Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 3
Today

- We will revisit classes, features and objects.
- We will see how program execution starts.
- We will play a role game.
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
      - functions
      - 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
      local variable type
    end

  function_name (a1: T1; a2, a3: T2): T3
    -- Comment
    do
    end

attribute_name: T3
  -- Comment

end
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.

...
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 (ie., 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.
Who are Adam and Eve?

- 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

<table>
<thead>
<tr>
<th><strong>General</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>roleplay</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>Abstract</td>
<td>False</td>
</tr>
<tr>
<td>Compilation Type</td>
<td>Standard (C/byte code)</td>
</tr>
<tr>
<td>Output Name</td>
<td></td>
</tr>
<tr>
<td>Root</td>
<td>DIRECTOR.prepare_and_play</td>
</tr>
</tbody>
</table>

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

![Image of software interface showing the process of changing the root class](image-url)
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.
Acrobat game

- We will play a little game now.
- Some of you will act as objects.
  - When you get created, please stand up and stay standing during the game
- There will be different roles
  - Acrobat
  - Acrobat with Buddy
  - Author
  - Curmudgeon
  - Director
You are an acrobat

- When you are asked to **Clap**, you will be given a number. Clap your hands that many times.
- When you are asked to **Twirl**, you will be given a number. Turn completely around that many times.
- When you are asked for **Count**, announce how many actions you have performed. This is the sum of the numbers you have been given to date.
You are an **ACROBAT**

class

ACROBAT

feature

*clap* (*n*: INTEGER)

do

-- Clap `n' times and adjust `count'.

end

*twirl* (*n*: INTEGER)

do

-- Twirl `n' times and adjust `count'.

end

*count*: INTEGER
You are an acrobat with a buddy

- You will get someone else as your Buddy.
- When you are asked to **Clap**, you will be given a number. Clap your hands that many times. Pass the same instruction to your Buddy.
- When you are asked to **Twirl**, you will be given a number. Turn completely around that many times. Pass the same instruction to your Buddy.
- If you are asked for **Count**, ask your Buddy and answer with the number he tells you.
You are an **ACROBAT_WITH_BUDDY**

class **ACROBAT_WITH_BUDDY**

inherit **ACROBAT**

redefine

```
twirl, clap, count
```

end

create

```
make
```

end

feature

```
make (p: ACROBAT)
do
  -- Remember `p' being
  -- the buddy, i.e. store
  -- value of `p' in `buddy'
end
```

```
clap (n: INTEGER)
do
  -- Clap `n' times and
  -- forward to buddy.
end
twirl (n: INTEGER)
do
  -- Twirl `n' times and
  -- forward to buddy.
end
```

```
count: INTEGER
do
  -- Ask buddy and return
end
```

```
buddy: ACROBAT
```

end
You are an author

- When you are asked to **Clap**, you will be given a number. Clap your hands that many times. Say “Thank You.” Then take a bow (as dramatically as you like).

- When you are asked to **Twirl**, you will be given a number. Turn completely around that many times. Say “Thank You.” Then take a bow (as dramatically as you like).

- When you are asked for **Count**, announce how many actions you have performed. This is the sum of the numbers you have been given to date.
class
  AUTHOR

inherit
  ACROBAT
  redefine clap, twirl end

feature
  clap (n: INTEGER)
    do
      -- Clap `n' times say thanks and bow.
    end

  twirl (n: INTEGER)
    do
      -- Twirl `n' times say thanks and bow.
    end
end
You are a curmudgeon

- When given any instruction (Twirl or Clap), ignore it, stand up and say (as dramatically as you can) "I REFUSE".
- If you are asked for Count, always answer with 0.
You are a **CURMUDGEON**

class
    **CURMUDGEON**

inherit
    **ACROBAT**
        redefine **clap**, **twirl** end

feature
    **clap** (*n*: INTEGER)
        do
            -- Say “I refuse”.
        end

    **twirl** (*n*: INTEGER)
        do
            -- Say “I refuse”.
        end

end
I am the root object

- I got created by the runtime
  - by executing my creation feature.
I am a \textit{DIRECTOR}

- I got created by the runtime
  - by executing my creation feature.

\begin{verbatim}
class \textit{DIRECTOR} \\
create \\
\textit{prepare\_and\_play} \\
feature \\
\textit{prepare\_and\_play} \\
do \\
\textit{prepare\_and\_play} \\
end
\end{verbatim}
Let’s play
I am the root object

prepare_and_play

    local
        acrobat1, acrobat2, acrobat3 : ACROBAT
        partner1, partner2: ACROBAT_WITH_BUDDY
        author1: AUTHOR
        curmudgeon1: CURMUDGEON
    do
        create acrobat1
        create acrobat2
        create acrobat3
        create partner1.make (acrobat1)
        create partner2.make (partner1)
        create author1
        create curmudgeon1
        author1.clap (4)
        partner1.twirl (2)
        curmudgeon1.clap (7)
        acrobat2.clap (curmudgeon1.count)
        acrobat3.twirl (partner2.count)
        partner1.buddy.clap (partner1.count)
        partner2.clap (2)
    end
# Concepts seen

<table>
<thead>
<tr>
<th>Eiffel</th>
<th>Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes with features</td>
<td>Telling person to behave according to a specification</td>
</tr>
<tr>
<td>Inheritance</td>
<td>All people were some kind of ACROBATS</td>
</tr>
<tr>
<td>Interface</td>
<td>Queries and commands that are applicable</td>
</tr>
<tr>
<td>Objects</td>
<td>People</td>
</tr>
<tr>
<td>Creation</td>
<td>People stand up</td>
</tr>
<tr>
<td>Entities</td>
<td>Names for the people</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>A name can refer to different kind of ACROBATSs</td>
</tr>
<tr>
<td>Dynamic binding</td>
<td>Telling people by name to do the same has different outcome</td>
</tr>
</tbody>
</table>
## Concepts seen

<table>
<thead>
<tr>
<th>Eiffel</th>
<th>Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command call</td>
<td>Telling people to do something</td>
</tr>
<tr>
<td>Query call</td>
<td>Asking a question to a person</td>
</tr>
<tr>
<td>Arguments</td>
<td>E.g. how many times to clap</td>
</tr>
<tr>
<td>Return value</td>
<td>E.g. count in ACROBAT_WITH_BUDDY</td>
</tr>
<tr>
<td>Chains of feature calls</td>
<td>E.g. partner1.buddy.clap (2)</td>
</tr>
</tbody>
</table>