National University of Singapore
School of Computing
CS1101S: Programming Methodology (JavaScript)
Semester I, 2012/2013

Discussion Group Exercises 6

The OOP Adventure Game

The questions in this section will explore two ideas: the simulation of a world in which objects are characterized by collections of state variables, and the use of *object-oriented programming* as a technique for modularizing worlds in which objects interact. These ideas are presented in the context of a simple simulation game like the ones available on many computers.

The basic idea of simulation games is that the user plays a character in an imaginary world inhabited by other characters. The user plays the game by issuing commands to the computer that have the effect of moving the character about and performing acts in the imaginary world, such as picking up objects. The computer simulates the legal moves and rejects illegal ones. For example, it is illegal to move between places that are not connected (unless you have special powers). If a move is legal, the computer updates its model of the world and allows the next move to be considered.

Our game takes place in a strange, imaginary world called NUS, with imaginary places such as the Central Library, COM1, AS6 and the Business Canteen. In order to get going, we need to establish the structure of this imaginary world: the objects that exist and the ways in which they relate to each other.

Initially, there are three constructors for objects:

```
function Thing(name, birthplace) {...}
function Place(name) {...}
function Person(name, birthplace, threshold) {...}
```

In addition, there are constructors for people and things, and functions that install them in the simulated world. The reason that we need to be able to create people and things separately from installing them will be discussed in one of the exercises later. For now, we note the existence of the functions

```
function MakeAndInstallThing(name, birthplace) {...}
function MakeAndInstallPerson(name, birthplace, threshold) {...}
```

Each time we make a place, or make and install a person or a thing, we give it a name. People and things are also created at some initial place. In addition, a person has a threshold factor that determines how often the person moves. For example, the function `MakeAndInstallPerson` may be used to create the two imaginary characters, `beng` and `bing`, and put them in their places, as it were.

```
var beng_office = new Place("beng office");
var bing_office = new Place("beng office");

var beng = MakeAndInstallPerson("beng", beng_office, 3);
var bing = MakeAndInstallPerson("bing", bing_office, 3);
```

All objects in the system are implemented as JediScript objects.

Once you load the system, you will be able to control `beng` and `bing` using appropriate methods. As you enter each command, the computer reports what happens and where it is happening. For instance, imagine we had interconnected a few places so that the following scenario is feasible:

```
beng.look_around();
// At beng-office : beng says -- I see beng-card (beng-card)

beng.go("north");
// beng moves from beng-office to com1-classrooms

beng.go("north");
// beng moves from com1-classrooms to com1-open-area

beng.place().exits();
// returns list("west", "south", "east", "north")

beng.go("north");
// beng moves from com1-open-area to lt15
// At lt15 : beng says -- Hi proffy

bing.go("down");
// bing moves from bing-office to lt15
// At lt15 : bing says -- Hi beng proffy
```

In principle, you could run the system by issuing specific commands to each of the creatures in the world, but this defeats the intent of the game since that would give you explicit control over all the characters. Instead, we will structure our system so that any character can be manipulated automatically in some fashion by the computer. We do this by creating a list of all the characters to be moved by the computer and by simulating the passage of time by a special function, `clock`, that calls a `move` method on each creature in the list.

Calling a `move` method on an object does not automatically imply that it will perform an action. Rather, like all of us, a creature hangs about idly until he or she (or it) gets bored enough to do something. To account for this, the third argument to the constructor `Person` specifies the average number of clock intervals that the person will wait before doing something (the threshold factor).

Before we trigger the clock to simulate a game, let's explore the properties of our world a bit more.

First, let's create a `jediscript_week10_description` and place it in `lt15` (where `beng` and `bing` now are).

```
var jediscript_week10_description
    = MakeAndInstallThing("JediScript Week 10 Description", lt15);
```

Next, we'll have `beng` look around. He sees the manual, `bing` and `proffy`. The manual looks useful, so we have `beng` take it and leave.

```
beng.take(jediscript_week10_description);
// At lt15 : beng says -- I take JediScript Week 10 Description

beng.go("north");
// beng moves from lt15 to forum
```

lstinlinebing had also noticed the manual; he follows beng and snatches the manual away. Angrily, beng sulks off to the Central Library:

```
bing.go("north");
// bing moves from lt15 to forum
// At forum : bing says -- Hi beng

bing.look_around();
// At forum : bing says -- I see scheme-manual beng (scheme-manual beng)

bing.take(jediscript_week10_description);
// At forum : beng says -- I lose JediScript Week 10 Description
// At forum : beng says -- Yaaaah! I am upset!
// At forum : bing says -- I take JediScript Week 10 Description

beng.go("up");
// beng moves from forum to central-library
```

Unfortunately for bing, the dungeon LT15 is inhabited by a troll named proffy. A troll is a kind of person; it can move around, take things, and so on. When a troll gets a move call from the clock, it acts just like an ordinary person—unless someone else is in the room. When proffy decides to act, it's game over for bing:

```
bing.go("south");
// bing moves from forum to lt15
// At lt15 : bing says -- Hi proffy \#t

proffy.move(); // proffy decides not to act (yet!)
```

After a few more moves, proffy acts again:

```
proffy.move();
// At lt15 : proffy says -- Growl.... I'm going to eat you, bing
// At lt15 : bing says -- I lose JediScript Week 10 Description
// At lt15 : bing says --
//                   Dulce et decorum est
//                   pro computatore mori!
// bing moves from lt15 to heaven
// At lt15 : proffy says -- Chomp chomp. bing tastes yummy!
```

Implementation The source code for this problem set is found in the file game.zip. The provided code contains a basic object system, functions to create people, places, things and trolls, together with various other useful functions and the skeleton of a simple game. You will be asked to extend the game by writing appropriate functions and extensions to the existing functions.

Problems:

1. (a) Define a function flip (with no parameters) that returns 1 the first time it is called, 0 the second time it is called, 1 the third time, 0 the fourth time, and so on.

(b) Define a class `Flip` that can be used to generate flip objects. That is, we should be able to write

```
var flip = new Flip();
```

Define a constructor `Flip` and a method `flip`. The first time you invoke the method `flip` on a `Flip` object, it returns 1, the second time 0, the third time 1, and so on.

(c) Draw an environment diagram to illustrate the result of evaluating the following sequence of expressions:

```
function Flip() {...}
var flip1 = new Flip();
var flip2 = new Flip();

flip1.flip(); // value: ?
flip2.flip(); // value: ?
```

2. Assume that the following definitions are evaluated, using the constructor function `Flip` from the previous exercise:

```
var flip = new Flip();
var flap1 = flip.flip();
function flap2() {
    return flip.flip();
}
var flap3 = flip;
function flap4() {
    return flip;
}
```

What is the value of each of the following expressions (evaluated in the order shown)?

```
flap1;

flap2;

flap3;

flap4;

flap1();

flap2();

flap3();

flap4();

flap1;

flap3();

flap2();
```

3. Draw a simple inheritance diagram showing all the kinds of objects (classes) defined in the adventure game system, the inheritance relations between them, and the methods defined for each class.

4. Draw a simple map showing all the places created by running `game.js` in the file `game.zip`, and how they interconnect. You will probably find this map useful in dealing with the rest of the problem set.

5. Suppose we evaluate the following expressions:

```
var ice_cream = new Thing("ice cream", arts_canteen);
ice_cream.set_owner(beng);
```

   At some point in the evaluation of the second expression, the expression

```
this.owner = new_owner;
```

   will be evaluated in some environment. Come up with expressions whose evaluation will reveal all the properties of `ice_cream` and verify that its owner is indeed `beng`.

6. Suppose that, in addition to `ice_cream` we defined above, we define

```
var rum_and_raisin = new NamedObject("ice cream");
```

   Are `ice_cream` and `rum_and_raisin` the same object (i.e., are they `===`)? If `beng` wanders to a place where they both are and looks around, what message will be printed?

7. Note how `install` is implemented as a method defined as part of both `mobile_object` and `person`. Notice that the `person` version puts the person on the clock list (this makes them "animated") then invokes the `mobile_object` version on **`this`**, which makes the `birthplace` where **`this`** is being installed aware that **`this`** thinks it is in that place. That is, it makes "**`this`**" and `birthplace` consistent in their belief of where "**`this`**" is. The relevant details of this situation are outlined in the code excerpts below:

```
// to be translated!!!
(define (make-person name birthplace threshold)
  (let ((mobile-obj (make-mobile-object name birthplace))
        \vdots)
    (lambda (message)
      (case message
            \vdots
            ((install)
             (lambda (self)
               (add-to-clock-list self)
               ((get-method mobile-obj 'install) self) ))   ; **
            \vdots))))

(define (make-mobile-object name place)
  (let ((named-obj (make-named-object name)))
    (lambda (message)
      (case message
            \vdots
            ((install)
             (lambda (self)
               (ask place 'add-thing self)))
            \vdots))))
```

Louis Reasoner suggests that it would be simpler if we change the last line of the `make_person` version of the `install` method to read:

```
mobile_obj.install();
```

Alyssa P. Hacker points out that this would be a bug. “If you did that,” she says, “then when you `MakeAndInstallPerson` `bing` and `bing` moves to a new place, he’ll thereafter be in two places at once! The new place will claim that `bing` is there, and `bing`’s place of birth will also claim that `bing` is there.”

What does Alyssa mean? Specifically, what goes wrong? You will likely need to draw an appropriate environment diagram to explain carefully.

Dynamic Programming and Memoization

Problems:

1. Consider the following function:

$$A(m,n) \quad = \quad \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

   Give a memoized version of this function in JavaScript.

   Warning: This function (known as Ackermann’s function) grows extremely quickly. For example, `A(4, 2)` gives $2^{65536}-3$. You may want to try out only very small values.

2. Let’s explore the algorithm a text editor may use to break up long lines to fit text between the margins. To have some measure of how good a layout configuration is, we associate each line with a cost which is the **square** of the number of spaces left over at the end of the line. For example:

```
--------------- line width: 15
A quick brown   cost = (2 * 2) = 4
fox jumps over  cost = (1 * 1) = 1
the lazy dog    cost = (3 * 3) = 9

Another example:

A quick         cost = (8 * 8) = 64
brown fox jumps cost = (0 * 0) = 0
over the lazy   cost = (2 * 2) = 4
dog             cost = (12 * 12) = 144
```

   In this case, the text layout shown in the first example is preferred (lower total cost).

   (a) Write the function `cost(word_list, line_width)` that takes in a list of words (represented as strings) and returns the cost of a line made up of the words from `word_list`. If the given words cannot fit into a line, return `Infinity`.

Note that `Infinity` is a predefined constant in JediScript Week 10. Also note that you can access the length of a string s by writing s`.length`.

```
cost(list("A", "quick", "brown"), 15)        => 4
cost(list("A", "quick", "brown", "fox"), 15) => Infinity (cannot fit)
cost([], 15)                                 => 225
```

(b) Write the function `f(word_list, line_width)` that finds the cost of the optimal solution to the problem of breaking a list of words into lines no longer than `line-width`.

(c) Time how long it takes to run your functions above using the `time` command. Apply memoization or dynamic programming and see how much you can speed up the computation.