References


Smalltalk Best Practice Patterns, Beck

Smalltalk With Style, Klimas, Skublics, Thomas

Reading

Smalltalk by Example, Alex Sharp,
   Chapter 2 Methods
   Chapter 8 Control Structures
   Chapter 4 Variables
   Chapter 5 Instance Creation
Control Messages
if

(boolean expression) ifTrue: trueBlock

(boolean expression) ifFalse: falseBlock

(boolean expression) ifFalse: falseBlock ifTrue: trueBlock

(boolean expression) ifTrue: trueBlock ifFalse: falseBlock

a < 1 ifTrue: [Transcript show: 'hi mom']

difference := (x > y)
    ifTrue: [ x - y]
    ifFalse: [ y - x]
## Boolean Expressions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or</td>
<td>a</td>
</tr>
<tr>
<td>And</td>
<td>a &amp; b</td>
</tr>
<tr>
<td>Exclusive or</td>
<td>a xor: (b &gt; c)</td>
</tr>
<tr>
<td>Negation</td>
<td>(a&lt; b) not</td>
</tr>
</tbody>
</table>

## Lazy Logical Operations

<table>
<thead>
<tr>
<th>Message</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or</td>
<td>a or: [b &gt; c]</td>
</tr>
<tr>
<td>And</td>
<td>a and: [c</td>
</tr>
</tbody>
</table>
This is not C

This is a runtime error

5 ifTrue: [1 + 3]

Of course you could just add the ifTrue: method to the Number class if you want to do the above.
A Style Issue

Both do the same thing

difference := (x > y)
  ifTrue: [ x - y]
  ifFalse: [ y - x]

(x > y)
  ifTrue: [difference := x - y]
  ifFalse: [difference := y - x]

The one on the left may seem strange. Other language do allow this. Some (many Smalltalkers) consider the one on the left to better convey the intent of the code.
**isNil**

Answers true if receiver is nil otherwise answers false

```plaintext
x isNil
    ifTrue: [ do something]
    ifFalse: [ do something else]
```

**Shortcuts**

```plaintext
ifNil:ifNotNil:
ifNotNil:ifNil:
ifNil:
ifNil:
ifNotNil:
```

```plaintext
x
    ifNil: [ do something]
    ifNotNil: [ do something else]
```
Blocks

A deferred sequence of actions – a function without a name
Can have 0 or more arguments
Executed when sent the message 'value'

Similar to
Lisp's Lambda-Expression
Erlang's funs
Ruby's Blocks
Python's lambda
Anonymous functions

[:variable1 :variable2 ... :variableN |
 | blockTemporary1 blockTemporary2 ... blockTemporaryK | expression1.
expression2.
...]
Blocks and Return Values

Blocks return the value of the last executed statement in the block

| block x |
block := [:a :b |
  | c |
  c := a + b.
  c + 5].
x := block value: 1 value: 2.

x has the value 8
Blocks know their Environment

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

a := 1.
b := 2.

aBlock := [a + b].
result := aBlock value

result is now 3

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

a := 5
result := aBlock value

result is now 6
Blocks and Arguments

Using the value: keyword message up to 4 arguments can be sent to a block.

[2 + 3 + 4 + 5] value
[:x | x + 3 + 4 + 5 ] value: 2
[:x :y | x + y + 4 + 5] value: 2 value: 3
[:x :y :z | x + y + z + 5] value: 2 value: 3 value: 4
[:x :y :z :w | x + y + z + w] value: 2 value: 3 value: 4 value: 5

valueWithArguments: can be used with 1 or more arguments

[:a :b :c :d :e | a + b + c + d + e ] valueWithArguments: #( 1 2 3 4 5)
[:a :b | a + b ] valueWithArguments: #( 1 2 )
Where is the Value Message

difference := (x > y)
ifTrue: [ x - y]
ifFalse: [ y - x]

In the False class we have:

ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
^falseAlternativeBlock value

In the True class we have:

ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
^trueAlternativeBlock value

This is an example of Polymorphism. More on this later.
While Loop

aBlockTest whileTrue
aBlockTest whileTrue: aBlockBody
aBlockTest whileFalse
aBlockTest whileFalse: aBlockBody

The last expression in aBlockTest must evaluate to a boolean

| x y difference | | count |
|----------------|------------------|
x := 8.
count := 0.
y := 6.
[count := count + 1.
difference := 0.
count < 100] whileTrue.
difference := difference + 1.
^difference

Transcript
clear;
show: count printString
More Loops

Transcript
clear.
3 timesRepeat:
[Transcript
  cr;
  show: 'Testing!'].
1 to: 3 do:
[ :n |
  Transcript
  cr;
  show: n printString;
  tab;
  show: n squared printString].
9 to: 1 by: -2 do:
[ :n |
  Transcript
  cr;
  show: n printString].

Transcript
Testing!
Testing!
Testing!
1   1
2   4
3   9
9
7
5
3
1
Classes
Objects & Classes - Smalltalk Language Details

Items to cover

Defining classes
Packages
Namespaces
Class names

Methods
• Instance
• Class

Variables
• Instance variables
• Class instance variables
• Shared variables

Inheritance

self & super
The Rules

Everything in Smalltalk is an object

All actions are done by sending a message to an object

Every object is an instance of a class

All classes have a parent class

Object is the root class
How do you Define a Class?

The previous slide gives the answer but you may not believe it.
Defining Point Class

Smalltalk.Core defineClass: #Point
  superclass: #{Core.ArithmeticValue}
  indexedType: #none
  private: false
  instanceVariableNames: 'x y '
  classInstanceVariableNames: "
  imports: "
  category: 'Graphics-Geometry'

Using the rules we send a message to an object. In this case we sent a message to the Namespace object that the class belongs. Some argue that we should sent a message to the classes parent (or super class). There are parts of the message that will not make sense now. Don't worry one does not have to type message. The browser will do it for you.
Terms

Superclass

Package (parcel)

Namespace
Class Names & Namespaces

Classes are defined in a namespace

Classes in different namespaces can use the same name

Full name of a class includes namespace

Root.Smalltalk.Core.Point

Use import to use shorter names

Workspace windows import all namespaces
Methods

All methods return a value

All methods are public

Placed a method in the "private" category to tell others to treat it as private
Instance methods

Sent to instances of Classes

1 + 2
'this is a string' reverse
Class Methods

Sent to Classes

Commonly used to create instances of the class

Array new
Point x: 1 y: 3
Float pi
Convention

ClassName>>methodName

String>>reverse

Point class>>x:y:
Naming Conventions
Class Names

Use complete words, no abbreviations

First character of each word is capitalized

SmallInteger
LimitedWriteStream
LinkedMessageSet
Simple Superclass Name

Simple words

One word preferred, two at maximum

Convey class purpose in the design

Number
Collection
Magnitude
Model
Qualified Subclass Name

Unique simple name that conveys class purpose
   When name is commonly used

   Array
   Number
   String

Prepend an adjective to superclass name
   Subclass is conceptually a variation on the superclass

   OrderedCollection
   LargeInteger
   CompositeCommand
Class Names and Implementation

Avoid names that imply anything about the implementation of a class

"A proper name that is stored as a String"

ProperName

ProperNameString

"A database for Problem Reports that uses a Dictionary"

ProblemReportDatabase

ProblemReportDictionary

"Not implemented with a Set, it is a specialized Set"

SortedSet
Method Names

Always begins with a lowercase first letter
Don't abbreviate method names
Use uppercase letters for each word after the first
Method Naming Guidelines

Choose method names so that statements containing the method read like a sentence

FileDescpriptor seekTo: work from: self position

Use imperative verbs and phrases for methods which perform an action

Dog
  sit;
  lieDown;
  playDead.

aFace lookSuprised
aFace surprised

Examples from Smalltalk With Style, page 10-11
Method Naming Guidelines

Use a phrase beginning with a verb (is, has) when a method returns a boolean

\[
\text{isString} \quad \text{aPerson isHungry}
\]

\[
\text{aPerson hungry}
\]

Use common nouns for methods which answer a specific object

\[
\text{anAuctionBlock nextItem}
\]

\[
\text{anAuctionBlock item} \quad \text{"which item"}
\]
Method Naming Guidelines

Methods that get/set a variable should use the same name as the variable

books
^books

getBooks
^books

books: aCollection
books := aCollection

setBooks: aCollection
books := aCollection
Inheritance

Smalltalk supports only single inheritance

Each class has single parent class

A class inherits (or has) all
   Methods defined in its parent class
   Methods defined in its grandparent class
   etc.
   Methods defined in any ancestor class
   Variables defined in any ancestor class
Terms

Parent Class
Superclass

Child class
Subclass
Object

Is the ancestor of all classes

Has no parent class

Contains important methods for all classes & objects
Inheritance and Name Clashes

Subclass can implement methods with same name as parent

This is called overloading the method

When message is sent to instance of the subclass, the subclass method is used

Subclass can not overload variable names

Actually you can force a subclass to overload a variable name. Nothing good comes from doing this.
Example

```
<table>
<thead>
<tr>
<th>aParent aChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>aParent := Parent new.</td>
</tr>
<tr>
<td>aChild := Child new.</td>
</tr>
<tr>
<td>aParent foo.</td>
</tr>
<tr>
<td>aChild foo.</td>
</tr>
</tbody>
</table>

'foo'

'bar'
```
Types of Variables

Temporary (Local) Variable
Named Instance Variable
Class Instance Variable
Shared Variable
Indexed Instance Variable
| a b sum |  
|------|------|------|
| a := 5. | b := 10. | sum := a + b.

Point>>grid: aPoint

"Answer a new Point to the nearest rounded grid modules specified by aPoint."

<table>
<thead>
<tr>
<th>newX newY</th>
</tr>
</thead>
<tbody>
<tr>
<td>aPoint x = 0</td>
</tr>
<tr>
<td>ifTrue: [newX := 0]</td>
</tr>
<tr>
<td>ifFalse: [newX := x roundTo: aPoint x].</td>
</tr>
</tbody>
</table>

| aPoint y = 0 |
| ifTrue: [newY := 0] |
| ifFalse: [newY := y roundTo: aPoint y]. |

^newX @ newY
Usage Convention

Do not use the same temporary variable name within a scope for more than one purpose

<table>
<thead>
<tr>
<th>aRecord</th>
</tr>
</thead>
<tbody>
<tr>
<td>aRecord := self indexRecord.</td>
</tr>
<tr>
<td>aRecord lock: 12.</td>
</tr>
<tr>
<td>aRecord := aRecord at: 12.</td>
</tr>
<tr>
<td>self update: (aRecord at: 1) with: self newData.</td>
</tr>
<tr>
<td>aRecord unlock: 12.</td>
</tr>
</tbody>
</table>
Named Instance Variable

Each object has its own copy of a named instance variable

Like
- Protected C++ data member
- Protected Java field

Accessible by
- Instance methods of the class
- Instance methods of subclasses of the class

Not accessible by
- Methods in non-subclasses
- Class methods
We now have two point objects. Each point object has a local copy of x and y. Values in the local copies are different.
We now have two point objects. Each point object has a local copy of x and y. Values in the local copies are different.
Adding Removing Instance Variables

Method 1 Edit Class Definition

Smalltalk defineClass: #ClassPoint
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'x y z w '
  classInstanceVariableNames: "
  imports: "
  category: "


Adding/Removing Instance Variables

Method 2: Use Browser's Class menu

When removing instance variables using the menu option will check to see if you are still using the variable before removing it.
self & super

self
Refers to the receiver of the message (current object)

Methods referenced through self are found by:
  Searching the class hierarchy starting with the class of receiver

super
Refers to the receiver of the message (current object)

Methods referenced through super are found by:
  Searching the class hierarchy starting the superclass of the class containing the method that references super
self and super Example

**Parent**

- Parent>>name
  - ^'Parent'

**Child**

- Child>>name
  - ^'Child'
- Child>>selfName
  - ^self name
- Child>>superName
  - ^super name

**GrandChild**

- GrandChild>>name
  - ^'GrandChild'

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>grandchild</td>
</tr>
<tr>
<td>grandchild := Grandchild new.</td>
<td></td>
</tr>
<tr>
<td>Transcript</td>
<td></td>
</tr>
<tr>
<td>show: grandchild name;</td>
<td>Grandchild</td>
</tr>
<tr>
<td>cr;</td>
<td></td>
</tr>
<tr>
<td>show: grandchild selfName;</td>
<td>Grandchild</td>
</tr>
<tr>
<td>cr;</td>
<td></td>
</tr>
<tr>
<td>show: grandchild superName;</td>
<td>Parent</td>
</tr>
<tr>
<td>cr;</td>
<td></td>
</tr>
</tbody>
</table>
How does this work

grandchild selfName

Receiver is grandchild object
Code in selfName method is ^self name
To find the method "self name" start search in Grandchild class

grandchild superName

Receiver is grandchild object
Code in superName method is ^super name
superName is implemented in Child class

To find the method "super name" start search in the superclass of Child
Why Super

Super is used when:

- The child class overrides a method
- Needs to call overridden method

Common Pattern

ClassPointSubclass >> initialize
super initialize.
z := 0.
Why doesn't super refer to parent class of the receiver?

If super referred to the parent class of the receiver the above code would result in an infinite loop. The receiver is a GrandChild object so the parent is Child. So in Child>>name "super name" would refer to Child>>name.
Class Methods

ClassPoint class>>origin
  ^self x: 0 y: 0

ClassPoint class>>x: xNumber y: yNumber
  ^(self new)
    x: xNumber;
    y: yNumber;
    yourself

ClassPoint class>>new
  ^super new initialize

center := ClassPoint origin.
center x
"Returns o"
new & initialize

ClassPoint>>initialize
    x := 0.
    y := 0.

ClassPoint class>>new
    ^super new initialize

ClassPoint new
    SomeParentClass new initialize
        SomeParentClass new returns a ClassPoint object
    aClassPointObject initialize
Initialization and Inheritance

Smalltalk.Core defineClass: #Parent
  superclass: #{Core.Object}
  instanceVariableNames: 'foo '

  Class Method

  new
  ^super new initialize

  Instance Methods

  initialize
  foo :=6.

  foo
  ^foo
Initialization of Subclass

How to initialize bar?

Smalltalk.Core defineClass: #Child
    superclass: #{Core.Parent}
    instanceVariableNames: 'bar'

Bad Idea 1 – Use Same pattern

Child class>>new
  ^super new initialize

Child>>initialize
  bar := 2.

Child>>bar
  ^bar
Why bad?

Does not work!

<table>
<thead>
<tr>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>test := Child new.</td>
</tr>
<tr>
<td>test foo “returns nil”</td>
</tr>
</tbody>
</table>

initialize is called twice

Child class>>new is not needed
Child class inherits an identical method
Bad Idea 2 – Subclass initializes Parent Variable

Child>>initialize
   bar := 2.
   foo := 6.

Why Bad?

Child class now involved in private affairs of the Parent

Changes to the Parent instance variables require changing Child
Solution

Parent class>>new
  ^super new initialize

Parent>>initialize
  foo := 6.

Parent>>foo
  ^foo

Child>>initialize
  super initialize
  bar := 2.

Child>>bar
  ^bar
Class Methods that Create Instances

Smalltalk does not have constructors like C++/Java

Use class methods to create instances

Place these class methods in "instance creation" category
Initial State of Instances

Create objects in some well-formed state

Class creation methods should:

- Have parameters for initial values of instance variables or
- Set default values for instance variables

Provide an instance method that:

- Sets the initial values of instance variables
- Place method in "initialize" or "initialize - release" category
- Use the name setVariable1: value variable2: ...
Disabling new

Point new
  Does not work

Point x: 1 y: 12
  This works

Implementers wanted users to specify initial value of a point
Class Instance Variables

A class has one instance of a class instance variable

Each subclass has a different instance

Accessible by
  Class methods of the class
  Class methods of subclasses
Example

Smalltalk.Core defineClass: #ClassInstanceVariableExample
   superclass: #{Core.Object}
   indexedType: none
   private: false
   instanceVariableNames: "
   classInstanceVariableNames: 'test '
   imports: "
   category: 'As yet unclassified'
Adding/Removing Class Instance Variables

Method 1

Edit the class definition directly

Method 2
Example

Smalltalk.Core defineClass: #Parent
superclass: #{Core.Object}
classInstanceVariableNames: 'test'

Parent class>>test
test isNil ifTrue:[ test := 0].
test := test + 1.
^test

Smalltalk.Core defineClass: #Child
superclass: #{Core.Parent}
classInstanceVariableNames: "

<table>
<thead>
<tr>
<th>Transcript</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>print: Parent test;</td>
<td>1</td>
</tr>
<tr>
<td>cr;</td>
<td></td>
</tr>
<tr>
<td>print: Parent test;</td>
<td>2</td>
</tr>
<tr>
<td>cr;</td>
<td></td>
</tr>
<tr>
<td>print: Child test;</td>
<td>1</td>
</tr>
<tr>
<td>flush</td>
<td></td>
</tr>
</tbody>
</table>
Lazy Initialization

Parent class>>test
    test isNil ifTrue:[ test := 0].
    test := test + 1.
    ^test
Indexed Instance Variable

Provides slots in objects for array like indexing

Used for Arrays

I have never added indexed instance variables

I have always used existing collection classes