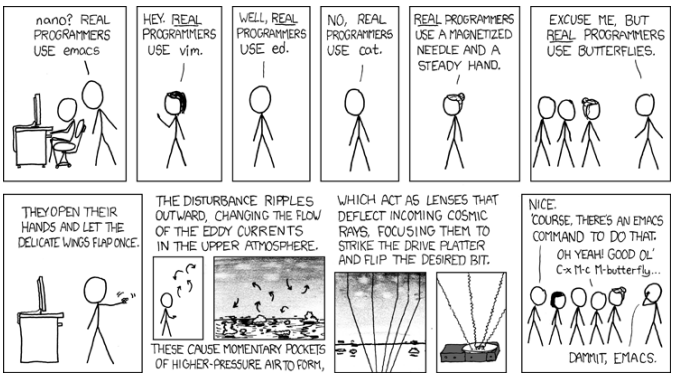
Assignment 1: Getting Started with classes, objects and features



Real Programmers c Randall Munroe (`http://xkcd.com/378`)

Published: August 23, 2016
Due: September 20, 2016

Goals

- Write more feature calls (learn to distinguish between queries and commands).
- Learn what makes up a valid feature call.
- Understand the difference between a class and an object.
- Read and write query call chains.
- Process user input.
- Add new features to a class.

1 ALPHABETICAL ORDER

To do

The question in this exercise is simple to state and corresponds to a concept that everyone knows:

Define alphabetical order.

No doubt you have an "operational" answer since you can find a word in a dictionary. But that is not the answer to the question. We do not want a process (look at the first letters of both words, if one of them is before the other its word is before the other alphabetically, otherwise if they are the same etc.). We want a definition. The answer must be a completion of the following sentence:

A word x is alphabetically before a word y if and only if ...

Replace the "..." by a proper condition on x and y.
To make the details precise: we restrict ourselves to words made of lower-case letters from the plain 26-letter Roman alphabet; letters are ordered ('a' precedes all the others, 'b' precedes all others except 'a' and so on).

To hand in

Hand in your answers.

2 BINARY SEARCH

This exercise is for students in advanced labs (optional for beginner groups).

To do

Which ones if any of the four given binary search programs, are correct?

(P1)

```
from
    i := 1; j := n
until i = j loop
    m := (i + j) // 2
    if t [m] <= x then
        i := m
    else
        j := m
    end
end
Result := (x = t[i])
```

(P2)

```
from
    i := 1; j := n; Result := False
until i = j and not Result loop
    m := (i + j) // 2
    if t [m] < x then
        i := m + 1
    elseif t [m] = x then
        Result := True
    else
        j := m - 1
    end
end
```

(P3)

```
from
    i := 0; j := n;
until i = j loop
    m := (i + j + 1) // 2
    if t [m] <= x then
        i := m + 1
    else
        j := m
    end
end
if i >= 1 and i <= n then
    Result := (x = t [i])
else
    Result := False
end
```

(P4)

```
from
    i := 0; j := n + 1;
until i = j loop
    m := (i + j) // 2
    if t [m] <= x then
        i := m + 1
    else
        j := m
    end
end
if i >= 1 and i <= n then
    Result := (x = t [i])
else
    Result := False
end
```

DETAILS: We have an array *t* of integers, indexed from 1 to *n*, and a value *x*, also an integer. We want to return **Result = True** if and only if *x* appears among the elements of the array. You may assume *n >= 1*. (If *n = 0* the array is empty and **Result** should be false.) You may also assume that the elements of *t* are in ascending order: *t [1] <= t [2] <= t [3]* and so on. This assumption is at the basis of the binary search algorithm, whose general idea is: take an element in the middle of the array, i.e. about at index *(1 + n) / 2*; if the element *x* we

are looking for is equal to it, return True and terminate; if it is less, repeat on the left part of the array; if it is greater, repeat on the right side. This is a reasonably fast algorithm because at each step we divide by 2 the number of elements to be examined.

NOTATIONS: a loop is written

from *a* **until** *c* **loop** *b* **end**

and its execution starts with *a* (the initialisation), then gets into the second part: if *c* is true, stop, otherwise execute *b* and repeat the second part (that is to say, stop if *c* is true, otherwise execute *b* and so on). Note that *a* is always executed, and *b* can be executed zero, one or more times depending on how quickly we get to a state satisfying *c*. Of course, that may be never: one way for a program to be wrong is that it never terminates.
For integers *m* and *n*, *m // n* is their integer division, for example *(6 // 2) = (7 //2) = 3*.
The *i*-th element of an array *t*, for *1 <= i <= n*, is written *t [i]*.
A search program (**P1**, **P2**, **P3** or **P4**) is correct if for any array *t* and any *x*:

- It always terminates.
- It does not modify the array or *x*.
- It never performs an illegal operation, such as division by zero, or trying to access a non-existent array element such as *t [0]* or *t [n+1]*), which would cause a crash.
- The result at the end of the execution is correct, i.e. **Result** is **True** if *x* appears in the array and **False** otherwise.

If you think a program is incorrect, please give an example of input (array *t* and value *x*) for which it will not work, and say why it does not work.

To hand in

Hand in your answers.

3 ZURICH NEEDS MORE STATIONS

In this task you will continue exploring Traffic and write more feature calls, with and without arguments.

To do

1. Download `https://drive.google.com/open?id=0B1GMHm59JFjqSjdPTHA5Q2Z0aHc`[1] and unzip it. Open `assignment_1_a.ecf` from within EiffelStudio. Look at the class

[1] you will need those files for the rest of the assignment

PREVIEW again: you will see that the feature explore is now empty. If you run the program, you will see the map of Zurich and nothing else going on (nothing is highlighted or animated). In this task you will add feature call instructions to explore (between do and end) and check how they affect the map.

2. Let us add a new station to one of the tram lines in Zurich. To achieve that you will have to call a feature on the predefined object Zurich. Try to find out which feature it is: type *Zurich* followed by a dot, and go through the pop-up list until you see something that fits. Note that this feature requires some arguments: you have to provide Traffic with the name for the new station as well as its x and y coordinates. The name can be any text string you like in double quotes, for example "Zoo". The coordinates are measured in meters starting from the city center. For example, $x = 1800$ and $y = -500$ would denote a location 1800 meters to the east and 500 meters to the south from the city center.

3. To add the new station to one of the existing tram lines invoke the feature connect station on Zurich, giving it as arguments the line number and the name of the station.

4. If you run the program now, you won't see any changes on the map. This is because in Traffic we distinguish two kinds of objects: the model of the city (in our case Zurich) and its visual representation, the view (in our case called Zurich map). Whenever you change the model, you have to let the view know that it has to update itself accordingly. Add a call to *Zurich_map.update* and you will see that the new station now appears on the map. If you would like the map to zoom automatically so that the new station fits on the screen, add a call to *Zurich_map.fit* to window.

5. Now let us attract some attention to the new station and make it blink. You can achieve that by highlighting and unhighlighting the view of the station several times in a row. In order to access the station view use the expression *Zurich_map.station.view (...)*, which takes a station as an argument; to get to the station you can use *Zurich.station (...)*, and provide the station name as an argument, for example "Zoo". To make the blinking visible, call the feature wait after each highlight and unhighlight instruction, giving as an argument the number of seconds you want to wait. Notice that wait is not invoked as the other features, by using an object name and then a dot, but just as it is (it is an unqualified call [Touch Of Class, page 134]).

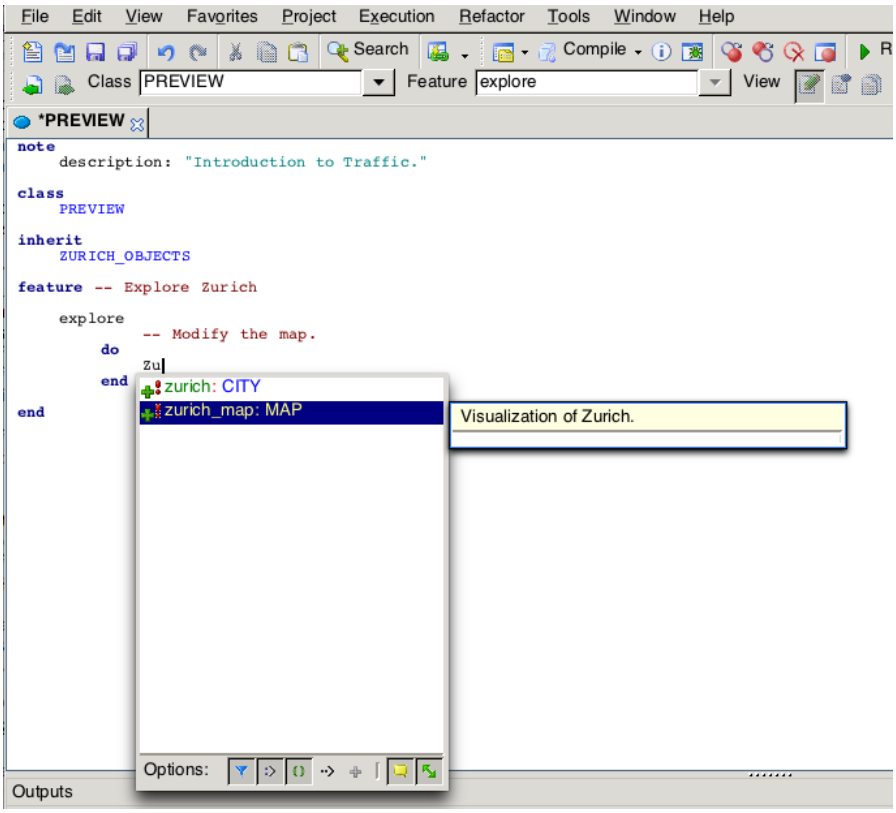
6. Let us find out where the feature wait comes from. As it appears in an unqualified call, it must be defined either in the same class or in an ancestor class. An ancestor class for a class *C* is a class that *C* inherits from. You may have noticed the **inherit** *ZURICH_OBJECTS* clause after class *PREVIEW*. It means that *PREVIEW* can use all the features defined in *ZURICH_OBJECTS*. In *PREVIEW* there is no wait, so let us check *ZURICH_OBJECTS*. Right-click on the class name *ZURICH_OBJECTS* in the editor and choose the option "Retarget to class ZURICH_OBJECTS". You can also type "zurich_objects" in the drop down box on the top left (labeled "Class"). Let us now check the features of class *ZURICH_OBJECTS*. On the bottom of the right panel select

the tab labeled “Features”. You should now see a list of all the features defined in class *ZURICH_OBJECTS*, including wait.

HINT There are two shorter ways to find wait. While in class *PREVIEW*, type “wait” in the drop down box labeled “Feature” above the editor window. Alternatively, right-click on wait in the program text and then select “Retarget to Feature wait”. This will bring up the desired feature in class *ZURICH_OBJECTS*.

HINT To check the list of all available features, press “Ctrl + Space” while in the editor window. To check the list of available features whose names start with a certain prefix, type this prefix and then press “Ctrl + Space” (see Figure 1). When you declare a variable or need a class name, to check the list of available classes whose names start with a certain prefix, type this prefix and then press “Ctrl + Shift + Space”.

Figure 1: EiffelStudio’s auto-completion feature.



To hand in

Hand in the code of feature explore (i.e. hand in the class *PREVIEW*, it correspond to the file `preview.e`).

4 Command or Query?

To do

Features listed below can be found in class *CITY* (from the library Traffic). Your task is to find out which features are commands and which features are queries [Touch Of Class, page 29]. You are supposed to base your decision on the feature definition rather then its name (the name can give you a hint, but it can also be misleading sometimes). If the feature definition appears in the class text in the form: *feature_name: CLASS NAME* or *feature_name (...): CLASS NAME*, then it is a query. If it appears in the form: *feature_name* or *feature_name (...)*, then it is a command.
Now for each of the following features in *CITY*, figure out whether it is a command or a query:

1. Feature *name*, as in *Zurich.name*.
2. Feature *buildings*, as in *Zurich.buildings*.
3. Feature *add_line*, as in *Zurich.add_line (2, "tram")*.
4. Feature *connecting_lines*, as in *Zurich.connecting_lines (central, polyterrasse)*
5. Feature *move_all*, as in *Zurich.move_all (0.5)*.
6. Feature *north*, as in *Zurich.north*.

To hand in

Hand in your answers.

5 Introducing yourself

In this task you will write your first standalone program (not based on Traffic). The program will introduce yourself to your assistant.

To do

1. Open "introduction.ecf" in EiffelStudio (you can find it in the file you downloaded in task 3). In the "Groups" tool on the right you can see that the whole project consists of a single class *APPLICATION*. Open this class in the editor. You will see that it has a single feature, *execute*, whose body is empty so far.
2. Modify the feature *execute* so that it prints the following text (replace the information about John Smith with your personal data):

Name: John Smith
Age: 20
Mother tongue: English
Has a cat: True

You can also add any other information you like.

To do the printing you will use the predefined object called *Io* (input-output). The features you can call on Io are defined in the class *STD_FILES*. Browse this class to find the features you need. In particular pay attention to:

- feature *put_string* that takes a text string (e.g. "Hello, world!") as an argument and prints it;
- feature *put_integer* that takes an integer number (e.g. 5) as an argument and prints it;
- feature *put_boolean* that takes a boolean value (True or False) as an argument and prints it;
- feature *new_line* that moves to the next line.

Compile and run your program.

HINT: the console window with the program output does not automatically pop out on all platforms. If your program appears to be doing nothing, look for a minimized window.

On Linux always start EiffelStudio from a console; then the output will be printed to the same console.

3. Until now you have compiled and executed a program without having the possibility to check what happened after every single instruction was executed. Now let us see how to use EiffelStudio in debug mode [Touch Of Class, page 170]. Being in debug mode means being able to observe the application execution instruction by instruction, therefore increasing the chances to discover errors ("bugs").

   Right-click on the feature name *execute* in the program text and choose "Pick feature execute". Now right-click in the context tool (the area below the editor). The code of execute should now appear in the context tool, with gray circles on the left (see an example in Figure 2). These circles identify instructions that will be executed. Click on the first gray circle; it should become red. You have just set a breakpoint, at which the program will pause execution.

   Now click the "Run" button (see Figure 3) or press "F5": the program will start, but almost immediately it will pause its execution at your breakpoint. Now you can observe the program behaviour step by step by clicking the "Step" button (or pressing "F10"). To resume the normal execution click on the "Run" button again (or press "F5").

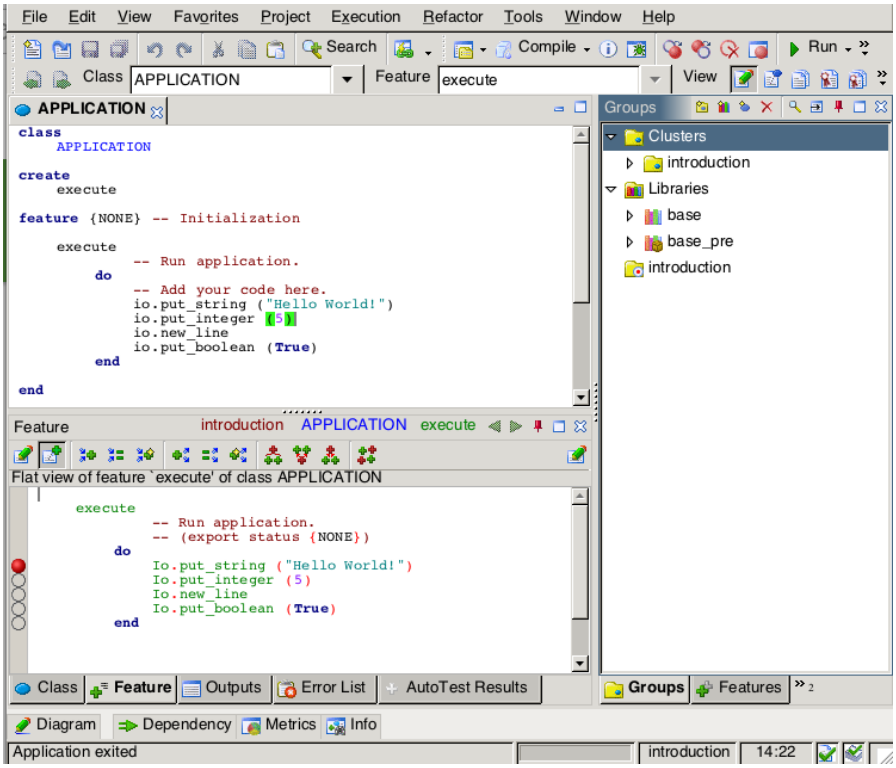


Figure 2: Setting a breakpoint.

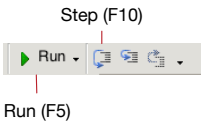


Figure 3: “Run” and “Step” buttons.

To hand in

Hand in the code of feature execute (i.e. `application.e`).

6 Classes vs. objects

To do

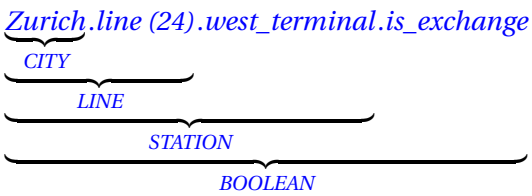
1. Describe the difference between a class and an object (1–2 sentences).
2. Find an analogy (not an example!) that illustrates the relationship between objects and classes in real life.

To hand in

Hand in your answers.

7 QUERY CALL CHAINS

As you already know, queries (unlike commands) always return an object. The type of this object can be determined from the signature of the query [Touch Of Class, page 220].
For example, *Zurich* is defined in the class *ZURICH_OBJECTS* as *Zurich: CITY*, which means that it always returns an object of type *CITY*. This, in turn, means that you can call features defined in class *CITY* on *Zurich*, as in *Zurich.name*, which returns an object of type *STRING*.
You can form even longer chains of query calls in the same way, as in:



This query call chain answers the question "Is the west terminal of line 24 in Zurich an exchange station?" and returns an object of type *BOOLEAN* (i.e., true or false).
It is also possible to use query call chains as arguments, e.g.,
Zurich.station ("Central").position.distance (Zurich.station ("Polyterrasse").position)
This query call chain answers the question "What is the distance between the positions of stations Central and Polyterrasse in Zurich?" and returns and object of type *REAL_64* – a 64 bit real number.

To do

1. To answer the questions below you will need to read feature declarations, open `assignment_1_b.ecf` from within EiffelStudio (you can find it in the file you downloaded in task 3). All query call chains in this task start with the query *Zurich*, so open *ZURICH_OBJECTS*, where the query is defined.
2. For each query call chain below, determine the type of the object it returns and state informally the question it answers, following the examples given above. To understand what individual queries mean, read the header comments in their declaration.
   a) *Zurich.line (5).kind.name*
   b) *Zurich.station ("Hardplatz").position.length*
   c) *Zurich.line (2).distance (Zurich.station ("Bellevue"), Zurich.line (2).west terminal)*
3. For each question below write a query call chain that answers the question. You can test your answers in the feature *explore* of class *PREVIEW* (following the examples that are already there).

a) How bright is the color of line 13?

b) How many meters to the north of the city center is the third station of line 31 located?

c) What is the next station of line 31 after Loewenplatz in the direction of its west terminal?

d) (Optional) How many public transportation lines go through station Paradeplatz?

e) (Optional) Does line 7 connect stations Paradeplatz and Rennweg directly?

HINT To navigate between classes and features in EiffelStudio, in addition to the mechanisms described above, you can use the "pick-and-drop" technique. Just 'pick' a class (or a feature) by holding down the [SHIFT] key and right-clicking on the class (feature) name. The cursor will change shape to an oval (or a thick cross in case you picked a feature). You can then 'drop' it in another tools pane within EiffelStudio by right-clicking again. When this is not possible, a thin red cross appears on the cursor.

To hand in

For the queries 2a – 2c hand in the type of the object they return and the question they answer; for the questions 3a–3e hand in the queries that answer the question.

8 IN AND OUT

In this task you will read some input data from the user and then output the processed data back to the user. You will also have to add new features to a class.

To do

1. Open *business_card.ecf* in EiffelStudio (you can find it in the file you downloaded in task 3). This project consists of a single class *BUSINESS_CARD*; open it in the editor. *BUSINESS_CARD* contains three attributes: *name*, *job* and *age*, which are meant to store the corresponding data about the business card owner. The class is also equipped with commands *set_name*, *set_job* and *set_age* that change the values of the corresponding attributes.

2. Modify the feature *fill_in* so that it prompts the user for his name (i.e., prints `Your name:` or `Please enter your name:`), then reads the user input and stores it in the attribute name using the command *set_name*.

   Like in the previous tasks, use the predefined object *Io* to perform input and output. To read input data use the commands *read_line*, *read_integer*, *read_character*, *read_real*, etc.

To access the data read by the last *read_x* command, use the queries *last_string*, *last_integer*, *last_character*, *last_real*, etc., accordingly. For example, the sequence of instructions:

```
Io.read_line
set_country (Io.last string)
```

reads a line and passes it as an argument to a command set country.

3. Modify the feature *fill_in* so that after asking the user for his name, it also asks for his job title and his age. Store this information in the corresponding attributes. After this step, a run of your program should look similar to this:

```
Your name:  John Smith
Your job:  Programmer Your age:  20
```

4. In the feature clause "Output" (i.e. in the code **feature** -- Output) declare a new procedure *print_card* [Touch Of Class, page 219]. This procedure should output a textual representation of the business card, containing all the data that it stores (name, job and age), each on a separate line.

   To assemble a string out of several parts use the string concatenation operator +. For example, an expression "Hello" + "," + "world!" results in a string "Hello, world!".

   Insert "%N" into a string at the position where you want a new line to be printed.

   Use the function *age_info* already defined in the class *BUSINESS_CARD* to output information about the age.

   Don't forget to format your code properly (for instance, using EiffelStudio's pretty-printing feature, CTRL-SHIFT-P) and add a header comment.

5. Now add a call to *print_card* at the end of the feature *fill_in*. After this step, a run of your program should look similar to this:

```
Your name:  John Smith
Your job:  Programmer
Your age:  20
John Smith
Programmer
20 years old
```

6. To make the output of your program look more like a business card letâĂŹs put a border around it. After this step a run of your program should look similar to this:

```
--------------------
|Your name:  John Smith     |
|Your job:  Programmer      |
|Your age:  20              |
--------------------
```

Modify the procedure *print_card* so that it also prints the border. For the top and the bottom border use the function *line (n: INTEGER): STRING* already defined in *BUSINESS_CARD*, which returns a horizontal line of length n. Use the constant attribute *Width* [Touch Of Class, page 250] every time you refer to the width of the card.

To output the right border you need to print a correct number of spaces between it and the text. For this, define a new function *spaces (n: INTEGER): STRING*, which would return a string consisting of n spaces. Use the function line as an example.

To get the number of characters in a string use the query *count*.

7. What is the benefit of using the constant attribute *Width*? What will happen with your program if the name entered by the user is longer than *Width*? How could you solve this problem (describe the general idea)?

8. (Optional). Surprise your assistant: add any information you like to the business card or change its style (for example, add a double border or a logo).

To hand in

Hand in the code of the class *BUSINESS_CARD* (i.e. `business_card.e` file) and your answers to the questions in point 7.