

Object-Oriented Programming 2

Rolf Haenni & Annett Laube

Exercises 2

1. Set Implementation

Implement a generic class `ArrayListSet<T>`, which implements the `Set<T>` interface. Internally, the elements of the set are stored in an instance of `ArrayList<T>`. Define all necessary methods, including `equals(Object obj)` and `toString()`. Define the following two constructors:

- `ArrayListSet()`: constructs an empty set
- `ArrayListSet(Collection<? extends T> c)`: constructs a set initialized with all elements of a given collection `c`

Retrieve the JUnit test class `ArrayListSetTest` from the GitHub repository to test your class.

2. Memoization

If a deterministic method which involves expensive computations is called multiple times with the same parameters, it may be useful for the overall performance to put the return values of each method call into a cache. For this, the method's surrounding class is extended with one or multiple static final variables, in which the return values are stored. If the method is called for the second time, the return values is extracted from the cache. This technique is called *memoization*.

1. The method `isProbablePrime(int certainty)` from the `BigInteger` class checks if a given `BigInteger` value is prime. If the value is not prime, the method returns `false`, but if the value is prime, it returns `true` with probability $1 - \frac{1}{4^{\text{certainty}}}$. Let's assume that `certainty = 80` is fixed for all possible applications. This implies

that the method's internal procedure loops 80 times. For large values (e.g., several hundred digits), this computation is relatively expensive.

Write a static method `boolean isPrime(BigInteger p)`, which checks if `p` is prime by calling `isProbablePrime(80)`, but which stores the result of the computation in one or multiple caches of type `Set<BigInteger>` to avoid repeating the computation in future calls of the method.

2. The following recursive method computes the *Fibonacci number* for a given input value $x \in \mathbb{N}$. For large input values, this solution is extremely inefficient (run some tests for $x = 40, 42, \dots$). The problem is the large number of recursive calls induced by the initial method call.

```
1 public static int fibonacci(int x) {  
2     if (x == 0) {  
3         return 0;  
4     }  
5     if (x == 1) {  
6         return 1;  
7     }  
8     return fibonacci(x-1) + fibonacci(x-2);  
9 }
```

Write a static method `int fibonacci(int x)`, which stores the results of each call in a cache of type `ArrayList<Integer>`.

3. Multiton-Pattern