ETH Zürich
Chair of Software Engineering

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 game.

Static view

- A program consists of a set of classes.
- Features are declared in classes. They define operations on objects constructed from the class.
    - Queries answer questions. They have a result type.
    - Commands execute actions. They do not have a result type.
- Terms "class" and "type" used interchangeably for now.

Dynamic view

- At runtime we have a set of objects (instances) constructed from the classes.
- An object has a type that is described in a class.
- Objects interact with each other by calling features on each other.

Static view vs. dynamic view

- Queries (attributes and functions) have a return type. However, when **executing** the query, you get an object.
- Routines have formal arguments of certain types. During the **execution** you pass objects as actual arguments in a routine call.
- During the **execution** local variables declared in a routine are objects. They all have certain types.

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
  - in the local clauses of routines

This is where you declare any objects that only the routine needs and knows.

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
            ...
        end

    attribute_name: T3
            -- Comment
end
```

formal argument types

local variable types

return type

return type

Exercise: Find the classes / objects

Hands-On

```
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

Hands-On

```
feature
	is_occupied (a_location: traffic_place): boolean
			-- Check if `a_location' is occupied by some flat hunter.
		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

Hands-On

```
    -- 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
```

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.
- You select a creation procedure of the root object as the first feature to be executed.

Acrobat game

Hands-On

- We will play a little game now.
- Everyone will be an object.
- There will be different roles.

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
end
```

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

feature
    make (p: ACROBAT)
        do
            -- Remember `p' being the buddy.
        end

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

You are an *ACROBAT_WITH_BUDDY*

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

    count: INTEGER
        do
            -- Ask buddy and return his answer.
        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.

You are an *AUTHOR*

```
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.
- Then sit down again if you were originally sitting.

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.
- I am executing the first feature.

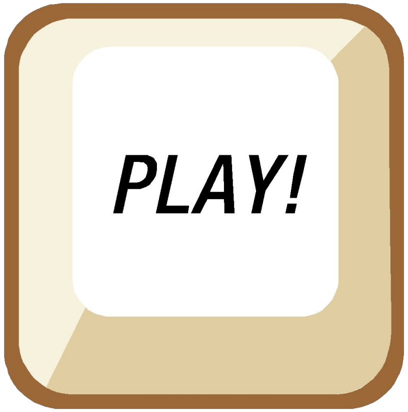
I am a *DIRECTOR*

```
class
    DIRECTOR

create
        prepare_and_play

feature
        prepare_and_play
            do
                -- See following slides.
            end
```

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

Eiffel	Game
Classes with Features	Telling person to behave according to a specification
Objects	People
Interface	What queries What commands
Polymorphism	Telling different people to do the same has different outcomes
Command Call	Telling a person to do something
Query Call	Asking a question to a person
Arguments	E.g. how many times to clap

Concepts seen

Eiffel	Game
Inheritance	All people were some kind of ACROBAT
Creation	Persons need to be born and need to be named
Return value	E.g. count in ACROBAT_WITH_BUDDY
Entities	Names for the people
Chains of feature calls	E.g. partner1.buddy.clap (2)

Advanced Material

The following slides contain advanced material and are optional.

Outline

- Invariants
  - Marriage problems
  - Violating the invariant

Invariants explained in 60 seconds

- Consistency requirements for a class
- Established after object creation
- Hold, when an object is *visible*
  - Entry of a routine
  - Exit of a routine

```
class
    ACCOUNT
feature
    balance: INTEGER
invariant
    balance >= 0
end
```

Public interface of person (without contracts)

```
class
    PERSON
feature
    spouse: PERSON
        -- Spouse of Current.

    marry (a_other: PERSON)
        -- Marry `a_other'.
end
```

```
class
    MARRIAGE
feature
    make
        local
            alice: PERSON
            bob: PERSON
        do
            create alice
            create bob
            bob.marry (alice)
        end
end
```

Write the contracts

Hands-On

```
class PERSON
feature
    spouse: PERSON

    marry (a_other: PERSON)
        require
            ??
        ensure
            ??

invariant
    ??
end
```

A possible solution

```
class PERSON
feature
    spouse: PERSON

    marry (a_other: PERSON)
        require
            a_other /= Void
            a_other.spouse = Void
            spouse = Void
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

invariant
    spouse /= Void implies spouse.spouse = Current
end
```

Implementing *marry*

```
class PERSON
feature
    spouse: PERSON

    marry (a_other: PERSON)
        require
            a_other /= Void
            a_other.spouse = Void
            spouse = Void
        do
            ??
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

invariant
    spouse /= Void implies spouse.spouse = Current
end
```

Implementing *marry* I

Hands-On

```
class PERSON
feature
    spouse: PERSON

    marry (a_other: PERSON)
        require
            a_other /= Void
            a_other.spouse = Void
            spouse = Void
        do
            a_other.spouse := Current
            spouse := a_other
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

invariant
    spouse /= Void implies spouse.spouse = Current
end
```

Compiler Error:

No assigner command

Implementing *marry* II

Hands-On

```
class PERSON
feature
    spouse: PERSON

    marry (a_other: PERSON)
        require
            a_other /= Void
            a_other.spouse = Void
            spouse = Void
        do
            a_other.set_spouse (Current)
            spouse := a_other
        ensure
            spouse = a_other
            a_other.spouse = Current
        end

    set_spouse (a_person: PERSON)
        do
            spouse := a_person
        end

invariant
    spouse /= Void implies spouse.spouse = Current
end
```

```
local
  bob, alice: PERSON
do
  create bob; create alice
  bob.marry (alice)
  bob.set_spouse (Void)
  -- invariant of alice?
end
```

Implementing *marry* III

Hands-On

```
class PERSON
feature
	spouse: PERSON

	marry (a_other: PERSON)
		require
			a_other /= Void
			a_other.spouse = Void
			spouse = Void
		do
			a_other.set_spouse (Current)
			spouse := a_other
		ensure
			spouse = a_other
			a_other.spouse = Current
		end

feature {PERSON}
	set_spouse (a_person: PERSON)
		do
			spouse := a_person
		end

invariant
	spouse /= Void implies spouse.spouse = Current
end
```

Invariant of a_other?

Violated after call to set_spouse

Implementing *marry* : final version

```
class PERSON
feature
	spouse: PERSON

	marry (a_other: PERSON)
		require
			a_other /= Void
			a_other.spouse = Void
			spouse = Void
		do
			spouse := a_other
			a_other.set_spouse (Current)
		ensure
			spouse = a_other
			a_other.spouse = Current
		end

feature {PERSON}
	set_spouse (a_person: PERSON)
		do
			spouse := a_person
		end

invariant
	spouse /= Void implies spouse.spouse = Current
end
```

Ending the marriage

Hands-On

```
class PERSON
feature
    spouse: PERSON

    divorce
        require
            spouse /= Void
        do
            spouse.set_spouse (Void)
            spouse := Void
        ensure
            spouse = Void
            (old spouse).spouse = Void
        end

invariant
    spouse /= Void implies spouse.spouse = Current
end
```

Violating the invariant

- See demo

What we have seen

- Invariant should only depend on Current object

- If invariant depends on other objects
  - Take care who can change state
  - Take care in which order you change state

- Invariant can be temporarily violated
  - You can still call features on Current object
  - Take care calling other objects, they might call back

Although writing invariants is not that easy, they are necessary to do formal proofs. This is also the case for loop invariants (which will come later).