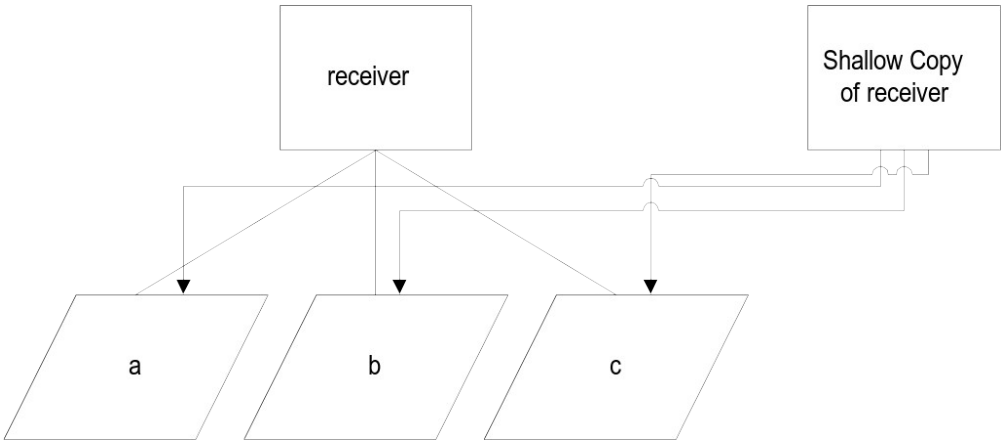
Lecture 7: The Object Class

- The Object class is the main class from which all other classes are derived.
- Any and every kind of object in Smalltalk can respond to the messages defined by the Object class
- All methods of the Object class are inherited to overridden
- **Functionality of an object**
  - Determined by its class
  - Two ways to test functionality
    - Comparing object to a class or superclass to test membership or composition
      - `receiver isKindOf: aClass`
        - tests if the receiver is a member of the hierarchy of aClass
        - `anInteger isKindOf: Integer` returns true
      - `receiver isMemberOf: aClass`
        - tests if the receiver is of the same class
        - `anInteger isMemberOf: Magnitude` returns false
      - `receiver respondsTo: aSymbol`
        - tests if the receiver knows how to answer aSymbol
        - `anInteger respondsTo: #sin` returns true
        - `anInteger respondsTo: #at:` returns false
    - Querying the object for its class
      - `receiver class`
      - `#(1 2 3) class` returns Array
- **Comparison of objects**
  - Comparison and equivalence are very similar, but should not be confused
  - == is used to test if the receiver and argument are the same object
    - `#(a b c) class == Array` returns true
    - `#(a b c) == #(a b c) copy` returns false
  - = is used to test if the receiver and argument represent the same component
    - `#(a b c) class = Array` returns true
    - `#(a b c) = #(a b c) copy` returns true
  - Other comparison operations
    - `receiver ~= anObject`
      - Not equal
    - `receiver ~~ anObject`
      - Not Equivalent
    - `receiver hash`
      - `hash` provides a nice way of telling objects apart, too much trust should not be placed in comparing objects of the same class, as hash is often trivialized (as in the example below, `Array` uses size has the hash function).
      - Ex:

        ```
        a := 3.147 hash. ← 132
        b := 3.14 hash. ← 287
        c := #(1 2 3) ← 3
        d: = #(3 4 5) ← 3
        ```
    - `receiver hashMappedBy: map`
- **Copying objects**
  - `deepCopy` has been removed since VisualWorks 1.0
  - Two methods for copying:
    - `copy` returns another instance just like the receiver. Usually `copy` is simply a shallow copy, but some classes override it.
      - Does not copy the objects that the instance variables contain, but ciopies the “pointer” to the objects.

- `shallowCopy` returns a copy of the receiver which shares the receiver's instance variables. This allows two objects to share one set of instance variables.



- `deepCopy` must be implemented in the rare cases in which it is needed
  - How should this be done? Create new instances of the member objects, then assign them to the new object.
- Example, shallow copies of arrays.:

```
| array1 array2 object1 object2 object3|
object1 := #('A').
object2 := #('B').
object3 := #('C').
array1 := Array with: object1 with: object2 with: object3.
array2 := array1 copy.
(array1 at: 1) at: 1 put: 'D'.
array1 inspect.
array2 inspect.
```

- **Accessing indexed variables**
  - `at: index` returns the object at index
  - `#(a b c) at: 2` returns 'b'

- `at: index put: anObject` puts anObject at index of the receiver
  - returns anObject
  - `#(a b c) at: 4 put: #d` returns 'd'
- `basicAt: index` is the same as `at: index` but cannot be overridden
- `basicAt: index put: anObject` – Same as above
- `size` returns the number of index in the receiver
  - `#(a b c d) size` returns 4
- `basicSize` same as size, but cannot be overriden
- `readFromString: aString` creates an object based on the contents of aString
- `Yourself` returns the receiver

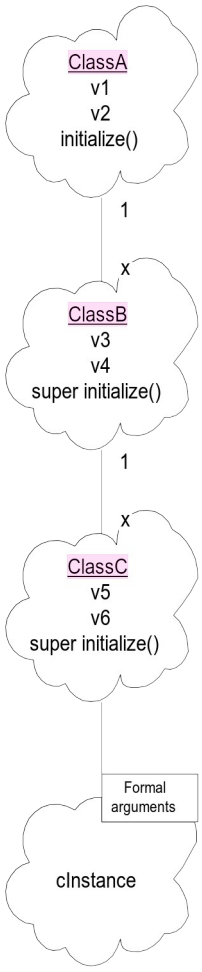
Lecture 8: Messages & Methods

- Messages are what is passed between objects
- Methods are what is defined in a class to act on an instance of the class
- **Message Expressions**
  - Receiver-object message-selector arguments
  - Unary
    - Receiver message-selector
    - Parsed left to right
    - Ex: `Time now.`
    - Ex: `8 squared.`
  - Binary
    - Receiver message argument
    - Parsed left to right
      - Ex: `1 + 2 * 3.` (Note: returns 9)
    - Parenthesis do the expected
      - Ex: `1 + (2 * 3).` (returns 7)
  - Keyword
    - Receiver message arguments
    - Ex:

      ```
      aString = 'ABC'.
      aString at: 3 put: $D.
      ```

      (Note: returns 'D', aString equals #(ABD))
      - Important to note that `'ABC' at: 3 put: $D` returns `$D`
      - `aString` is the object
      - `at` is the keyword message-selector
      - `3` is the argument
        - `'C'` is the object
        - `put` is the keyword message-selector
        - `$D` is the argument ($D is a literal)
  - Parentheses change order
  - Precedence *always* left to right
  - Separated by periods, unless temp variable declaration or comment
- **Method Lookup**
  - A method and a message-selector must be exactly the same, or no method will be found by the method lookup
  - The methods defined for the receiver's class first
  - If no match, the superclass is searched
  - Path continues through Object unless a method is found.
  - `self` refers to receiver, lookup starts within the class of the reciever
  - `super` refers to receiver, lookup starts in superclass of receiver

- Example
  - What is the order of initialization? (v1, v2, v3, v4, v5, v6)
  - Why? (initialize()'s look to superclass, then return to call their own initialize because they are implemented as `super initialize)`



Lecture 9: Variables and Return values

- A variable is a reference to any kind of object
- **Method arguments**
  - Accessibility: private
  - Scope: statements within the method
  - Extent: life of the method
  - Declaration: define with method name on first line of method `(name: aString)`
  - Assignment: Assigned by sender of the message (`aNode name: 'Node2'`)
  - Accessing : Directly by name
  - Ex: `anInteger raisedToInteger: 4.`
    - To understand this, it is easiest to look at literals and constants used as method arguments. The argument 4 is only visible to the object and the method- it cannot be accessed outside of the method. This coincides with the life of the variable, as it dies after the method call.
- **Temp variables**
  - Accessibility: private
  - Scope: statements within the method
  - Extent: life of the method
  - Declaration: use vertical bars
  - Assignment: use 'gets' operator
  - Accessing : Directly by name
  - Example:

```
cubeWithInteger
| x |
       x = self raisedToInteger: 3.
```

    - x is created in the method using the vertical bars, and is released once the method is finished.
- **Instance variables**
  - Accessibility: private
  - Scope: Instance methods of the defining class & subclasses
  - Extent: life of the instance
  - Declaration: define on the instance side of the class template

```
Object subclass #Node
       InstanceVariableNames: 'name nextNode'
       ClassVariableName: ''
       PoolDictionaries:''
       Category: ''
```

  - Assignment: write a method that sets the value
  - Accessing : write a method that gets the value
    - Can be either named or keyed
      - If keyed, then they can be accessed through ordinary `at:put:` messages
- **Class instance variables**
  - Accessibility: private
  - Scope: Class methods of the defining class & subclasses
  - Extent: life of the defining class
  - Declaration: define on the class side of the class template

```
Account class
       InstanceVaraibleNames: 'interestRate'
```

  - Assignment: write a class **initialize** method in the defining class and all of its subclasses
  - Accessing : Write a class method that returns the value

- **Class Variables**
  - Accessibility: shared
  - Scope: Instance and class methods of the defining class & subclasses
  - Extent: life of the defining class
  - Declaration: define on the instance side of the class template
  - Assignment: write a class **initialize** method
  - Accessing : Write a class method that returns the value
  - Always begin with uppercase
- **Global Variables**
  - Accessibility: shared
  - Scope: all objects, all methods
  - Extent: while in Smalltalk dictionary
  - Declaration: with assignment
  - Assignment: with declaration

    ```
    Smalltalk at: #MyTranscript put: TextCollector new.
    ```
  - Accessing : Directly by name
  - Don't use, unless absolutely necessary. Bloated images, anti-OO code, incorrect code are the consequences.
- **Return Values**
  - Method always returns an object
    - Default return value is `self`.
    - Use ^ to explicitly return a different object
    - Can use both implicit and explicit returns in a method (i.e. in a conditional)

Lecture 10: Blocks and Branching

- **Blocks**
  - Contains a deferred sequence of expressions
  - Used in many of the control structures
  - Instance of `BlockClosure`
  - Returns the result of the last expression (similar to lisp)
    - Ex: `[3+4. 5*5. 20-10]` returns a Home Context with value of 10.
      - `[3+4. 5*5, 20-10] value` returns 10.
    - Ex: `['Visual','Works'] value` returns 'VisualWorks' (comma is binary method)
  - Syntax

    ```
    [:arg1 :arg2 … :arg255 | |temp vars| executable expressions]
    ```
  - A block can contain:
    - 0 to 255 arguments
    - temp variables
    - executable expressions
  - Block with no arguments: sequence of actions takes place every time value message is received by the block
  - Block with arguments: action takes place every time block receives messages value, value: value, etc.
  - block variables scope is only within defining block
  - NOTE: temp variables inside declared blocks have not been successfully tested with Smalltalk Express or GNU Smalltalk.
  - Examples
    - `[ :x :y | x + y / 2 ] value: 10 value: 20` (returns 15)
    - `[ |x| x := Date today. x day ]` value (returns the day to today's Date)
      - `[Date today day ] value` returns same value & is more succinct
    - `[ :y | |x| x := y *2. x * x ] value: 5` (returns 100)
    - `#(5 10 15) collect: [ :x | x squared]` (returns #(25 100 255)
      - sends 1 argument 3 times and collects the results into an array
- **Class Boolean**
  - Classes `True` and `False` are subclasses of Boolean
  - Logical operators can be used for testing
    - The 'and' operator: `&`
    - The 'or' operator: `|`
    - The negation operator: `not`
      - `not` is a unary operator
    - The equivalence operator: `eqv`
    - The exclusive or operator: `xor`
  - The `Boolean` classes are used in branching
  - `and`: and `or`: methods used with alternative blocks returns values of alternative blocks
  - `ifTrue:` and `ifFalse` are used with blocks to provide if-then support
    - can be used together in either order, or separately
- **Branching (Control Structures)**
  - Boolean classes `True` and `False` understand keyword messages:
  - `ifTrue:`
  - Ex: (result: anArray = #('a' 'b' 'c'))

    ```
    | anArray |
    anArray := #('a' 'b' 'd').
    (anArray at: 3) asString > 'c'
       ifTrue: [anArray at: 3 put 'c'].
    ```

  - `ifFalse:`
  - `ifTrue: ifFalse:`
  - Ex: (result: upperArray = #('A' 'B' 'D'))

```
| anArray upperArray |
anArray := #('a' 'B' 'd').
upperArray := Array new.
upperArray := anArray collect:
    [:aString | aString asUpperCase = aString
          ifTrue: [aString]
          ifFalse: [aString asUpperCase]].
```

- `ifFalse: ifTrue:`
- These messages demand zero argument blocks as their arguments
  - Ex:

```
abs
      ^self < 0
            ifTrue: [0 - self]
            ifFalse: [self]
```

  - What happens here?
    - `self` is compared to 0
    - corresponding block is executed
    - `(-self)` or `self` is returned depending on which block was executed
- Repetition
  - `timesRepeat:` message
    - Ex: `5 timesRepeat [ Transcript show: 'This is a test'; cr ]`
  - `to:` message (similar to for loop)
    - Ex: `1 to: 15 by: 3 do: [:item | Transcript show: item printString; cr]`
- Conditional Iteration
  - Blocks can be used as arguments in messages and can be receiver objects
  - `whileTrue:` and `whileFalse:` messages
    - get sent to blocks. `ifTrue:` and `ifFalse:` get sent to Boolean
    - Ex (receiver):

```
Initialize: myArray
| index |
index := 1.
[ index <= myArray size]
      whileTrue:
      [myArray at: index put: 0.
      index := index + 1]
```

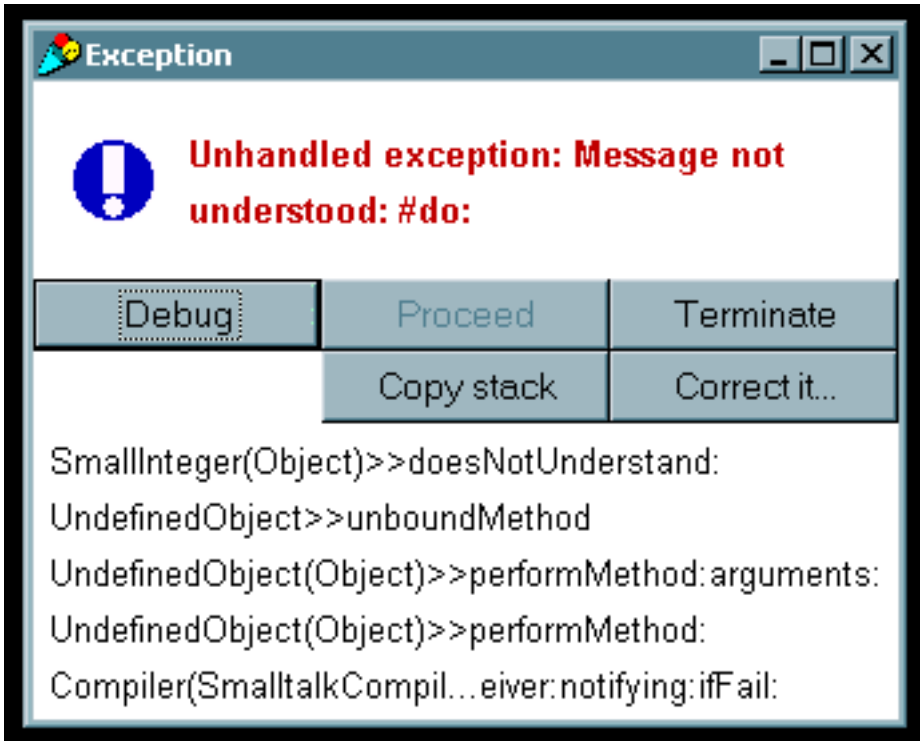
    - Ex (argument):

```
Initialize: myArray
| index |
index := 1.
[ myArray at: index put: 0.
      index := index + 1.
      index  <= myArray size] whileTrue;
```

Lecture 11: Reporting Errors and Debugging techniques

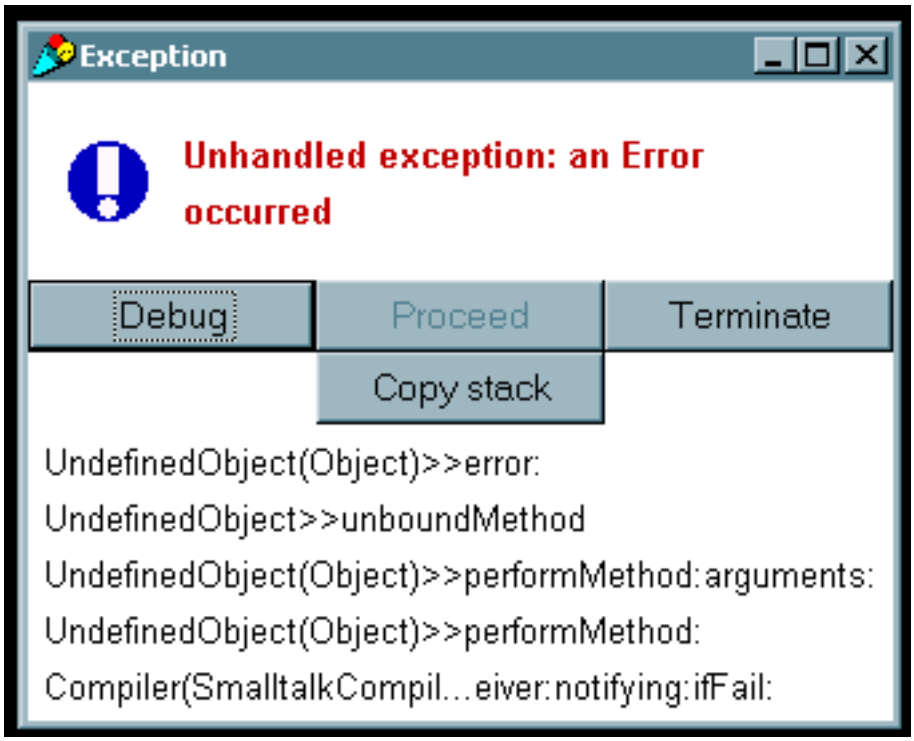
- **Error Handling**
  - Smalltalk's interpreter handles all errors
  - An error is reported by an object sending the interpreter the message `doesNotUnderstand: aMessage`
  - There are some common error messages supported in the Object class, but implementation is dependant on the system
    - `doesNotUnderstand: aMessage`
      - Lets look at an example of trying to use a method that an object of the class SmallInteger cannot understand.

```
| anInteger |
anInteger := 0.
self doesNotUnderstand: (anInteger do:[]).
```

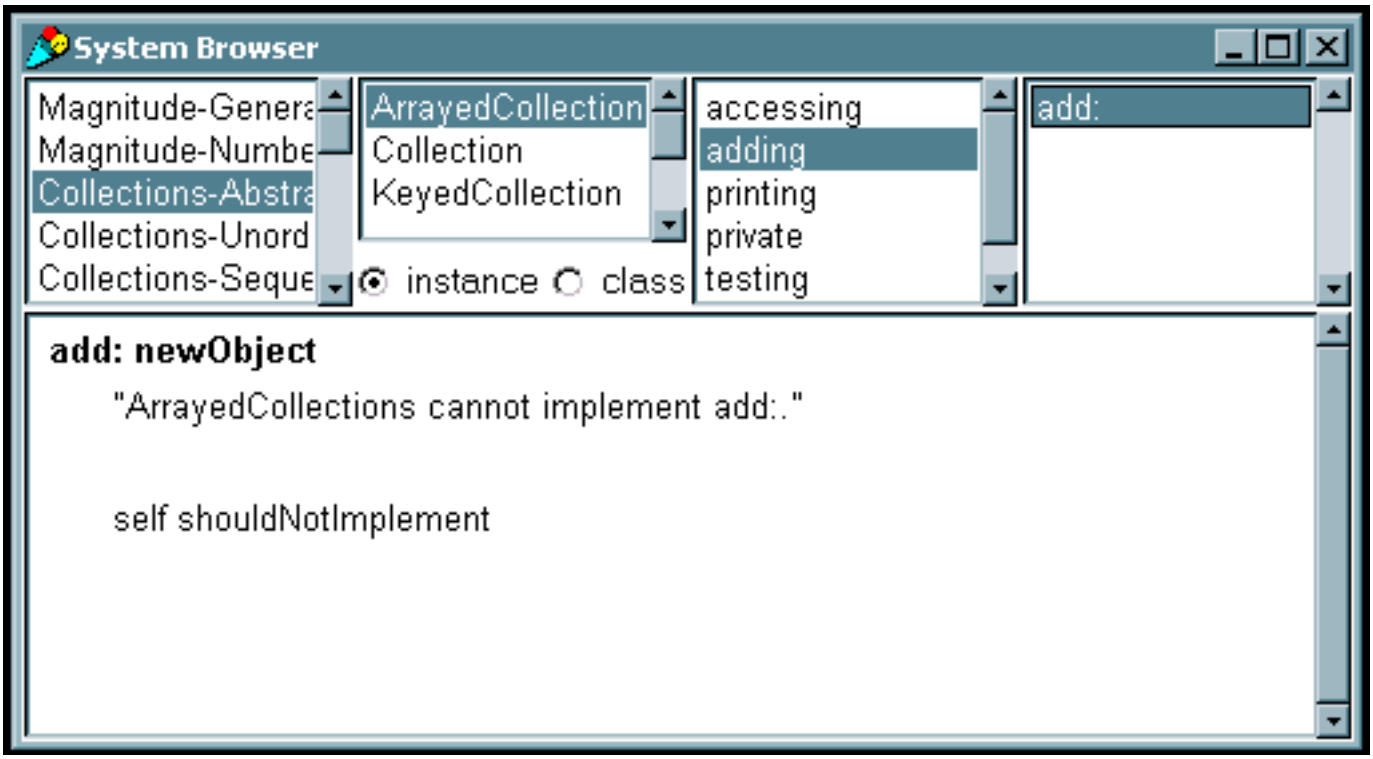


- `error: aString` uses aString in the report the user sees

```
self error: 'an Error occurred'.
```



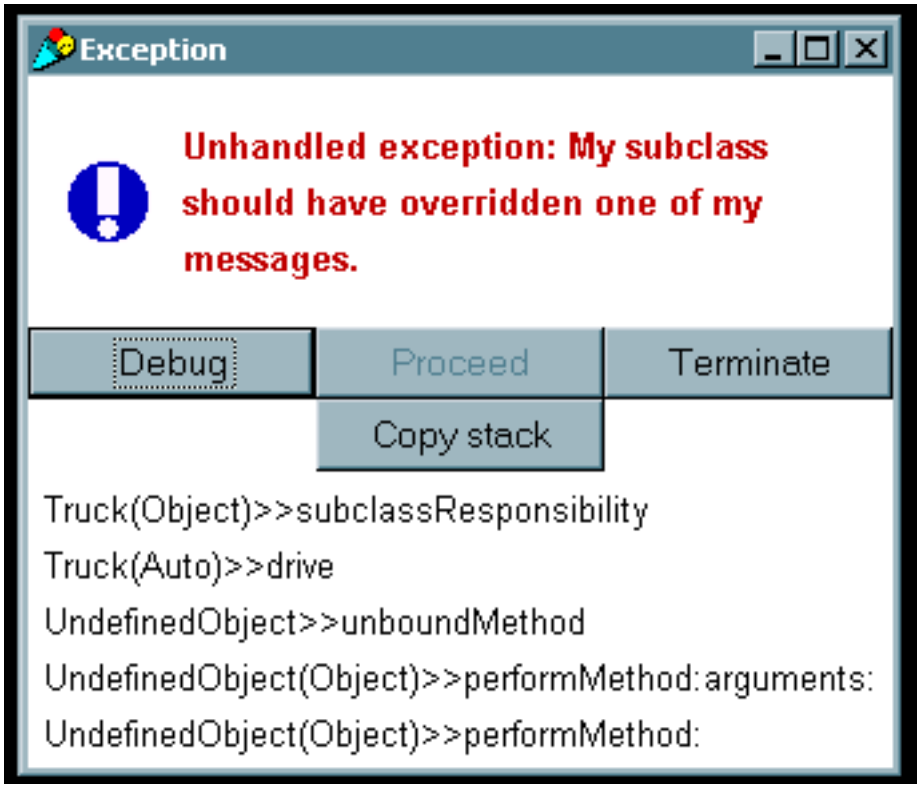
- `primitiveFailed` reports that a method implementing a system primitive failed
- `shouldNotImplement` reports that the superclass says a method should be implemented in the subclasses, the subclasses do not handle it correctly.
  - This method is utilized throughout the collection classes. If we look at the Array class, we'll see this method is used inside the add: method.
    - Arrays are statically sized collections, and the add: method is used to grow the size of collections.



- `subclassResponsibility` reports that a subclass should have implemented the method
  - This method is used extensively in abstract classes. This method allows all objects in the hierarchy to implement a method differently, while reporting an error if the method was not defined.
    - Example: Class Auto defines a method `drive`, but only calls the `subclassResponsibility` method. We define a subclass Truck, but do not define the method `drive`. If we then define a Truck object and call the drive method, then Smalltalk will try to pass the drive message up the tree until a parent class knows how to implement it- in this case displaying a `subclassResponsibility` error message.

```
Object subclass: #Auto
    instanceVariableNames: 'speed '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Examples-General'!


!Auto methodsFor: 'creation'!

withSpeed: aSpeed

    self subclassResponsibility! !

!Auto methodsFor: 'driving'!

accelerate

    speed := speed + 1.!

decelerate
```

```
    speed := speed + 1.!

drive

    self subclassResponsibility! !

Auto subclass: #Truck
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Examples-General'!


!Truck methodsFor: 'creation'!

withSpeed: aSpeed

    speed := aSpeed.! !


| aTruck |
aTruck := (Truck new) withSpeed: 5.
aTruck drive.
```



- **Message Handling**
  - Used to send messages to objects, usually only created when an error occurs
  - `perform:` is the method called to pass messages, takes many different arguments, or just aSymbol.
    - A good example of this can be seen in the Goldberg book (page 245).
      - Suppose we wish to write a simple calculator that checks to make sure each operator is a valid operator.

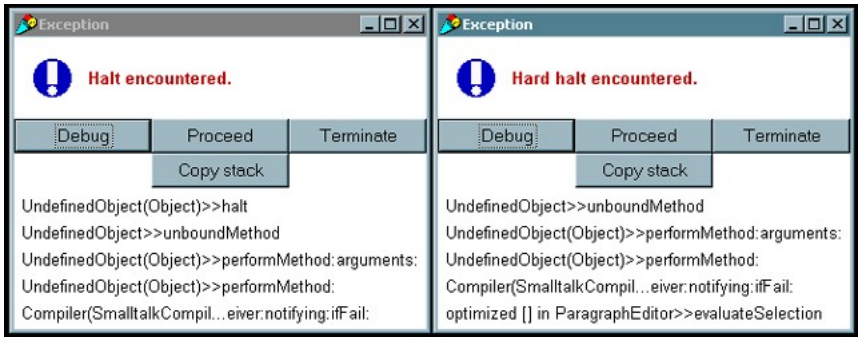
```
Object subclass: Calculator
   instanceVariableNames: 'result operand'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples-General'!

!Calculator methodsFor: 'creation'!

new
   ^super new initialize


!Calculator methodsFor: 'accessing'!

result
   ^result

!Calculator methodsFor: 'calculating'!

apply: operator
   (result respondsTo: operator )
          ifFalse: [self error: 'operation not understood'].
   operand isNil
          ifTrue: [result := result perform: operator]
          ifFalse:
               [result := result perform: operator with: operand]

clear
   operand isNil
          ifTrue: [result := 0]
          ifFalse: [operand := nil]

operand: aNumber
   operand := aNumber

!Calculator methodsFor: 'private'!

initialize
   result := 0
```

- The following code shows an example of how to use the class Calculator

```
| aCalculator |
aCalculator := Calculator new. "result = 0"
aCalculator operand: 3.
aCalculator apply: #+. "result = result + 3 ← 3"
aCalculator apply: #squared. "result = 3 ^ 2 ← 9"
aCalculator operand: 4.
aCalculator apply: #-. "result = result - 4 ← 5"
```

- System Primitive Messages
  - Messages in class Object used to support system implementation
  - `InstVarAt: anInteger` and `instVarAt: anInteger put: anObject` are examples which are used to retrieve and store instance variables.
  - In general, these will not be used, but are important to how Smalltalk works.
- **Class UndefinedObject**
  - the object `nil` represents a value for uninitialized variables
  - `nil` also represents meaningless results
  - Testing an object's initialization is done through `isNil` and `notNil` messages

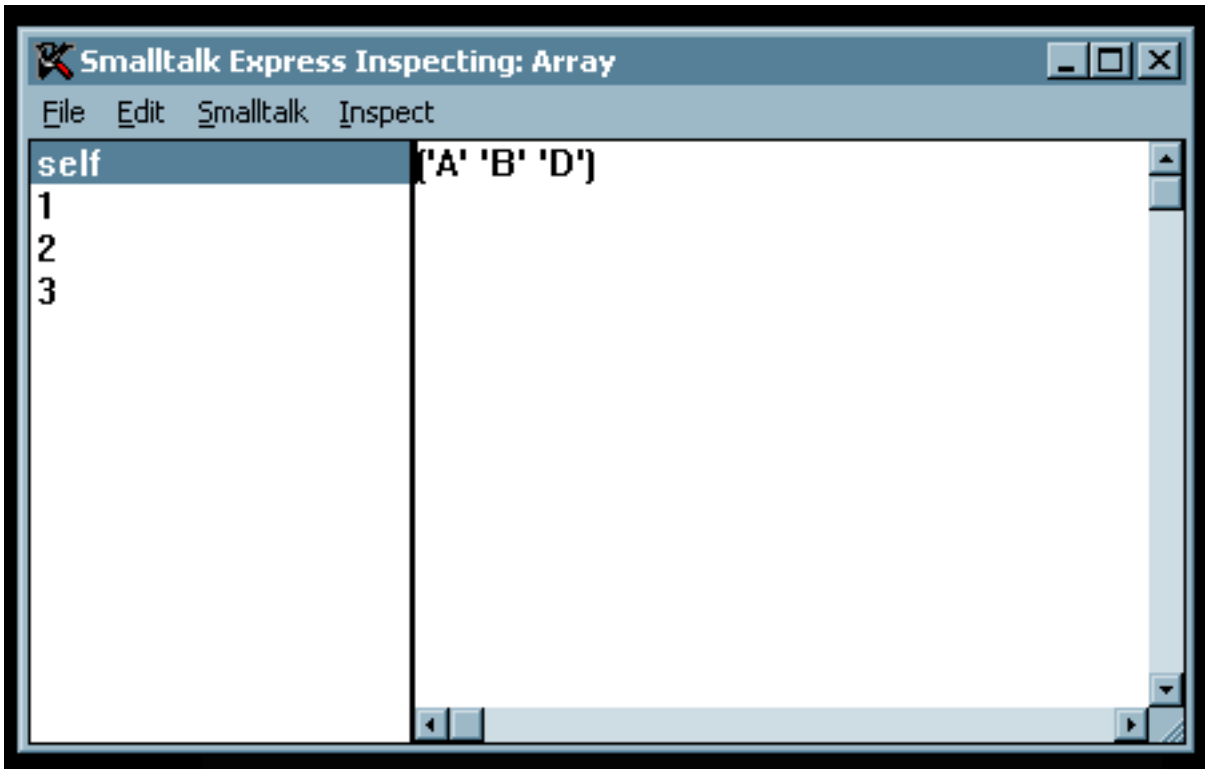
- **Debugging**
  - Smalltalk has a small set of methods for error handling and are useful to debugging. These messages are implemented by passing Signals.
    - Whats a signal? A signal is an Exception passed to the VM. A signal will stop the execution and show a window with a message and has several qualities, such as whether or not the exception is proceedable. An example of this is the `halt: aString` message, which raises a `haltSignal` with the context of the receiver and the error message of `aString`.
      - `errorSignal`
      - `messageNotUnderstoodSignal`
      - `haltSignal`
      - `subclassResponsibilitySignal`
  - `confirm`: similar to `notify:` method, brings up a window asking for confirmation, not in all implementations . In VW 3.0 and above, the `confirm`: method belongs to class Dialog.
    - Ex: `(Dialog confirm: 'Quit ?') ifTrue:[aBlock].`
  - **`halt`**
    - `halt` shows the debug window, with 'halt encountered' or similar message as the primary error. Useful for setting a breakpoint to check value of variables
      - Ex: `self halt.`
      - Ex: It is possible to stop other objects `Transcript halt.`
    - `halt: aString` implements halt, bringing up a window with the label from `aString`
    - `halt`: appears very similar to notify:, but with one difference. `halt`: allows invariants related to multiple processes to be restored.
      - How could I do this? When execution halts, use the workspace to restore the values.
    - `hardHalt` halts the execution without passing a signal.



  - `notify: aString`: shows a message dialog window with aString as the label. This method is not available in Smalltalk Express.
    - Ex: `self notify: 'custom error message'.`
  - `inspect` displays a window showing the object and all of its variables
  - Ex:

```
| anArray upperArray |
anArray := #('a' 'B' 'd').
upperArray := Array new.
upperArray := anArray collect:
      [:aString | aString asUpperCase = aString
            ifTrue: [aString]
```

```
            ifFalse: [aString asUpperCase]].
upperArray inspect.
```



Lecture 12: Designing and implementing classes

- **Steps to develop a specification**
  1. Decide what we want the program to do
  2. Decide on the data structures
  3. Decide on the operations we want to apply to these data structures
- **The message protocol**
  - Class Protocol: A description of the protocol understood by a class
  - Typically contains protocols for creating and initializing new instances of the class
  - Instance Protocol: A description of the protocol understood by instances of a class
  - Messages that may be sent to any instance of the class
  - **Steps to implementing a class**
  1. Deciding on a suitable representation for instances of the class.
  2. Selecting and implementing efficient algorithms for the methods or operations
  3. Deciding on class variable and instance variables
- **Describing a class**
  - Class name: A name that can be used to reference the class
  - Superclass name: name of the superclass
  - Class variables: variables shared by all instances
  - Instance variables: variables found in all instances
  - Pool dictionaries: Names of lists of shared variables that are to be accessible to the class and its instances. Can also be referenced by other unrelated classes
  - Class methods: operations understood by the class
  - Instance methods: operations that are understood by instances
  - Example: A class for complex numbers
    - Step 1: What do we want to be able to do?
      - Specify real and complex parts
      - Do simple operations of complex and real parts
    - Step 2: What do we want to use?
      - Specify real and complex parts
    - Step 3: How are we going to use the data structures?
      - Creating a complex number
      - Accessing complex and real parts
      - Adding and Multiplying Complex numbers
    - The Class Description (for more detail refer to LaLonde pages 44-45)

```
Class Complex
Class name                  Complex
Superclass name                    Object
Instance variable names     realPart imaginaryPart

Class methods

Instance creation

newWithReal: realValue andImaginary: imaginaryValue
       "Returns an initialized instance"
       | aComplex |
       aComplex := Complex new.
       aComplex realPart: realValue;
              imaginaryPart: imaginaryValue.
              ^aComplex


accessing

realPart
       "Returns the real component of the reciever"
       ^realPart
```

```
imaginaryPart
       "Returns Imaginary part"
       ^imaginaryPart

operations

+ aComplex
       "Returns the receiver + aComplex"
       | realPartSum imaginaryPartSum |
       realPartSum := realPart + aComplex realPart.
       imaginaryPartSum := imaginaryPart + aComplex imaginaryPart.
       ^ Complex newWithReal: realPartSum andImaginary:
              imaginaryPartSum.

* aComplex
       "Returns the receiver * aComplex"
       | realPartProduct imaginaryPartProduct |
       realPartProduct := (realPart * aComplex realPart) -
                     (imaginaryPart * aComplex imaginaryPart).
       ComplexPartProduct := (realPart * aComplex imaginaryPart) +
                     (imaginaryPart * aComplex realPart).
       ^ Complex newWithReal: realPartProduct andImaginary:
              imaginaryPartProduct.
```

- The following code shows how to use this new class. The code computes the magnitude of the complex number. After multiplying the number by its conjugate, there is only a real part, so we just take the square root.

```
| aNumber |
aNumber := (Complex new) newWithReal: 1 andImaginary: 1.
aNumber := aNumber * (Complex new)
       newWithReal: (aNumber realPart)
       andImaginary: (0 - aNumber imaginaryPart).
(aNumber realPart) sqrt.
```