Constants, once routines, and helper functions

these slides contain advanced material and are optional
Basic constants

• Defining constants for basic types in Eiffel

```eiffel
class CONSTANTS
feature
  Pi: REAL = 3.1415926524
  Ok: BOOLEAN = True
  Message: STRING = "abc"
end
```

• Usage of constants

```eiffel
class APPLICATION
inherit CONSTANTS
feature
  feature
    foo do print (Pi) end
end
```
Pitfalls of constants

• Basic strings are not expanded, they are mutable

```ruby
class APPLICATION
  feature
    Message: STRING = "abc"
    foo
      do
        Message.append ("def")
        -- "Message" is now "abcdef"
      end
  end
end
```

• There is a class READABLE_STRING_GENERAL that exposes the read-only interface
Constants in OO programming

• What about user-defined types?

```plaintext
class CONSTANTS
feature
  i: COMPLEX = ?
  Hans: PERSON = ?
  Zurich: MAP = ?
end
```

• Need a way to initialize complex and constant objects
• Other languages use static initializers
• In Eiffel, we use once routines
What are *once* routines?

- Executed when first called
- Result is stored
- In further calls, stored result is returned

```plaintext
foo: INTEGER
  once
  Result := factorial (10)
end

test_foo
  do
    io.put_integer (foo) -- 3628800, calculated
    io.put_integer (foo) -- 3628800, from storage
  end
```
Once for whom?

- Computation is once per **class hierarchy**
- Flag to specify that execution is
  - Once **per thread** (default)
  - Once **per system**
  - Once **per object**

```plaintext
once_per_thread
  once ("THREAD")
  ...
end

also_once_per_thread
  once
  ...
end

once_per_system
  once ("GLOBAL")
  ...
end

once_per_object
  once ("OBJECT")
  ...
end
```
Use of once routines

• Constants for non-basic types

```plaintext
i: COMPLEX
    once create Result.make (0, 1) end
```

• Lazy initialization

```plaintext
settings: SETTINGS
    once create Result.load_from_filesystem end
```

• Initialization procedures

```plaintext
Initialize_graphics_system
    once ... end
```
Shared objects

• Sometimes you need to share data among objects
  – Global settings, caching, operating on shared data structures
  – See singleton pattern

• Other languages use static variables for this

• In Eiffel, this can be achieved with once routines
  – A once routine returning a reference always returns the same reference
  – You can create a SHARED_X class to share an object and inherit from it when you need access to the object
Is it guaranteed that there will only be one instance of X?
• What is the result of the following function calls?

```plaintext
double (i: INTEGER): INTEGER
  require
    i > 0
  once
    Result := i * 2
  ensure
    Result = i * 2
end
test_double
  do
    print (double (3)) -- ?
    print (double (7)) -- ?
    print (double (-3)) -- ?
  end
```

What about now?
ECMA Eiffel call rule

8.23.26 Semantics: General Call Semantics

The effect of an Object_call of feature sf is, in the absence of any exception, the effect of the following sequence of steps:

1. Determine the target object O through the applicable definition.
2. Attach Current to O.
3. Determine the dynamic feature df of the call through the applicable definition.
4. For every actual argument a, if any, in the order listed: obtain the value v of a; then if the type of a converts to the type of the corresponding formal in sf, replace v by the result of the applicable conversion. Let arg_values be the resulting sequence of all such v.
5. Attach every formal argument of df to the corresponding element of arg_values by applying the Reattachment Semantics rule.
6. If the call is qualified and class invariant monitoring is on, evaluate the class invariant of O’s base type on O.
7. If precondition monitoring is on, evaluate the precondition of df.
8. If df is a once routine, apply the Once Routine Execution Semantics to O and df.
9. If the call is qualified and class invariant monitoring is on, evaluate the class invariant of O’s base type on O.
10. If postcondition monitoring is on, evaluate the postcondition of df.
Pitfalls of once routines II

• What is the result of the following function calls?

```plaintext
recursive (x: INTEGER): INTEGER

Result := 3
if x > 1 then
    Result := Result + recursive (x - 1)
end
end

test_recursive
do
    print (recursive (3)) -- ?
    print (recursive (7)) -- ?
    print (recursive (73)) -- ?
end
```

What about now?
8.23.22 Semantics: Once Routine Execution Semantics

The effect of executing a once routine $df$ on a target object $O$ is:

1. If the call is fresh: that of a non-once call made of the same elements, as determined by Non-Once Routine Execution Semantics.

2. If the call is not fresh and the last execution of $f$ on the latest applicable target triggered an exception: to trigger again an identical exception. The remaining cases do not then apply.

3. If the call is not fresh and $df$ is a procedure: no further effect.

4. If the call is not fresh and $df$ is a function: to attach the local variable Result to the latest applicable result of the call.
• Do you see a problem here?

```plaintext
array: ARRAY [INTEGER]

pointer: POINTER
    once
        create array.make_filled (0, 1, 10)
        Result := $array
    end
```

• The $-operator can be used to get the memory address and interface with external C code
Once routines summary

• Once routines can be used
  – To cache complex computations
  – To create constants objects
  – To share data
  – To implement the singleton pattern

• Once routines should
  – Not have arguments
  – Not have complex postconditions
  – Not be recursive
  – Not use return type POINTER
Helper functions

• Helper functions are used for
  – Functionality that is used by different clients
  – Functionality that is not tied to an object

• Example: mathematical computations

• Other languages use **static functions**

• In Eiffel, two variants
  – Via inheritance
  – Via expanded classes
class MATH
feature {NONE}
   log_2 (v: REAL): REAL
   do
      Result := log (v) / log ({REAL} 2.0)
   end
end

class APPLICATION
inherit {NONE} MATH
feature
   foo do print (log_2 (1.2)) end
end
expanded class MATH
feature
    log_2 (v: REAL): REAL
do
        Result := log (v) / log (REAL 2.0)
    end
end

class APPLICATION
feature
    foo
        local
            m: MATH
do
                print (m.log_2 (1.2))
        end
end