Exercise 2:
Feature Composition

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering Design & Construction
WS 2016/17 - Dr. Michael Eichberg, M.Sc. Matthias Eichholz

The second task of this exercise will be graded. Please submit your solution until the **22nd of November, 23:59** via email to eichholz@st.informatik.tu-darmstadt.de. Make sure you zip your complete sbt project and make sure that it works out of the box by running `sbt run` and `sbt test`.

Although the other exercises are not graded, it is highly recommended to also do them on your own. Just looking at a solution is much easier in comparison to actually coming up with it. Support can be found in the forum:
`https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewforum.php?f=234`

Task 1 Functional Sets

In the last exercise, we used binary search trees to represent sets. Another way to represent sets is via a single function `Set => Boolean` that defines whether a given element is in the set. In Scala, we can define a type alias, so that we can use the alias `Set` instead of the longer function type:

```
type Set = Int => Boolean
```

We can write a `contains` function as follows:

```
def contains(s: Set, elem: Int): Boolean = s(elem)
```

Since a set of type `Set` is just a function, `contains` simply applies that function to determine whether the element is in the set or not.

Task 1.1 Implementation

Implement a constructor, with the following signature, that creates a set containing a single element:

```
def Set(elem: Int): Set
```

Implement set union, intersection and difference with functions of the following signatures:

```
def union(s: Set, t: Set): Set
def intersect(s: Set, t: Set): Set
def diff(s: Set, t: Set): Set
```

Implement a filter function for our functional set representation similar to the filter method above:

```
def filter(s: Set, p: Int => Boolean): Set
```

Implement a function that maps every element in a given set S using a function f, so that the result is the set $\{f(x) \mid x \in S\}$:

```
def map(s: Set, f: Int => Int): Set
```

You can assume that for all elements $x \in S$: $-1000 \leq x \leq 1000$.

Task 2 A Testing Framework for Performance Tests (Graded)

The goal of this exercise is to write a small testing framework similar to ScalaMeter[1] that can be used to run performance tests. We want to have an easily extensible performance measurement framework which provides user a small domain specific language to specify the measurement tasks at hand. The primary mechanism offered by the framework to enable user to customize the behavior are “stackable" traits.

For now, the framework should support two different kinds of performance evaluations, but of course it should be possible to later extend the framework with further performance evaluations. First, it should be possible to measure the time it takes to execute a program. For example, measuring the time it takes to allocate an array with 50000 random numbers should look like the following.

```
object TimeTestSuite extends TimeEvaluation {
  measure("Allocate␣large␣array␣of␣random␣numbers") {
    val a = new Array[Int](50000)
    val r = new Random(42)
    for (i <- a.indices) a(i) = r.nextInt()
  }
}
```

In a similar way, it should be possible to evaluate the memory that is allocated during the execution of a program. The test specification should look similar to the one for the run time evaluation.

```
object TimeTestSuite extends MemoryEvaluation {
  measure("Allocate␣large␣array␣of␣random␣numbers") {
    val a = new Array[Int](50000)
    val r = new Random(42)
    for (i <- a.indices) a(i) = r.nextInt()
  }
}
```

It should also be possible to measure both time and memory, by mixing both traits together:

```
object TimeAndMemoryTestSuite extends TimeEvaluation with MemoryEvaluation {
  ...
}
```

Disclaimer: Please keep in mind that this task focuses on software design and not on performance testing. In general, performance evaluation (of software systems) is a complex topic and the evaluation methods used here might not be suitable for practical use.

Task 2.1 Performance Evaluations

In the code template, we already provided the trait `PerfSpec` that should be used as base trait for all performance evaluations. Provide a trait `TimeEvaluation` and `MemoryEvaluation` for the two types of performance evaluations respectively which can then be mixed into the test suite depending on the kind of evaluation that should be performed. The run time evaluation should measure the execution time in nano seconds while the memory evaluation should measure the amount of memory allocated on the Heap. For the memory evaluation make sure you trigger the garbage collector before starting to measure, to prevent inconsistent results.

Hint: You can use the interface `java.lang.management.MemoryMXBean` to get access to the memory system of the Java virtual machine. To obtain the current time in nano seconds you can use `System.nanoTime()`.

Task 2.2 Serialization of Evaluation Results

The framework should provide the developer some flexibility with regard to the serialization mechanism to use. Your task is to provide three different serializers, an `InMemorySerializer`, which stores the serialized data in the main memory, a `FileSerializer` which stores the serialized data in a file, and a `JSONSerializer` which stores the data in a JSON file. For JSON, use the json4s library (`https://github.com/json4s/json4s`; **Hint:** make sure to edit the sbt build script to include the library). Use the provided `Serializer` trait as a basis for your implementations. The `serialize` method uses the `name` parameter as a key and stores a the provided value, the current date and the unit of the value ("ns" for time evaluation, "b" for memory evaluation). The `deserialize` method retrieves all date-value-unit tuples stored for the given key.

[1] `https://scalameter.github.io/`

```
  trait Serializer {
    def serialize(name: String, value: Long, unit: String) = {}
    def deserialize(name: String): List[(Date, Long, String)]
  }
```

The `FileSerializer` is supposed to write to a different file for each test. Therefore implement the following method:

```
def fileForName(name: String): java.nio.file.Path
```

The `JSONSerializer` is supposed to write all results to a single JSON file (define a field `file` inside JSONSerializer). The JSON file could structurally look like the following:

```
{
  "MyTestJSONSerializer":{
    "test1":[{
      "date":1477988775131,
      "value":42,
      "unit":"b"
    },
    {
      "date":1477988775136,
      "value":44,
      "unit":"b"
    }]
  }
}
```

where `MyTestJSONSerializer` is the name of the class and `test1` is the name of the test. Anyhow, you can choose any structure you like, just be sure that test names can be equal in different classes and should still be considered unique. This obviously also holds for `FileSerializer`.

Task 2.3 Comparison of Evaluation Results

You also want to run the same test multiple times and compare how the results have changed in comparison to previous runs. Provide another trait `PastComparison` that can be mixed into a performance evaluation suite to provide this functionality. For this to work, you need a way to memorize previous evaluation results by serializing them. Make sure that your trait `PastComparison` relies on the previously implemented `Serializer` trait, but is **not** fixed to a single implementation.

Provide a method **`def`** `lastExecutionWorse(name: String): Boolean` that calculates if the current measurement is more than `comparisonThreshold` % worse than the worst previous result (higher or lower). Make sure that you group the history by unit, so you don't accidentally compare two history results of different units. So this method should return true if a last history value for any unit is worse than the previous history.
The test should be considered as failed (throw an exception) if the current measurement is `comparisonThreshold` % different compared to the maximum or minimum of previously measured results. The `comparisonThreshold` should not be fixed in the implementation of `PastComparison`, but only determined in the actual test like shown in this example:

```
object MemoryTestSuite extends MemoryEvaluation with InMemorySerializer with PastComparison {

  def comparisonThreshold = 0.05

  measure("Allocate␣large␣array") {
    val a = new Array[Int](50000)
    val r = new Random(42)
    for (i <- a.indices) a(i) = r.nextInt()
  }

  // This should fail, as we allocate more than 5 percent more memory...
  // Usually you woudn't use the framework this way, but reusing the same test name simulates
  // two test runs...
  measure("Allocate␣large␣array") {
    val arr = new Array[Int](500000)
    val r = new Random(42)
    for (i <- arr.indices) arr(i) = r.nextInt()
  }
}
```

Task 2.4 Trait order

Please describe shortly whether the order you combine `TimeEvaluation` and `MemoryEvaluation` matters when combining them for a test. If it does, state which order you think is better. If it doesn't, explain why.
Hint: Answer this question at the corresponding *`//TODO:`* section in `PerfSpec.scala`.

Task 2.5 Runnable Tests

Write at least 4 different tests that use a combination of the previously implemented testing features and make sure that those can be run using `sbt run`.

Task 2.6 Testing your implementation

With the project you received a number of tests that can be run with `sbt test`. Test your implementation with those tests and write more to test all important aspects of your implementation.
Hint: Your project has to compile, run, and successully run the tests, meaning both `sbt run` and `sbt test` have to complete without errors!

Task 3 Performance Testing Framework in Java 8

Think about how you would implement the same testing framework as above in Java 8. Is it possible to implement it in a similar way in Java 8 using interfaces with default methods? If not, how would the design change?