More Smalltalk

CSE 413: Programming Languages

Michael Ringenburg

miker@cs.washington.edu

Today's Plan

- Brief review of Wednesday's lecture
- Blocks and control structures
- Self, super, inheritance and dynamic dispatch
- A Smalltalk case study in object-oriented design

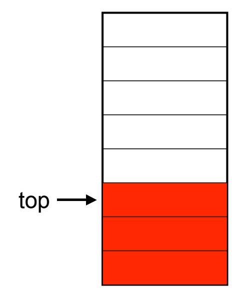
Review - A Stack Class

```
Object subclass: #Stack
  instanceVariableNames: 'anArray top'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'CSE 413-Stack Example'
```

Stack Methods

```
push: item
  top := top + 1.
  anArray at: top put: item
```

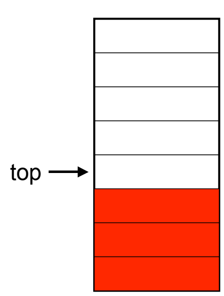
```
pop
  | item |
  item := anArray at: top.
  top := top - 1.
  ^item
```



Stack Methods

```
push: item
  top := top + 1.
  anArray at: top put: item
```

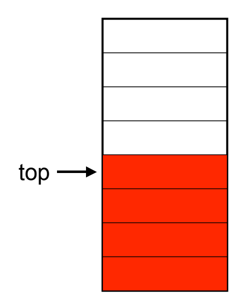
```
pop
  | item |
  item := anArray at: top.
  top := top - 1.
  ^item
```



Stack Methods

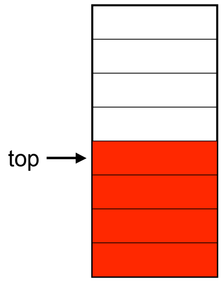
```
push: item
  top := top + 1.
  anArray at: top put: item
```

```
pop
  | item |
  item := anArray at: top.
  top := top - 1.
  ^item
```



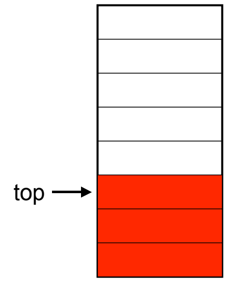
Stack Methods

```
push: item
  top := top + 1.
  anArray at: top put: item

pop
  | item |
  item := anArray at: top.
  top := top - 1.
  ^item
```



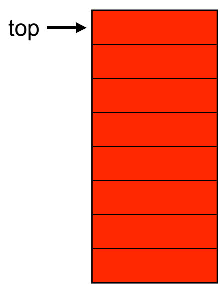
Stack Methods

```
push: item
  top := top + 1.
  anArray at: top put: item

pop
  | item |
  item := anArray at: top.
  top := top - 1.
  ^item
```

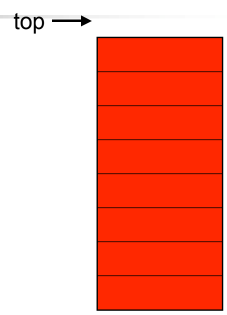


What If The Array Is Full?

```
push: item
  top := top + 1.
  anArray at: top put: item

pop
  | item |
  item := anArray at: top.
  top := top - 1.
  ^item
```



What If The Array Is Full?




```
push: item
  top := top + 1.
  anArray at: top put: item

pop
  | item |
  item := anArray at: top.
  top := top - 1.
  ^item
```

Adding Error Checking

```
push: item
  | save |
  top := top + 1.
  top > anArray size ifTrue:
   [save := anArray.
    anArray := Array new: 2 * save size.
    1 to: save size do:
     [:k | anArray at: k put: (save at: k)]].
  anArray at: top put: item
```

Blocks

- Blocks are Smalltalk objects that contain unevaluated code.
- The unevaluated code in a block may take arguments.
- Blocks can be passed around as arguments to messages.
- Syntax for blocks:

```
[ :arg1 :arg2 | <statement sequence> ]
```

Example Blocks

```
[ :item1 :item2 | item1 print .
                  item2 print ]

[ :i | x := x + i ]

[ :x :y | x + y ]

[ 'hello world' ]
```

Block Details

- Sending the `value` message to a block causes the code to be evaluated.
  - `['hello world'] value`
  - `[:x :y | x + y] value:1 value:3`
  - `[:x :y :z|x+y+z] valueWithArguments: #(1 5 6)`
  - If a block takes more than 4 arguments, we must use the valueWithArguments form.
- Evaluated blocks return the result of their last expression (unless they contain a return(^)—see next slide).

Block Scoping

```
foo: aBlock
  | x y |
  y := 3.
  x := 5.
  aBlock value
  ^ x
```

```
bar
  | x y z|
  x := 10.
  y := 11.
  z := self foo:[x := y]
  ^ x
```

- Blocks are *lexically-scoped*:
  - Variables bindings are determined by the scope the block is created in, *not* the scope the block is evaluated in.
  - Returns (^) cause the method the block was created in to return.

Block Scoping

```
foo: aBlock
  | x y |
  y := 3.
  x := 5.
  aBlock value
  ^ x
```

```
bar
  | x y z|
  x := 10.
  y := 11.
  z := self foo:[^5]
  ^ x
```


- Blocks are *lexically-scoped*:
  - Variables bindings are determined by the scope the block is created in, *not* the scope the block is evaluated in.
  - Returns (^) cause the method the block was created in to return.

Control Structures

- Control structures in Smalltalk are implemented as messages that take blocks as arguments.
- Conditionals are implemented as messages to instances of the `True` and `False` classes.
- While loops are implemented as messages to blocks that evaluate to `true` or `false`.
- For loops are implemented as messages to integer objects.

ifTrue and ifFalse messages

Example:

```
x = 0 ifTrue: ['Can't divide by 0' print]
      ifFalse: [y := 1.5 / x]
```

Methods for `True` class:

```
ifTrue: block
  ^ block value
```

```
ifFalse: block
  ^ nil
```

```
ifTrue: tBlock ifFalse: fBlock
  ^ tBlock value
```

Class Exercise

Class exercise - implement the corresponding methods
For the `False` class:

Class Exercise

Class exercise - implement the corresponding methods
For the `False` class:

```
ifTrue: block
  ^ nil
```

```
ifFalse: block
  ^ block value
```

```
ifTrue: tBlock ifFalse: fBlock
  ^ fBlock value
```

whileTrue and whileFalse

Example:

```
[x < 10] whileTrue: [x print. x := x+1]
[x < 0] whileFalse: [y := y+1. x := x-2]
```

The `whileTrue` method for `Block` class:

```
whileTrue: bodyBlock
  self value
    ifTrue: [bodyBlock value.
             self whileTrue: bodyBlock].
```

whileTrue and whileFalse

Example:

```
[x < 10] whileTrue: [x print. x := x+1]
[x < 0] whileFalse: [y := y+1. x := x-2]
```

The `whileFalse` method is analogous:

```
whileFalse: bodyBlock
  self value
    ifFalse: [bodyBlock value.
              self whileFalse: bodyBlock]
```

For Loops

- For loops in Smalltalk use the `to` and `do` messages.
- The `to` message creates a collection containing all the integers between the receiver and the argument.
- The `do` message iterates over a collection. Each iteration uses the next element of the collection as an argument to a value message sent to the block. Example:

```
1 to: 10 do: [ :i | i print ]
```

Inheritance

- Subclasses inherit all variables and methods from their superclass.
- So, we could create a 3D point class from the 2D point class as follows:

```
Point subclass: #Point3D
  instanceVariableNames: 'z'
  classVariableNames: 'originZ'
  poolDictionaries: ''
  category: 'CSE 413-Point Examples'
```

Point3D Methods

- We can create new methods to read and set the z-coordinate.
- We also want to override (i.e., replace) the addition and scaleBy methods with methods that operate on all three coordinates.
- However, the existing addition and scaleBy methods already do part of the work—they correctly compute the x and y coordinates. How can we reuse this code?

The Answer: `super`!

- Sending a message to `super` invokes a method in the superclass.
- We can use `super` to implement scaleBy for the Point3D class:

Invokes the 2D Point `scaleBy` →

```
scaleBy: factor
  super scaleBy: factor.
  z := z * factor
```

Addition

- We could try the same thing for addition:

```
+ anotherPoint
  | result |
  result := super + anotherPoint.
  result setz: z + anotherPoint getz.
  ^ result
```

- But there's a problem; `super +` invokes the addition method of our 2D point class, which returns a 2D point. When we try to set its z coordinate, a runtime error occurs!

Fixing the problem

- To fix the problem, we need to change the addition method for 2D points:

```
+ anotherPoint
  | result |
  result := Point new.
  result setx: x + anotherPoint getx.
  result sety: y + anotherPoint gety.
  ^ result
```

- Why does this work? Dynamic dispatch!

Fixing the problem

- To fix the problem, we need to change the addition method for 2D points:

```
+ anotherPoint
  | result |
  result := self class new.
  result setx: x + anotherPoint getx.
  result sety: y + anotherPoint gety.
  ^ result
```

- Why does this work? Dynamic dispatch!

Dynamic Dispatch

- When we invoke a method on a receiver object, the `self` variable is bound to that object.
- Sending a message to self invokes methods of `self`'s class—even if `self` is used in a method inherited from a superclass.
- In our addition example:
  - Calling the 2D addition will return a new 2D point if it is invoked by a 2D point.
  - If, however, a 3D point invokes the 2D addition as part of the 3D addition, a 3D point will be returned—because `self` will be a 3D point.

Dynamic Dispatch

- This is called *dynamic dispatch* because the method to be invoked (dispatched) is chosen at runtime based on the class of `self`.
- Dynamic dispatch is a fundamental element of object-oriented programming. It make large-scale code reuse by subclasses possible.

Abstract vs. Concrete Classes

- The `Point` class provides an interface (the messages or methods) and an implementation.
- We can provide more flexibility by splitting these:
  - Abstract superclasses provide methods but no instance variables.
  - Concrete subclasses provide instance variables and additional accessor methods.

(Thanks to Craig Chambers)

An Abstract Interface Class

```
Object subclass: #Point
  instanceVariableNames: '' …
```

```
+ anotherPoint
  | result |
  result := self class new.
  result setx: self getx + anotherPoint getx.
  result sety: self gety + anotherPoint gety.
  ^ result
```

```
getx
setx: newX
  self subclassResponsibility
```

```
gety
sety: newY
  self …
```

(Thanks to Craig Chambers)

A Cartesian Implementation

```
Point subclass: #CartesianPoint
  instanceVariableNames: 'x y' …
```

```
getx
  ^ x
```

```
gety
  ^ y
```

```
setx: newX
  x := newX
```

```
sety: newY
  y := newY
```

(Thanks to Craig Chambers)

A Polar Implementation

```
Point subclass: #PolarPoint
  instanceVariableNames: 'rho theta' …
```

```
getrho
  ^ rho
```

```
gettheta
  ^ theta
```

```
setrho: newRho
  rho := newRho
```

```
settheta: newTheta
  theta := newTheta
```

We also need to provide the `getx`, `setx`, `gety` and `sety` methods mentioned in the interface.

(Thanks to Craig Chambers)

A Polar Implementation

```
Point subclass: #PolarPoint
  instanceVariableNames: 'rho theta' …
```

```
getrho
  ^ rho
```

```
gettheta
  ^ theta
```

```
setrho: newRho
  rho := newRho
```

```
settheta: newTheta
  theta := newTheta
```

```
getx
  ^ rho * theta cos
```

```
gety
  ^ rho * theta sin
```

(Thanks to Craig Chambers)

Object-oriented Design

- Steps in building an object-oriented program:
  - Identify the major data abstractions. These are the objects.
  - Identify the major operations on the data abstractions. These are the interfaces.
  - Identify commonalities among the abstractions, and organize an inheritance hierarchy.
  - Implement the design.
  - Repeat (as necessary).
- Design for the long term—make sure it's easy to build on and add to your design.

(Thanks to Craig Chambers)

Summary

- Smalltalk was the first pure object-oriented language.
- Everything is an object, and every operation is a message send to an object.
- Blocks are objects that contain unevaluated code, and are used to implement control structures.
- Smalltalk is useful to study because it contains all the key features of modern OO languages, without much of the complexity.