Chapter 11 (4th Ed.)
Support of OOP in Languages

- **OOP**
- **Design Issues for OO Languages**
- **Smalltalk**
- **C++, Java, and Ada**

OOP

- **Background**
  - → **Procedural approach focused on subroutines and data**
  - → **Functional decomposition and data analysis**
- **ADTs**
  - → **OO languages focus on data types**
  - → **Subroutines are associated with particular data types, not just functional decomposition**
- **Inheritance**
- **Dynamic binding**

Inheritance

- **Reuse**
- **Extant types**
  - → **Modules**
  - → **“Not quite right”**
  - → **Independent, at same “level,” global**

Inheritance

- **Terminology**
  - → **Classes**
  - → **Objects - instances of classes**
  - → **Base class, superclass, parent class**
  - → **Derived class, subclass, child class**
  - → **Methods**
  - → **Messages - method invocation**
  - → **Method signatures**

Methods

- **Overloading methods**
- **Overriding inherited methods**
- **Instance variables & methods**
- **Class variables & methods**

More Inheritance Issues

- **Access control**
  - → **public, protected, private**

- **Single & multiple inheritance**

The better a class hierarchy matches the problem domain the more natural and better the complete solution will be.

Polymorphism and Dynamic Binding

- **Not the same as previous descriptions of dynamic binding**
- **Most methods can be determined statically**
- **Polymorphic variable, pointer**
  - → **Defined to refer to a base class**
  - → **Can also refer to any subclass**
  - → **Real type of object determined at run time so that proper method can be invoked**
- **Virtual methods, virtual class, pure virtual methods**

Computing in an OO Language

- **Messages are sent between objects**
- **Replies are objects**
- **Simulates computers communicating**
- **Each object is an abstraction of a computer**
  - → **Stores data**
  - → **Processes the data it contains**
  - → **Can send and receive messages**
- **Modeling real-world “objects”**

Design Issues of OO Languages

- **Exclusivity of objects**
  - → **EVERYTHING is an object**
  - → **Advantage: Elegant and pure uniformity**
  - → **Disadvantage: SLOW**
  - → **No distinction between predefined and user-defined types**
- **Subclasses and Subtypes**
  - → **Is-a relationship**
  - → **Subclass can only add variable and methods and override inherited in compatible ways**

More Design Issues

- **Type of Inheritance**
  - **Interface inheritance - only public interface is visible even to the derived classes**
  - **Implementation inheritance - implementations details also visible to derived classes**
    - Changing parent class requires recompilation of all derived classes
    - Defeats the purpose of information hiding
    - Can make code more efficient

More Design Issues

- **Type Checking and Polymorphism**
  - → **Polymorphic pointers require dynamic binding**
  - → **What about type checking?**
    - Compiler can check if signature and return types are the same (or compatible).
    - Dynamic type checking is much more expensive
- **Single and Multiple Inheritance**
  - → **Rarely necessary**
  - → **Complexity, e.g. name collisions**
  - → **Large programs $\Rightarrow$ large, complex class hierarchies and systems**

Even More Design Issues

- **Allocation and Deallocation**
  - → **Are all variable reference variables?**
  - → **Can one *new* an object?**
  - → **Can objects be statically declared?**
  - → **Is allocation implicit?**
  - → **Is deallocation implicit or explicit?**
- **Method Binding**
  - → **Are all binding of messages (method calls) to methods dynamic?**

Overview of Smalltalk

- **The definitive OO language**
- **EVERYTHING is an object, treated uniformly**
  - → **e.g. constant 2**
- **Messages**
- **All object from the heap**
- **All variables are reference variables**
- **All deallocation is implicit**
- **Designed from the beginning to be OO, unlike most other OO languages**
- **Its purity of purpose is reflected in its simple elegance and uniformity of design.**

Smalltalk Environment

- **Program editor**
- **Compiler**
- **OS**
- **Virtual machine**
- **Original graphical user interface**

Intro to Smalltalk

- **Expressions**
- **Methods**
- **Assignment Statements**
- **Blocks & Control Structures**
- **Classes**

Smalltalk Expressions

- **Literals**
- **Variable Names**
- **Message Expressons**
- **Blocks**

Smalltalk Expressions - Literals

- **Numbers**
  - → **e.g. Integer**
  - → **Typical arithmetic operators, among others**
- **Strings**
  - → **Comparisons, substrings, etc**
- **Keywords**
  - → **Identifiers with a trailing :**
  - → **e.g. at:**

- **All of these are objects, have a message protocols**

Smalltalk Expressions - Variables

- **Sequence of letters and digits that begin with a letter**
- **Private**
  - → **Local to the object**
  - → **Begins with lower case**
- **Shared**
  - → **Visible outside of the object**
  - → **Begins with upper case**
- **References, not static**
- **Typeless**
- **Named or indexed (arrays?)**

Smalltalk Expressions - Messages

- **Have the form of Expressions**
- **2 parts: the object and the message**
- **The message consists**
  - → **The method**
  - → **0 or more parameters**
- **Replies to messages are objects**

Smalltalk Expressions - Messages (cont)

- **3 categories**
  - **Unary**
    - e.g. firstAngle sin
  - **Binary**
    - e.g. 21 + 2
  - **Keyword**
    - e.g. firstArray at: 1 put: 5
    - No specific name for the method
    - Identified by its **selector** - at:put:
- **Precedence**
  - **Unary, binary, keyword**
- **Cascading**
  - **ourPen home; up; goto: 500@500; down; home.**

Smalltalk Methods & Assignment

- **General syntatic form**
  - → **message_pattern [| temp vars |] statements**

- **Message_pattern : function header**

- **Return value indicated by a ^**

- **Assignment statements**
  - → **similar in appearance to other languages; always assign pointers**

    total <- 22

    sum <- total -- will result in both total and sum pointing to 22

  - → **can use to capture information returned from a method**

    index <- index + 1

    salesTax <- deducts grossPay: 350.0 dependents: 4

Blocks

- **Blocks — are unnamed literal objects**
  - → **used to group expressions:**

    [index <- index + 1. sum <- sum + index]

  - → **executed by sending the unary message “value” to the block**

    addIndex <- [sum + index]

    sum <- addIndex **value**

  - → **are always executed in the context of their definition, even when passed as a parameter (similar to ALGOL 60 pass-by-name)**
  - → **can be parameterized**

    [:x : y | sum <- x + 10. total <- sum * y]

Control Flow Constructs

- **Reasonable set, all implemented via message passing paradigm (not simply!)**

- **Iteration**

  - **logical pretest loops — invokes keyword method whileTrue: of the block object containing the boolean condition; parameter is code to be executed if true**

```
count <- 0.
sum <- 0.
[count <= 20]
    whileTrue: [sum <- sum + count.
                count <- count + 1]
```

  - **repetition**

```
xCube <- 1.
3 timesRepeat: [xCube <- xCube * x]
```

  - **“for”**

```
2 to: 10 by: 2 do: [:even | sum <- sum + even]
```

- **Selection**

```
total = 0
    ifTrue: [averate <- 0]
    ifFalse: [average <- sum // total]
```

An Example Class Definition

```
class name               Polygon
superclass               Object
instance variable names  ourPen
                         numSides
                         sideLength
"Class methods"
"Create an instance"
new
    ^ super new getPen

"Instance methods"
"Get a pen for drawing polygons"
getPen
    ourPen <- Pen new defaultNib: 2

"Draw a polygon"
draw
    numSides timesRepeat: [ourPen go: sideLength;
                                  turn: 360 // numSides]
"Set length of sides"
length: len
    sideLength <- len

"Set number of sides"
sides: num
    numSides <- num
```

```
example use
   |MyPoly|
   MyPoly <- Polygon new
   MyPoly length: 60.
   3 to: 8 do: [:sides | MyPoly sides:
      sides.  MyPoly draw]
```

DeptCodes Example

class name	**DeptCodes**
superclass	**Object**
instance variable names	**names**
	codes

```
"create an instance"
new
   ^ super new

"instance methods"
"number of table entries"
size
   ^ names size

"fetch the code for a department"
at: name | index |
   index <- self indexOf: name.
   Index = 0
      ifTrue: [self error: 'Error-Nmae not in table']
      ifFalse: [^codes at: index]
```

DeptCodes Continued

```
"install a new code;  create entry if necessary"
at: name put: code |index|
  index <- self indexOf: name.
  index = 0
    ifTrue: [index <- self newIndexOf: name].
  ^ codes at: index put: code

"lookup index of a given dept name"
indexOf: name
  1 to: names size do: [:index |
      (names at: index) = name ifTrue: [^index]].
  ^ 0

"create new entry w/ given name and return index"
newIndexOf: name
  self grow.
  names at: names size put: name.
  ^ names size
```

DeptCodes Continued

```
"stretch table by 1 element ant put in new name"
grow | oldNames oldCodes |
  oldNames <- names.
  oldCodes <- codes.
  names <- Array new: names size + 1.
  codes <- Array new: codes size + 1.
  names replaceFrom: 1 to: oldNames size with: oldNames.
  codes replaceFrom: 1 to: oldCodes size with: oldCodes.

"test for inclusaiotn of a given name"
includes: name
  ^ (self indexOf: name) ~= 0

"test for empty"
isEmpty
  ^ names isEmpty

"create initial empty arrays"
initialize
  names <- Array new: 0.
  codes <- Array new: 0
```

Using the DeptCodes Class

Expression	***Result***
dCodes <- DeptCodes new	**Creates new instance**
dCodes initialize	**Creates empty arrays**
dCodes isEmpty	**true**
dCodes at: ‘Physics’ put: 100	**100**
dCodes at: ‘Chemistry’ put: 110	**110**
dCodes at: ‘Biology’ put: 120	**120**
dCodes isEmpty	**false**
dCodes size	**3**
dCodes at: ‘Chemistry’	**110**
dCodes includes ‘Physics’	**true**
dCodes includes ‘Computing’	**false**

Smalltalk/C++ Comparison

- **Programming environments**
  - → **Smalltalk is part in an integrated software development system — includes program editor, compiler, complete runtime support.**
  - → **C++ is a conventional compiled language**
- **Control/data structures**
  - → **Smalltalk uses message passing model. (elegant but slow)**
  - → **C++ uses usual collection of data and control structures (much faster)**
- **Binding**
  - → **Smalltalk has dynamic binding. C++ can use either (*virtual* specifies dynamic), but virtual functions with same name must have same type protocols — as significant restriction to polymorphism.**
  - → **However, dynamic binding is inefficient and means type errors are not detected until runtime.**
- **Classes as types**
  - → **C++ classes are types. Permits C++ objects to access private members of other objects of the same class. Also restricts polymorphism.**

Java

- **Not all types are objects, primitive scalar types**
  - → **Use wrapper class**
- **All classes have parent classes**
- **All Java objects are explicit heap dynamic**
- **Inheritance**
  - → **Single inheritance only**
  - → **Final keyword, cannot be overriden**
- **Dynamic Binding**
  - → **All methods are dynamically bound, except for final methods**
- **Encapsulation**
  - → **Classes & packages**