Assignment 2: References and Loops

Real Programmers c Randall Munroe (`http://xkcd.com/292`)

Published: September 20th, 2016
Due: October 25th, 2016

Goals

- Understand assignments.
- Learn to program with loops and conditionals.
- Start to work on a more complex application.
- Test your understanding of recursion.

1 REFERENCE ASSIGNMENTS

In this task you can test your understanding of assignment instructions. Consider the following class:

```
class PERSON

create make

feature -- Initialisation
    make (s: STRING)
            -- Set 'name' to 's'
        require
            s_non_empty: s /= Void and then not s.is_empty
        do
            name := s
        ensure
            name_set: name = s
        end

feature -- Access
        name: STRING
            -- Person's name.

        loved_one: PERSON
            -- Person's loved one.

feature -- Basic operations
        set_loved_one (p: PERSON)
            -- Set 'loved_one' to 'p'.
              do
                  loved_one := p
              ensure
                  loved_one_set: loved_one = p
              end
invariant
        has_name: name /= Void and then not name.is_empty
end
```

Below is the code of the feature *tryout*. It contains a number of declarations and creation instructions, and it is defined in a class different from *PERSON*. All features of class *PERSON* as shown above are accessible by feature *tryout*.

```
tryout
        -- Tryout assignments
    local
        i, j: INTEGER
        x, y, z: PERSON
    do
        create x.make("Anna")
        create y.make("Ben")
        create z.make("Chloe")
        x.set_loved_one (y)
        y.set_loved_one (z)
        -- Here the code snippets from below are added
    end
```

To do

In Table 1 you will find a number of subtasks, each containing a code snippet and statements. For each subtask, list the correct statements (there might be more than one).
Assume that the code snippet is inserted at the location indicated in feature *tryout* above. If it produces a compilation error, choose option (a); otherwise decided for each statement whether it is correct or incorrect `after the code snippet has been fully executed.` To make the answers easier to read, we call `Anna` the object whose *name* attribute is set to "Anna", and accordingly `Ben` and `Chloe` for subtasks 6-9.

1	*i* := 7 *i* := *i* + *j*	(a) Compilation Error. (b) *i* has value 10. (c) *i* has value 13. (d) *i* has value 7.
2	*i* := -1 *j* := 5 *i* := 2 *j* := 3	(a) Compilation Error. (b) *i* has value 2 and *j* has value 3. (c) *i* has value 1 and *j* has value 8. (d) *i* has value -1 and *j* has value 5.
3	*i* := -7 *j* := 5 *i* := *j* *j* := *i*	(a) Compilation Error. (b) *j* has value 5 and *i* holds no value anymore. (c) *i* has value 5 and *j* has value -7. (d) both *i* and *j* have value 5. (e) both *i* and *j* have value -7.
4	*i* := 3 *j* := *i* + 7 *i* := 8	(a) Compilation Error. (b) *i* has value 8 and *j* has value 10. (c) *i* has value 8 and *j* has value 11. (d) both *i* and *j* have value 8.

5	*y* := *x* *x* := *y*	(a) Compilation Error. (b) *y* is a **Void** reference and *x* is attached to *Anna*. (c) *x* is attached to *Ben* and *y* is attached to *Anna*. (d) *Ben* is not attached to any local variable of *tryout* any more. (e) *x* and *y* are both attached to *Anna*.
6	*y* := *z* *y.loved_one* := *x.loved_one*	(a) Compilation Error. (b) *y* is a attached to *Chloe*. (c) *Ben*'s *loved_one* is *Ben*. (d) *Chloe*'s *loved_one* is *Ben*.
7	*x* := *z* *y* := *x* *x.set_loved_one (z)*	(a) Compilation Error. (b) *y* is a attached to *Chloe*. (c) *Anna* loves herself. (d) *Chloe* loves herself.
8	*y* := *x.loved_one* *x.set_loved_one (z)* *z* := *y*	(a) Compilation Error. (b) *x* is attached to *Anna* and her *loved_one* is Ben. (c) *y* is attached to *Chloe*. (d) Nobody loves *Ben*. (e) *y* is attached to *Ben*.
9	*z.set_loved_one (x)* *y.set_loved_one (x.loved_one.loved_one)* *x* := *x.loved_one.loved_one*	(a) Compilation Error. (b) *y* is a attached to *Chloe*. (c) *Anna* loves herself. (d) *Chloe* loves herself.
10	*z* := *x.loved_one* *z.set_loved_one (x)* *y* := *y.loved_one.loved_one*	(a) Compilation Error. (b) *Anna* loves herself. (c) *y* is **Void** and *z* is attached to *Ben*. (d) *x* is attached to *Ben*. (e) *y* and *z* are both attached to *Ben*. (f) *Anna* and *Ben* love each other.

Table 1: Subtasks

To hand in

Hand in your answers to the questions above.

2 Next station loops

In this task you will equip public transportation in Traffic with route information displays.

To do

1. Download `https://drive.google.com/open?id=0B1GMHm59JFjqaV9EeklaYTVMQVk`,

unzip itand open `assignment_2.ecf` from within EiffelStudio. Open class *DISPLAY* in the editor.

2. First let us add some public transport to Zurich. In the feature *add_public_transport* write a loop that iterates through all lines in Zurich and adds one transportation unit to each line[1].

   Run the application to check if you can see trams and buses. Try double-clicking on the map to toggle animation on and off and left-clicking on trams and buses to select them.

3. Now let us implement feature *update_transport_display (t: PUBLIC_TRANSPORT)*. The application is programmed to call this feature every time you select a transport, and, if the animation is running and a transport is selected, whenever some short amount of time has passed. The selected transport is passed through the argument *t*.

   The feature has to output route information of the selected transport to the console below the map. In particular, for the next three stops of the transport, it has to show the time it takes to reach the stop and the station name. Then it has to show the same information for the final destination (unless the destination is already listed among the next stops). See an example in Figure 1.

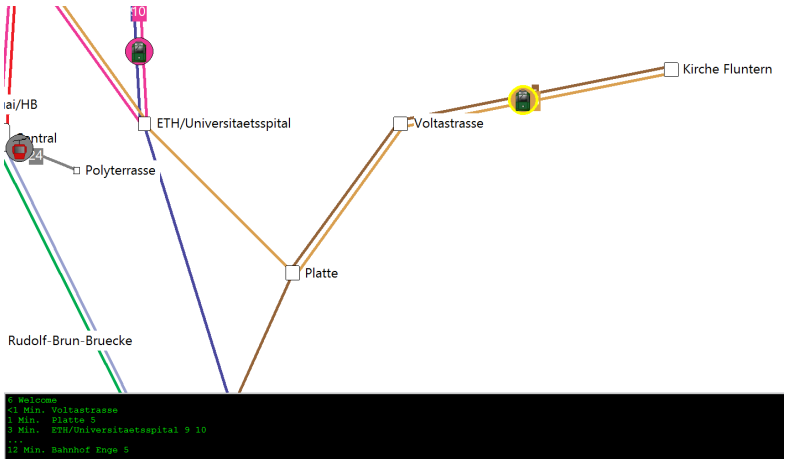

Figure 1: Route information display in Traffic

Note that sometimes there are less than three stops left until the end of the line. In this case the display only shows as many stops as there are left.

To iterate through stations on a line in a given direction (denoted by a terminal station), sometimes it is more convenient to use the feature *next_station* of class *LINE*

[1] You can use **across loop**

example:

```
local
    s: STATION
do
    from
        s := Zurich.station ("Central") -- Start from Central
    until
        s = Void -- Until the end of the line (add your condition here)
    loop
        -- Move to the next station of line 6 in the direction of its west terminal
        s := Zurich.line (6).next station (s, Zurich.line (6).west terminal)
    end
end
```

4. Next to each upcoming station, output all connections available at that station. Note that a line is not considered a connection, if its next stations in both directions are covered by the line we are already on.

HINT:

- To do the output you might want to call features *clear* and *append_line* on the object console. Use "%T" to print a tabulation character. Use the query *out* to get a string representation of any object.
- To convert time from seconds to minutes, use the integer division operator (//): *time_minutes := time_seconds // 60*.
- You might want to define a separate function *stop_info(t: PUBLIC_TRANSPORT; s: STATION): STRING*, which returns the data to be displayed for a given station.

To hand in

Hand in the code of class *DISPLAY* (i.e. the file `display.e`).

3 AN INFECTIOUS TASK

You are the boss of a company concerned about health of your employees (especially in winter – the time of flu epidemics). To take a better decision about the company's health policy, you decide to simulate the spreading of the flu in a program. For this you assume the following model: if a person has a flu, he spreads the infection to only one coworker, who then spreads it to another coworker, and so on.
The following class *PERSON* models coworkers. The class *APPLICATION* creates *PERSON* objects and sets up the coworker structure.

```
class PERSON

create
    make

feature -- Initialisation
    make (a_name: STRING)
            -- Create a person named 'a name'.
        require
        a_name_valid: a_name /= Void and then not a_name.is_empty
    do
        name := a_name
    ensure
        name_set: name = a_name
    end

feature -- Access
    name: STRING

    coworker: PERSON

    has_flu: BOOLEAN

feature -- Element change
    set_coworker (p: PERSON)
            -- Set 'coworker' to 'p'.
    require
        p_exists: p /= Void
        p_different: p /= Current
    do
        coworker := p
    ensure
        coworker_set: coworker = p
    end

set_flu
            -- Set 'has_flu' to True.
    do
        has_flu := True
    ensure
        has_flu: has_flu
    end
```

```
invariant
    name_valid: name /= Void and then not name.is_empty
end
```

```
class APPLICATION
create
    make

feature -- Initialization
    make
            -- Simulate flu epidemic.
    local
        joe, mary, tim, sarah, bill, cara, adam: PERSON
    do
        create joe.make ("Joe")
        create mary.make ("Mary")
        create tim.make ("Tim")
        create sarah.make ("Sarah")
        create bill.make ("Bill")
        create cara.make ("Cara")
        create adam.make ("Adam")
        joe.set_coworker (sarah)
        adam.set_coworker (joe)
        tim.set_coworker (sarah)
        sarah.set_coworker (cara)

         bill.set_coworker (tim)
        cara.set_coworker (mary)
        mary.set_coworker (bill)

        infect (bill)
    end
end
```

Table 2 shows four different implementations of feature *infect*, which is supposed to infect a person p and all people reachable from p through the coworker relation.

To do

1. For each version of infect answer the following questions:
    - Does it do what it is supposed to do?

Version 1

```
infect (p: PERSON)
        -- Infect 'p' and coworkers.
    require
        p_exists: p /= Void

    local
        q: PERSON
    do
        from

          q := p.coworker
          p.set_flu
        until
          q = Void

        loop
          if not q.has_flu then
            q.set_flu
          end
        ---------------------------
          q := q.coworker
        end
    end
```

Version 2

```
infect (p: PERSON)
        -- Infect 'p' and coworkers.
    require
        p_exists: p /= Void

    do
        if p.coworker /= Void and then not
            p.coworker.has_flu then
          infect (p.coworker)
        ---------------------------
          p.coworker.set_flu
        end
        p.set_flu
    end
```

Version 3

```
infect (p: PERSON)
        -- Infect 'p' and coworkers.
    require
        p_exists: p /= Void

    do
        p.set_flu
        if p.coworker /= Void and then
            not p.coworker.has_flu then
        ---------------------------
          infect (p.coworker)
        end
    end
```

Version 4

```
infect (p: PERSON)
        -- Infect 'p' and coworkers.
    require
        p_exists: p /= Void

    do
        if p.coworker /= Void and then not
            p.coworker.has_flu then
          p.coworker.set_flu
        ---------------------------
          infect (p.coworker)
        end
        p.set_flu
    end
```

Table 2: Different versions of feature *infect*

- If yes, how? (One to two sentences.)
- If no, why? (One to two sentences.)

Note: this is a pen-and-paper task; you are not supposed to use EiffelStudio.

2. The class *PERSON* above assumes that each employee can only infect one coworker. This is unfortunately too optimistic. Rewrite the class *PERSON* in such a way that an employee can have (and infect) an arbitrary number of coworkers. Implement a correct recursive feature infect for this new setting. Note: you may use a loop to iterate through the list of coworkers.

To hand in

Hand in your answers to the task 1 and the code (the .e files) of class *PERSON* and feature *infect* for the task 2.

4 LOOP PAINTING

To do

Write a program that does the following:

1. Asks the user to input a positive number.
2. Displays, using consecutive numbers, a rectangle-triangle having as hypotenuse a number of digits equal to the user input. It should also display the triangle's mirror (see Figure 2). Numbers and white spaces should be alternating.
3. Take into consideration that the user might not always input values you expect. Make sure your program does not crash, no matter what the user inputs.

```
1                          1
 2                        2
3  4                    4  3
 5  6                  6  5
7  8  9              9  8  7
 10 11 12          12 11 10
13 14 15 16      16 15 14 13
 17 18 19 20    20 19 18 17
21 22 23 24 25 25 24 23 22 21
 26 27 28 29 30 30 29 28 27 26
```

Figure 2: Example with value 10

To hand in

Hand in your class text (the `.e` files).

5 CONNECT FOUR

The game "Connect Four" has a mxn board where two players take turns dropping tokens from the top of the board. The tokens will fall from the top of each column and come to rest on the bottom of the board or on top of the topmost token in that column. A player wins when four of their tokens line up in a row either horizontally, vertically, or diagonally. There is not winner in case the board is full of tokens. Figure 3 shows a snap of the game.

To do

Your task is to implement in EiffelStudio a version of the Connect Four:

1. Asks the user to input two positive numbers, the size of the board: mxn. The board needs to be big enough for one of the users to win.
2. The game will have two players. Plays alternates strictly between them.
3. The game will ask users (alternatively) for the move. Each user need to input the column where he/she wants to drop the token.
4. The game will be displayed on the command line

One of the hardest part of the implementation is to check whether the move that a player makes is a winning move. You may want to represent the board with a list of lists so it is easier to check the winner of the game.

Figure 3: Connect Four

To hand in

Hand in the code of the classes you defined (i.e. files `.e` and the `.ecf` file).