Elisa Gonzalez Boix (email:egonzale@vub.ac.be, office:10F731)
Jorge Vallejos (email:jvallejo@vub.ac.be, office:10F724)

1 First Steps in AmbientTalk

The aim of this session is to get you familiar with the sequential part of AmbientTalk. Even though AmbientTalk is a domain-specific language for distributed programming, it remains a full-fledged object-oriented language in its own. This exercise introduces AmbientTalk's basic building blocks (i.e. functions, objects, messages).

1.1 Material

Ambienttalk's tutorial and language reference are available at `http://soft.vub.ac.be/amop/`. The lab session material is available at `http://soft.vub.ac.be/amop/teaching/dmpp`

1.2 Functions, Tables and control flow primitives

Question 1: Write a recursive function for calculating greatest common divisor:
gcd(a,b) = a if a=b
gcd(a,b) = gcd(a-b,b) if a>b
gcd(a,b) = gcd(a,b-a) if b>a
Question 2: Write a function that returns the length of a table (without using table.length of course ;)
Question 3: Write a function that returns the reverse of a given table.
Question 4: Write a `makeSetUnion` function that takes two tables and returns the concatenation of them without duplicates.
Usage example: makeSetUnion([1,2,4,4], [4,5]) returns [1,2,4,5]
Question 5: Write an `average` function that calculates the average of the variable amount of given numbers.
Question 6: Write a function that takes a table as argument and returns a table without the first element. Try to do it with 2 statements :)

1.3 Object-oriented programming

From this section on, you will need to remove the TODO prefix in the unit test method corresponding to a question in order to check that the requested functionality works. Sometimes, you will also need to complete the unit test implementation.

Question 7: Implement a prototypical `Counter` object with methods to increment, decrement and get its current value of the counter.

a) Create a new instance of a Counter and increment it.
Note: Check the current value of the prototypical Counter object after incrementing the new instance. Is it 0?

b) Extend the `Counter` object to increment or decrement the value of counter by a given value. Add the necessary code in the `testCounter` method to test this functionality.
Hint: use optional arguments.

c) Extend the `Counter` object with a method `doUpToCounter:` which takes a block and executes it as many times as the counters current value. Add the necessary code in the `testCounter` method to test this functionality.
Hint: take a look at the methods implemented by native number objects.

d) Extend the `Counter` object with a method `doWithCurrentValue:` which takes a block as argument and executes the block with the current value of the counter. Add the necessary code in the `testCounter` method to test this functionality.

e) Make two extensions to the Counter object: `ForwardCounter` and `BackwardCounter` objects and define the following methods: `setTo`, `isZero` and `step`. Add the necessary code in the `testCounter` method to test this functionality.

Question 8: Implement a prototypical `Polygon` that has one method called `perimeter` which takes an arbitrary number of arguments representing the n-sides of a simple polygon, and returns its perimeter (i.e. the sum of the sides).

a) Extend the `Polygon` object to create a `Rectangle` object. The `Rectangle` object has two methods `area` and `perimeter` to calculate its area and perimeter, respectively.

b) Extend the `Rectangle` object to create a new `Square` object.

Question 9: Use the observer pattern to create a *future*. A future is an object that acts as a placeholder for a result of an operation that is initially unknown, usually because its value has not been computed yet. Once the return value is computed, it "replaces" the future object, and the future is said to be *resolved* with the value. Computations that require access to the actual resolved value of the future before it is resolved can be "suspended" in the form of an observer registered to the future. To this end, programmers can use the `when:becomes:` function as shown in the code snippet below that illustrates the use of a future object to display the current location of a user:

```
// Return a future object representing a location
def futureLocation := makeFuture();
// Install an observer on the future's resolved value
when: futureLocation becomes: { |location|
   system.println("Your current location is " + location);
};
// Resolve the future
futureLocation.resolve("VUB");
// This should return "Your current locations is VUB"
```

Implement the `makeFuture` and `when:becomes:` functions which allow for the following behaviour:

- The `makeFuture` function creates a future object that has a method named `resolve`, which can be used by a programmer to resolve the future, and a `subscribe` method to register an observer object with a future.
- The `when:becomes:` function takes as parameter a future, and a closure which will be applied to the future's resolved value when it is computed.

As you will see later in the theory class, futures (also known as promises) are a frequently recurring abstraction in concurrent languages. They are used to reconcile asynchronous message passing with return values.

1.4 Modular Programming

Question 10: Implement a stack data type using the vector module found in `/at/collections/vector.at`.

a) You will have to implement a constructor to create an empty stack and methods to manipulate it (`push`, `pop` and `top`).

b) Raise a `StackUnderflow` and `StackOverflow` exception when trying to pop an empty stack or push into a full stack, respectively.

c) Complete the implementation of the unit test called `testStack` that check that the `StackUnderflow` exception is raised correctly. *Hint:* take a look at the file `/at/lang/exceptions.at` for creating exceptions.

Question 11: Implement a `Comparable` trait with the following methods that allow objects to be compared: `equalTo`, `smallerThan`, `greaterThan`, `smallerOrEqualTo` and `greaterOrEqualTo`.

a) You will also have to implement a `Date` object which uses this trait to compare dates as follows: `oneDate.smallerThan(anotherDate)`

b) Re-implement the `Comparable` object now using the AmbientTalk's traits library found in `/at/lang/traits.at`.

Hint: Check the AmbientTalk tutorial, section Language Extensions in the Appendix for information about the library.

1.5 Symbiosis with Java

Question 12: Implement an small "AmbientTalk widget" which converts temperature from Fahrenheit to Celsius ($C = (F - 32) * 5/9$). The widget uses a simple Java GUI which accepts Fahrenheit input from the user, and has a button to display the conversion results. Hence, the temperature widget consists of an AmbientTalk object in charge of the logic to convert the temperature measures, and a Java object which implements the GUI of the widget.

a) Create the GUI of the widget as a Java instance of a `TemperatureWidget` class, which takes as parameter an AmbientTalk object (called `Conversor`) to calculate the conversion.
Hint: Check the `ATConversorInterface` to find out the method that the `TemperatureWidget` Java object expects a `Conversor` object to implement to calculate the temperature in Celsius.

b) Modify your program in such a way that the result of `calculateCelsius` is displayed on the GUI only when the `Conversor` object explicitly calls the `setCelsiusTemperature` method on the `TemperatureWidget` object.