Introduction to Programming

Lab Session 2

(With material from the ETH Zurich course “Introduction to Programming”)

August 30, 2016

INNOPOLIS UNIVERSITY

News

1. About assignments in general
2. About assignment 1

News

1. About assignments in general
2. About assignment 1
3. Do not forget Quiz 1

News – about assignments in general

Information taken from the Moodle web-page (Introduction to Programming I course):

News – about assignments in general

Information taken from the Moodle web-page (Introduction to Programming I course):

Assignment Policy

If a submitted assignment contains work other than student's one it is necessary to explicitly acknowledge the source. It is encouraged to refer and quote other works, but it has to made clear which words and ideas are property and creation of the student, and which ones have come from others (which must not correspond to more than 30% of the work). If two or more assignments show evidence of being produced by unauthorized cooperative work, i.e. copied from fellow students, they will be all failed without further investigation on who produced the results and who actually copied.

News – about assignments in general

News – about assignments in general

Submission Policy

Each assignment defines a hard deadline. Any submission after the deadline will not be taken into account.

All assignment must be submitted using the Moodle system. Any other means of submission will not be taken into consideration.

News – about Assignment 1

- Published on: August 23rd, 2016

News – about Assignment 1

- Published on: August 23rd, 2016
- To be submitted on: September 20th, 2016.

News – about Assignment 1

What to submit?

News – about Assignment 1

What to submit? The assignment explicitly states what needs to be hand in: every section has a subsection called **To hand in**.

News – Assignment 1: What to hand in

Section 1 – ALPHABETICAL ORDER: *Hand in your answers.*

Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

News – Assignment 1: What to hand in

Section 1 – ALPHABETICAL ORDER: *Hand in your answers.* Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

Section 2 – BINARY SEARCH: *Hand in your answers.* Meaning

News – Assignment 1: What to hand in

Section 1 – ALPHABETICAL ORDER: *Hand in your answers.* Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

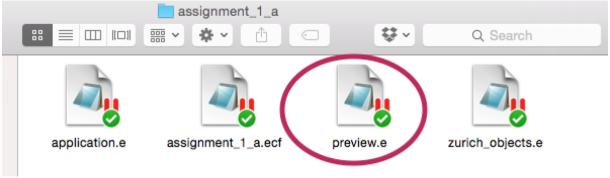
Section 2 – BINARY SEARCH: *Hand in your answers.* Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

News – Assignment 1: What to hand in

Section 1 – ALPHABETICAL ORDER: *Hand in your answers.* Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

Section 2 – BINARY SEARCH: *Hand in your answers.* Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

Section 3 – ZURICH NEEDS MORE STATIONS: *Hand in the code of feature explore (i.e. hand in the class PREVIEW, it corresponds to the file `preview.e`).* Meaning, you have to provide the file `preview.e`.


assignment_1_a
Search
application.e
assignment_1_a.ecf
preview.e
zurich_objects.e

News – Assignment 1: What to hand in

Section 4 – COMMAND OR QUERY?: *Hand in your answers.*

Meaning

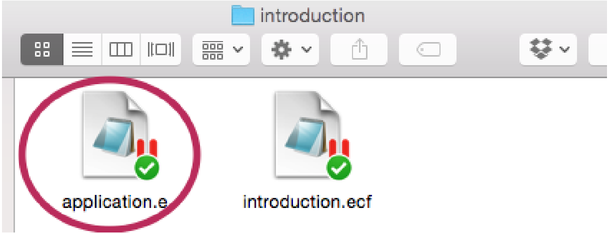
News – Assignment 1: What to hand in

Section 4 – COMMAND OR QUERY?: *Hand in your answers.*
Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

News – Assignment 1: What to hand in

Section 4 – COMMAND OR QUERY?: *Hand in your answers.*
Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

Section 5 – INTRODUCING YOURSELF: *Hand in the code of feature execute (i.e. `application.e`).*


introduction
application.e
introduction.ecf

News – Assignment 1: What to hand in

Section 4 – COMMAND OR QUERY?: *Hand in your answers.*
Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

Section 5 – INTRODUCING YOURSELF: *Hand in the code of feature execute (i.e. `application.e`).*

News – Assignment 1: What to hand in

Section 4 – COMMAND OR QUERY?: *Hand in your answers.*
Meaning, you have to write down your answer in any text editor (e.g. a doc file, a pdf file).

Section 5 – INTRODUCING YOURSELF: *Hand in the code of feature execute (i.e. `application.e`).*

Section 6 – CLASSES VS. OBJECTS: *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.*

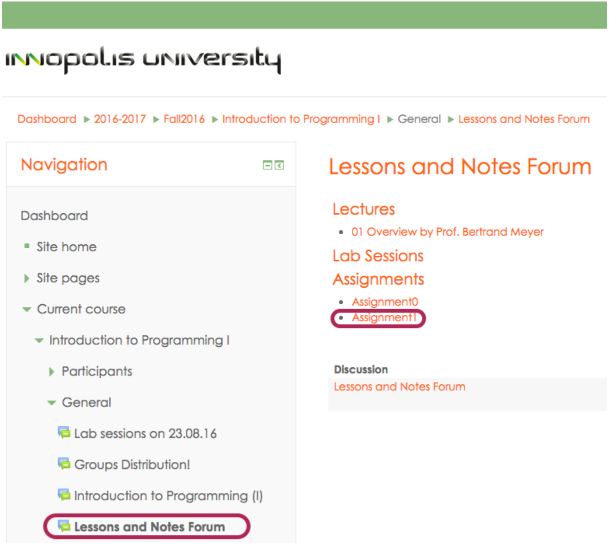
News – Assignment 1: What to hand in

Section 7 – IN AND OUT: *Hand in the code of the class BUSINESS_CARD (i.e. `business_card.e` file) and your answers to the questions in point 7.*

News – Assignment 1: What to hand in

When you already have all your answers: create an archive file (e.g. assignment_1.zip) with all your answers and submit that file. Moodle allows you to submit only one file.

News – about Assignment 1: how to submit



innopolis university

Dashboard ▸ 2016-2017 ▸ Fall2016 ▸ Introduction to Programming I ▸ General ▸ Lessons and Notes Forum

Lessons and Notes Forum

Lectures

- 01 Overview by Prof. Bertrand Meyer

Lab Sessions

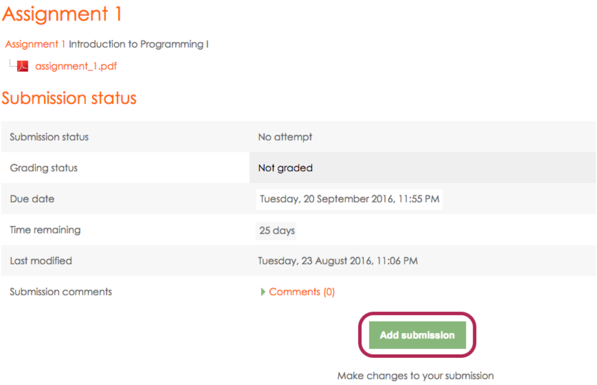
Assignments

- Assignment0
- Assignment1

Discussion

Lessons and Notes Forum

News – about Assignment 1: how to submit



Assignment 1

Assignment 1 Introduction to Programming I

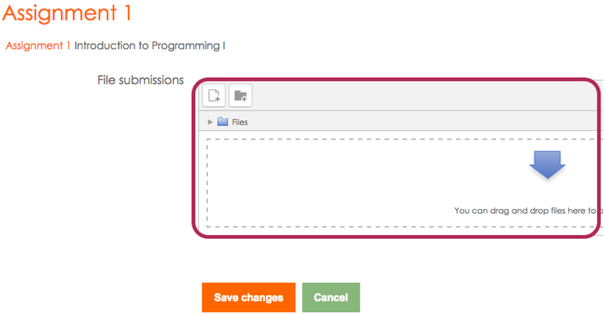
assignment_1.pdf

Submission status

Submission status	No attempt
Grading status	Not graded
Due date	Tuesday, 20 September 2016, 11:55 PM
Time remaining	25 days
Last modified	Tuesday, 23 August 2016, 11:06 PM
Submission comments	Comments (0)

Make changes to your submission

News – about Assignment 1: how to submit

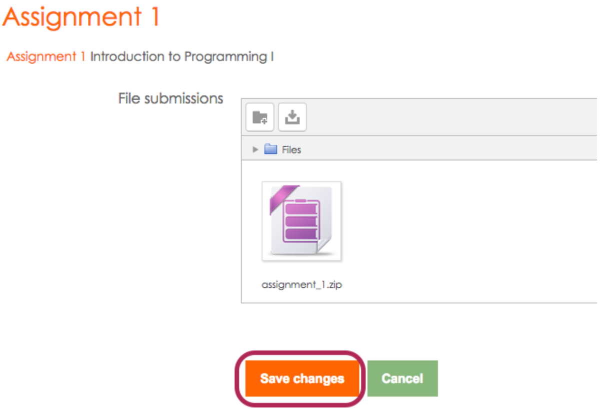


Assignment 1

Assignment 1 Introduction to Programming I

File submissions

News – about Assignment 1: how to submit

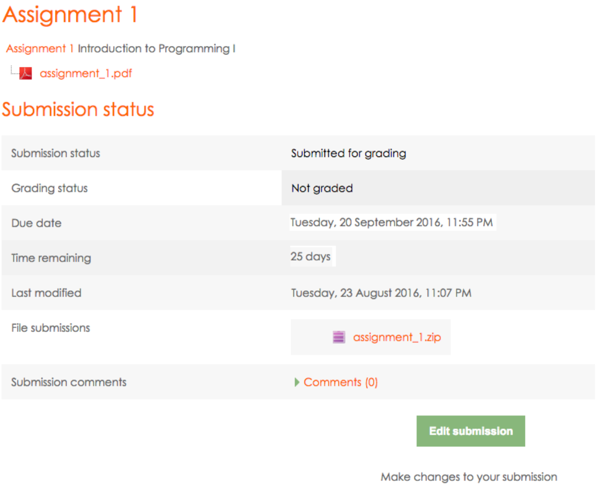


Assignment 1

Assignment 1 Introduction to Programming I

File submissions

News – about Assignment 1: how to submit



Assignment 1

Assignment 1 Introduction to Programming I

assignment_1.pdf

Submission status

Submission status	Submitted for grading
Grading status	Not graded
Due date	Tuesday, 20 September 2016, 11:55 PM
Time remaining	25 days
Last modified	Tuesday, 23 August 2016, 11:07 PM
File submissions	assignment_1.zip
Submission comments	▸ Comments (0)

Make changes to your submission

In this Lab

- We will revisit classes, features and objects.
- We will see how program execution starts.
- Understanding contracts: preconditions, postconditions (a first attempt)

Static view

- A program consists of a set of classes.
- Features are declared in classes. They define operations on objects created from classes.
  - Queries answer questions. The answer is provided in a variable called **Result**.
  - Commands execute actions. They do not return any result, so there is NO variable called **Result** that we can use.
- Another name for a class is `type`.
- Class and Type are not exactly the same, but they are close enough for now, and we will learn the difference later on.

Declaring the type of an object

- The type of any object you use in your program must be declared somewhere.
- Where can such declarations appear in a program?
  - in feature declarations:
    - formal argument types;
    - return type for queries: i.e. functions and attributes;
    - in the **local** clauses of routines.

Declaring the type of an object

- The type of any object you use in your program must be declared somewhere.
- Where can such declarations appear in a program?
  - in feature declarations:
    - formal argument types;
    - return type for queries: i.e. functions and attributes;
    - in the **local** clauses of routines.

Here is where you declare objects that only the routine needs and knows about.

Declaring the type of an object

```
class DEMO
feature
 procedure_name (a1: T1; a2, a3: T2)
     -- Comment
   local
     l1: T3
   do
     ...
   end
 function_name (a1: T1; a2, a3: T2): T3
     -- Comment
   do
     Result := ...
   end
 attribute_name: T3
   -- Comment
```

Declaring the type of an object

```
class DEMO
feature
 procedure_name (a1: T1; a2, a3: T2) ← formal argument type
     -- Comment
   local
    l1: T3
   do
    ...
   end
 function_name (a1: T1; a2, a3: T2): T3
     -- Comment
   do
    Result := ...
   end
 attribute_name: T3
    -- Comment
```

Declaring the type of an object

```
class DEMO
feature
 procedure_name (a1: T1; a2, a3: T2)      formal argument type
     -- Comment
   local
    l1: T3      local variable type
   do
    . . .
   end
 function_name (a1: T1; a2, a3: T2): T3
     -- Comment
   do
    Result := . . .
   end
 attribute_name: T3
   -- Comment
```

Declaring the type of an object

```
class DEMO
feature
 procedure_name (a1: T1; a2, a3: T2)      formal argument type
    -- Comment
   local
    l1: T3      local variable type
   do
    ...
   end
 function_name (a1: T1; a2, a3: T2): T3      return type
    -- Comment
   do
    Result := ...
   end
 attribute_name: T3
   -- Comment
```

Exercise: Find the classes / objects

```
class GAME
feature
 map_name: STRING
     -- Name of the map to be loaded for the game
 last_player: PLAYER
     -- Last player that moved
 players: PLAYER_LIST
     -- List of players in this game.
```

Exercise: Find the classes / objects

```
feature
 is_occupied (a_location: TRAFFIC_PLACE): BOOLEAN
     -- Check if ‘a_location’ is occupied.
   require
     a_location_exists: a_location /= Void
   local
     old_cursor: CURSOR
   do
     Result := False
      -- Remember old cursor position.
     old_cursor := players.cursor
     . . .
```

Exercise: Find the classes / objects

```
   -- Loop over all players to check if one occupies 'a_location'.
  from
    players.start
     -- do not consider estate agent, hence skip the first
     -- entry in 'players'.
    players.forth
  until
    players.after or Result
  loop
    if players.item.location = a_location then
      Result := True
    end
    players.forth
  end
   -- Restore old cursor position.
  players.go_to(old_cursor)
end
```

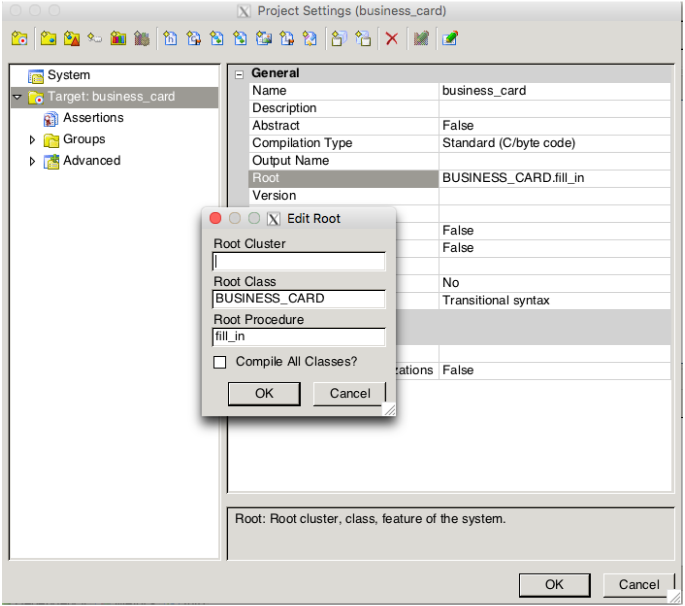
Dynamic view

- At runtime (i.e. during the program execution), we have a set of objects (instances) created from the classes (types).
- The creation of an object implies that a piece of memory is allocated in the computer to represent the object itself.
- Objects interact with each other by calling features on each other.

How all starts?

- Who creates the first object?
  - The runtime creates a so-called root object.
  - The root object creates other objects, which in turn create other objects, etc.
  - You define the type of the root object in the project settings.
- How is the root object created?
  - The runtime calls a creation procedure of the root object.
  - You define this creation procedure in the project settings.
  - The application exits at the end of this creation procedure.

Changing the root class



Project Settings (business_card)

General

Name	business_card
Description	
Abstract	False
Compilation Type	Standard (C/byte code)
Output Name	
Root	BUSINESS_CARD.fill_in
Version	
	False
	False
	No
	Transitional syntax
ations	False

Edit Root

Root Cluster

Root Class: BUSINESS_CARD

Root Procedure: fill_in

□ Compile All Classes?

Root: Root cluster, class, feature of the system.

Static view vs. dynamic view

- Queries (attributes and functions) have a result type. When executing the query, you get an object of that type.
- Routines have formal arguments of certain types. During the execution you pass objects of the same (or compatible) type as actual arguments to a routine call.
- Local variables are declared in their own section, associating names with types. During the execution, local variables may hold different values of their respective types at different points in time.

Contracts

Why do we need contracts?

- They are executable specifications that evolve together with the code.
- Together with tests, they are a great tool for finding bugs.
- They help us reason about an Object Oriented (O-O) program at the level of classes and routines.
- Proving (part of) programs correct requires some way to specify how the program *should* operate. Contracts are a way to specify the program.

Assertions

$$\underbrace{\overbrace{\text{not\_too\_small}}^{\text{Assertion Tag}} : \overbrace{\quad i >= 1 \quad}^{\text{Condition}}}_{\text{Assertion}}$$

Assertions

$$\underbrace{\overbrace{\text{not\_too\_small}}^{\text{Assertion Tag}} : \overbrace{i >= 1}^{\text{Condition}}}_{\text{Assertion}}$$

- The assertion tag is optional, but recommended: if present, it is used to construct a more informative error message when the condition is violated.

Assertions

$$\underbrace{\overbrace{\text{not\_too\_small}}^{\text{Assertion Tag}} : \overbrace{\quad i >= 1 \quad}^{\text{Condition}}}_{\text{Assertion}}$$

- The assertion tag is optional, but recommended: if present, it is used to construct a more informative error message when the condition is violated.
- The condition is required.

Precondition

Property that a feature imposes on clients:

Precondition

Property that a feature imposes on clients:

```
root_number (x: REAL): REAL
            -- Returns the root number of ‘x’
```

Precondition

Property that a feature imposes on clients:

```
root_number (x: REAL): REAL
          -- Returns the root number of ‘x’
     require
          x_positive: x >= 0
```

Precondition

Property that a feature imposes on clients:

```
root_number (x: REAL): REAL
        -- Returns the root number of 'x'
```

```
    require
        x_positive: x >= 0
```

A feature without a **require** clause is always applicable, as if the precondition reads

```
require
    always_ok: True
```

Postcondition

Property that a feature guarantees on termination

Postcondition

Property that a feature guarantees on termination

```
root_number (x: REAL): REAL
            -- Returns the root number of 'x'
```

Postcondition

Property that a feature guarantees on termination

```
root_number (x: REAL): REAL
        -- Returns the root number of ‘x’
    require
        x_positive: x >= 0
```

Postcondition

Property that a feature guarantees on termination

```
root_number (x: REAL): REAL
          -- Returns the root number of 'x'
   require
      x_positive: x >= 0
   ensure
      result_value: x = Result * Result
```

Postcondition

Property that a feature guarantees on termination

```
root_number (x: REAL): REAL
        -- Returns the root number of ‘x’
  require
    x_positive: x >= 0
  ensure
    result_value: x = Result * Result
```

A feature without an **ensure** clause always satisfies its postcondition, as if the postcondition reads

```
  ensure
    always_ok: True
```

Exercises

Pre- and postcondition example

Add pre- and postconditions to:

```
smallest_power (n, bound: NATURAL): NATURAL
            -- Smallest x such that ‘n’^x is greater or equal ‘bound’.
    require
        ???
    do
         -- to implement later
    ensure
        ???
    end
```

A possible solution

Add pre- and postconditions to:

```
smallest_power (n, bound: NATURAL): NATURAL
          -- Smallest x such that ‘n’^x is greater or equal ‘bound’.
   require
      n_large_enough: n > 1
      bound_large_enough: bound > 1
   do
       -- to implement later
   ensure
      ???
   end
```

A possible solution

Add pre- and postconditions to:

```
smallest_power (n, bound: NATURAL): NATURAL
          -- Smallest x such that ‘n’^x is greater or equal ‘bound’.
   require
      n_large_enough: n > 1
      bound_large_enough: bound > 1
   do
       -- to implement later
   ensure
         greater_equal_bound: n^Result >= bound
         smallest : n^(Result - 1) < bound
   end
```

Go to `https://drive.google.com/open?id=0B1GMHm59JFjq0GJoWEpKT3kzRzA`. Next exercises will use the zip files you can find there.

Implementing a class *COURSE*

Unzip and open the Eiffel project `university.zip`.
Fill in class *COURSE*. The class should implement a specific course given by the University. It must contain:

- a name (e.g. Introduction to Programming I),
- an identifier (e.g. 1801),
- and a schedule (e.g. Wednesdays all day),
- a maximum number of students that can be enrolled to the course (max student)

A course in the university has a minimum number of students (3 students) to start the course. Implement the class *COURSE* and add a new feature *create_class (. . .)* that creates a class (do not forget to add the pre- and postconditions).

Implementing a class *COURSE*

Unzip and open the Eiffel project `university.zip`.
Fill in class *COURSE*. The class should implement a specific course given by the University. It must contain:

- a name (e.g. Introduction to Programming I),
- an identifier (e.g. 1801),
- and a schedule (e.g. Wednesdays all day),
- a maximum number of students that can be enrolled to the course (max student)

A course in the university has a minimum number of students (3 students) to start the course. Implement the class *COURSE* and add a new feature *create_class (. . .)* that creates a class (do not forget to add the pre- and postconditions).
Make class *COURSE* the *root class* and feature *make* the *root procedure*. Try to call the feature *create_class (. . .)*.

Implementing a special bank account

Unzip and open the Eiffel project `bank.zip`.
Fill in class *SPECIAL_BANK_ACCOUNT* to have the following features:

- the name of the owner;
- the balance (the balance cannot be less than 100) – for the purpose of the exercise suppose the currency is Rubles;
- a feature to deposit money;
- a feature to withdraw money.

This account is special since it does not let the owner to have more than 1000000 Rub. Implement the class *SPECIAL_BANK_ACCOUNT* (do not forget to add the pre- and postconditions).

Implementing a special bank account

Unzip and open the Eiffel project `bank.zip`.
Fill in class *SPECIAL_BANK_ACCOUNT* to have the following features:

- the name of the owner;
- the balance (the balance cannot be less than 100) – for the purpose of the exercise suppose the currency is Rubles;
- a feature to deposit money;
- a feature to withdraw money.

This account is special since it does not let the owner to have more than 1000000 Rub. Implement the class *SPECIAL_BANK_ACCOUNT* (do not forget to add the pre- and postconditions).
Make class *SPECIAL_BANK_ACCOUNT* the *root class* and feature *make* the *root procedure*. Try out your implementation.

Thank you!