullCase.IGNORE);

   // Definition of the enum:
   enum NullCase {
       IGNORE, SET;
   }
   ```

   In the HVAC system we can use three enums: `HeaterSetting.HEATER_ON/HEATER_OFF`, `CoolerSetting.COOLER_ON/COOLER_OFF` etc. Then the call would look like this: `java hvac.set(HEATER_ON, COOLER_OFF, FAN_ON);` Furthermore, Java's type-checking will make sure that we enter those values in the correct order, as we cannot use `HEATER_ON` where a `CoolerSetting` constant is expected.

Command-Query separation

Methods broadly fall into two categories:

1. Methods that have what we call **side-effects**. For example, they could be setting an object's state, printing something to the console, opening up a file etc. We call these **commands**. These are methods that if called repeatedly may cause problems.
2. Methods that return a value but otherwise have no side-effects. These could simply be returning the value of a field, or perform some computation. We call these methods **queries**. In theory one should be able to call a query multiple times and not expect anything to change.

The **command-query separation** principle says that every method should do play one of those two roles, but not both:

- A method that returns a value should not have side-effects.
- A method that has side-effects should not return a value.

Group activity: Think back on the methods we implemented for the Circle class (the names might not match exactly what we used then). Which of these are queries, and which are commands? Do any of them act as both queries and commands?

- getCenter()
- getRadius()
- getArea()
- getPerimeter()
- contains(Point point)
- shiftTo(Point newCenter)
- multiplyRadiusBy(int value)

Group activity: The following methods are part of the ArrayList<E> class. E here is a *class variable* which stands for the class of the elements to be stored in the list. Which of these methods are pure queries, which are pure commands, and which violate the command-query separation rule?

- boolean add(E e) Appends the specified element to the end of this list. Returns true.
- void add(int index, E element) Inserts the specified element at the specified position in this list.
- boolean addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator. Returns true if the list changed as a result of the operation, and false otherwise.
- void clear() Removes all of the elements from this list.
- Object clone() Returns a shallow copy of this ArrayList instance.
- boolean contains(Object o) Returns true if this list contains the specified element.
- void ensureCapacity(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
- void forEach(Consumer<? super E> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
- E get(int index) Returns the element at the specified position in this list.
- int indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
- boolean isEmpty() Returns true if this list contains no elements.

- Iterator<E> iterator() Returns an iterator over the elements in this list in proper sequence.
- E remove(int index) Removes the element at the specified position in this list. Returns the removed element.
- boolean remove(Object o) Removes the first occurrence of the specified element from this list, if it is present. Returns true if the list contained the specified element.
- boolean removeAll(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection.
- protected void removeRange(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
- void replaceAll(UnaryOperator<E> operator) Replaces each element of this list with the result of applying the operator to that element.
- E set(int index, E element) Replaces the element at the specified position in this list with the specified element. Returns the element previously at the specified position.
- int size() Returns the number of elements in this list.
- void sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator.
- Spliterator<E> spliterator() Creates a late-binding and fail-fast Spliterator over the elements in this list.
- List<E> subList(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
- Object[] toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
- void trimToSize() Trims the capacity of this ArrayList instance to be the list's current size.