Object Oriented Paradigm Languages

- The central design goal is to build **inherent abstraction** into the system, moving all the abstract implementation details from the user level (ad-hoc) to the system level (inherent/built-in). In addition to the built-in abstraction security.
- The OOP aims also at code sharing via the "***inheritance***" mechanism, to match ***genericity*** in the block-structured languages.
- The basic philosophy of the OOP is to include the data objects' manipulating code within the objects themselves (i.e., as their behavior).
- Hence, shifting the algorithmic solution encoding from the traditional user domain (in the block-structured domian) to the lively data OO domain, in a message based system.
- Data objects will receive messages in the program code (the user) ordering them to act on themselves changing their states (and possibly other associated data objects states, in the process), to achieve a desired final state (solution).
- Conceptually, all data elements in the system are built as objects, each with a ***state*** (memory) and ***behavior*** (protocol). Hence, each object is an Abstract Data Type (ADT) of the block-structured domain.

- Hence, data can be viewed as an ***alive*** object; since they have states and protocol (set of commands), they change their state upon receiving a message.

- Everything in the system is considered to be an ***object***, (ADT), which is an instance of some ***class*** that groups similar behavior objects (with different states).

- We can easily draw the relation between data as ADTs and live objects.

ADT: Data Structure (to hold the type values) Set of Operations (works on the type)

Object: "State" (current object's value) "Protocol" (Behavioral Commands)

OOP's Vocabulary:

1) **"Object":** Encapsulated "state" and a "protocol" (behavior).

2) **"Class":** An abstraction (object) that describes a set of other objects that have the same behavior (protocol), but might be at different states (values), at a given time. Such objects can be spawned (instantiated) from the class via an instantiation process.

3) **"Instance":** An object instantiated from some class, its protocol is defined in its instantiating class, yet it has its own ***state***.

4) **“object’s protocol”:** The set of commands or messages that the object can respond to, defining its behavior.

5) **“messages”:** They are commands which are sent from one object to another, asking it to act on its state or other objects states.

6) **“Methods:** They are the concrete implementation (body code) of the above messages. A ***message*** is a procedure like ***call*** and the ***method*** is the ***implementation/definition*** of such procedure.

7) **“Inheritance”:** The Smalltalk system defines a class hierarchy of classes with each having zero or more subclasses. Each subclass “inherits” all of its parent class internal definitions (**violation of abstraction!**), and it might redefined some of them, superseding the inherited definitions. Hence, we can easily define new classes as subclasses of exiting classes, without extensive duplication of code, i.e., sharing all parent classes’ definitions, for free.

“**Subtyping**”: Inheritance can be used to define subtype S of type T.

T is the parent class and S is its subclass, that extends the behavior of T instances, i.e., an instant of S can do everything an instant of T can do plus more.

Squeak

Our Smalltalk window system is called **"Squeak"** with the following major windows:

A) **System Browser**:

Lists all classes categorized into groups, each of number of classes, when a class name is selected, you will have the choice of looking the protocol of the class itself or its instances (by selecting "class" or "instance" at the bottom of the window). When a protocol's message is selected, the implementation of its corresponding method is displayed at the bottom window. In the system browser window you can search for specific class.

B) **File List**:

Very useful window which allows you to look into the entire system directory, including yours, and manage your Smalltalk files, creating/deleting/saving/moving and so on. At the same time, you can save your finalized workspace code into files and to run later. Moreover, you can write and save Smalltalk code files and run them within your directory then resave them.

C) **Work Space**:

Scratchpad to write temp Smalltalk programs (on the fly), when you decide to keep, just save it in the Smalltalk directory"queak3.10.2-7179" for later access.

Types of Objects: They are expressed in four ways.

1) **Literals**: Five classes—

1) **Characters**: $a $? $$ $z
2) **Numeric**:
   i) integer: 1 -14 217
   ii) float: 6.25 -14.3 2.7e12 33.5e-10
   iii) fraction: 1/7 -7/721 4/9
3) **String**: 'This is a String literal' 'special @ + : $' 'It is ' also'
4) **Symbol**: #aSymbol #= #S
5) **Array**: #(1 $a 'example' 2.4 7/9 $*)

2) **Reserved Words**: *true* and *false* represent the only objects of classes True and False, respectively; which are subclasses of the class ***Boolean***.
"***nil***" represent the only object of the class ***undefindObject***.
self and ***super*** are used to indicate the receiving objects, i.e., the recipient object or its super class, respectively.

3)**Variables**: There are 6 kinds of variables, all initially bound to "*nil*"→

- **Instance Variables**: Defined in the instantiating class to hold the private "state" of each of its instances. Hence, the instantiation of an object will result in private copy of such variables to be attached with such instantiated object (to hold its state); though all instances share the same

(instances) protocol. They accessible from within the instance's methods only.

- **Class Variables**: Defined in the class and accessible by all instances and methods of the class and its descendant classes.

- **Temporary Variables**: Local variables declaration for use only within a defined method, or temporary code (scratchpad).

- **Global Variables**: System (dictionary) defined names (globals).

- **Pool Variables**: Variables defined for use by only a group o use.

- **Parameters**: Formal method parameters.

Variable Assignment:

a ← 6. "'a' is bounded to the "integer" class and the value 6"

a ← b. "binds a to the same class of b, it does not create a copy of b"

or a := b. "also accepted as an assignemt"

4) **Expressions**: An expression is the grouping of messages in a single statement.

6 + 3 – 7 * 9 "sending the message ***** with parameter **9** to the object that result from sending the message – with parameter 7 to the object that result from sending the message + with parameter 3 to object 6"

We notice that the regular precedence that we are used to is not applied here, and we need to use () to enforce it.

(6 + 3) – (7 * 9) " send the message – to the object that results from sending the message + to object 6 with parameter 3, with parameter that results from sending the message * with parameter 9 to the object 7.

There are three types of messages:

1) **Unary**: (parameterless)

6 factorial " send the parameterless message factorial to the object"

2) **Binary**: (one parameter)

6 + 5

3) **Keyword**: (one or more keyword/formal parameter pairs)

6 **gcd**:21 "send the message gcd- greatest common divisor- to the integer object 6 with parameter 21. The result is 3"

'test string' **at**: 3 **put**: $x.
"sensd the message **at:put:** to the left string to place x at the 3ed passion, replacing s"

Blocks: An object that is represented as a sequence of executable messages, separated by '.' And placed between [].

zeroOut ← [a := 0. b := 0. c := 0]

The above assignment binds the name "zeroOut" to the "block" class with the value of the block of messges that bind objects a, b, and c to the integer object 0.
To execute the block stored in zeroOut, just send it the message "value" to the block object zeroOut:

zeroOut **value**. "zero the objects a, b, and c"

This is an example of a message that is sent to en object to affect other objects, not itself.

Conditional Expressions With Block Parameters:

(x < y) **ifTrue:** [z ← y]
ifFalse: [z ← x].

" sends the keyword message **"ifTrue:ifFalse:"** to a *Boolean* object (x <y) that might return *true* or *false* of the classes True/False repectively."

| x y z bb cc| "declaring local variables"

y := 7. x := 5. bb := [z := y]. cc := [z := x].

(x > y) **ifTrue**: [bb value.]
ifFalse: [cc value].

Indefinite Looping With Block Parameters:

<"BlockContext"> **whileTrue:** <parameter-block>

|logn temp n|

n := 64. logn := 0. temp := n // 2.
[temp > 0] **whileTrue:** [temp := temp // 2.
logn := logn + 1].

Definite Looping With Block Parameters:

2 to:10 **by:**2 **do: [:even| sum := sum +even].**

Sending the message "to:by:do:" to an instant of class "number".

3 timesRepeat: [xCube := xCube * x].

Class Definition-

An Example is class "Account" in figure 12.7, page 385 in your Handout:

1) **Class Name**: Conventionally starts with uppercase letter.

   Line 1: #Account

2) **Super Class**: the name of immediate super class of the defined class, where all messages, variables, and methods are inherited.

   Line 1: **Object**

3) **Class Variables**: they hold the class "state".

   Line5: **TotalBalance** and **NumberOfAccounts**

4) **Class Methods**: class protocol commands implementation.

initialaize and reportTotal

5) **Instance Methods**: instances protocol commands implementation.

balance , deposit:of:on: , monthend , open:for: , withdraw:of:on:

4 and 5 are pointers to message dictionaries, each entry has three fields:

msg name – msg machine code – msg source code

Run Time Method Activation:

For every method invocation there will be an "object" activation record (**AR**) created to hold its "state".

An **AR** will have:

1) **Environment Part**:

i) **Locals**—parameters, temp (local) variables, intermediate results.

ii)**Non-locals**—static link (SL) that points to environment of the activated method, i.e., pointer to its activating instance object.

2) **Instruction Part**:

i) pointer to the caller method object code
ii) relative offset within the method code

3) **Sender Part:** A pointer to the AR of the caller (sender) method.

Method Activation:

When an instant object receives a message generated by another method, the following will take place:

1) An AR for the new method will be created.
2) The received message name is looked in the defining class inheritance hierarchy of the recipient method (following the static chain, SLs). If found (until the root class "object"), then proceed to step 3, otherwise, terminate with an error "undefined message".
3) Transfer message actual parameters (at the caller method) into their corresponding formal parameters listed the callee method's header. If any parameter does not match, terminate with an error, otherwise proceed to step 5.
4) Suspend; and save the current "state" of the caller method in its AR.
5) Set up the DL in the callee's AR to point to the caller's AR, then activate the callee.

Method Return:

1) Transfer the result objects (e.g., if needed as part of an expression) from the callee to the caller.
2) Resume execution at the caller by restoring its state from its AR into the hardware. There is no de-allocation of the callee AR since:
   i) we do not have stack model of computation (a heap is used), and
   ii) de-allocation of free spaces is "implicit", using the reference counter approach as in Lisp.

Question: We know that the usual "type" checking in the imperative domain is replaced by "protocol" checking, in Smalltalk.
Is Smalltalk still a "secure" "statically" typed language?
Yes, it is *secure*, but with ***dynamic "protocol"*** checking.

Question: What is the major difference in the run time semantics process of getting to a "non-local" name in the imperative block structures (Pascal, Algol, Modula-2, Ada) and object-oriented languages?
In the block-structured languages we have nested scopes due the nesting of procedures/functions, but in object oriented domain we have inheritance hierarchy to follow in order to get to non-locals; hence in block-structured languages we follow the static chain of static links (sequence of activation

records’ pointers) whereas in the object-oriented languages we follow pointers to classes’ run-time implementation structures of the inheritance mechanism.