Object-Oriented Programming Languages

- Basic ideas:
  - programs are collections of objects, each of which has associated data and operations
  - objects are instances of object classes
    e.g. Joe is an instance of the class man
    my_list is an instance of the class list
  - classes can be defined in hierarchies — subclasses inherit the data and operations of their parent class
    e.g., cat is a subclass of animal
    octagon is a subclass of polygon
    but, my_octagon is an instance of octagon
  - reflects a data-oriented approach to programming

- Evolution:
  - SIMULA 67 — classes and hierarchies
  - Abstract Data Types — objects grouped with local data and operations (Ada, Modula-2)
  - OO Languages — add inheritance (objects can inherit data and operations from parent classes) and dynamic binding
Categories of OO Languages

- “pure” OO languages — all computation is based on message passing
  - static type binding, e.g. Eiffel
  - dynamic type binding, e.g., Smalltalk

- “hybrid” languages — use procedure calls rather than message passing, and a mixture of static and dynamic binding, e.g., CLOS, C++
Smalltalk — Fundamental Concepts

- Smalltalk programs consist entirely of objects
  - integers are objects
    - e.g., 3 is an instance of the class integer
  - an entire file-handling system can be an object

- All objects have:
  - local memory (state)
  - inherent processing ability (methods --- operations that can be performed on the object)
  - the capability to communicate with other objects
  - the ability to inherit characteristics from ancestors

- Objects do not have names; they are anonymous and can be referenced only by pointers
  - can be passed as parameters and returned from other objects

- Objects communicate with other objects by sending them messages. A message requests an operation that the other object provides.
  - e.g. could send a message to an octagon object to request that it (compute and) return its area
Smalltalk Expressions

- **Literals**
  - numbers, strings, keywords, etc.
  - but note, they are implemented as objects that are instances of a class
    e.g., only one 2 exists in the system; all “2” values point to it

- **Variables**
  - can be private (local to an object) or shared (visible outside the object)
  - are implemented as pointers to the actual object
  - are typeless; any variable can point to any object

- **Message Expressions**
  - used to send a message asking another object to carry out a specified operation (see next slide)

- **Block Expressions**
  - a sequence of expressions that can be executed as a unit or passed as a parameter
Message Expressions

- **Unary messages**
  
  → no parameters; just name the receiving object and the method in that object, e.g.,
  
  mh_octagon area
  
  anglex sin
  
  flag is_true

- **Binary messages**
  
  → single parameter (an object) that is passed to the specified method of the receiver, e.g.,
  
  $21 + 2$ — means “pass the object 2 to the + method of object 21”

  → mainly used for arithmetic operations

- **Keyword messages**
  
  → one or more parameters are passed, with keywords used to indicate correspondence with formal parameters in the method

  → methods are not named; keywords are used to identify the intended method, e.g.,

  arrayc at: 1 put: 5 — means “pass the parameters (1,5) to the at:put method of object arrayc”
Methods

- Define the operations that an instance of the class will execute when a message corresponding to the method is received (analogous to function definitions)

- General syntactic form (brackets mean optional):
  
  message_pattern [ | temporary variables | ] statements

- Can return a value, indicated by preceding expressions that describe it with ^, e.g.,
  
  currentTotal
  ^\text{\textasciicircum}(\text{oldTotal} + \text{newValue})

- Keyword methods — specify the pattern of keywords that identify the particular method, e.g.,
  
  x: \text{xCoord} y: \text{yCoord}  -- matches an \text{x:y:} message selector
  ourPen up; goto xCoord @ yCoord; down.  -- sends all three messages to ourPen

  → example call to this method:
  
  ourPen x: 300 y: 400
Assignment Statements/Blocks

- **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 — 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]
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.