CS 474—Spring 2014
Introduction to Smalltalk (Part II)

Ugo Buy
Dept. of Computer Science
University of Illinois at Chicago

January 21, 2014

Smalltalk

- First general-purpose OO language
  - Golberg/Robson 1980
  - Follows Simula (1966), an OO language for writing simulation software
  - Today: Seldom used, but big part of Objective C, the language for programming iPhones, Macs, iPads, etc.
- Non-commercial version is publicly available
  http://www.cincomsmalltalk.com/main/developer-community/trying-cincom-smalltalk/try-cincom-smalltalk/

Smalltalk

- Designed with GUIs in mind
- Window-based environment consisting of
  - Many predefined classes
  - IDE (Interactive Development Environment) with class browser, debugger, stepper, code inspector
  - Interpreted/compiled program execution

Smalltalk: Main features

- Pure OO language
  - Every type is a class (not even a few Java-style primitive types)
  - Every expression is a message expression
  - Every function is a method
  - No main program, execution starts by sending msg to an object
- Single inheritance model
- Dynamically typed
- Statically scoped
- Garbage collected

Literals

1. Numbers
   - As usual (e.g., 3, 10.50, -140, etc.)
   - Radix prefix is "r", e.g., 1
     - 16rFF = 8r377 = 255
     - 16r-10 = -16
   - Scientific notation: 16r8e2 = 2,048
2. Characters
   - $ notation, e.g., $1, $a, $A, $$, etc.

5

Literals

3. Strings
   - Single quote notation, e.g.,
     - 'this is a string'
   - Two single quotes put quote character in string e.g.,
     - 'don''t do this'
4. Symbols: Unique strings
   - One object for each print name
   - # notation
   - Case sensitive
   - Examples: #aSymbol, #jack

6

Literals

5. Arrays (other than strings)
   - One dimensional, nesting gets multiple dimensions
   - Dynamic typing allows elements to be different types
   - #( ) notation
   - Nested arrays: Don't repeat # symbol
   - Array with 5 elements:

     #($a $b ($c 18 $d) 'gesundheit' ($f $g $h))
   - Empty array: #( )
   - Elements numbered from 1 on: Count like humans do, not like C/C++ do

7

Literals

6. Comments
   - Enclosed in double quotes

     "This is a comment"

8

Literals

7. Variables
   - Sequences of letters and digits, starting with a letter
   - Nothing else allowed (e.g., no underscores, dashes)
   - Case sensitive
   - First char determines type of identifier
     - Lowercase: Private identifier, accessible in just one object (e.g., instance variable name)
     - Uppercase: Global identifier (e.g., class name)

9

Literals

8. Special symbols, denoting frequently used instances
   - nil Bottom object (an instance of class UndefinedObject)
   - true Boolean object
   - false Boolean object
   - self Message receiver (similar to this in Java)
   - super Again message receiver, but special search for method used)

10

Assignments

- Left arrow notation in original definition, later replaced by more traditional colon-equal notation
  - x <- y "old notation"
  - x := y "new notation"
  - colors := #('red' 'green' 'blue')
- Assignment is a value-returning expression, can be cascaded (right associative, as in C/C++)
  - x := y := 1 "This is the same as (x := (y := 1))"

11

Message expressions

- Syntax must specify: (1) receiver, (2) selector, and (3) arguments (if any)
  - Message sender implicitly defined as object executing message expression
  - Recall that *all* expressions are message expressions in Smalltalk
- Examples:
  - 3 + 4 "Receiver is 3, selector is +, argument is 4"
  - x > y "Receiver is x, selector is >, argument is y"
  - x sqrt "Receiver is x, selector is sqrt"

12

Kinds of message expressions

Three kinds of message expressions:

1. **Unary** expressions
   - No arguments
2. **Binary** expressions
   - Arithmetic and relational expressions: One argument
3. **Keyword** expressions
   - Typical case: One or more arguments

- Parenthesized expressions are possible too

13

Unary expressions

- Receiver followed by message selector (id)
- Two items separated by blank char (space, newline, …)
- No arguments
- Receiver can be literal constant, identifier bound to receiver object, or expression returning the receiver
- Examples:
  - x sqrt "receiver = object bound to x"
  - (x + y) sqrt "receiver = object returned by expr (x + y)"
  - 4 sqrt "receiver is literal constant 4"

14

Binary expressions

- Three tokens
  1. Receiver
  2. Message selector
  3. Argument
- Three tokens separated by blank chars
- Receiver and argument can be literal constants, bound identifiers, or expressions returning object
- **Selector must consist of one or two non-alphanumeric characters**

15

Binary expressions

- Examples of binary expressions:
  - (x sqrt) + y "receiver is object returned by (x sqrt)"
  - x >= 0 "receiver is object bound to x"

16

Keyword messages

- Receiver as usual
- Message selector consists of one or more keywords, each ending with a colon
  - at: "Return array element at argument index position"
  - at:put: "Change array element at index position."
- Arguments are interspersed in message selector
- Examples:
  - anArray := #($a $b $c).
    anArray at: 2 put: $x. "anArray now contains #($a $x $c)"
  - anArray at: 3 "this returns $c"

17

Parsing message expressions

- Each expression always returns a value
  - What expressions do anyways, right?
- Simple rules for parsing message expressions
  - None of operator precedence nonsense from C and C++

18

Parsing rules

1. Unary and binary expressions parse left-to-right
2. Unary expressions have higher precedence than binary expressions, which have higher precedence than keyword expressions
3. Parenthesized expressions have higher precedence than all other

- Simple riddles: What do these expressions return?
  - 12 - 2 * 10 – 1
  - 10000 sqrt log "log returns logarithm base 10 of receiver"

19

Caveats

1. Do not concatenate keyword expressions, e.g.,
   - Try assignment A[i] := A[j]

     a at: i put: a at: j is incorrect because message selector at:put:at: does not exist!

     a at: i put: (a at: j) is correct!
2. Can mix different kinds of expressions, e.g.,
   - bigFrame width: smallFrame width * 2
   - This ends up being:

     bigFrame width: ((smallFrame width) * 2)

20

Dynamic typing

- Smalltalk is both *dynamically typed* and *strongly* typed
  - Dynamically typed

    Type information associated with objects (at run-time), not identifiers (at compile-time)
  - Strongly typed

    Every object *has* a type (unlike assembly language)
- Examples:
  - 3 < 4 + 5 error (cannot send Boolean obj message +)
  - 3 + 4 < 5 returns Boolean object false

21

Dynamic typing

- Run-time type checking
  - Undefined method results in message doesNotUnderstand: being sent to receiver
  - This means that a bad message was sent to a receiver
  - The run-time method search mechanism of Smalltalk could not find an appropriate method after searching the class of the receiver and its superclasses

22

Dynamic typing

- Dynamic storage allocation and deallocation required
  - Compiler does not know type of referents
  - Handled transparently by run-time system
  - Allocation done automatically for literal expressions appearing in source code:

    a := ' hello there'
  - Deallocation done automatically with garbage collection (no free, delete operators)

23

Pointer semantics

- Assignment uses pointer, not copying, semantics
  - a := 'hello there' .

    b := a. "Now a and b point to same obj"
- Same as in Java, different from value and reference identifiers in C++
- If copying is really necessary, use one of two copying methods, **shallow copy** and **deep copy**
  - Shallow copying copies only first level of receiver
  - Deep copying copies recursively entire receiver

24

Pointer semantics

- Copying examples:
  - list1 := #($a $b ($c $d) ($e $f) $g).
    list2 := list1 shallowCopy.
    list3 := list1 deepCopy.
  - (list2 at: 4) at: 1 put: $z.
    (list1 at: 4) at: 1. "will return $z"
    (list3 at: 4) at: 1. "still returns $e"
- Deep copying stops only at immutable objects, which are unique (e.g., instances of classes Boolean, SmallInteger, Symbol, Character, or instance nil)

25

Comparisons

- Logical equality
  - Syntax: =

    Logical comparisons of receiver and argument, returns true if two operands are logically equivalent
- Physical identity
  - Syntax: ==

    Physical comparison, returns true only if the receiver and the argument are the same instance (aka object)

26

Examples of equality messages

- Typical evaluations
  - string1 := 'hello'.
    string2 := 'hello'.
    string3 := string1.
  - string1 = string2.
    string1 == string2.
    string1 == string3.
- Negation of == is ~~
- Negation of = is ~=

27

More on equality messages

- Message == defined in class Object, the root of Smalltalk's class hierarchy
- Message = defined in class Object to be the same as ==
- Consequence: Since every class inherits from Object (including your classes), you must redefine = whenever you care about = being different from == (e.g. String class)
- Beware of potential asymmetries
  - Possible to define = in classes B and C, so that
    instB = instC "may return true"
    instC = instB "may return false"
  - Please do not do this!

28

Atomic objects

- Objects that cannot be broken down into parts
- Always unique objects, like instances of Symbol (not atomic)
- Roughly speaking, all instances of Boolean, SmallInteger, Character, and instance nil
- Examples:
  - 2.0 == 2.0 "returns false"
    2.0 = 2.0 "returns true"
    2 == 2 "always returns true"
    #hello == #hello "again true, symbols are unique"
    'hello' == 'hello' "false, strings are not unique"

29

Control flow

- Message expressions for defining flow of control within methods
- Block: Basic unit for aggregating message expressions
- Alternation (choice), e.g., by messages ifTrue: ifFalse:, etc.
- Iteration (repetition), e.g., by messages whileTrue: whileFalse: do: to:do, timesRepeat: etc.
- Also, language is recursive
- Exception handling messages
- Messages for concurrency

30

Blocks

- Block: Sequence of message expressions
- Syntax: Message expressions are enclosed in square brackets and separated by periods.
  - Beware of difference between statement *terminator* (e.g., semicolon in C++) and statement *separator* (e.g., semicolon in Pascal, period in Smalltalk)
- Example:

```
[ index := index + 1.
  array at: index put: 0. ]
```

31

Block semantics

- All blocks are first-class language structures
  - In VisualWorks, instances of class BlockClosure
  - Blocks can be created dynamically, assigned to identifiers, passed/returned to/from methods, and called (executed) under program control
- Processing of block literal causes block object to be created, but not executed
- Possible because Smalltalk is *dynamically linked*

32

Block semantics

- Block object executed by sending it message value
  - **resetNextElement := [index := index + 1. array at: index put: 0.].** "Block defined and bound to id"

    resetNextElement value. "Block executed"
- Block execution returns value of last message expression executed in block
  - This would be 0 in the example above
- Block can contain return statement, which terminates block and enclosing method
- Syntax uses caret character (^<expr> syntax)

33

Block examples

- Example of block object creation + binding to identifier

  "Block object created, but not executed."

  energyFormula := [e := m*c*c.]
- Example of block execution

  energyFormula value. "Multiplications and assignment take place."
- Caveat 1: Identifiers c, e, m must have been declared and in scope when block was defined (not necessarily when block executed)
- Caveat 2: Identifers c, m must hold numeric values when block executed

34

Block identifiers

Three kinds of identifiers can appear in blocks (all dynamically typed):

1. Local identifiers

   Statically scoped, visible only within defining block
2. Block parameters

   Statically scoped, visible only within defining block, bound to arguments when block executed
3. Free variables

   Statically scoped, visible in defining block and context in which block was defined

35

Block parameters

- All parameters declared at beginning of block
- Declarations end with vertical bar
- Each parameter identifiers preceded by colon (in declaration only)
- Arbitrary number of parameters
- Block with parameters executed by messages value:, value:value:, etc. depending on number of parameters
- Message valueWithArguments: passes array containing all arguments to block receiver
- Wrong argument number results in run-time error

36

Block parameters

- Block parameters cannot appear in left-hand side of assignments
- All parameters declared at beginning of block

```
aBlock := [ :x :y | x + y + 3].
a := aBlock value: 5 value: 10.      "a is bound to 18 after block ev."
```

- `aBlock valueWithArguments: #(5 10).` "Same effect."
- More examples

```
energyFormulaWithParams := [ :m :c  | e := m * c * c. ].
energyFormulaWithParams value: 5 value: 10.      "This will return
    500."
```


Local variables

- Declared after block parameters at beginning of block
- Called *temporary* variables in Smalltalk jargon
- Can be freely used (assigned and read) within defining block and enclosing blocks
- Enclosed in vertical bars
- Lexically scoped inside the block
  - Scope rules based on nesting of units within units
- Block syntax

```
[ :par1 :par2 … | | temp1 temp2 temp3|  message expressions… ]
```


Free variables

- A block may contain "free" variables, not defined in the block, e.g.,
  - Instance variables, class variables, method parameters, parameters and temporary variables of enclosing blocks, etc.
- Lexical (static) scoping: Reference to a free variable will be resolved in the context where the block was defined, not the context where the block is executed
- Figuring out correct reference can be tricky

  We will see next while just how difficult…

Example of free block variables

- What is value of (C1 new) m2: 36?

```
"assume methods m1, m2 are in class C1"
m1: x      "returns a block"
    |a b c|
    a := 10.
    b := 20.
    c := [:z | b := a + x. a := b - z.].
    a := 99.
    ^c.
m2: x    "returns an integer"
    |a b c|
    a := 5.
    b := 50.
    c := self m1: b - a.
    ^(c value: x).
```

Control flow: Alternation

- Conditional choice supported with four keyword methods defined in class Boolean
  - ifTrue:
  - ifFalse:
  - ifTrue:ifFalse:
  - ifFalse:ifTrue:
- All methods have boolean receivers and zero-argument blocks for arguments
  - ifTrue: executes argument block if receiver is true
  - ifFalse: executes argument block if receiver is false

41

Control flow: Alternation

- Additional conditional methods arguments
  - ifTrue:ifFalse: executes first argument block if receiver is true, second argument otherwise
  - ifFalse:ifTrue: executes first argument block if receiver is false, second argument otherwise
- Method ifTrue: returns nil if receiver is false
- Method ifFalse: returns nil if receiver is true
- In all other cases, conditional methods return value of executed block (i.e., the block's last msg expression)

42

Control flow: Alternation

- Examples of conditional expressions:

```
(number \\ 2 = 0)                    "The standard form"
      ifTrue: [parity := 'even']
      ifFalse: [parity := 'odd']

parity := (number \\ 2 = 0)          "Taking advantage of syntax"
            ifTrue: ['even']
            ifFalse: ['odd']

(aStream atEnd)                      "Another example"
      ifTrue: [aStream reset]
      IfFalse: [c := aStream next]
```

43

Logical operators

- Logical operators are Boolean messages
- Following operators always evaluate argument (if any), in addition to receiver
  - & logical conjunction (and)
  - | logical disjunction (or)
  - not logical negation (postfix, of course)
  - eqv: logical equivalence (Boolean arg)
  - xor: exclusive or (Boolean argument)

44

Lazy logical operators

- Following operators do not evaluate argument unless they have to
- **Argument must be a block** (not always evaluated)
  - ifTrue:
  - ifFalse:
  - and: lazy conjunction (like ifTrue:)
  - or: lazy disjunction (like ifFalse:)
  - When and: and or: return false, ifTrue: and ifFalse: return nil, otherwise identical behavior
- Code example:

```
nextElement := (aStack isEmpty not) and: [aStack pop]
```

45

Iteration constructs

- Three kinds of iteration messages:
  1. Conditional iteration
  2. Indexed iteration
  3. Iteration over collections

46

Conditional iteration

- Messages whileTrue: and whileFalse:
- **Receiver must be a block**
- Message whileTrue: evaluates receiver (block)
  - If receiver is true, evaluate argument block and repeat
  - If receiver is false, return nil
  - whileFalse: works in similar way

47

Example of conditional iteration

- Compute sum of array elements

```
total := 0.
index :=1.
[ index <= array size ] whileTrue: [ total := total + (array at: index).
                                     index := index + 1.].
```

48

Method implementation

- This is how whileTrue: is implemented in class BlockClosure (taken directly from VWNC)

```
whileTrue: aBlock
    "Evaluate the argument, aBlock, as long as the value of the
    receiver is true."
^self value
    ifTrue:
        [aBlock value.
        [self value] whileTrue: [aBlock value]]
```

49

Indexed iteration

- Messages to:do: and to:by:do:
  - Receiver is an integer
  - to: argument is an integer
  - do: argument is a block with one argument
  - by: argument is an integer (second message only)
- Semantics: Index is generated ranging from receiver to to: argument
  - by: argument specifies increment
- Argument block executed once for each index value
- Index value passed as an argument to block

50

Example of Indexed iteration

- Increments elements at odd positions of an array

```
1 to: array size by: 2
    do: [ :index | array at: index put: (array at: index) + 1]
```

- nil is returned

51

Iteration over collections

- Collections are instances of concrete subclasses of abstract superclass Collection
- Main Collection subclasses:
  - Bag (concrete)
  - Set (concrete)
  - SequenceableCollection (abstract)
- All such instances take message do:
- Argument is one-argument block, which is executed with each element of the receiver as an argument
- Receiver is returned

52

Example of do: method

- Adding all elements of an array of numbers

```
| total |
total := 0.
#( 10 20 30 ) do: [ :x | total := total + x.]
  "Now total holds 60…."
```


Additional iteration messages

Most concrete subclasses of Collection have following messages

- collect: returns a collection (same type as receiver) of values returned by each iteration, e.g.,

  'hello' collect: [:x | x asUppercase] will return **'HELLO'**

  #(3 6 12) collect: [:elem | elem odd] returns **#(true false false)**

- select: returns objects whose block evaluation returned true, e.g.,

  #(3 6 9 12) select: [:elem | elem odd] returns **#(3 9)**

- reject: return objects whose evaluation returned false

Additional iteration messages

- Integer object responds to message timesRepeat:
- Argument is a zero-argument block
- Argument block as many times as value of receiver
- Return nil

Instance creation

- Most classes respond to **class** message new
- Differences between class and instance methods
- Instance method:
  - Associated with an instance of a class
  - Can access directly and modify instance variables of message receiver
  - Can access directly and modify class variables of receiver's class
  - Cannot access instance variables of other instances of receiver's class (in Smalltalk)

Instance creation (cont' d)

- Class method:
  - Associated with an entire class
  - Can access directly and modify class variables
  - Smalltalk: No access to any instance variable
  - C++ (Java): Functions (methods) declared static
  - C++, Java: Qualified access to instance variables when access restrictions and visibility rules permit (e.g., anInstance.publicVar)
  - No qualified access in Smalltalk; method only has access to receiver's (instance and class) variables, nobody else's variables

57

Instance creation (cont' d)

- Class method new generally returns a new instance of the receiver (a class)
  - Assume C1 is a defined class, expression below creates new instance and binds to identifier

    instC1 := C1 new.
- Class method new is defined in Object to do following:
  1. Allocate space for new instance of receiver (a class object different from Object)
  2. Initialize all variables of new instance to nil
  3. Return new instance

58

Instance initialization

- If different initialization actions required, programmer can either:
  1. Refine Object's new to perform additional initialization actions for class under consideration
     - *Method refinement:* Subclass redefines inherited method to (1) perform all actions of superclass's method and (2) some additional actions
     - Discussed later in the course

59

Instance initialization (cont' d)

- Additional option for initialization:
  2. Define new class methods for instance creation with specific initialization actions
     - Use programmer defined method instead of new
     - Similar to constructors in C++ and Java, but must invoke instance creation method explicitly, e.g.,

```
C1 newx: initialX y: initialY
    |temp|
    temp := C1 new.
    temp x: initialX.      "Assume C1 has modifier methods x: y: "
    temp y: initialY.
    ^temp.
```

60

Instance initialization (cont' d)

- Yet another option for initialization:
  3. Define instance methods for initialization
  - Client' s responsibility to invoke such methods, after new instance created

    Client of a class: Code using that class (to define new instances, work with existing instances, etc.)
  - Suppose C1 defines instance method initialize, then client could use code below:

    instC1 := C1 new initialize.

61

Class definition example

- Classical LIFO stack implemented with OrderedCollection

```
Object subclass: #Stack
      instanceVariableNames: 'items '
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Ugo-examples'

"Stack class methodsFor: 'instance creation"
new
      "overrides super new "
      | temp |
      temp := super new.
      temp initialize.
      ^temp.
```

62

Class definition example

- Instance methods of Stack

```
initialize
      "initialize instance variable items to empty ordered collection"
      items := OrderedCollection new.
      ^self.

isEmpty
      "test for emptiness, return Boolean instance"
      ^items isEmpty.

pop
      "stack pop operation, old stack top is returned"
      (self isEmpty)
         ifTrue: [self error: 'Attempt to pop an empty stack']
         ifFalse: [^items removeFirst]
```

63

Class definition example

- Instance methods of Stack (continued)

```
push: x
      "push operation, returns the items in the receiver"
      items addFirst: x.
      ^items first.

top
      "returns top of receiver without modifying it"
      ^items first.
```

64

System defined classes

- Root of class hierarchy is Object
- Main subclasses of Object:
  - BlockClosure
  - Boolean
  - Magnitude
  - Collection
  - Stream
  - ProcessorScheduler, Delay, SharedQueue
- User-defined classes are merged seamlessly with predefined classes

65

Class definition

- Type class definition in browser
- Must specify following items:
  1. Class identifier (a symbol)
  2. Superclass name (capital identifier)
  3. Whether class has named or indexed instance variables
  4. Instance variable names
  5. Class variable names
  6. Class package
  7. Class namespace

66

Syntax of class definition

- Define class identifier (item 1), superclass identifier (2), and type of instance variables (3) by using keyword subclass: or variableSubclass:
- Class with named instance variables:

```
<superclass_id> subclass: <class_symbol>
                instanceVariableNames: …
```

- Example:

```
Object subclass: #Stack …
```

- Class with indexed instance variables:

```
<superclass_id>  variableSubclass: <class_symbol>
```

- Main difference: Methods available to new class

67

Classes with indexed variables

- Method for class creation is new: as opposed to new
- Example:

```
aStack := Stack new.          "Instance vars initialized to nil"
anArray := Array new: 4.      "Array with 4 indexed variables"
```

- Indexed variables still initialized to nil

  anArray bound to #(nil nil nil nil) after above expression
- Accessed by their position, otherwise anonymous
- Messages at: and at:put: available to class instances defined with indexed instance variables

68

Named instance variables

- Declared after instanceVariableNames: selector in class definition
  - Smalltalk string listing the names of all instance variables
- Initialized to nil
- Only accessible within defining object
- Accessed by name (no qualification) in code of defining class and its subclasses
  - Always refer to a receiver variable

69

Named instance variables

- Similar to protected access level of C++, except:
  - Cannot access variables of other instances in the receiver's class
  - This is possible even for private members in C++

70

Class variables

- Declared after classVariableNames: keyword in class definition
  - String listing the names of all class variables
- Follows instance variable declarations
- Capitalized
- One copy of variable shared by all instances of a class and its subclasses
- Class variables shared only by class (but copied for each subclass) are called *class instance variables*

71

Example of class definition

- Define a Node class for a linked list
- Subclass of object, two named instance variables, no class variables

```
Object subclass: #Node
    instanceVariableNames: 'element next'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CS474-Examples'
```

- More Node code later on

72

Semantics of class definition

- Create new class by selecting "accept" option in action menu of Smalltalk browser, or by using menu option *new class…*
- New class defined
- Class object created
- Metaclass object created as well
- Method dictionary and class method dictionary created
- Class objected denoted by name of Class (capitalized)
- Ready to create new instances
- Can add methods later

73

Method definitions

- Again use class browser to define methods
- Syntax:
  1. Message pattern (selector and parameters)
  2. Temporary variables between vertical bars
  3. Sequence of message expressions separated by periods
- Up arrow character ^ returns a value and exits
  - Method with no return expression will return self

74

More on method definitions

- Method parameters cannot appear in left-hand side of assignments
- Scope of temporary variables and parameters is method body
- Extent is method execution
- Within method refer to instance variables of receiver by their name (or with at: at:put: for indexed vars)
- Within method refer to class variables of receiver's class or superclass(es) by their name
- Private method category is not enforced!

75

More on method definitions

- Can create unary, binary or keyword methods uniformly
- Class vs. instance methods defined by selecting appropriate type of method in browser
  - Class vs. instance tab in latest versions of VWNC Smalltalk
  - Radio button in earlier versions

76

Examples of method definitions

- Methods of Node class defined earlier
- A class method for instance creation

```
new: newElement
    ^super new element: newElement
```

1. Invoke new method defined in superclass (Object), which allocates space for new instance, initializes variables to nil, and returns reference to new instance
2. Send message element: to new instance to change element variable

77

More on new: example

- Precedence rules parse code as follows:

```
new: newElement
    ^((super new) element: newElement)
```

- Newly created and initialized instance returned
- Class Node inherits new class method from Object
  - Thus, in this case instances can be created either with new or new:

78

Examples of instance methods

- Accessor and modifier methods of Node class

```
element
    ^element

next
    ^next

element: x
    element := x

next: y
    next := y
```

79

A Queue example

```
Object subclass: #Queue
    instanceVariableNames: 'front back'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CS474-Examples'

"instance methods"
isEmpty
    ^front = nil

add: newElement
    self isEmpty
        ifTrue:  [ front := back := Node new: newElement ]
        ifFalse: [ back next: (Node new: newElement).
                   back := back next ]
```

80

A Queue example (continued)

```
"instance methods continued"
remove
    | returnElem |
    self isEmpty
        ifTrue:  [ ^self error: 'queue underflow' ]
    returnElem := front element.
    front := front next.
    front = nil
        ifTrue: [ back := nil ]
    ^returnElem
```

81

Inheritance in Smalltalk

- Traditional single inheritance model
- Subclass inherits all variables (whether instance or class variables) and methods (whether instance or class methods) from its superclass
- Defer a method by message subclassResponsibility
- Abstract class Collection defers definition of add:

```
add: x
    self subclassResponsibility
```

82

Inheritance in Smalltalk

- Subclass has right to refuse inherited method, with message shouldNotImplement
- Class ArrayedCollection, a subclass of Collection, refuses inherited method add: because you cannot add elements to an array
- Implementation of add: in ArrayedCollection

```
add: x
    self shouldNotImplement
```

83

Method refinement

- Method refinement by special identifier super
- Identifier denotes the same receiver as self, but message sent to super causes different search for method to be executed

84

Alternative Queue implementation

```
Object subclass: #Queue
    instanceVariableNames: 'items '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CS474Examples'!

"Queue class methodsFor: 'instance creation' "

new
    "initialize elements to empty ordered collection"
    ^((super new) items: (OrderedCollection new)).
```

85

Instance methods

- Methods for queue operations

```
dequeue
    "dequeue operation"
     ^items removeFirst.

enqueue: item
    "enqueue operation"
    items addLast: item.
    ^items.

items: x
    items := x.
    ^self.
```

86

Queue subclass: Priority queue

- Queue subclass that implements a priority queue
- Elements removed depending on some ordering, rather than FIFO order
- Use a two-argument block to compare elements stored in receiver (a PriorityQueue instance)
- Class definition:

```
Queue subclass: #PriorityQueue
    instanceVariableNames: 'comparisonBlock '
    classVariableNames: 'DefaultBlock '
    poolDictionaries: ''
    category: 'CS474Examples'
```

87

Class methods for Priority queue

```
new
    "refines method inherited from Queue"
    | temp |
    temp := super new.
    temp comparisonBlock: DefaultBlock.
    ^temp.

new: aBlock
    "refines method inherited from Queue"
    | temp |
    temp := super new.
    temp comparisonBlock: aBlock.
    ^temp.
```

88

Class variable initialization

- Special **class** method called initialize performs initialization of class variables
- Invoked by Smalltalk run-time system whenever new class created
- Must return receiver (class object of class being created)

```
"Note: This is class method initialize, not instance method initialize"
initialize

  "initialize DefaultBlock todo trivial comparison"
  DefaultBlock := [:x :y| x <= y].
  ^self.
```


Instance methods for PriorityQueue

```
comparisonBlock
  ^comparisonBlock.

comparisonBlock: aBlock
  comparisonBlock := aBlock.

enqueue: item
  "override inherited method from Queue to reflect item priorities."
  1  to: items size
     do: [ :i | (comparisonBlock value: item value: (items at: i))
                  ifTrue: [  items add: item beforeIndex: i.
                             ^items.]].
  items addLast: item.
  ^items.
```


self vs. super

Both denote same (current) receiver, but different searches for method to be executed

Search with self is dynamic, super is static (can be resolved at compile-time)

1. self starts search in class of receiver, even if message expression containing self (i.e., the method currently executing) is in a superclass of the receiver's class
2. super always starts search for method to be executed in the immediate superclass of the class containing the method that executes expression with receiver super

Examples of self vs. super

```
Object subclass: #A
  …
  who
    ^$A
  whom
    ^self who
```

```
B subclass: #C
  …
  who
    ^$C
  why
    ^super whom
```

```
A subclass: #B
  …
  who
    ^$B
  what
    ^super who
  where
    ^super whom
```

```
instB := B new.
instC := C new.
instB whom.
instC whom.
instB what.
instC what.
instB where.
instC where.
instC why.
```

Riddle: Can you guess the values returned by these message expressions?

Answers to the riddle

```
instB := B new.
instC := C new.
instB whom.     returns $B
instC whom.     returns $C
instB what.     returns $A
instC what.     returns $A
instB where.    returns $B
instC where.    returns $C
instC why.      returns $C
```

How may did you guess correctly?

93

What is different?

- Identifier self must return down to receiver's class even when inherited method executes message expression containing self
  - Proper interpretation of message polymorphism
  - Reason why "instC why" returns "$C" in earlier example
- Identifier super uses executing method as a starting point from which to start search
  - Search starts in superclass of class containing method
  - Static search! (Can be resolved at C-T)

94

Why the difference?

- The behavior of super allows for chains of invocations
  - Proper interpretation of message polymorphism
  - Reason why "instC why" returns "$C" in earlier example
- Identifier super uses executing method as a starting point from which to start search
  - Allow chains of invocations of method defined in multiple superclasses in inheritance chain of receiver

95

Example with super

- Suppose Person class has subclass Student, and Student has subclass Undergraduate
- All 3 classes define method print as follows:

```
Class Person:       print: outStream
                    "prints Person variables"

Class Student:      print: outStream
                    super print: outStream.
                    "prints Student noninherited variables"

Class Undergraduate: print: outStream
                     super print: outStream.
                     "prints Undergraduate noninherited vars"
```

96

Example (cont' d)

- Static definition of super works well, but using original receiver as reference would cause infinite recursion in class Student when receiver is Undergraduate instance
- Recall code:

```
Class Person:        print: outStream
                     "prints Person variables"

Class Student:       print: outStream
                     super print: outStream.
                     "prints Student noninherited variables"

Class Undergraduate: print: outStream
                       super print: outStream.
                       "prints Undergraduate noninherited vars"
```

97

Instance implementation

- Every instance represented by an object (like C++)
- Every class is represented by a *class object* (extending the concept of vtbl for polymorphic classes in C++)
- Every instance has a pointer to the corresponding class object (similar to vptr in C++)
- Class object denoted by (capitalized) shared variable
- Follow link to class object by sending message class to an instance, e.g.,

```
anArray class.    "This will return Array"
```

98

Class objects

- Class object holds following information on behalf of all instances of that class
  - *Method dictionary*. Keys are selectors, values are compiled code (similar to vtbl, except only instance methods defined in class are stored in dictionary)
  - *Instance variable names*. Needed when creating new instances
  - *Class variables*. Just like instance objects hold instance variables…

99

Class object implementation

- Class variables hold method dictionary and string with instance variable names
  - Method dictionary accessed with class method **methodDictionary**
- Class object has references to superclass and subclasses
  - Accessed with class messages superclass and subclasses

```
Array superclass.                  "returns ArrayedCollection"
Array superclass superclass.       "returns SequenceableCollection"
Array superclass superclass superclass.  "returns Collection"
```

100

Message dispatching (method search)

- Simple, but time-consuming, algorithm:
  1. Follow class pointer from receiver to class object
  2. Search method dictionary for method
  3. If found, execute method
  4. Otherwise, follow superclass pointer and repeat from Step 2
- If method not found after searching Object's method dictionary, raise run-time error
- Note similarities and differences with C++

101

Implementing class methods

- Class methods stored in dictionary, just like instance methods
  - Where should class method dictionary be stored?
  - For sake of uniformity, there must be a class object for the class object, where method dictionary held
- *Metaclass* object holds class method dictionary
  - Metaclass = the class of the class
- One metaclass object for each class object
  - Support different class protocols for different classes

102

Metaclasses

- A class object is the (unique) instance of its metaclass
- One metaclass object for each class object and vice versa
- Metaclass and class hierarchies mirror each other
- Both objects created when class defined by programmer
- Metaclass objects hold class method dictionary of class
- Anonymous objects, find by sending message class to class object

Array class **"This returns Array metaclass"**

103

More on metaclasses

- Metaclass objects have class behavior (because they have an instance, the corresponding class object)
- In particular, metaclass objects have superclass pointer, to support symbol super and inheriting class methods
  - This is why metaclass hierarchy and class hierarchy mirror each other
- Metaclass Object class has the root protocol of all class objects (just like class Object has root protocol of all instance objects)
- Metaclass Object class is a subclass of its instance (Object)

104

More on metaclasses

- Three subclasses are interposed between Object and Object class in Smalltalk's inheritance hierarchy
  1. Behavior – Define method dictionary and instance creation protocol (method new)
  2. ClassDescription – Defines InstanceVariableNames: variable and additional info for browser
  3. Class – Defines variable holding name of class. All metaclasses are subclasses of Class
- Together these superclasses of Object class define general behavior and implementation of class objects

105

Metaclass implementation

- Where do class pointers for metaclass objects lead?
- Unique object that must support creation of metaclass (and class) objects
- In this case, one object will do
- This object is a metaclass, bound to shared variable Metaclass
- The instances of Metaclass are all the metaclass objects
- Every time you create a new class, you add an instance to Metaclass

106

Two last questions

1. What is the class of Metaclass?

   Unique metaclass whose only reason for existing is for Metaclass to have a class

   The class of Metaclass is Metaclass class (an anonymous obj)

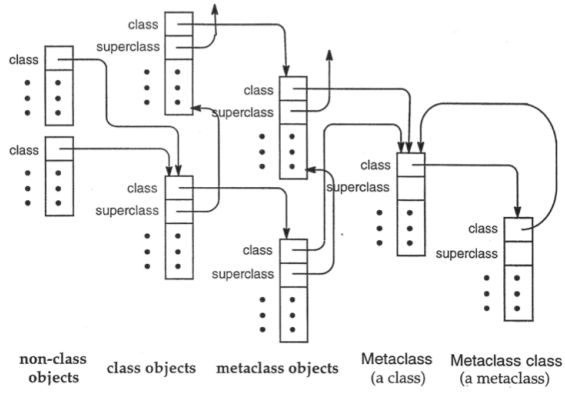
2. What is the class of Metaclass class?

   Since Metaclass class is a metaclass, its class is Metaclass, the class of all metaclasses (of course!)

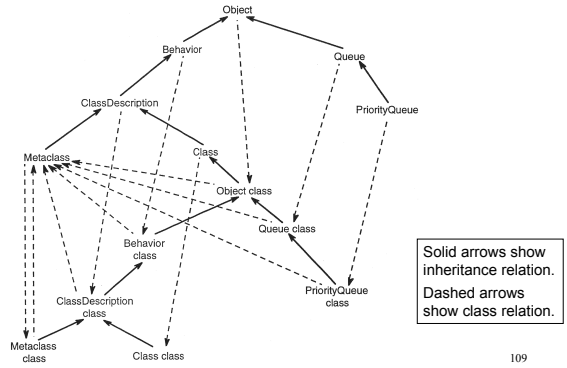
   Point of circularity: Metaclass and Metaclass class are each other's class

107

Class/instance relations (Drake's book)



Class/metaclass hierarchy (Drake's book)



Done with Smalltalk basics

Read Chapters 5 and 6 in Drake's book!

Object instance protocols

Copying

- copy (should be overridden), shallowCopy, …

Comparing

- ==, ~~, = (should be overridden), ~=
- hash (returns integer useful for hashing)
- == and hash are primitive (part of Smalltalk kernel)

Error handling

- doesNotUnderstand:, error: (general error)

Object instance protocols

Class membership

- isMemberOf: (true if arg is class of receiver),
- isKindOf: (arg is receiver's class or a superclass of it),
- respondsTo: (receiver responds to arg, a message)

Accessing

- at:, at:put: (in terms of primitive methods basicAt:, basicAt:put:)

Testing

- isNil, notNil, isBehavior, isNumber, isString, isSymbol, generally return false, suitably overridden in classes

Object instance protocols

Printing

- printString (returns a string, suitable for output; uses printOn: below)
- printOn: (prints on argument stream, you should redefine for your classes)
- isLiteral: (true for arrays, numbers, etc.)

Object class protocols

Instance creation (class methods defined in Object class)

- new, new: (Defined in Behavior, an ancestor of Object class)
- readFrom:, readFromString:, (create instances from a string or a stream), e.g.,

```
|a|
a := Array readFromString: '#( (1 2) 3 (4 5))'.
^a.
```

- Note: Implemented in Object class
- Recall inheritance hierarchy: Object → Behavior → ClassDescription → Class → Object class

Dependents access protocol

- Again, defined in Object
- Support Model-View-Controller (MVC) paradigm
  - A decomposition method for software systems (a software architecture)
  - Divide software into three components (subsystems)
  - Model: Classes implementing the software's logic
  - View: Classes formatting and displaying information for user's benefit (charts, diagrams, etc.)
  - Control: Classes for handling I/O with user
- Automatically propagate information among components

Dependents access protocol

- Perhaps the first realization of the Observer design pattern defined in Gang-of-4 book (Gamma, Helm, Johnson, and Vlissides)

Dependents notification protocol

- Two phase protocol:
  1. Master is informed that it has changed
  2. In response, master sends notification to all dependents
- Grand total: Three sets of methods
  - Dependents list maintenance
  - Master notification (changing protocol)
  - Dependent notification (updating protocol)

Dependents access protocol

- Methods for dependents list maintenance
  - Each instance has a list of dependents (usually empty)
  - Protocol includes addDependent:, removeDependent:, release, dependents
- Methods for master notification
  - self changed, self changed:, self changed:with: send messages update, update: or update:with: to each dependent

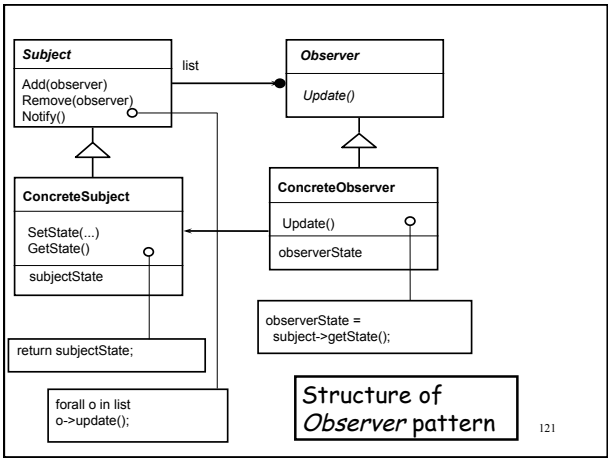
Dependents access protocol

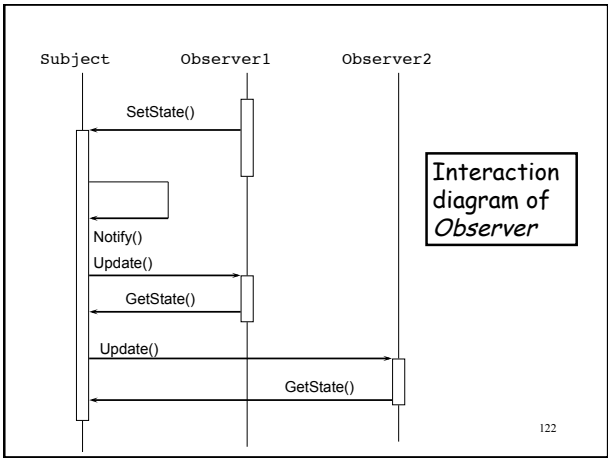
- Methods for dependent notification:
  - Messages update, update: or update:with:
  - broadcast: sends selector (the arg string) to each dependent
  - broadcast:with: sends selector (first arg) with an argument (second arg) to each dependent (messages performUpdate: and performUpdate:with:)

Dependents access protocol

- Some notification and dependent methods:

Changed message to master	Updating message sent to dependent
changed	update: nil
changed: aSymbol	update: aSymbol
changed: aSymbol with: obj	update: aSymbol with: obj
changeRequest	updateRequest
changeRequest: aSymbol	updateRequest: aSymbol
broadcast: aSymbol	performUpdate: aSymbol
broadcast: aSymbol with: obj	performUpdate: aSymbol with: obj





Message handling protocol

- Decide at run-time what message to send to an object
  - avoid "hard-coding" message id in source code
  - perform: takes symbol as argument which indicates selector (e.g., message name) to be sent to receiver
  - perform:with:, perform:with:with:, etc., also send message arguments to be appended to selector, e.g.,

```
|a|
a := #( $a $b $c).
a perform: #at: with: 3.          "returns $c".
```

  - useful, e.g., in response to menu selections

Object's system protocol

- How system actually accesses objects
  - Primitive methods basicAt:, basicAt:put:
  - Do not use these!

Class UndefinedObject

- One instance only, denoted by special symbol nil
- Responds to testing methods (e.g., isNil),
- Class creation message new overridden to return an error
- Copying protocol returns ^self (no new instances)

125

Class Magnitude

- Abstract class aimed at providing basic functions for value types
- Main subclasses: Date, Time, Character, ArithmeticValue, LookUpKey
  - <, =, hash are subclassResponsibility
  - Concrete methods: >, >=, <=, between:and: max: min: are defined in terms of deferred methods
  - Important consequence: All subclasses above respond to these messages

126

Classes Date and Time

- Date has many class methods for general dates
  - nameOfDay:, nameOfMonth: take integer args
  - dayOfWeek:, indexOfMonth: take strings, return integers
  - leapYear: (returns 0 or 1)
  - daysInYear: (returns integer)
- Class variables:
  - MonthNames := #(#January #February…)
  - WeekDayNames := #(#Monday #Tuesday…)

127

Classes Date and Time

- Class messages for Time
  - millisecondClockValue (since 1901)
  - totalSeconds (since 1901)
  - millisecondsToRun: (executes a block and returns the amount of time it took to execute)

128

Classes Date and Time

- Instance creation for Date (class methods)
  - today
  - newDay:month:year:

129

Classes Character and Number

- Lots and lots of interesting methods, beyond time allowed for this course
- Double dispatching is efficient technique for executing arithmetic expressions
  - A realization of Visitor pattern from Go4 book
- Subclass Fraction of Number handles rational numbers accurately (with instance variables numerator and denominator)
- Literal representation for Fraction uses slash character (e.g., 2/3 is instance with numerator 2 and denominator 3)

130

Evaluating arithmetic expressions

- Binary message expressions with a receiver, an operator (+, -, *, /), and an argument

  Example: Arithmetic message: 3 + 5.5 has receiver 3, selector +, and argument 5.5
- Of course, there are many numeric types, each with its own method for addition, subtraction, etc.
  - Integer addition will be different from Float, and so on
  - Expression 3 + 5.5 invokes Integer addition, but expression 5.5 + 3 invokes Float addition
- Method selection based on two data types, not one

131

Evaluating arithmetic expressions

- Pure OO approach to arithmetic expression evaluation must address two problems:
  1. Making sure that result is independent of order of operands
     - Always return 3 + 5.5 = 5.5 + 3 = 8.5
  2. Efficiency of operations
     - Find quickly right code for the two operands
- Competing approaches:
  1. *Coercive generality*
  2. *Double dispatching*

132

Coercive generality

- Assign each numeric type a *generality index*
  - Large integers more general than small integers, etc.
  - Index grows with generality of corresponding class
- Examples of generality (VWNC):

  SmallInteger: 20

  LargeNegativeInteger, LargePositiveInteger: 40

  Fraction: 60

  Float: 80

  DoublePrecisionFloat: 90

133

Coercive generality

- Basic algorithm for 4 arithmetic operations
  - Compare generality of receiver and argument
  - Same generality: Do operation
  - Different generality: Convert lower generality operand to generality of other operand, and do operation
- Implementation uses methods:
  1. generality -- (unary) generality of receiver
  2. coerce: -- (keyword, 1 arg) tries converting argument to class of receiver

134

Implementing coercive generality

- Example of addition for class Fraction uses method retry:coercing:
- Instance method in class Fraction

```
+ aNumber
    "Add receiver and argument aNumber"
    (aNumber isMemberOf: Fraction)
        ifTrue: ["Code for adding fractions..."]
        ifFalse: [ self retry: #+ coercing: aNumber]
```

135

Implementing coercive generality

- Code for retry:coercing:
- Instance method in class Number

```
retry: aSymbol coercing: aNumber
    "Add receiver and argument aNumber"
    (aSymbol == #= and: [(aNumber isKindOf: Number) == false])
        ifTrue: [^false].
    self generality < aNumber generality
        ifTrue: [ ^(aNumber coerce: self)  perform: aSymbol
                                           with: aNumber. ]
    self generality > aNumber generality
        ifTrue: [ ^self perform: aSymbol with: (self coerce: aNumber).]
    self error: '...' .
```

136

Double dispatching

- Receiver of arithmetic operation (numeric instance) sends message encoding both its class and operator invoked to original argument
- Basic algorithm:
  - Receiver of arithmetic operation sends message, encoding operation and its own type, to argument of operation
  - Original argument receives message and does appropriate operation

137

Example of double dispatching

- Consider arithmetic message: 3 + 5.5
- Method for addition in class SmallInteger:

```
+ arg
    arg sumFromInteger: self.
```

- Next message: 5.5 sumFromInteger: 3
- Method sumFromInteger defined in Float knows that:
  - Argument is an integer (3) because of method's name
  - Receiver (5.5) is a float (method is defined in Float class)
  - Two numbers must be added (see method's name)

138

Example of double dispatching

- Given above information, sumFromInteger in class Float will take following actions
  1. Convert integer argument (3) to a float (3.0)
  2. Add receiver (5.5) and conversion result (3.0)
  3. Return sum (8.5) to + method in class SmallInteger
- "Double dispatching" means two methods executed for each arithmetic operation
  - Original receiver and argument are switched in second call
- Advantage: Fast!!!

139

Comparing two approaches

- Double dispatching is much faster (3 messages will get you to desired method)
- Double dispatching requires $O(mn^2)$ methods, for m operations and n data types involved
  - Say you must support 4 arithmetic ops for 4 numeric types
  - Full implementation requires 16 sumFrom…, subtractFrom, etc. methods in each type
  - This is in addition to the 4 basic ops in each type
  - 80 ops total
- Coercive generality cannot be implemented when total ordering does not exist

140

Class Collection

- Testing protocol
  - includes: (membership test, returns Boolean)
  - isEmpty (returns Boolean)
  - size
  - occurrencesOf:
- Deferred methods:
  - add:
  - do:
  - remove:ifAbsent:

141

Class Collection

- Conversion protocol
  - asBag, asSet, asOrderedCollection
  - asSortedCollection
  - asSortedCollection: <aBlock> (uses two arg block that returns a Boolean to keep new instance sorted)
- Copying protocol
  - copyEmpty: Creates an empty instance of the receiver's class

142

Collection subclasses

- Class Set
  - Implement set abstraction as a hash table
  - Elements cannot be duplicated
- Class Bag
  - Implement set abstraction as a hash table
  - Elements can be duplicated
- Class Dictionary
  - A set with elements of class Association

143

Dictionary protocol

- Methods at: and at:put: work on keys and values

  aDictionary at: customer put: customerInformation.
- Method do: works only on values
- Other Dictionary methods
  - keys (a set) and values (a bag)
  - keysDo: (iterate on keys)
  - associationsDo: (iterate on associations)

144

Class SequenceableCollection

- A subclass of Collection
- Main subclasses OrderedCollection, LinkedList, Interval, ArrayedCollection
- Main SequenceableCollection methods
  - first
  - last
  - indexOf: identityIndexOf: (return 0 if not found)
  - indexOf:ifAbsent:
- Rich copying protocol (uses shallow copying, though)

145

Class OrderedCollection

- Concrete subclass of SequenceableCollection
- Stored contiguously
- Main OrderedCollection methods
  - addFirst: addLast:
  - addAllFirst: <aSubcollection>
  - addAllLast: <aSubcollection>
  - removeFirst removeLast
  - removeFirst: <anInteger>, removeLast: <anInteger>
  - before: and after:

146

Implementation of OrderedCollection

- Method new: gives initial size
- Instance variables firstIndex, lastIndex give positions of starting and ending points of used portion of collection
- Method grow used when initial allocation used completely
- Method size gives only portion used
- Methods at: at:put: redefined because of offset

147

Method at: in OrderedCollection

- From Cincom Smalltalk:

```
at: anInteger

    "Answer the element at index anInteger.   at: is used by a
     knowledgeable client to access an existing element"

    anInteger isInteger ifFalse: [^self nonIntegerIndexError: anInteger].
    ^(anInteger < 1 or: [anInteger + firstIndex - 1 > lastIndex])
        ifTrue: [self subscriptBoundsErrorFor: #at: index: anInteger]
        ifFalse: [super at: anInteger + firstIndex - 1]
```

148

A deepCopy example

```
Object subclass: #BST
    instanceVariableNames: 'left right value '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Tree-examples'

copy
    |temp|
    (value isNil) ifTrue: [^BST new.]
    ifFalse: [temp := BST new.
            temp value: value.
            (left isNil) ifFalse: [temp left: (left copy)].
            (right isNil) ifFalse: [temp right: (right copy)].].
  ^temp.
```

149

Class SortedCollection

- A subclass of OrderedCollection, ensuring that instances are always kept sorted
- Instance variable sortBlock bound to two argument block used for comparisons in sorting method (quick sort algorithm) and insertion methods
- Class variable DefaultSortBlock bound to default block: [:x :y | x <= y]
- Class message sortBlock: creates new instance with argument block
- Instance message asSortedCollection converts all other types (arg is block again)

150

Class SortedCollection

- Insertion message that may break ordering of elements are disallowed
  - at:put:, addFirst:, addLast:, etc. all overridden as ^self shouldNotImplement

151

Additional Collection subclasses

- ArrayedCollection: Abstract superclass of contiguously-stored, fixed-length collection subclasses
- Element add and remove messages are disallowed
  - add:, addFirst:, addLast:, remove: etc. all overridden as shouldNotImplement
- Main (concrete) subclasses
  1. Array
  2. String (element type = Character)
  3. Symbol
  4. Text

152

Smalltalk I/O

- Handled by Stream classes, similar to most modern programming languages
- Both internal and external streams supported
- Full discussion is beyond the scope of this course
- Binary Object Storage System (BOSS)
  - A group of external stream classes that implement persistent objects
  - Useful to store objects on file
- Class Random

Done with Smalltalk!

Read Chapters 7 and 8 in Drake's book!