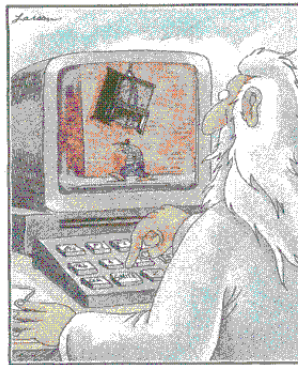


Assignment 4: Object creation

ETH Zurich

Hand-out: 10 October 2008

Due: 16 October 2008



God at His computer

Copyright FarWorks, Inc. Gary Larson

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".

The declaration of a new attribute is always placed outside of other feature declarations, but between the keyword **feature** (line 8 in Listing 1) and **end** (line 26 in Listing 1) of the class text. It is composed of the identifier (name for the new object), a colon, and the type of it. In our example the identifier is *station* and the type is **TRAFFIC_STATION** (see line 22).

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** and calling features on it will result in a program crash displaying the message "Feature call on Void target". If you like you can set additional properties of the object.

To detect the available creation procedures, look at the text of 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 or 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 an argument.

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.

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 city item types how they should be put into the according container:

Example declaration	Adding it to the map
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
  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
3
4 inherit
5 HASHABLE
6   redefine
7     out
8   end
9
10 TRAFFIC_CITY_ITEM
11   undefine
12     out,
13     add_to_map,
14     remove_from_map
15   end
16
17 create
18   make, make_with_location, make_with_point
19
20 feature {NONE} -- Initialize
21
22   make (a_name: STRING) is
```

```
24  -- Initialize 'Current'.
    require
26      a_name_exists: a_name /= Void
      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  make_with_point (a_name: STRING; a_point: TRAFFIC_POINT) is
44  -- 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
    end

54  feature -- Access
56  ...
58  end
```

To do

1. Download http://se.inf.ethz.ch/teaching/2008-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 the `TRAFFIC_PASSENGER` that walks along `Route3` (step 2) and call feature `go` on the created passenger (this will make it start moving). 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](#) following *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 a coordinate, 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](#) that has the same x and y values as the station. To do this, access *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*.
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's why you have to create a point randomizer before creating the free moving. 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 that travels along this list of generated points. Again, you will need to call feature *start* on it.
Generate a new point list using the point randomizer and create an object of type [TRAFFIC_TAXI](#). Start it.
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, this should be "Tourist line"; the second is the type of line, in our case this should be 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 a 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.

To hand in

Submit the class text of [OBJECT_CREATION](#) to your assistant.

2 It's Logic !

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 examples that illustrate your reasoning for:
 - and
 - and then
 - or
 - or else

To hand in

Hand in your solution to the questions above.

3 Temperature application (optional)

In this task you will write an application which converts temperatures between Celsius and Fahrenheit 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 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:

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

- The formula for conversion:

$$Fahrenheit = (9/5) * Celsius + 32$$

- A function computes a result and returns it to the caller (e.g. *fahrenheit_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.

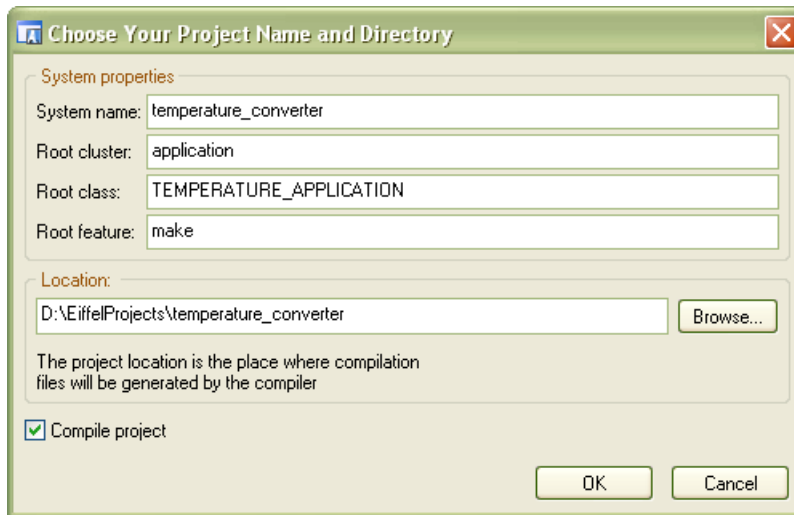


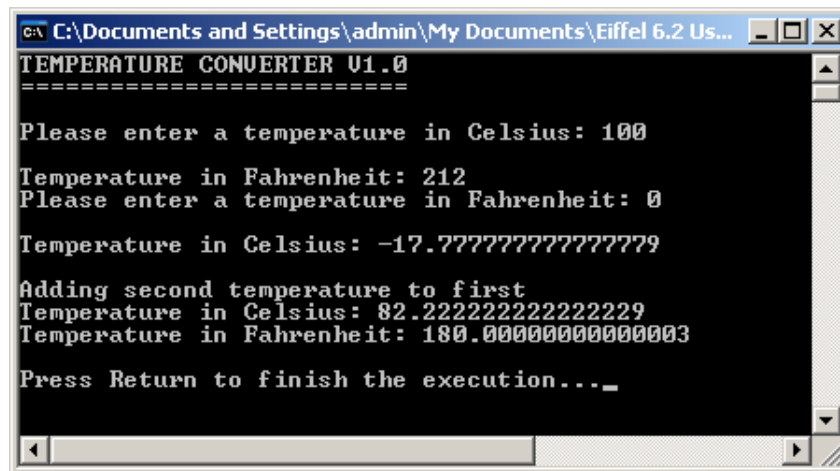
Figure 1: New project

- Download the skeleton class for *TEMPERATURE* from <http://se.inf.ethz.ch/teaching/2008-H/eprog-0001/exercises/temperature.e> and put it into your project.
- Fill in the missing pieces of class *TEMPERATURE* as indicated through the comments.
- 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 fahrenheit value of it.
 4. Repeat points 1–3 for a temperature in Fahrenheit.
 5. Add the second temperature to the first one and output the resulting celsius and fahrenheit values.

The execution of your application should yield the result shown in figure 2.

To hand in

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



```
GA C:\Documents and Settings\admin\My Documents\Eiffel 6.2 Us...
TEMPERATURE CONVERTER U1.0
=====
Please enter a temperature in Celsius: 100
Temperature in Fahrenheit: 212
Please enter a temperature in Fahrenheit: 0
Temperature in Celsius: -17.777777777777779
Adding second temperature to first
Temperature in Celsius: 82.222222222222229
Temperature in Fahrenheit: 180.000000000000003
Press Return to finish the execution..._
```

Figure 2: Console