Goals

- Understand and read loops and conditionals.
- Use loops and conditionals to solve tasks.
- Use nested loops.

1 Reading loops

The structure of a loop contains multiple clauses:

- The **from** clause is required (but may be empty) and specifies the loop initialization instructions.
- The **invariant** clause is optional and contains boolean expressions that are ensured to hold by the initialization instructions and preserved by each execution of the **loop** clause as long as the exit condition is **False**.
- The **variant** clause is optional and contains an integer expression that is setup by the initialization instructions to be a non-negative integer and decreases (to a still non-negative value) by each execution of the **loop** clause as long as the exit condition is **False**.
- The **until** clause captures the exit condition of the loop and contains a boolean expression; as soon as the expression returns **True** the execution of the loop is finished. Note that defining the correct exit condition is one of the main challenges in writing loops. Always ensure that the loop does not terminate (1) too early (e.g. forgetting the last iteration), or (2) too late (e.g. executing it once more than intended)!
The loop clause contains instructions that are repeatedly executed until the exit condition of the loop is fulfilled. Always ensure that the loop body contains instructions that let the loop advance - forgetting these happens very often and results in endless loops!

The structure of a loop is shown in Listing 1 and an example of a loop in Eiffel is given in Listing 2.

Listing 1: Loop structure

```eiffel
from
  initialization_instructions
invariant
  invariant_clause
variant
  variant_clause
until
  exit_condition
loop
  loop_instructions
end
```

Listing 2: Loop example

```eiffel
loop_example is
  -- Execute a loop that prints numbers
  -- from 1 to 100.
  local
    count: INTEGER
  do
    from
      count := 1
    invariant
      count >= 1
      count <= 101
    variant
      101 - count
    until
          count > 100
    loop
      io.put.integer (count)
      io.put.new_line
      count := count + 1
    end
  end
```

To do

Assume that the two code extracts in Listing 3 and Listing 4 intend to loop through a list of places and search for the place named "place Cite Universitaire" and highlight it.

1. For each version (Listing 3 and Listing 4) decide whether it does what it is supposed to do.

2. If you think it is not OK, then correct the errors.

You may assume for this exercise that all the entities are not Void (i.e. they are all attached to an object). The feature `start` for container objects sets the internal cursor position to the beginning of the list, feature `item_for_iteration` returns the object at the cursor position, feature `forth` advances the cursor by one position, and `after` is a boolean query that returns `True` if the cursor position is past the last element. Note that `name = "place Cite Universitaire"` is not the same as `name.is_equal("place Cite Universitaire")`.

To hand in

This is a pen-and-paper exercise: you do not need to write code in EiffelStudio. Hand in your answers and if necessary the corrected versions of Listing 3 and Listing 4.
2 Equipping Paris with trams and buildings

It happens very often that you want to iterate through all the items of a container in Traffic (e.g. through Paris.places, Paris.lines, or Paris.passengers). To do this you can use the following scheme (here for Paris.lines, similar for the other containers in TRAFFIC_MAP):

Listing 5: Looping through map item containers

To do


2. Download http://se.ethz.ch/teaching/2007-F/eprog-0001/exercises/assignment_5.zip and extract it in traffic/example. You should now have a new directory traffic/example/ev_assignment_5 with assignment_5.ecf directly in it (it is important that the location corresponds to the description here!).

3. Open and compile this new project. Open class LOOPINGS and solve the tasks below.
4. Write a loop in feature `generate_trams_for_line8` that for every station belonging to `Line8` creates a tram that starts moving at this station. Use feature `set_to_station` (a place: `TRAFFIC_PLACE`) to set the initial position of a tram to a certain station (this feature is available in `TRAFFIC_TRAM` by inheriting from `TRAFFIC_LINE_VEHICLE`). Call `generate_for_line8` in feature `explore_on_click`. Don’t forget to add the generated trams to `Paris`.

5. Implement feature `generate_trams` that does the same for all tram lines of `Paris`. To find out whether a `TRAFFIC_LINE` object is a tram line, you need to create an object of type `TRAFFIC_TYPE_TRAM` and see if the line’s attribute `type` is equal to it. Call `generate_trams` in feature `explore_on_click`. You can remove the call to `generate_trams_for_line8` in `explore_on_click` since it will be done as part of `generate_trams`.

6. Feature `generate_buildings_along_road` should do the following: for the road with id `15` it should generate buildings (of type `TRAFFIC_VILLA`) along its polyline. This should result in two lines of buildings: one above the road with distance `24.0` and one below also with distance `24.0`. The distance between two buildings next to each other on a row should also be `24.0`. Figure 1 shows a schematic image of it. To access the list of points through which a road passes do the following: declare an identifier of type `TRAFFICROAD_CONNECTION` and assign `Paris.roads.item (15).one_way` to it. The attribute `polypoints` is a container holding all coordinates through which the road goes. Call `generate_buildings_along_road` in feature `explore_on_click`. Don’t forget to add the generated buildings to `Paris`.

```
local
  r: TRAFFICROAD_CONNECTION
  c: TRAFFICCOORDINATE
  do
    r := Paris.roads.item (15).one_way
    c := r.polypoints.item (1)
  end
```

![Diagram](image)

Figure 1: Buildings along a road

Note that `TRAFFIC_COORDINATE` provides so called infix-features (`+`, `−`, `∗`). They can be used to do vector calculations:
<table>
<thead>
<tr>
<th>Operation</th>
<th>Variables</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector addition</td>
<td>a, b, c: TRAFFIC_COORDINATE</td>
<td>c := a + b</td>
</tr>
<tr>
<td>Vector subtraction</td>
<td>a, b, c: TRAFFIC_COORDINATE</td>
<td>c := a - b</td>
</tr>
<tr>
<td>Scalar multiplication</td>
<td>a, b: TRAFFIC_COORDINATE, f: DOUBLE</td>
<td>b := a * f</td>
</tr>
</tbody>
</table>

(Note: the scalar needs to be the second operator)

7. Implement feature `generate\_buildings` that does the same for all roads in Paris. Call `generate\_buildings` in feature `explore\_on\_click`. You can remove the call to `generate\_buildings\_along\_road` in `explore\_on\_click` since it will be done as part of `generate\_buildings`.

8. The algorithm you implemented in step 7 will put buildings on top of line connections, cluttering the city with buildings. To prevent this you can put the following code that marks fields with line connections as occupied at the beginning of feature `generate\_buildings`:

```java
local
grid: TRAFFIC\_GRID
l: TRAFFIC\_LINE\_CONNECTION
do
create grid.make (200, Paris.center, Paris.radius)
from
   Paris.line\_sections . start
until
   Paris.line\_sections . after
loop
   l := Paris.line\_sections . item\_for\_iteration
   grid.mark\_polyline (l.polypoints, 7.0, True)
   Paris.line\_sections . forth
end
```

Now add a conditional to your code that only creates and adds a building if it does not collide with occupied grid fields. You can use the following code to do this (assuming `building\_coordinate` is the center position of the new building):

```java
if not grid. has\_rectangle\_collision (building\_coordinate, 18.0, 16.0) then
   −− Create and add building
end
```

9. In the end your application should display something similar to Figure 2.

![Figure 2: Buildings along all roads and trams](Image-Link)
To hand in
Hand in the code of class *LOOPINGS*.

3 Loop painting

![Image](image.png)

Figure 3: Example with size 7

To do
1. Write a program that asks the user to input a value, and then displays a checkered triangle of the given size as in Figure 3.

2. Be aware that stars and white spaces should be alternating.

To hand in
Hand in your class text.