Extreme | True | Pure Objects

This presentation aims to cover the approach which is based on the statement – everything is an object. That is why term object is the essential one. Set of operations on objects is fixed; object life cycle has just 3 stages; object immutability is defined. Term attribute is introduced as it is a key part of any object internal structure. Relations between objects are defined. Two atomic objects are introduced as an introduction to constant objects concept. What is class type in the object world? Brief introduction into active objects concept is given.

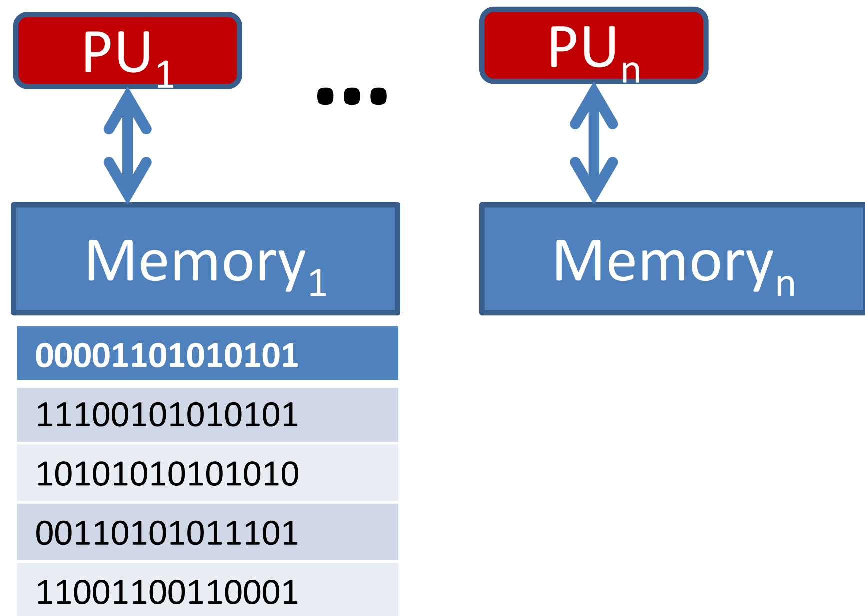
Alexey Kanatov,
alexey.v.kanatov@gmail.com
LinkedIn
May 2023

Content

- Introduction
- Key definitions, object structure
- Object life cycle
- Object operations
- Object type
- 2 atomic objects -> constant objects
- Active objects
- Summary

Introduction to … object thinking

- What kind of computers do we have? Processor+memory model
- Memory structure – set of cells with an address per cell. Cell is an ordered set of bits – here comes binary digits 0 and 1
  - Ordered set of QWORDS, DWORDS, WORDS, BYTES, bits
- Object – region of the computer memory. Its logical structure in most cases is different from the memory structure
  - Set of named attributes
  - Object is flat!



- (name:”A”, age:66) === (age:66, name:”A”)
- Objects are easily mapped into memory

name: “Alexey V. Kanatov”
age: () **do return** 55 **end**
address: “Moscow”
company: **ref to** Huawei.Object

Only … objects

- Everything is an object (both code and data). There is nothing except objects!

- Object: (a-la struct {…})
  - name or address (this-self-Current)
  - region(s) of the computer memory
  - set of attributes. What is an attribute?
  - invariant: set of predicates which guard object consistency

- Attribute:
  - Tag: var or const (deep const)
  - Name (number-offset): String // must be unique in the object
  - Value: Any // value is an object too !
  - [Type is defined by the value]

Object structure - example

- Here comes ... an object

```
// Notation 1
object Kanatov
  var name: val “Alexey V. Kanatov”  // attribute #1
  const age: val ():Integer do return 55 end // attribute #2
  var spouse: ? // attribute #3: value not defined yet
  var company: ref Huawei              // attribute #4
end
// Notation 2
object Kanatov {
  var name: val “Alexey V. Kanatov”  // attribute #1
  const age: val ():Int {return 55}  // attribute #2
  var spouse: ? // attribute #3: value not defined yet
  var
}
```

Aha: attribute value can be of **val** (value) or **ref** (reference) nature, and value can be undefined

Object structure - implication

Serialization and persistency => distribution

```
object Kanatov
  var name: val "Alexey V. Kanatov"
  const age: val ():Integer do return 55 end
  var spouse: ?
  var company: ref Huawei
end
```

- Object type descriptor + object raw data (binary)
- XML
- JSON
- What ever …

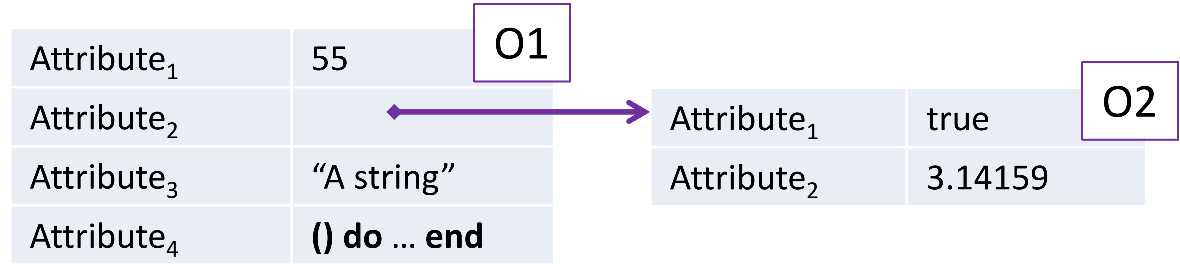
Object life cycle

1. Object creation
   - its attributes initialization
   - who (what) can do it ?
     - runtime creates root object and calls **main** for example
     - 3rd party code
     - program code (using **object** … **end** or **new** instruction)
2. Life time loop:
   - attributes’ activations:
     - access (read) data attributes (fields)
     - call routines (methods + function type attributes)
   - On object operations
3. Destruction
   - automatic (GC, on scope left - whatever)
   - manual (mark or free)

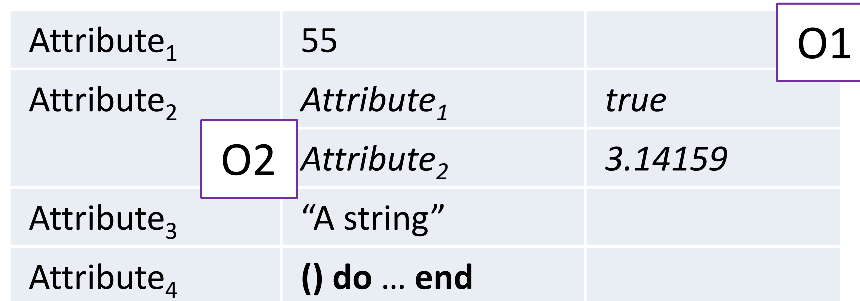
Note: No write into object attributes is the basis for object consistency

Object relations

- Compile time and runtime relations
  - There are no objects at compile time !!!
  - It is all about runtime !!!
- Refers: O1 refers to O2

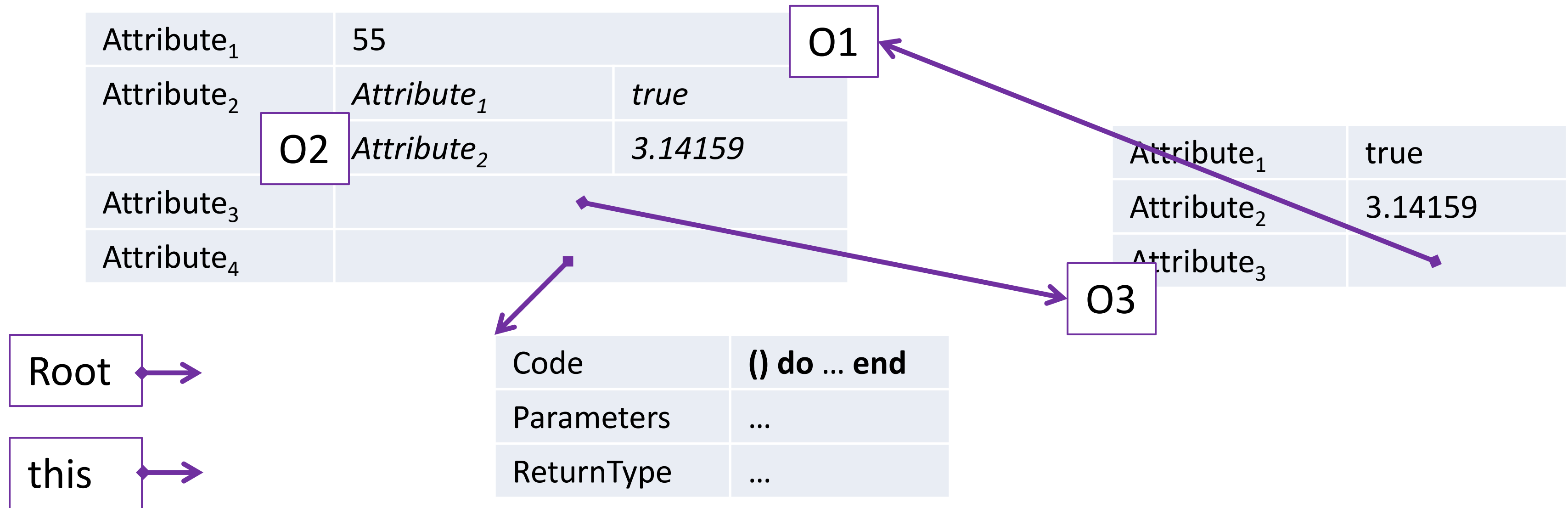


- Includes: O1 includes (contains) O2; (no cycles!)



Only 2 relations -> Only 2 kinds of attributes – ref and val

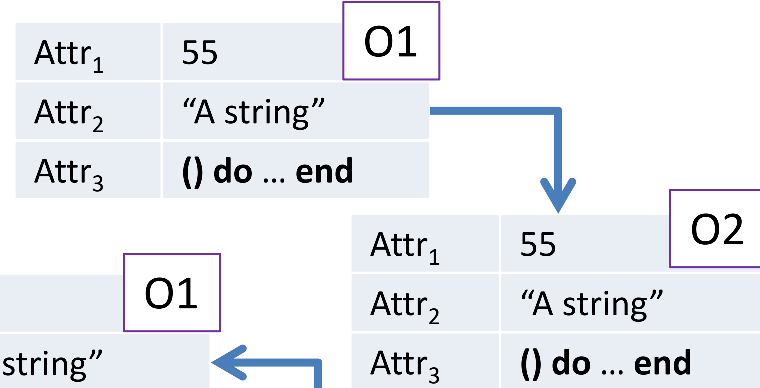
Object world - program

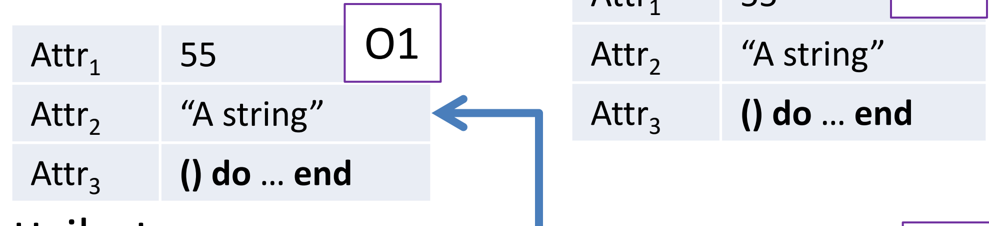


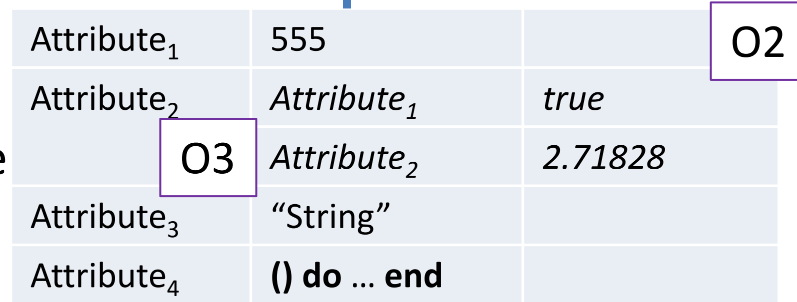
Map of the world ☺

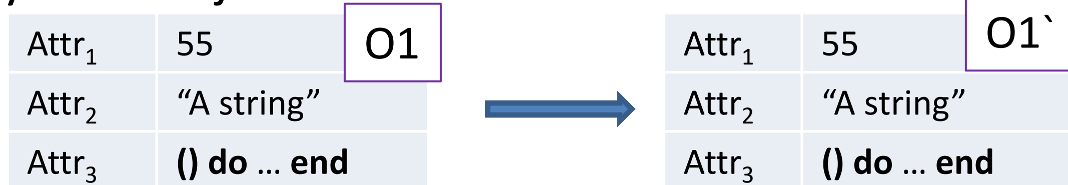
Object operations

- Compare: How to compare objects?
  - Attribute by attribute compare
  - Or programmer-defined
- Copy: (assignment-like)
  - Same type: 'memmove'
  - Different type
    - Type are related: attribute by attribute copy
    - Types are unrelated: conversion – some action(s) to be performed
- Clone: create a copy of an object – shallow or deep







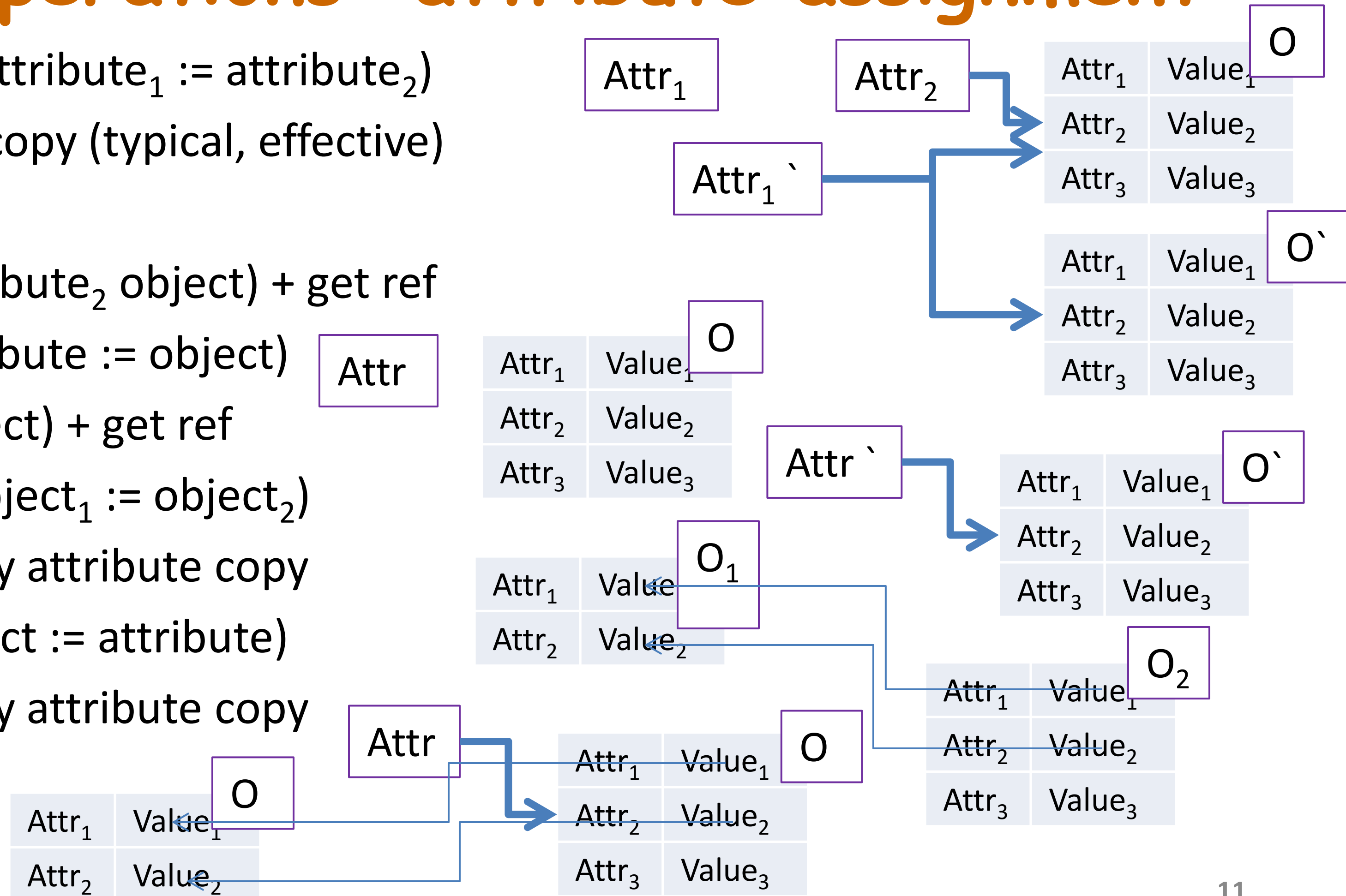


Object operations: attribute assignment

1. <u>ref1 := ref2</u> (attribute$_1$:= attribute$_2$)
   - reference copy (typical, effective)

   or

   - clone (attribute$_2$ object) + get ref
2. <u>ref := val</u> (attribute := object)
   - clone (object) + get ref
3. <u>val1 := val2</u> (object$_1$:= object$_2$)
   - attribute by attribute copy
4. <u>val := ref</u> (object := attribute)
   - attribute by attribute copy



Object type

- Group of objects form a type. Group may be defined differently
  - 1 | 4 | 6 | 7 // Just name all objects of the same kind
  - 1 | "a string" | true | (1,2,3,4) | () **do** … **end** /* All objects of different kinds */
  - minInteger **..** maxInteger // Range
  - true | false // Just name all objects of the same kind
  - {0b0 | 0b1} 8 // Regular expression
  - Boolean | String // Union
  - **type** Name constructors* attributes* **end** // Text form
- If all objects in a type of the same kind (have common subset of attributes) we know what we can do with these objects (set of operations is known) otherwise type check is required
- What is the type of object **1** ? Integer.1 or Cardinal.1 or LongInt.1?

Object type - implications

- Group of objects form a type …
- Options
  - 1 object ⬄ 1 type (module, singleton per program, per hierarchy, per routine (function)) IO.put (“string”)
  - N objects ⬄ 1 type (range, explicit, … ?)
  - Unlimited # of objects ⬄ 1 type (**new**…)

Note on notation …

```
object x
  const procedure : () do … end
  const function : ():Type do … end
end

type X
  const procedure = () do … end
  const function = ():Type do … end
end
```

```
object x
  procedure() do … end
  function():Type do … end
end

type X
  procedure() do … end
  function():Type do … end
end
```

- Less to type
- The way we used to

2 atomic objects: starting from 0 and | …

```
type Bit const 0b0, 0b1 end
  &, and (other: Bit): Bit=>if this=0b0 do 0b0 elseif other=0b0 do 0b0 else 0b1
  |, or (other: Bit): Bit=> if this=0b1 do 0b1 elseif other = 0b1 do 0b1 else 0b0
  ^, xor (other: Bit): Bit=> if this = other do 0b0 else 0b1
  ~, not (): Bit => if this = 0b0 do 0b1 else 0b0
  + (other: Bit): Bit do // Definition of addition
    if this = 0b0 do return other
    elseif other = 0b1 do raise String."Bit overflow"
    else return 0b1
    end // if
  end // +
  - (other: Bit): Bit do // Definition of subtraction
    if this = other do return 0b0
    elseif this = 0b1 do return 0b1
    else raise String."Bit underflow"
    end // if
  end // -
end // Bit
```

0b0 and 0b1 are names of objects of type Bit.

All attributes are explicitly defined

Constant objects

Every type in the text form may define its constant objects

```
type Boolean
  const false(Cardinal.0), true(Cardinal.1) end
     // false is new Boolean(Cardinal.0)
  …
  Boolean (v: Cardinal)
  require v in Cardinal.0 ..
    data is if v = zero do 0b0 else 0b1
  end
  data: Bit <Platform.booleanBitsCount>
invariant
  this and this = this // idempotence of 'and'
  this or this = this // idempotence of 'or'
  this and not this = false // complementation
  this or not this = true // complementation
end // Boolean
```

Number of objects of the type is not less than number of its constant objects

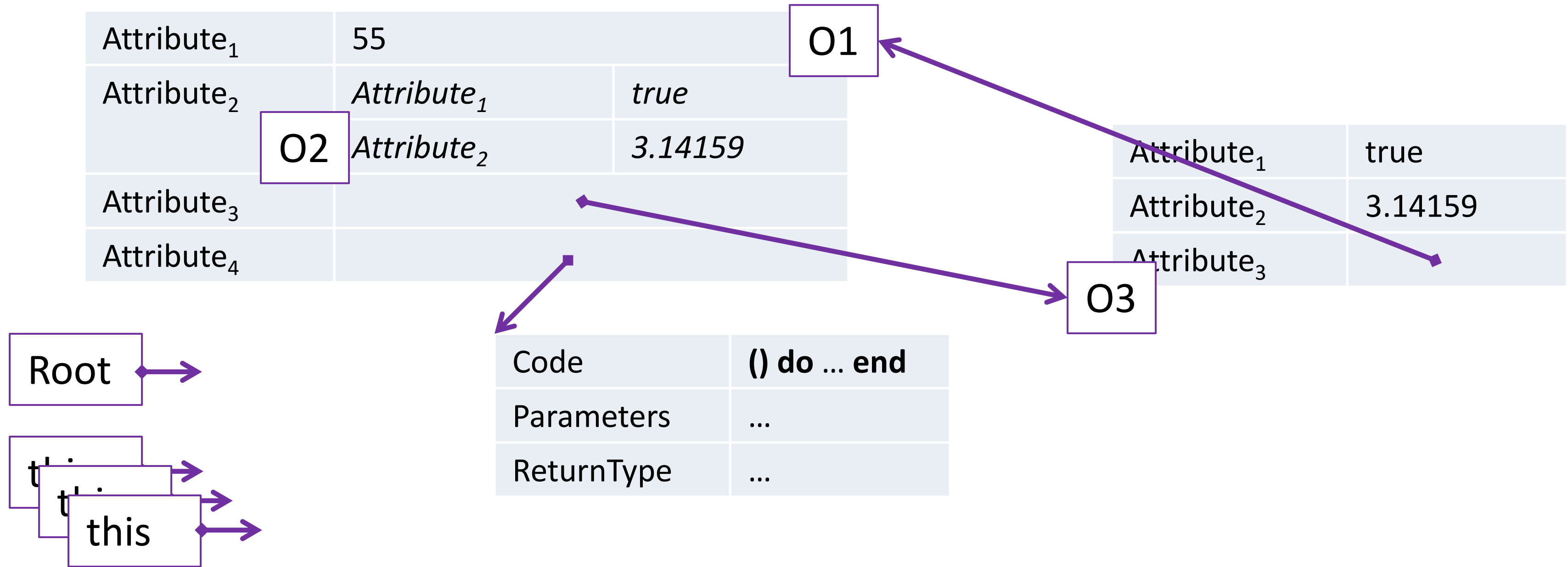
Passive objects

```
object o1
  const proc1 : () do … end
  var proc2 : () do … end
  const proc3 : () do … end
  var func : ():Type do … end
end
o1.proc1 () // synchronous call
…
o1.proc2 () // synchronous call
…
o1.proc3 () // synchronous call
…
o1.proc1 () // synchronous call
…
result = o1.func() // synchronous call
```

Active objects

```
active object o1
  const proc1 : () do … end
  var proc2 : () do … end
  const proc3 : () do … end
  var func : ():Type do … end
end
o1.proc1 () // async call
…
o1.proc2 () // async call
…
o1.proc3 () // async call
…
o1.proc1 () // async call + queue for proc1
…
result = o1.func() // synchronous call
```

Real world



Map of the active real world ☺

Summary (I): raw facts

There are only objects and nothing except objects

Objects have attributes and life cycle

Objects have relations with other objects

Every object knows its type. Type defines group of objects

Bit.1b and Bit.0b are two cornerstone objects. Every type in the text form may define its constant objects

Objects can be active and passive

Summary (II): outcomes

- Object thinking maps to existing hardware architectures well
- It is simple – just one concept, simple life cycle
- It guarantees type safety and protected encapsulation
- It covers concurrent programming
- It covers distributed programming
- It has no hidden magic – all is explicitly defined
- Convergence of imperative and functional paradigms

Summary (III): implications

static

public

global data

null

Thank you !
Q&A