# Lecture 2: Emerald Objects and Types

## An OO Language for Distributed Applications

Oleks Shturmov

<olekss@uio.no> / <oleks@oleks.info>

University of Oslo

January 31, 2019

Lecture slides derived from the 2018 lectures slides byEric Jul:

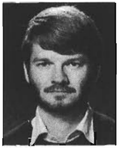
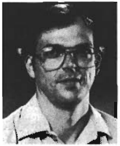
https://www.uio.no/studier/emner/matnat/ifi/INF5510/v18/lectures-v18/f2/

The source code for these slides is maintained here:

https://github.com/emerald/in5570v19/tree/master/lectures/2

# The People Behind Emerald

- Work done at the University of Washington in the early to mid-1980s
- See HOPL paper for more details on the history of Emerald[1]

| | Runtime and Mobility | Language Design |
|---|---|---|
| **PhD Students** |  [2]<br>Eric Jul |  [3]<br>Norm Hutchinson |
| **Faculty** |  [4]<br>Hank Levy |  [5]<br>Andrew Black |

[1] `https://www.uio.no/studier/emner/matnat/ifi/INF5510/v16/material/emerald-hopl.pdf`

[2] Today, Professor at the University of Oslo

[3] Today, Associate Professor at the University of British Columbia

[4] Today, Chair in Computer Science & Engineering at the University of Washington

[5] Today, Professor at Portland State University

# Principle: Everything Is an Object[6]

- Basic types (integers, booleans, strings, etc.) are objects
- Classes are objects (in Emerald, mere syntactic sugar)
- Types are objects (of a special built-in type, `Signature`)
- Language constructs however, are **not** objects
  (e.g., declarations, if-statements, for-loops, programs)

**Alternative interpretation:**
Every valid expression evaluates to an object

Consequently:

- Type names and declarations are expressions
- Class names and declarations are expressions

[6] Well, almost everything

# Some Non-Objects: Trivial Emerald Programs

- An Emerald program is a list of constant declarations
- Each bearing a name, an expression, and optionally, a type
- The following (trivial) programs produce no output

With type inference:

```
const a <- 4
const b <- true
const c <- 'x'
const d <- "Hello, World!\n"
```

With type annotations:

```
const a : Integer <- 4
const b : Boolean <- true
const c : Character <- 'x'
const d : String <- "Hello, World!\n"
```

# Some Hello-World Objects (1/3)

Time for some output!

```
const main <- object main
  initially
    stdout.putstring["Hello, World!\n"]
  end initially
end main
```

To compile and run:

```
$ ec hello.m    # Assuming you call the above file hello.m
$ emx hello.x   # Assuming ec went well, you'll get a hello.x
```

- The use of the name(s) "`main`" is purely conventional
- Emerald merely evaluates the declarations of a program (and their expressions) in order, from top to bottom
- An `initially`-block can contain a list of declarations and statements, and end in fault-handling code; more on fault-handling in subsequent lectures

# Some Hello-World Objects (2/3)

The following is also a valid Emerald program:

```
const alice <- object female
  initially
    stdout.putstring["Hello, I am Alice!\n"]
  end initially
end female

const bob <- object male
  initially
    stdout.putstring["Hello, I am Bob!\n"]
  end initially
end male
```

Compile and run:

```
$ ec hello.m
$ emx hello.x
Hello, I am Alice!
Hello, I am Bob!
```

# Some Hello-World Objects (3/3)

So is this:

```
const main <- object main
  initially
    stdout.putstring["Hello, World!\n"]
    stdout.putstring["Hello?\n"]
    stdout.putstring["Is there anyone out there?\n"]
  end initially
end main
```

Compile and run:

```
$ ec hello.m
$ emx hello.x
Hello, World!
Hello?
Is there anyone out there?
```

# A More Elaborate Object (1/3)

```
% A random number generator
% Derived from https://stackoverflow.com/a/3062783/5801152
const rand <- object rand
  var seed : Integer <- 123456789
  const a <- 1103515245
  const c <- 12345
  const m <- 2147483648
  op next -> [retval : Integer]
    seed <- (a * seed + c) # m
    retval <- seed
  end next
  initially
    stdout.putstring[rand.next.asstring || "\n"]
    stdout.putstring[rand.next.asstring || "\n"]
    stdout.putstring[rand.next.asstring || "\n"]
  end initially
end rand
```

- Many built-in types define an `asstring` method
- Append a line break (`|| "\n"`) to flush stdout

# A More Elaborate Object (2/3)

If we **export** the operation, we can use it outside:

```
const rand <- object rand
  var seed : Integer <- 123456789
  const a <- 1103515245
  const c <- 12345
  const m <- 2147483648
  export op next -> [retval : Integer]    % See here
    seed <- (a * seed + c) # m
    retval <- seed
  end next
end rand                                   % Here
                                           %
const main <- object main                  %
  initially
    stdout.putstring[rand.next.asstring || "\n"]
    stdout.putstring[rand.next.asstring || "\n"]
    stdout.putstring[rand.next.asstring || "\n"]
  end initially
end main                                   % And here
```

# A More Elaborate Object (3/3)

Now, with a bit more **class**:

```
const rand <- class rand                          % See here
  var seed : Integer <- 123456789
  const a <- 1103515245
  const c <- 12345
  const m <- 2147483648
  export op next -> [retval : Integer]
    seed <- (a * seed + c) # m
    retval <- seed
  end next
end rand

const main <- object main
  initially
    const r <- rand.create                        % And here
    stdout.putstring[r.next.asstring || "\n"]
    stdout.putstring[r.next.asstring || "\n"]
    stdout.putstring[r.next.asstring || "\n"]
  end initially
end main
```

# What Is A Class (in Emerald) Anyway?

A class declares (1) an object type, and
(2) a means to create instances of that type

Consequently, an Emerald class `C` is **syntactic sugar** for an Emerald object exporting the following methods:

```
getSignature -> Signature
create [p1, p2, ...] -> C
```

where

- **Signature** is a built-in type of all type objects
- The value (object) returned by `create` will “conform to” the signature returned by `getSignature`

More on type objects and conformity after an example

# A More Elaborate (Class) Object

The class from before, without syntactic sugar:

```
const rand <- object RandCreator
  const RandType <- typeobject RandType
    op next -> [seed : Integer]
  end RandType
  export function getSignature -> [r : Signature]
    r <- RandType
  end getSignature
  export op create -> [r : RandType]
    r <- object Rand
      var seed : Integer <- 123456789
      const a <- 1103515245
      const c <- 12345
      const m <- 2147483648
      export operation next[] -> [r : Integer]
        seed <- (a * seed + c) # m
        r <- seed
      end next
    end Rand
  end create
end RandCreator
```

# Type Objects

- Special objects of the built-in type **Signature**
- Constructed using the **`typeobject`** keyword
- Every Emerald object has an associated type object
- Use the **`typeof`** operator to fetch the type of an object
- Use the `getSignature` method to fetch the type of a class
- Compare type objects with the `*>` (conforms to) operator

# Type Objects: Examples (1/3)

```
const RandType <- typeobject RandType
  op next -> [seed : Integer]
end RandType

const RandObject <- object RandObject
  export op next -> [retval : Integer]
    retval <- 42
  end next
end RandObject

const main <- object main
  initially
    const r : Boolean <- (typeof RandObject) *> RandType
    stdout.putstring[r.asstring || "\n"]
  end initially
end main
```

# Type Objects: Examples (2/3)

```
const RandType <- typeobject RandType
  op next -> [seed : Integer]
end RandType

const RandClass <- class RandClass
  export op next -> [retval : Integer]
    retval <- 43
  end next
end RandClass

const main <- object main
  initially
    const r : Boolean <- RandClass.getSignature *> RandType
    stdout.putstring[r.asstring || "\n"]
  end initially
end main
```

# Type Objects: Examples (3/3)

```
const RandClass <- class RandClass
  export op next -> [retval : Integer]
    retval <- 42
  end next
end RandClass

const RandObject <- object RandObject
  export op next -> [retval : Integer]
    retval <- 43
  end next
end RandObject

const main <- object main
  initially
    const r <- (typeof RandObject) *> RandClass.getSignature
    stdout.putstring[r.asstring || "\n"]
  end initially
end main
```

# Type Conformity Is Everywhere

When using an object where a particular type is expected,
the object type must **conform to** the expected type

- Conformity is checked at compile time; no runtime costs!

```
const RandType <- typeobject RandType
  op next -> [seed : Integer]
end RandType

const RandClass <- class RandClass
  export op next -> [retval : Integer]
    retval <- 43
  end next
end RandClass

const main <- object main
  initially
    const r : RandType <- RandClass.create
  end initially
end main
```

# Type Conformity: Definition

A type $S$ conforms to a type $T$ iff for each operation

$$o\,[\,p_1^T, p_2^T, \ldots p_n^T\,] \text{->} [\,r_1^T, r_2^T, \ldots, r_m^T\,] \quad \text{in } T$$

there is a corresponding operation[7]

$$o\,[\,p_1^S, p_2^S, \ldots p_n^S\,] \text{->} [\,r_1^S, r_2^S, \ldots, r_m^S\,] \quad \text{in } S$$

where

1. $p_i^T$ conforms to $p_i^S$, for all $i \in 1, 2, \ldots n$, and
2. $r_i^S$ conforms to $r_i^T$, for all $i \in 1, 2, \ldots n$

NB! Formal parameters conform one way, while results the other.

[7] Having the same name, number of formal parameters, and results.

# Some Special Cases: Any and None

**Any** and **None** are special built-in types
**None** is the type of the keyword (expression) **nil**

They have the following interesting properties:

1. Everything conforms to **Any**
2. **None** conforms to anything
3. Nothing conforms to **None**

Notably, **nil** conforms to **Any**, and anything at all

# Type Conformity: Example (1/3)

Consider the types of some waste bins, which we can pick at:

```
typeobject AnyBin
  op Pick -> [Any]
end AnyBin
```

```
typeobject PaperBin
  op Pick -> [Paper]
end PaperBin
```

Now, imagine being a waste picker[8]:

- If you accept any trash (i.e., `AnyBins`), then you are also willing accept specialized trash (e.g., `PaperBins`).
- If you only accept specialized trash (e.g., `PaperBins`), then you are not willing to accept any trash (i.e., `AnyBins`).

Hence, `PaperBin` conforms to `AnyBin`, but not vice-versa.

[8] Waste-picking is an admirable profession for an autonomous drone

# Type Conformity: Example (2/3)

Now, instead consider bins we can throw something into:

```
typeobject AnyBin
  op Throw[Any]
end AnyBin
```

```
typeobject PaperBin
  op Throw[Paper]
end PaperBin
```

- A bin that accepts anything (i.e., `AnyBin`), can also act as a specialized bin (e.g., `PaperBin`).
- A specialized bin however (e.g., `PaperBin`), cannot act as a bin for anything (i.e., `AnyBin`).

Hence, `AnyBin` conforms to `PaperBin`, but not vice-versa.

# Type Conformity: Example (3/3)

Combining the two examples however, yields non-conforming bins:

```
typeobject AnyBin
  op Pick -> [Any]
  op Throw[Any]
end AnyBin
```

```
typeobject PaperBin
  op Pick -> [Paper]
  op Throw[Paper]
end PaperBin
```

This makes sense:

- You cannot throw anything into a `PaperBin`, so it cannot act as an `AnyBin`.
- You can throw anything into an `AnyBin`, so it cannot act as a `PaperBin`, from which you only ever want to pick `Paper`.

Hence, neither `AnyBin` conforms to `PaperBin`, nor vice-versa.

# Further Reading


- Raj, Tempero, Levy, Black, Hutchinson, and Jul (1991), Emerald: A general-purpose programming language. Software: Practice and Experience, Vol. 21, No. 1.

  https://www.uio.no/studier/emner/matnat/ifi/INF5510/v15/pensum/SPE-paper-1991.pdf

- Raj, Tempero, Levy, Black, Hutchinson, and Jul (1991), Technical Report: The Emerald Programming Language

  https://www.uio.no/studier/emner/matnat/ifi/INF5510/v15/pensum/Report.pdf