Object-Oriented Programming

Exercise Session 4

Magnus Madsen

magnus@cs.aau.dk

Problem 1. Explain, in your own words, the concept of *polymorphism*.

Problem 2. Relate, in your own words, the concepts of *inheritance* (subtype polymorphism) to the concept of *generics* (parametric polymorphism).

(*Hint: When would you use inheritance over generics and vice versa?*)

Problem 3. Write a generic method `select` with a type parameter `T` that takes two arguments x and y of type `T` and a boolean b, and returns x if b is true and y otherwise.

Problem 4. Write a generic method `extract` with a type parameter `T` that takes an argument of type `T[]` and an argument of type `T`. The method returns the first element of the array, if it is non-empty. Otherwise, it returns the second argument.

Problem 5. Write a generic method `copy` with a type parameter `T` that takes two arguments of type `T[]` and copies the content of one array to the other array. Throw an exception if the arrays have unequal lengths.

Problem 6. Write a generic method `shuffle` with a type parameter `T` that takes an argument of type `T[]`. Permute the array using the following algorithm: Repeatedly generate two random numbers i and j, where i and j must be a valid array indices and then swap the entry i with the entry j. Perform this operation n times where n is the length of the array.

(*Hint: The `Random` class has a `nextInt` method that you may find useful.*)

Problem 7. Write a class `Box` with a single type parameter `T`. A box contains *exactly* one element of type `T` (i.e. it should have a field of type `T`). Add an appropriate constructor. Add getter and setter methods. Ensure that the box can never contain the `null` value. Throw exceptions if a user attempts to put `null` in the box.

Problem 8. Write a class `Pair` with two type parameters `A` and `B` to represent a pair of values (i.e. the class should have two fields of type `A` and `B`). Add an appropriate constructor and getter methods. Do *not* add any setters, as the class should be *immutable*.

Problem 9. Add a method `swap` to the `Pair` class. The `swap` method should return a new pair where the first component becomes the second component and vice versa. For example, for the pair (`true`, 42) the method should return (42, `true`).

(*Hint: You will have to swap the type parameters in the return type.*)

Problem 10. Add methods `setFst` and `setSnd` to the `Pair` class. Each method should take a type parameter `C` and return a new pair where the appropriate component has been updated. For example, calling `setFst` with the integer 42 on the pair (`true`, `"Hello World"`) should return (42, `"Hello World"`).

Problem 11. Write a class `Dict` that takes two type parameters `K` and `V`. The class should represent a dictionary, i.e. a mapping from items of type `K` to items of type `V`. Internally, the dictionary should maintain a single array of pairs of type `Pair<K, V>`. The dictionary should support the operations: `get(K key)` and `put(K key, V value)`. The `get` method takes a key argument, searches through the array for an element with that key, and returns its value. If the key is not present, it should throw an exception. The `put` method updates the array with a new pair for the mapping from `key` to `value`.

(*Hint: If a pair with the key is already in the map it must be updated or removed.*)

Problem 12. Explain, in your own words, the concepts of a *functional interface* and a *lambda expression*.

Problem 13. Explain, in your own words, the concept of a *higher-order function*.

Problem 14. Write a functional interface `Joinable` with a method `join` that takes two string arguments and returns a string. Write a method `reduce3` that takes three strings and a `Joinable` and joins the strings from left to right. For example,

- `reduce3("a", "b", "c", (x, y) -> x + y) = "abc"`
- `reduce3("a", "b", "c", (x, y) -> x + "." + y) = "a.b.c"`
- `reduce3("a", "b", "c", (x, y) -> x) = "a"`

Problem 15. Write a method `joinAll` that takes a non-empty array of strings and a `Joinable` and joins all strings in the array. For example,

- `joinAll(new String[]{"a", "b", "c"}, (x, y) -> x + y) = "abc"`
- `joinAll(new String[]{"a", "b", "c"}, (x, y) -> x + "." + y) = "a.b.c"`
- `joinAll(new String[]{"a", "b", "c"}, (x, y) -> x) = "a"`

Problem 16. Write a generic method `exists(Predicate<T> f, T[] a)` that takes a type parameter `T` and two arguments: a unary lambda expression f and an array a of type `T`. The method should return `true` if the array contains an element for which the predicate evaluates to true. Otherwise, it should return false.

Problem 17. Write a generic method `twice(UnaryOperator<T> f, T v)` that takes a type parameter `T` and two arguments: a unary lambda expression f and a value v of type `T`. The method should return the result of applying the lambda function twice to the argument. For example, `twice(x -> x * 2, 1) = 4`.

Weekly Hand-in

Write a class `Queue` to represent a *first-in first-out* queue. The queue should take a type parameter `T` and store elements of type `T` in an array. A queue is like a check-out line in the super-market. You enter the queue, and eventually you are the one who has waited the longest, and then you exit the queue.

The `Queue` should support the following operations:

- `void enqueue(T t)` adds the element t to the back of the queue.
- `T dequeue()` removes and returns the first element in the queue. Throws an exception if the queue is empty.
- `T drain(int n)` removes the first n elements in the queue. Returns the last element removed. Does *not* throw an exception if the queue contains less than n elements, but drains the queue until it is empty and returns the element removed last.
- `void drainWhile(Predicate<T> f)` repeatedly removes elements from the queue as long as the predicate f is `true`. Stops when f returns `false`. For example, `drainWhile(x -> true)` should completely drain the queue.

Hint: You have two options for how to implement the queue:

- **Easier (fixed array):** Use an array of type `T` with fixed size. The size could be given as an argument to the constructor of `Queue`. Maintain two integer fields: `current` and `next` which are both indices pointing into the array. `current` should point to the oldest element and `next` should point to the next available index. Maintain these during enqueue and dequeue operations. When `current` goes beyond the size of the array, wrap around, and start from zero. Throw an exception if an enqueue operation would overflow the array.
- **Harder (dynamic array):** As above, but instead the size of the array should double when ever an enqueue operation is about to overflow.

Write unit tests to ensure the correctness of your queue.