Introduction to Eiffel

Martin Nordio, Christian Estler
ETH Zurich

Distributed Software Engineering Lab
Overview

Part 1: Language Constructs
- Basics: class definition, if then else, expressions, loops and across, creation procedures
- Inheritance: redefinition and multiple inheritance
- Exception Handling
- Once Routines
- Style rules
- Generics
- Information Hiding

Part 2: Contracts
- Preconditions, postconditions and class invariants
- Contracts in inheritance

Part 3: Tuples and Agents
Preparation

Go to:

http://codeboard.io

If you don’t have an account yet, please sign-up and sign-in before doing the exercises.

Once you’re done with a programming exercise, submit your solution.
Part 1: Language constructs

1.1 BASICS
Class declaration: Eiffel vs Java:

```eiffel
class ACCOUNT
end
```

```java
public class Account {
}
```
Constructors

class ACCOUNT
create
  make,
  make_balance
feature
  make
    do ...
  end
  make_balance (i: INTEGER)
    do ...
  end
end

class Account
public
  Account() {...
  public Account (int b) {...
end
Constructors

public class Account {
    public Account() {...}
    public Account (int b) {...}
    public Account (string s) {...}
}

Constructors can have any name; use the create clause to declare a routine as constructor
Overloading

class
   PRINTER

feature
   print_int (a_int: INTEGER)
     do ...
     end

   print_real (a_real: REAL)
     do ...
     end

   print_string (a_str: STRING)
     do ...
     end
end

class Printer {
   public void print(int i) {...}
   public void print(float f) {...}
   public void print(String s) {...}
}

Eiffel does not support overloading!
Creating Objects

class BANK

feature pay_bill
  local b1: ACCOUNT
  do
    create b1.make
  end
end

public class Bank {
  public void payBill() {
    Account b1 = new Account();
  }
}
Creating Objects

class BANK

feature pay_bill

local b1, b2: ACCOUNT

do
create b1.make
create b2.make_balance (2)
end
end

public class Bank {
public void payBill() {
    Account b1 = new Account();
    Account b2 = new Account (2);
}

Create objects using the create keyword; declare the local variables in the local clause
Let’s code…

Go to:

https://codeboard.io/projects/16

**Task**: create a local ACCOUNT object in the constructor of the APPLICATION class

**Task**: modify the creation procedure of ACCOUNT to print a confirmation that an account was created

**Task**: write a new creation procedure in class ACCOUNT that lets you create an account with an initial balance; use it from APPLICATION
Creating Objects: default create

class MAIN

feature root
local
  b1: BANK
do
  create b1
  —— corresponds to
  —— create b1.default_create
  b1.pay_bill
end
end

class BANK

feature pay_bill
  do
    ...
  end
end

All classes inherit from ANY (Object in Java). If no creation procedure is specified, default_create is used (inherited from ANY)
Creating Objects: default create

class BANK
inherit ANY
  redefine default_create
end

create
default_create

feature ...
end

The routine default_create can be redefined
Let’s code…

Go to:

https://codeboard.io/projects/16

Task: override the default_create in class CUSTOMER to print a confirmation message

Task: create a customer object in the APPLICATION class

Task: write a creation procedure for class CUSTOMER that takes, name, first_name and age as arguments; use it to create a customer
Features

class ACCOUNT

feature -- Initialization
  make    do ... end
  make_balance (i: INTEGER)
    do ... end
  make_name (s: STRING)
    do ... end

feature -- Basic operations
  deposit (i: INTEGER) do ... end
  withdraw (i: INTEGER) do ... end
  transfer (b: ACCOUNT) do ... end

feature -- Access
  balance: INTEGER do ... end
end

public class Account {
  public Account() {...
  public Account (int b) {...
  public Account (string s) {...
  public void deposit (int i) {...
  public void withdraw (int i) {...
  public void transfer(Account b) .
  public int balance() {...
}

The feature clause is used to group routines and for information hiding (see 1.8)
Expressions and Conditionals

```
feature
    foo
    do
        if b and (c or d) then
            x := 5
            ...
        end
    end
end

public foo() {
    if (b & (c | d)) {
        x = 5;
        ...
    }
}

public foo() {
    if (b && (c || d)) {
        ...
    }
}
```

```
foo
    do
        if b and then (c or else d) then
            ...
        end
    end
end
```

```
public foo() {
    if (b && (c || d)) {
        ...
    }
}
```
Task: write a condition that only allows to withdraw money if the balance is sufficient; otherwise print an error message; make two withdraws that show the regular and the exceptional behavior
Return and breaks

class B

feature
  foo: INTEGER
    do
      Result := 5
    end
end

epublic class B {
  public int foo() {
    return 5;
  }
}

Eiffel does not support neither breaks, continues nor return
Loops

public class Printer {
    public void print() {
        for(int i=0;i<10;i++) {
            ...
        }
    }
}

print
    local
        i: INTEGER
    do
        from
            i := 1
        until
            i >= 10
    loop
        ...
            i := i + 1
    end
end
Loops: Example 2

print

local
  i: INTEGER

do
  from
    i := 1
  until
    i >= 10
loop
  ...
  i := i + 1
end
end

public class Printer {
  public void print() {
    int i=0;
    while(i<10) {
      i++;
    }
  }
}
Let’s code…

Go to:

https://codeboard.io/projects/24

Task: implement the ‘print_log’ functionality for in the class ACCOUNT; complete class ACCOUNT to log deposits and withdraws
Loops: Traversing a list

```java
public class Printer {
    public void print() {
        for(Element e : list) {
            e.print();
        }
    }
}
```

```
print_using_from
    do
        from list.start
        until list.after
        loop
            list.item.print
            list.forth
        end
    end

print_using_across
    do
        across list as e loop
            e.item.print
        end
    end
```
## Basic Types

<table>
<thead>
<tr>
<th>Eiffel</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>boolean</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>char, byte</td>
</tr>
<tr>
<td>INTEGER</td>
<td>short, int</td>
</tr>
<tr>
<td>INTEGER_64</td>
<td>long</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>STRING</td>
<td>String</td>
</tr>
</tbody>
</table>
Part 1: Language constructs

1.2 INHERITANCE
Deferred Class (abstract class)

Deferred class
ACCOUNT

feature
deposit (a_num: INT)
deferred
end
end

abstract class Account {
    abstract void deposit(int a);
}

A class must be deferred if it has at least one deferred routine. A class can be deferred without any deferred routines.
Simple Inheritance

class ACCOUNT
inherit ANY
end

public class Account
extends Object {
}

Go to:

https://codeboard.io/projects/25

Task: create a deferred class PERSON; move the properties ‘name’ and ‘age’ from class CUSTOMER into the deferred class PERSON; make sure the program behavior did not change
Feature redefinition

class ACCOUNT
inherit ANY
redefine out end

feature
  out: STRING
  do
    Result := “abc”
  end
end

public class Account
  extends Object {
    String toString() {
      return “abc“;
    }
  }

All routines that are redefined must be listed in the inherit clause.
class ACCOUNT
inherit ANY
redefine out end

feature

out: STRING
do
Result := Precursor {ANY}
end
end

class Account extends Object {
String toString() {
    return super();
}
}
Multiple Inheritance

Option 1:
class C
  inherit A
  B rename foo as foo_b end
end

Option 2:
class C
  inherit A
  B undefined foo end
end

Class C will have two features foo and foo_b

foo from B becomes deferred; implemented in C by foo from A
Go to:

https://codeboard.io/projects/26

**Task:** redefine the ‘print_self’ routine in class B to print the correct message

**Task:** redefine the ‘print_self’ routine in class C to print the correct message; what happens when you try to compile?

**Task:** resolve the conflict that was created due to multiple inheritance (hint: there are the 2 options to do that?)
Structure of inherit clause

```
inherit
  A
  rename
    ...
    ...
  undefine
    ...
    ...
  redefine
    ...
    ...
end

B
  rename
    ...
    ...
  undefine
    ...
    ...
  redefine
    ...
    ...
end
```

A `redefine` clause must be structured in the order `rename`, `undefine`, `redefine`. 
Frozen class / frozen routine

frozen class ACCOUNT
inherit ANY
end

class ACCOUNT
feature
  frozen deposit (a_num: INT)
do
  ...
end
end

final class Account
  extends Object {
  }

class Account {
  final void deposit(int a) {
    ...
  }
  }

A frozen class cannot be inherited; a frozen routine cannot be redefined.
Expanded class

expanded class

   MY_INT

end

int, float, double, char