Introduction to Bluebottle

What is Bluebottle?
Bluebottle is a new operating system (it is sometimes referred to as "Aos" or "Active Oberon"). The kernel of Bluebottle is called Aos (Active Object System). It is the runtime system of Active Objects. The entire system is based on Active Objects. Active Objects are programmed in Active Oberon, an extension of the well known Oberon programming language.

Object Oriented Programming (OOP) in 15 Minutes…
What are Objects?
Objects are like records that can contain *methods* (procedures) that manipulate the *state* (fields) of the *object* (record). To be more precise: an object is like a POINTER TO RECORD. That means an object needs to be created (also called *instantiated*) before it can be accessed.

Example:
```
TYPE
   MyFirstObject = OBJECT (* ← this creates an object type called MyFirstObject*)
   VAR field : LONGINT;
      PROCEDURE ManipulateState;
      BEGIN
         INC(field)
      END ManipulateState;
   END MyFirstObject;
```
This is the type definition of a simple but complete object type. *Object types* are called *classes* in Java, C++ or C#.

How to create an instance of this object type?
An object instance is created the same way as an instance of POINTER TO RECORD is created: with the **NEW** keyword.
Example:
```
VAR x : MyFirstObject;
BEGIN
   NEW(x); (* ← this instantiates the object *)
```

How to access fields and methods of an object ?
Fields of objects are accessed exactly the same way as fields of records.
Examples:
```
   x.field := 3;
   INC(x.field);
```
Methods are accessed analogously.
```
   x.ManipulateState;
```

If there are two instances of a MyFirstObject object type (`a` and `b`) then `a.ManipulateState` accesses `a.field`, while `b.ManipulateState` accesses `b.field`.

Is this all one needs to know about OOP?
No. The most important feature of OOP is the possibility to create sub-types (*sub-classes*) of object types. Creating a *sub-type/sub-class* is analogous to creating a type extension of a given record. The sub-type contains all the fields and methods of the *super-type/super-class*. Objects of a sub-type can be assigned to variables of super-types. It is therefore possible to access fields and to call methods of object types that are "not known", as long as these object types are sub-types of known types.

How can a Sub-type/Sub-class *(Type-Extension)* be created?
Example:
```
TYPE
   MySubObj = OBJECT(MyFirstObject) (* ← MyFirstObject is the super-class *)
   VAR additionalField : LONGINT; (* ← this is an additional field *)
      PROCEDURE AdditionalMethod; (* ← this is an additional method *)
      BEGIN
         INC(additionalField);
         DEC(field)     (* ← access to fields and methods of the super-type are possible *)
      END AdditionalMethod;

      PROCEDURE ManipulateState; (* ← this is a refined method *)
      BEGIN
         field := field + additionalField
      END ManipulateState;
   END MySubObj;
```

In this example, the sub-type replaces the ManipulateState method of the super-type.
```
VAR x, a : MyFirstObject;
   b : MySubObj;
BEGIN
   NEW(a); NEW(b);
   x := a; x.ManipulateState;
   x := b; x.ManipulateState;
```
Both a and b can be assigned to x.
`x.ManipulateState` calls a different method depending on the *actual* type of x.

What makes the Objects "Active" ? (And therefore better ;-))
Active Objects contain a process that is associated with them. When an active object is created, a new process is created at the same time. The new process runs in the "Body" of an Active Object.
Example of an Active Object:
```
TYPE
   MyActiveObjectClass = OBJECT
   VAR field : LONGINT;
      PROCEDURE Method;
      BEGIN
      END Method;
   BEGIN {ACTIVE} (* ← This is called the body of an Active Object *)
      LOOP
         [...]
         Method;
         [...]
      END
   END MyActiveObjectClass;
```

Since an Active Object starts running as soon as it is created there needs to be a way to give it an initial state. How is this accomplished ?
This is done with a constructor. A constructor is called right after the object is created but before it starts running. The constructor is defined with the marker "&" in front of its name.
Example of an Active Object with constructor:
```
TYPE
   MyActiveObjectClass = OBJECT
   VAR field : LONGINT;
      PROCEDURE &Init(param : LONGINT); (* ← constructor *)
      BEGIN
      END Init;

      PROCEDURE Method;
      BEGIN
      END Method;
   BEGIN {ACTIVE} (* ← body of the Active Object *)
      LOOP
         [...]
         Method;
         [...]
      END
   END MyActiveObjectClass;
```

This Active Object has a constructor called Init. Init can also be called as a normal method of the object. If called, it does not restart the object's process. Non-Active Objects can also have a constructor, defined exactly the same way.

How do the parameters get into the constructor ?
NEW(x *{, parameter}*);

How is an Active Object stopped ?
As soon as the process leaves the body *(calling a procedure or method does not leave the body)*, it is terminated. A terminated Active Object is like a normal object. It stays in memory until it is collected by the garbage collector.

What is Oberon in Bluebottle ?
Oberon is a single process program that runs on Bluebottle, just as Oberon can run on Windows or Linux. Actually it is just a special "Active Object". Currently Oberon is used as development environment for Bluebottle (just like Visual Studio or Delphi on Windows).

How can a Bluebottle command/program be called from Oberon ?
There is a module called Aos that can communicate with the Aos Kernel, from inside Oberon. To call a Bluebottle program, invoke
Aos.Call *AosModuleName.AosCommand Parameters* ~
The module Aos tells the Aos kernel to search for the specified module *(AosModuleName)* and command *(AosCommand)*. If found, it passes the parameters *(Parameters)* to the message.

How can I see the result of my Bluebottle command ?
Does it appear in the System Log? No.
The result can not directly appear in Oberon since this would mean an asynchronous call into the single-threaded Oberon object. There is a special log program (inside Oberon), called KernelLog that handles this problem by synchronously polling a log-queue. The kernel log can be started with **System.OpenKernelLog** (→ in the System popup).

How to write to the Kernel Log?
Import AosOut and call AosOut.String, AosOut.Int etc.
Example:
```
AosOut.String("Result is "); AosOut.Int(result, 0); AosOut.Ln;
```

Could I open a new Window for the Output?
Yes, sure. Even with fancy shapes and transparency. But this is not the topic of this document ;-) → Find out … the source of all the example-programs is available.

How can I read the command Parameters?
A Bluebottle command has the signature: *Name(par:PTR):PTR.* Aos.Call passes an AosCommands.Parameters as par. Use a TypeGuard (as in normal Oberon) to convert PTR (a generic object instance) to AosCommands.Parameters. AosCommands.Parameters has a field "str" that is a POINTER TO ARRAY OF CHAR that contains the parameter-string.
Example:
```
PROCEDURE Start*(par : PTR) : PTR;
VAR s : AosCommands.Parameters;
BEGIN
   s := par(AosCommands.Parameters);
   AosOut.String("Echo : "); AosOut.String(s.str^)
   RETURN NIL
END Start;
```

What can I do to make the String-Parsing easier ?
It is possible to open a stream on the string and use the stream's methods of formatted reading.

What is a Stream?
In Bluebottle, the streams are defined in AosIO. There are two different kinds of streams, the reader streams and the writer streams. Streams can be opened on anything like device-drivers, protocols, strings or files as long as a special Receiver respectively Sender procedure is available on which the stream can be opened. A TCP connection as an example offers a **Send** and a **Receive** method, on which a respective stream can be opened.
The stream offers methods to read or write characters, integers represented as decimal ASCII characters, tokens and many more.

Example:
```
VAR sr : AosIO.StringReader;
[...]
(* this opens a stream on a string... *)
AosIO.OpenStringReader(sr, LEN(s.str^)); sr.Set(s.str^);
```

Now you can read tokens, integers, strings etc. from the stream.
Example:
```
   sr.Token(serverAdr); sr.SkipWhitespace(); sr.Int(number, FALSE);
```
This reads a token (number of characters without a white-space), then skips all white-spaces and reads an integer.
Have a look at the definition of AosIO for more detailed information about streams.
Watson.ShowDef AosIO.Mod ~

In the Lab installation, all your work is "saved" in the RAM disk… So be sure to backup it to Floppy/FTP/Mail regularly.

To backup via FTP to you n.ethz. account:
1. FTP.Open username@rz.n.ethz.ch ~ (middle click on FTP.Open)
2. First time per session, the system does not know your password. It places the command "NetSystem.SetUser …" into the Oberon log (not the kernel log). Execute this command. Type your password + <ENTER>
3. Execute 1. again. Use Put and Get in the FTP browser to backup and restore your files

To send your exercise via mail to your tutor:
1. Open the Mail Panel
2. Clicks Setting (into the little box up left)
3. Adjust your email address
4. Leave settings (click little x top left of the settings panel)
5. Select the files you like to send. Press AsciiCode in the mail panel
6. Fill out the To: field in the new text. Select the text with F1 and press Send * in the mail panel

Programming examples:

```
MODULE HelloWorld;

IMPORT
   AosOut;

PROCEDURE Hello*(par:PTR):PTR;
BEGIN
   AosOut.String("Hello!");
   RETURN NIL
END Hello;

END HelloWorld.

Aos.Call HelloWorld.Hello ~
```

```
MODULE MyCalc;
IMPORT
   AosIO, AosCommands, AosOut;

PROCEDURE Add*(par : PTR): PTR;
VAR sr : AosIO.StringReader;
   s : AosCommands.Parameters;
   a, b : LONGINT;
BEGIN
   s := par(AosCommands.Parameters);
   AosIO.OpenStringReader(sr, LEN(s.str^));
   sr.Set(s.str^);
   sr.SkipWhitespace; sr.Int(a, FALSE);
   sr.SkipWhitespace; sr.Int(b, FALSE);
   AosOut.String("Result: ");
   AosOut.Int(a + b, 0); AosOut.Ln;
   RETURN NIL
END Add;

END MyCalc.

Aos.Call MyCalc.Add 5 7
```