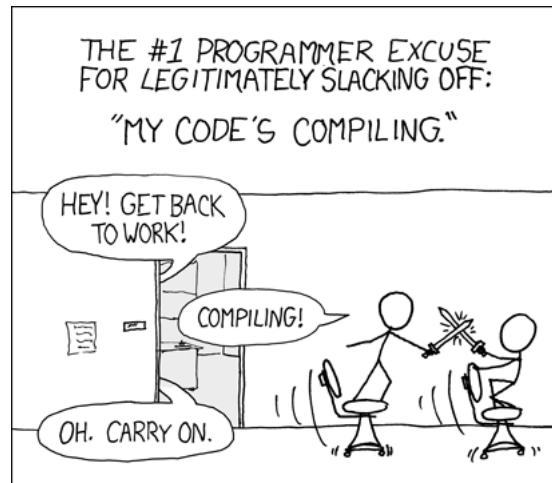


Assignment 4: Object creation

ETH Zurich

Hand-out: 9 October 2009

Due: 20 October 2009



Copyright Randall Munroe <http://xkcd.com>

Goals

- Create objects in Traffic.
- Repeat the difference between strict and semi-strict boolean operators.

1 Creating objects in Traffic

Up to now you have always worked with existing, predefined objects of Paris. In this assignment you will create new objects and add them to Paris. To build new city objects in Traffic such as passengers, trams, places, lines, or roads you can follow a general scheme:

1. **Declare an attribute or local variable of the according type.** This step is needed, so that the software knows that the identifier you will be using in the code is valid and what type of object it can denote. If you forget this step, the Eiffel compiler will produce an error saying that you are using an "Unknown identifier".

At line 22 in Listing 1 an attribute is declared. An attribute declaration is composed of the identifier (name for the new object), a colon, and the identifier type. In our example the identifier is *station* and the type is `TRAFFIC.STATION`. A local variable exists only within a feature body, and so is declared at the beginning of the feature body itself, in

a section called, surprisingly, **local**. This goes before the **do**. The syntax of declaring a local variable is the same as described before for an attribute. If you need an example, look in class *APPLICATION*, feature *prepare*.

2. **Create the object using one of the creation procedures declared in the according class.** If you forget this step, the declared identifier will be **void**, meaning that there is not an object attached to it. Calling features on it will result in a program crash displaying the message "Feature call on Void target". This will not happen if the type of the identifier you have declared is "expanded". In this case the object will be automatically created for you, so you don't have to worry. Example of expanded types are *INTEGER*, *BOOLEAN* and *DOUBLE*. This distinction between reference and expanded types is there mainly for efficiency reasons. In other programming languages like Java the types just mentioned are not even objects (they are called "primitive types"). For a more detailed explanation on why it is important to have types that can be void, see paragraphs 6.2 and 6.3 of Touch of Class.

To find all the available creation procedures, look at the class *TRAFFIC_STATION*. They are listed in the creation clause of the class header (see lines 17 and 18 of Listing 2). In our example a *TRAFFIC_STATION* can be created using either *make*, *make_with_location*, or *make_with_point* as a creation procedure. *make_with_location* requires three arguments: a *STRING* object and two *INTEGER* objects.

Note that for *STRINGS* there is a fast track to object creation: Just put the text you want in the *STRING* object between double quotation marks such as in "My new string object". For *INTEGERS* and *DOUBLES* this fast track is done by writing the number at the appropriate position such as in *16* and in *12.76*. For objects of type *BOOLEAN* you may use *False* or *True*.

There might be cases, where you have to create other objects first because they are needed as arguments to the creation feature. The creation procedure *make_with_point*, for example, takes a *TRAFFIC_POINT* object as second argument. To use this creation procedure you would need to first create a *TRAFFIC_POINT* object, that is passed to *make_with_point* as a second argument. If you fail to create the *TRAFFIC_POINT* object, you should not be surprised to get, at runtime, a "Feature call on Void target" message when trying to use the object.

3. **Add the object to the city by adding it to the according container.** If you forget this step, there will be no compiler error and no program crash, but you won't see the object on the displayed map. These are rather tricky errors to detect and fix, because neither the compiler nor the runtime system can help you.

The class *TRAFFIC_CITY* provides several containers for the various types of city objects and for each there is a command that allows you to put a new object into the container. To make things easier, we list them here and show you, for most of the city item types, how they should be put into the right container:

Adding objects to containers

<i>Object declaration</i>	<i>Adding the object to the map</i>
v: <i>TRAFFIC_VILLA</i>	<i>Paris.put_building (v)</i>
a: <i>TRAFFIC_APARTMENT_BUILDING</i>	<i>Paris.put_building (a)</i>
s: <i>TRAFFIC_SKYSCRAPER</i>	<i>Paris.put_building (s)</i>
b: <i>TRAFFIC_BUS</i>	<i>Paris.put_bus (b)</i>
f: <i>TRAFFIC_FREE_MOVING</i>	<i>Paris.put_free_moving (f)</i>
l: <i>TRAFFIC_LANDMARK</i>	<i>Paris.put_landmark (l)</i>
li: <i>TRAFFIC_LINE</i>	<i>Paris.put_line (li)</i>
p: <i>TRAFFIC_PASSENGER</i>	<i>Paris.put_passenger (p)</i>
r: <i>TRAFFIC_ROAD</i>	<i>Paris.put_road (r)</i>
ro: <i>TRAFFIC_ROUTE</i>	<i>Paris.put_route (ro)</i>
st: <i>TRAFFIC_STATION</i>	<i>Paris.put_station (st)</i>
t: <i>TRAFFIC_TAXI</i>	<i>Paris.put_taxi (t)</i>
to: <i>TRAFFIC_TAXI_OFFICE</i>	<i>Paris.put_taxi_office (to)</i>
tr: <i>TRAFFIC_TRAM</i>	<i>Paris.put_tram (tr)</i>

Listing 1: Creating a new *TRAFFIC_STATION* object

```

1 class
2   CREATION_EXAMPLE
3
4   inherit
5
6   TOURISM
7
8   feature -- Explore Paris
9
10    explore is
11      -- Create a new station.
12      do
13        Paris.display
14
15        -- Step 2: Creation of the new object
16        create station.make_with_location ("Home", 600, 700)
17
18        -- Step 3: Adding new object to the map
19        Paris.put_station (station)
20      end
21
22      station: TRAFFIC_STATION
23      -- Step 1: Declaration of attribute
24      -- New station
25
26 end
  
```

Listing 2: Class `TRAFFIC_STATION` (shortened and slightly adapted)

```
class
2  TRAFFIC_STATION

4  inherit
    HASHABLE
6    redefine
    out
8    end

10 TRAFFIC_CITY_ITEM
    undefine
12    out,
    add_to_map,
14    remove_from_map
    end
16
create
18  make, make_with_location, make_with_point

20 feature {NONE} -- Initialize

22  make (a_name: STRING) is
    -- Initialize 'Current'.
24    require
    a_name_exists: a_name /= Void
26    a_name_not_empty: not a_name.is_empty
    ensure
28    name_set: equal (a_name, name)
    location_exists : location /= Void
30    end

32  make_with_location (a_name: STRING; a_x, a_y: INTEGER) is
    -- Initialize 'Current' with name 'a_name' and location 'a_x', 'a_y'.
34    require
    a_name_exists: a_name /= Void
36    a_name_not_empty: not a_name.is_empty
    ensure
38    name_set: equal (a_name, name)
    location_exists : location /= Void
40    location_set : location.x = a_x and location.y = a_y
    end
42

44  make_with_point (a_name: STRING; a_point: TRAFFIC_POINT) is
    -- Initialize 'Current' with name 'a_name' and location 'a_point'.
    require
46    a_name_exists: a_name /= Void
    a_name_not_empty: not a_name.is_empty
48    a_point_exists : a_point /= Void
    ensure
50    name_set: equal (a_name, name)
    location_exists : location /= Void
```

```
52     location_set : location.x = a_point.x and location.y = a_point.y
53     end
54 feature -- Access
55
56     ...
57
58 end
```

To do

1. Download http://se.inf.ethz.ch/teaching/2009-H/eprog-0001/exercises/assignment_4.zip and extract it in `traffic/example`. You should now have a new directory `traffic/example/assignment_4` with `assignment_4.ecf` directly in it. It is important that the location corresponds to the description here!
2. Open and compile this new project. Open class `OBJECT_CREATION` and solve the tasks below.
3. Declare an attribute of type `TRAFFIC_PASSENGER` in class `OBJECT_CREATION` (step 1). Then create an object of type `TRAFFIC_PASSENGER`. Make it walk along `Route3` (step 2) by calling feature `go` on it. As a final step add it to Paris (step 3). If you want the passenger to walk back and forth on his route you can call feature `set_reiterate` (`True`) on the passenger. Run your program. Note: The code that you just produced is very similar to the one found in feature `animate` of `TRAFFIC_ROUTE`.
4. Create an object of type `TRAFFIC_TRAM`, so that it will follow `Line1`. To make it start moving, call feature `start` on the created tram and add it to Paris.
5. Create a new landmark for the Gare de Lyon railway station. The creation procedure expects as first argument the coordinate of the landmark center, as a second argument a name for the landmark, and as third argument a path to an image file. To obtain the coordinate of the landmark center for the first argument, create a `TRAFFIC_POINT` object having the same x and y values as the station. To get these values, use query `location` of the `TRAFFIC_STATION` object you obtain by calling `Station_Gare_de_Lyon` from `TOURISM`. For the third argument (the image path) use `"train_station.png"` since the image file should be located in the root directory `traffic/example/assignment_4`. Add the landmark to Paris.
6. As a next step, you will create an object of type `TRAFFIC_FREE_MOVING`. To do this, you first need to create an object of type `TRAFFIC_POINT_RANDOMIZER`. A point randomizer object can generate a list of points within city bounds. Such a list of points is needed as an argument to the creation procedure of `TRAFFIC_FREE_MOVING`. That is why you have to create a point randomizer object before creating the free moving object. The creation procedure of `TRAFFIC_POINT_RANDOMIZER` expects the city center and the city radius of `Paris` as arguments. Have a look at `TRAFFIC_CITY` and you will find the needed features. You do not need to add the point randomizer object to Paris, since it is only a temporary helper object. To generate a new point list, use `generate_point_array` (`n`) (where `n` stands for the number of points in the list). The generated list is accessible through the feature `last_array`.
After you have created the point randomizer and generated a new list of points, you create a free moving object that travels along this list of generated points. Again, you will need to call feature `start` on it and add it to Paris.

Create an object of type `TRAFFIC_TAXI` associating it to a newly generated point list using the point randomizer, start it and add it to Paris.

7. Create a new line of type `TRAFFIC_LINE`. Make sure to use the creation procedure `make_with_terminal`, otherwise you will get a precondition violation when using the feature `extend` on it. The creation procedure has three arguments: the first one is the name of the new line, that should be "Tourist line"; the second is the type of line, in our case a bus line, so use an object of type `TRAFFIC_TYPE_BUS` as second argument; the third argument defines the starting place of the line, which in our case is `Station_Gare_de_Lyon`. Use the feature `extend` to add `Station_St_Michel_Notre_Dame`, `Station_Champs_de_Mars_Tour_Eiffel_Bir_Hakeim`, `Station_Charles_de_Gaulle_Etoile`, `Station_Palais_Royal_Musee_du_Louvre` as stops to the tourist line. To make the display of the tourist bus line more eye-catching, associate an object of type `TRAFFIC_COLOR` to the line (e.g. with RGB-values 255, 160, 0).
8. Create a new object of type `TRAFFIC_BUS` that moves along the tourist bus line. To make the bus drive back and forth infinitely, call feature `set_reiterate (True)` on the bus. We may sound repetitive, but don't forget to add it to Paris!

To hand in

Submit the class text of `OBJECT_CREATION` to your assistant.

2 It's Logic!

Read Touch of class, paragraph 5.3. Here are some examples:

- `if (x >= 0)and (x <= 10)then ... end`
- `if (x >= 0)and then (x.square_root >= 5)then ... end`
- `if (x < 0) or (x > 10) then ... end`
- `if (x < 0) or else (x.square_root < 5) then ... end`

To do

1. Describe the difference between semi-strict and strict boolean operators.
2. Explain when you would prefer semi-strict operators over strict operators and when you would prefer strict operators over semi-strict operators.
3. Give other examples, also involving boolean conditions, that illustrate the following:
 - and
 - and then
 - or
 - or else

To hand in

Hand in your solution to the questions above.

3 Temperature application

In this task you will write an application which converts temperatures between Celsius and Kelvin units. The application should consist of two classes: *TEMPERATURE* and *TEMPERATURE_APPLICATION*. The latter is the root class.

Things you need to know

- To print something in the console window, use *io.put_string*, *io.put_integer*, *io.put_boolean* and so on, depending on the type of the argument. To go to a new line, use *io.put_new_line*. To read user input, use *io.read_....* Use *io.last_....* to retrieve the value that was last read. As an example, reading an *INTEGER* from the console and then displaying it on the screen, looks as follows:

```
read_and_display_int is
  -- Read integer and display it.
local
  i: INTEGER
do
  io.read_integer
  i := io.last_integer
  io.put_integer (i)
  io.put_new_line
end
```

- The formula for conversion we are interested in has been simplified:

$$T_{Celsius} = T_{Kelvin} - 273$$

- A function is a query that computes a result and returns it to the caller (e.g. *kelvin_value* in *TEMPERATURE* is a function). The mechanism to define the return value of a function is based on the special entity **Result**. The object that **Result** refers to at the end of a function execution, is the return value of the function. As an example, in class *TRAFFIC_LINE* you find a feature *count* that returns the number of stations of a line. It calculates it based on the list of stops that the line stores.

```
count: INTEGER is
  -- Number of stations in this line
do
  Result := stops.count
end
```

To do

- Launch EiffelStudio. Create a new project of type “Basic application (no graphics library included)”, using the settings shown in figure 1.
- Download the skeleton classes for *TEMPERATURE* and *TEMPERATURE_APPLICATION* from http://se.inf.ethz.ch/teaching/2009-H/eprog-0001/exercises/temperature_application.e and <http://se.inf.ethz.ch/teaching/2009-H/eprog-0001/exercises/temperature.e> and put them into your project.
- Fill in the missing pieces of classes *TEMPERATURE* and *TEMPERATURE_APPLICATION* according to the comments.

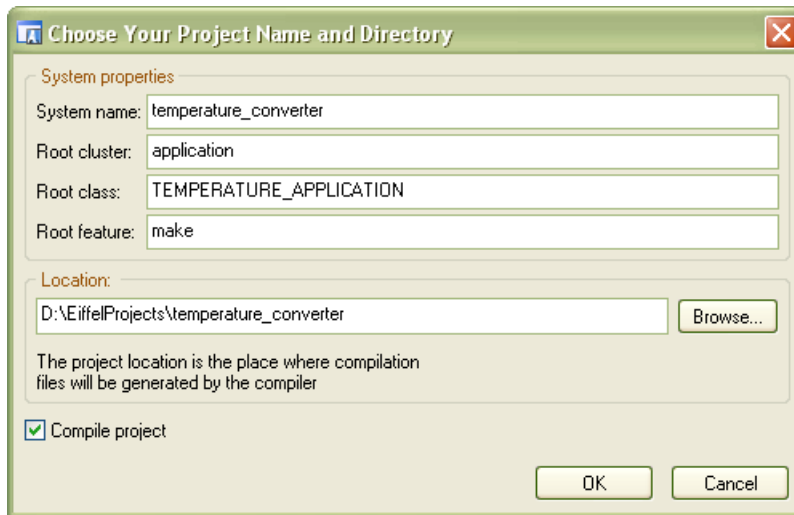


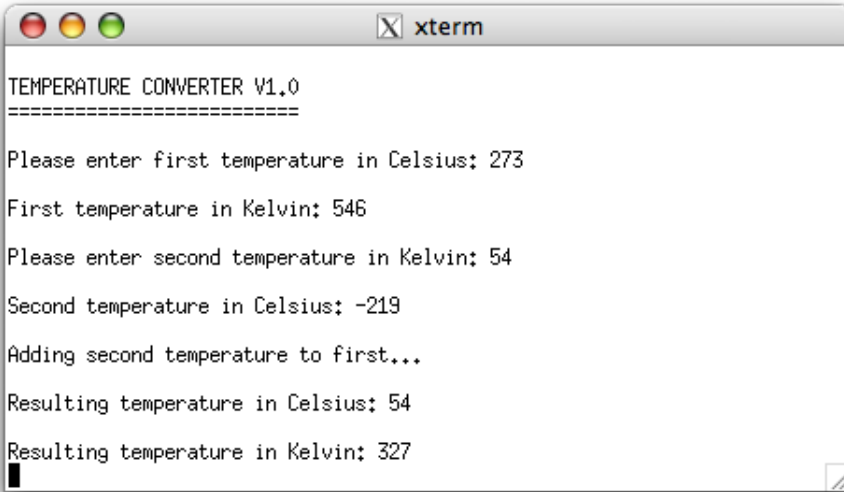
Figure 1: New project

- Do not forget to add contracts.
- Feature `make` in class `TEMPERATURE_APPLICATION` should use the `TEMPERATURE` class to do the following:
 1. Ask the user to enter a temperature in Celsius.
 2. Create a temperature object with the input value.
 3. Output the Kelvin value of it.
 4. Repeat points 1–3 for a temperature in Kelvin.
 5. Add the second temperature to the first one and output the resulting Celsius and Kelvin values.
- A final piece of advice: use the debugging features of EiffelStudio. They will help you to understand what's going on.

A sample execution of your application could yield the result shown in figure 2.

To hand in

Submit the class files for `TEMPERATURE` and `TEMPERATURE_APPLICATION`.



```
TEMPERATURE CONVERTER V1.0
=====
Please enter first temperature in Celsius: 273
First temperature in Kelvin: 546
Please enter second temperature in Kelvin: 54
Second temperature in Celsius: -219
Adding second temperature to first...
Resulting temperature in Celsius: 54
Resulting temperature in Kelvin: 327
```

Figure 2: Console