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
```