Assignment 5: Assignments and control structures

ETH Zurich

Handout: Monday, 12 October 2015
Due: Wednesday, 21 October 2015

GOTO © Randall Munroe (http://xkcd.com/292)

Goals

- Understand assignments.
- Learn to program with loops and conditionals.
- Start to work on a more complex application.

1 Assignments

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

```
class PERSON
create make
feature -- Initialization
  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

Below you will find a number of subtasks, each containing a code snippet and statements. For each subtask, list all 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 decide 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.	$i := 7$ $i := i + 3$	(a) Compilation error. (b) i has value 13. (c) i has value 10. (d) i has value 7.
2.	$i := 3$ $2 := i$	(a) Compilation error. (b) i has value 2. (c) i has value 3. (d) i has value 5.
3.	$i := -1$ $j := 5$ $i := 2$ $j := 3$	(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.
4.	$i := -7$ $j := 5$ $i := j$ $j := i$	(a) Compilation error. (b) i has value 5 and j has value -7. (c) both i and j have value -7. (d) both i and j have value 5. (e) j has value 5 and i holds no value any more.
5.	$i := 5$ $j := i + 7$ $i := 8$	(a) Compilation error. (b) i has value 8 and j has value 15. (c) i has value 8 and j has value 12. (d) both i and j have value 8.
6.	$y := x$ $x := y$	(a) Compilation error. (b) x is attached to `Ben` and y to `Anna`. (c) y is a void reference and x is attached to `Anna`. (d) x and y are both attached to `Anna`. (e) `Ben` is not attached to any local variable of *tryout* any more.
7.	$y := z$ $y.loved\_one := x.loved\_one$	(a) Compilation error. (b) y is attached to `Chloe`. (c) `Chloe`'s *loved_one* is `Ben`. (d) `Ben`'s *loved_one* is `Ben`.
8.	$y := x.loved\_one$ $x.set\_loved\_one\ (z)$ $z := y$	(a) Compilation error. (b) y is attached to `Chloe`. (c) y is attached to `Ben`. (d) x is attached to `Anna` and her *loved_one* is `Ben`. (e) Nobody loves `Ben`.
9.	$z := x.loved\_one$ $z.set\_loved\_one\ (x)$ $y := y.loved\_one.loved\_one$	(a) Compilation error. (b) y and z are both attached to `Ben`. (c) y is Void and z is attached to `Ben`. (d) x is attached to `Ben`. (e) `Anna` and `Ben` love each other. (f) `Anna` loves herself.

To hand in

Hand in your answers to the questions above.

2 Reading loops

To review the structure and semantics of loops refer to section 7.5 of Touch of Class.

It happens very often that you want to iterate through all the items of a container, for example all lines in Zurich or all stations of a particular line. To do this you can use the

following scheme:

```
-- Make all lines pink:
across
  Zurich.lines as i -- Create a new cursor 'i'
loop
  -- 'i.item' denotes the current line:
  i.item.set_color ("F010B0")
end
```

If you need to do something more complicated, for example remove some lines from *Zurich* as you go, you will need the full form of the loop. Here is the same loop in its full form:

```
local
    i: like Zurich.lines.new_cursor -- Local variable to store the cursor
do
    -- Make all lines pink:
    from
        -- Create a new cursor at the beginning of the container
        -- and assign it to 'i':
        i := Zurich.lines.new_cursor
    until
        i.after -- 'i' traversed the whole container
    loop
        i.item.set_color ("F010B0")
        i.forth -- Move 'i' one position forward
    end
end
```

The type declaration for i means "i has the same type as the expression *Zurich.lines.new_cursor*". The actual type of i in this case is *V_ITERATOR* [*LINE*] ("an iterator over lines").

To do

Assume that the two code extracts in Listing 1 and Listing 2 intend to iterate through all stations in Zurich, find one named "Central" and move it to the exact center of the city (coordinates [0, 0]).

1. For each version (Listings 1 and 2) decide whether it does what it is supposed to do.
2. If you think it does not, then correct the errors.

Hints

Operator = used on expressions of a reference type compares two references. If they are pointing to the same object the result is true, otherwise the result is false. In contrast, operator ~ compares the content of two objects.

For this exercise you may assume the following:

- There are no compilation problems
- *Zurich* and all its stations are not Void

Listing 1: Version A

```
explore
    -- Move "Central".
  local
    station: STATION
  do
    across
      Zurich.stations as i
    loop
      if i.item.name = "Central" then
        station := i.item
      end
      if station /= Void then
        station.set_position ([0.0, 0.0])
      end
    end
  end
```

Listing 2: Version B

```
explore
    -- Move "Central".
  local
    i: like Zurich.stations.new_cursor
  do
    from
      i := Zurich.stations.new_cursor
    until
      i.item.name ~ "Central" or i.after
    loop
    end
    if not i.after then
      i.item.set_position ([0.0, 0.0])
    end
  end
```

To hand in

This is a pen-and-paper exercise: you do not need to write code in EiffelStudio. Hand in the corrected versions of Listing 1 and Listing 2.

3 Next station: loops

In this task you will equip public transportation in Traffic with route information displays, similar to the ones real trams and buses have in Zurich (Figure 1).

Figure 1: Route information display in a tram

To do

1. Download `http://se.inf.ethz.ch/courses/2015b_fall/eprog/assignments/05/traffic.zip`, unzip it, copy the dir assignment_5 into directory traffic/examples and open `assignment_5.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. You can use the **across** loop example from task 2.

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 2.

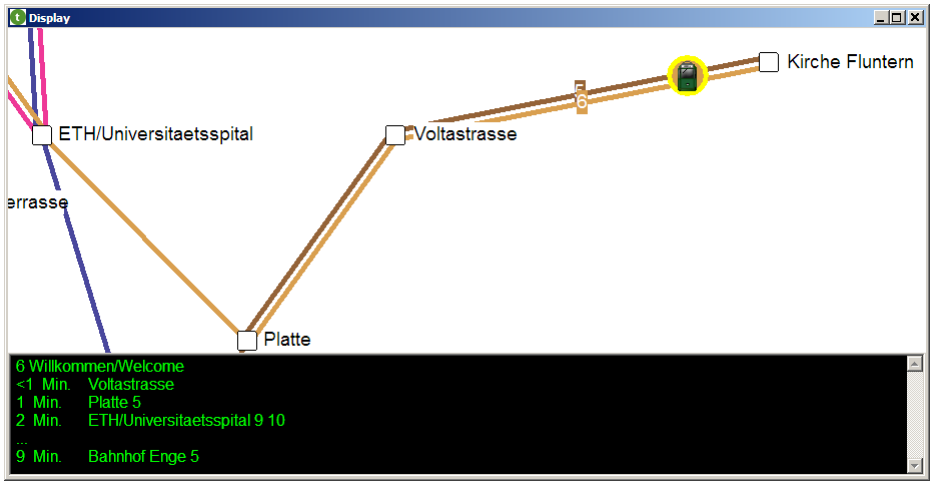

Figure 2: Route information display in Traffic

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*, as an alternative to the schemes described in task 2. For 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. *(Optional)* 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.

Hints

- 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*.

4 Board game: Part 1

In this task you will start a small project from scratch. We will proceed in iterations, starting with a simplified problem and then progressively enriching it. This first part will focus on choosing the right classes.

The idea is to program a prototype of a board-game. It comes with a board, divided into 40 squares, and a pair of dice; the game can accommodate 2 to 6 players. It works as follows:

- All players start from the first square.
- One at the time, players take a turn: roll the dice and advance their respective tokens on the board.
- A round consists of all players taking their turns once.
- The winner is the player that first advances beyond the 40th square.

To do

Make up a list of classes that you would use to model the board game.

Choosing the right abstractions

To suggest classes, you have to ask yourself: *What are the relevant abstractions in the problem domain?* A good source of abstractions is the problem description: names (nouns) in the description often identify important concepts in the problem domain.

Unfortunately, some names in the problem description might not deserve to become classes (like "idea" or "program"); also there might be relevant abstractions that are not expressed as names in the specific text we are looking at.

Whether an entity in the problem domain deserves its own class, depends on its relevant properties and behavior. If you are modeling a door, whose only relevant property is being open or closed, use a boolean variable. If you are programming a game with trapdoors and magic doors that trigger special behavior, then you might need a class for it. Thus the second question you should ask yourself about each candidate abstraction is: *Is there any meaningful data (attributes) and behavior (routines) associated with the abstraction?*

Finally, you should take into account that the problem description (the requirements) is almost never final. When reading the description think about things that are likely to change and new functionality that is likely to be added. Sometimes a concept doesn't have enough associated behavior in the present version of the requirements, but if you think it is likely to gain more in the future, it might still deserve its own class.

To hand in

Hand in your candidate list of class names together with short descriptions of their associated properties and behavior.

5 MOOC: Assignment, control structures

To do

1. Access the main MOOC course web page at `http://se.ethz.ch/mooc/programming`.
2. Listen to lecture number 6 "References, Assignment, and Object Structure" and take the corresponding quiz.
3. Listen to lecture number 8 "Control Structures", and take the corresponding quizzes.
4. Solve the programming exercises.

Your goal is to provide all correct answers to the quizzes and pass all the tests in the programming exercises. You can take the quizzes and attempt the programming exercises as many times as you want. If you succeed, you will be awarded a badge for each correctly solved quiz.