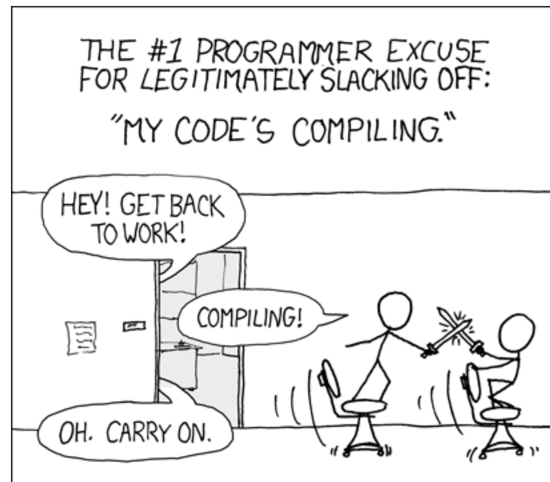
Assignment 4: Object creation

ETH Zurich

Hand-out: 8 October 2010
Due: 19 October 2010



Compiling © Randall Munroe (xkcd.com)

Goals

- Create new objects.
- Create new classes.
- 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 on the Paris map. In this assignment you will create new objects and add them to Paris. To add new items such as passengers, trams, places, lines, or roads to a city in Traffic you can follow a general scheme:

1. Declare an *attribute* [Touch Of Class, page 238] or a *local variable* [Touch Of Class, page 233] of the required type. For example:

```
class
    OBJECT_CREATION

feature -- Explore Paris
```

```
    station: TRAFFIC_STATION
            -- An example of an attribute.

    explore is
            -- Create new objects for Paris.
        local
            p: TRAFFIC_PASSENGER -- An example of a local variable.
        do
            ...
        end
end
```

Prefer local variables over attributes unless the object is used by more than one feature or you need to retain the object value between feature executions.

2. Create the object using one of the creation procedures [Touch Of Class, page 122] declared in the corresponding class. For example:

```
class
    TRAFFIC_STATION

create
    make, make_with_location

feature {NONE} -- Initialization
    make (a_name: STRING)
            -- Create station with name 'a_name'.
        ...
        end

    make_with_location (a_name: STRING; a_x, a_y: INTEGER)
            -- Create station with name 'a_name' at location ('a_x', 'a_y').
        ...
        end
end

-- In your code:
create station.make_with_location ("Central", 0, 100)
```

You don't have to create the object if its type is "expanded". In this case the object will be automatically created for you. Examples of expanded types are *INTEGER*, *BOOLEAN* and *DOUBLE*.

3. Add the object to the city by calling an appropriate command on the city. If you forget this step, you won't see the object on the displayed map. To make things easier, in Table 1 we list the commands for adding objects of the most widely used types to a city.

To do

1. Download http://se.inf.ethz.ch/teaching/2010-H/eprog-0001/assignments/04/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.

2. Open and compile this new project. Open class *OBJECT_CREATION* and solve the tasks below.

3. Declare a command *add_passenger* in *OBJECT_CREATION*, which adds a new passenger (object of type *TRAFFIC_PASSENGER*), walking along *Route3*, to *Paris*. To make the passenger start moving, call feature *go* on it. If you want the passenger to walk back

Table 1: Adding objects to a city

Object type	Command
TRAFFIC_VILLA	*put_building*
TRAFFIC_APARTMENT_BUILDING	*put_building*
TRAFFIC_SKYSCRAPER	*put_building*
TRAFFIC_BUS	*put_bus*
TRAFFIC_FREE_MOVING	*put_free_moving*
TRAFFIC_LANDMARK	*put_landmark*
TRAFFIC_LINE	*put_line*
TRAFFIC_PASSENGER	*put_passenger*
TRAFFIC_ROAD	*put_road*
TRAFFIC_ROUTE	*put_route*
TRAFFIC_STATION	*put_station*
TRAFFIC_TAXI	*put_taxi*
TRAFFIC_TAXI_OFFICE	*put_taxi_office*
TRAFFIC_TRAM	*put_tram*

and forth on his route you can call *set_reiterate* (*True*) on it. Add a call to the feature *add_passenger* in *explore* and run your program[1].

4. Declare a command *add_tram*, which adds a new tram following *Line1* to *Paris*. To make it start moving, call feature *start* on it.

5. Declare a command *add_landmark*, which adds a new landmark for the Gare de Lyon railway station. The creation procedure expects the coordinate of the landmark center, a name, and a path to an image file as arguments. Use the location of *Station_Gare_de_Lyon* as the coordinate of the landmark center and **"train_station.png"** as the image path (the image file should be located in the directory `traffic/example/assignment_4`).

6. Declare a command *add_free_moving*, which adds a new free moving object to *Paris*.

   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; upon creation you should give it the center point and the radius of *Paris*. 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*. The generated list is accessible through the feature *last_array*.

   After you have created the point randomizer and generated a new list of points, create a free moving object that travels along this list of generated points. Again, you will need to call feature *start*.

7. Declare a command *add_line*, which adds a new **bus** line called "Tourist line" to *Paris*. Make sure to use the creation procedure *make_with_terminal*. The line should go from Gare de Lyon to St Michel Notre Dame, then to Champs de Mars Tour Eiffel Bir-Hakeim, then to Charles de Gaulle Etoile and end at Palais Royal Musee du Louvre. To make the tourist line more eye-catching, change its color, e.g., to purple.

8. Declare a command *add_bus*, which adds a new bus driving back and forth along the tourist line.

[1]Repeat this step for all the new features you declare later in this task.

To hand in

Hand in the code of the class *OBJECT_CREATION*.

2 It's Logic!

Review the section "Semistrict boolean operators" [Touch Of Class, page 89] and answer the following questions.

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. In a boolean expression

   ```
   x /= 0 and then y \\ x = 0
   ```

   it is essential to use **and then** because the integer remainder operation would fail if $x = 0$. Give other examples of boolean expressions using **and**, **and then**, **or** and **or else** and explain why it is essential or optimal to use the chosen boolean operator in each example.

To hand in

Hand in your examples and explanations.

3 Temperature application

In this task you will write an application which converts temperatures between Celsius and Kelvin scales using the following formula:

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

The application should consist of two classes: *TEMPERATURE* and *APPLICATION*. Class *TEMPERATURE* encapsulates the notion of temperature; it hides from its clients implementation details such as how temperatures are converted between different scales. Class *APPLICATION* is a root class and a client of *TEMPERATURE*; it provides the user interface to the application.

To do

1. Launch EiffelStudio. Create a new project of type "Basic application (no graphics library included)", using the settings shown in figure 1.
2. Download the skeleton classes for *TEMPERATURE* and *APPLICATION* from http://se.inf.ethz.ch/teaching/2010-H/eprog-0001/assignments/04/temperature.e and http://se.inf.ethz.ch/teaching/2010-H/eprog-0001/assignments/04/application.e and put them into your project directory.
3. Fill in the missing pieces of classes *TEMPERATURE* and *APPLICATION* according to the comments. Feature *execute* of class *APPLICATION* must use the class *TEMPERATURE* to perform the conversion.

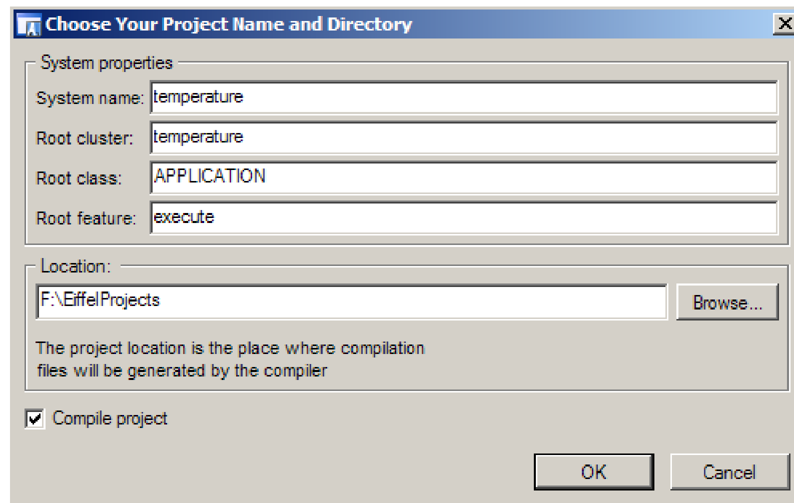


Figure 1: New project

4. Express the following properties using contracts:
   - *make_celcius* sets the Celsius temperature to the provided value;
   - *make_kelvin* sets the Kelvin temperature to the provided value;
   - a temperature can never be below zero in the Kelvin scale.

A sample execution of your application could yield the following result:

```
Enter the first temperature in Celsius: 0
The first temperature in Kelvin is: 273
Enter the second temperature in Kelvin: 283
The second temperature in Celsius is: 10
The average in Celsius is: 5
The average in Kelvin is: 278
```

To hand in

Hand in the code of *TEMPERATURE* and *APPLICATION*.

4 Ein ticket für alles

Imagine that you work for ZVV. You are writing an application that collects and stores information about customers and determines which customers can have discount on their seasonal tickets. The rule is that a person is eligible for discount if he or she is under 25. However, you don't want to store the age of each customer explicitly (otherwise you will have to update the data every time someone has a birthday). Instead, you want to store the birth date of a customer and calculate his or her age based on the birth date and the current date.

Todo

1. Launch EiffelStudio. Create a new project of type "Basic application (no graphics library included)", with a name "ticket", root class *APPLICATION* and root creation procedure *execute*.

2. To store and manipulate dates use the class *DATE*, which is defined in a library called "time". Using the "Add a library" button at the top of the "Groups" panel, add the "time" library to your project. Choose the file `$ISE_LIBRARY\library\time\time.ecf`, not the `$ISE_LIBRARY\library\time\time-safe.ecf`! Compile the project again.

3. Using the "Add a new class" button at the top of the "Groups" panel, create a new class *CUSTOMER*, which would encapsulate information about customers. Add attributes to store the customer's first name, last name and birth date. Add a creation procedure that accepts a first name, a last name and a birth date as arguments and stores them in the corresponding attributes.

4. Declare a function *age*: *INTEGER*, which returns customer's age in years. Hint: to calculate the age create another object of type *DATE* storing today's date and then calculate the difference between the two dates using features of class *DATE*.

5. Declare a function *has_discount*: *BOOLEAN*, which answers the question, whether the customer is eligible for discount.

6. Declare a function *info*: *STRING*, which returns the information about the customer (first name, last name, age, discount eligibility) as text.

7. In the feature *execute* of class *APPLICATION* ask the user to input customer's first name, last name and age. Store this information in a newly created object of type *CUSTOMER*. Then output the information about the customer using the feature *info*.

   Hint: probably the easiest way to create a *DATE* object to store the birth date is to use the creation procedure *make_from_string_default*. This procedure converts strings in the format "mm/dd/yyyy" into dates; ask the user to input the date in this format and assume for simplicity that he always does it correctly.

8. **Optional.** Instead of asking the user to input the first name on one line and the last name on another line, ask to input the whole full name at once (in the format `"Firstname Lastname"`). Before creating the *CUSTOMER* object separate the resulting string into the first name and the last name. Hint: you can use features *index_of* and *substring* of class *STRING*. You can assume for simplicity that the full name always consists of two words with exactly one space in between.

A sample execution of your application could yield the following result:

```
Enter full name: John Smith
Enter birth date as mm/dd/yyyy: 06/15/1989

First name: John
Last name: Smith
21 years old
Eligible for discount: True
```

To hand in

Hand in the code of classes *CUSTOMER* and *APPLICATION*.