Defining a new class

Example: 3-D Points

class definition:

```
Object subclass: #Point3D
   instanceVariableNames: 'x y z'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Graphics-Primitives'
```

No special syntax for class definition
• evaluate expression using Browser to build class

Defining some methods

instance methods:

```
+ anotherPoint
    | result |
    result := Point3D new.
    result x: x + anotherPoint x.
    result y: y + anotherPoint y.
    result z: z + anotherPoint z.
    ^ result

scaleBy: factor "modifies receiver (unlike in Squeak)"
    x := x * factor.
    y := y * factor.
    z := z * factor.
    ^ x
    x: newX
    x := newX.
    y ... z ...
    y: ... z: ...
```

plus many other methods

Class methods

A class (e.g. Point3D) is an object
• it has methods it inherits (e.g. new)
• it can have user-defined methods

To create an instance of a class, send the class a new message:
```
p := Point3D new.
```

Can define your own class methods for e.g. initialized creation

In Point3D class methods:
```
x: x y: y z: z
    | p |
    p := self new.
    p x: x.
    p y: y.
    p z: z.
    ^ p
```

A use:
```
p := Point3D x: 3 y: 4 z: 5.
```

Using inheritance instead

Define Point3D as a subclass of Point

class definition:
```
Point subclass: #Point3D
   instanceVariableNames: 'z'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Graphics-Primitives'
```

instance methods:

```
+ anotherPoint
    same as before
scaleBy: factor
    same as before
z ... z: ...
```

Summary:
• inherit x and y instance variable declarations
• inherit x, x:, y, y:, etc., methods
• add z, z: methods
• override +, scaleBy: methods
Sends to *self*

If a message is sent to *self*, then method lookup starts at the object that *self* refers to, not the current class.

Example: In *Point*:
```plaintext
double
  ^ self + self
```

This behavior is crucial to object-oriented programming.

---

Differential programming for methods

There’s redundancy in current implementation

In *Point*:
```plaintext
scaleBy: factor
  x := x * factor.
  y := y * factor.
```

In *Point3D*:
```plaintext
scaleBy: factor
  x := x * factor.
  y := y * factor.
  z := z * factor.
```

---

Super sends

Can use *super* send to avoid code duplication:

In *Point3D*:
```plaintext
scaleBy: factor
  super scaleBy: factor.
  z := z * factor.
```

Send to *super* is just like send to *self*, except method lookup starts in superclass.

• *super* can only appear as a message receiver.

---

A pitfall

In *Point*:
```plaintext
+ anotherPoint
  | result |
  result := Point new.
  result x: x + anotherPoint x.
  result y: y + anotherPoint y.
  ^ result
```

Current *Point3D*:
```plaintext
+ anotherPoint
  | result |
  result := Point3D new.
  result x: x + anotherPoint x.
  result y: y + anotherPoint y.
  result z: z + anotherPoint z.
  ^ result
```

“Better” *Point3D*:
```plaintext
+ anotherPoint
  | result |
  result := super + anotherPoint.
  result z: z + anotherPoint z.
  ^ result
```
Inserting sends to self

Increase reusability of Point + method by replacing hard-wired constant with send to self

In Point:
+ anotherPoint
  | result |
  result := self pointClass new.
  result x: x + anotherPoint x.
  result y: y + anotherPoint y.
  ^ result
pointClass
  ^ Point

In Point3D:
+ anotherPoint
  | result |
  result := super + anotherPoint.
  result z: z + anotherPoint z.
  ^ result
pointClass
  ^ Point3D

Another example of inheritance: PolarPoint

Goal: define a 2-D point that represents values using polar coordinates (rho and theta)

Idea: implement by subclassing existing cartesian Point class
class definition:
Point subclass: #PolarPoint
  instanceVariableNames: 'rho theta'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Graphics-Primitives'
instance methods:
  rho ... theta ...
  rho: ... theta: ...
  x
    ^ rho * theta cos
  y
    ^ rho * theta sin
  x: ... y: ...

A problem

Example:
| p1 p2 |
| p1 := PolarPoint new.
p1 rho: 1.
p1 theta: 60.
p2 := PolarPoint new.
p2 rho: 1.5.
p2 theta: 170.

p1 + p2

produces a message-not-understood error:
  sending + to an instance of UndefinedObject (i.e., nil)

Why?

A solution

Old:
+ anotherPoint
  | result |
  result := self class new.
  result x: x + anotherPoint x.
  result y: y + anotherPoint y.
  ^ result

New:
+ anotherPoint
  | result |
  result := self class new.
  result x: self x + anotherPoint x.
  result y: self y + anotherPoint y.
  ^ result

Theme: adding sends to self increases flexibility later
Abstract vs. concrete classes

Point defines both an interface and an implementation

For better flexibility, split these two components apart
• abstract superclass containing methods
  but no instance variables
• concrete subclass providing the instance variables and
  some accessor methods

Abstract classes represent interfaces; can’t be instantiated
Concrete classes flesh out abstract classes with full
implementations

The interface

class definition:
Object subclass: #Point
  instanceVariableNames: ‘’ ...
instance methods:
+ anotherPoint
  | result |  
  result := self class new.
  result x: self x + anotherPoint x.
  result y: self y + anotherPoint y.
  result z: self z + anotherPoint z.
  ^ result
scaleBy: factor
  self x: self x * factor.
  self y: self y * factor.
  self z: self z * factor.
  x
  self subclassResponsibility
x: newX
  self subclassResponsibility
y ... y: ...

The implementation

class definition:
Point subclass: #CartesianPoint
  instanceVariableNames: ‘x y’ ...
instance methods:
  x ... y ...
x: ... y: ...

Another implementation

class definition:
Point subclass: #PolarPoint
  instanceVariableNames: ‘rho theta’ ...
instance methods:
rho ... theta ...
rho: ... theta: ...
x
  ^ rho * theta cos
y
  ^ rho * theta sin
x: newValue
  ...
y: newValue
  ...