```
myFinances spend: 100.00 for: 'food'
```

A brief tour of Smalltalk

(based on a textbook by Adele Goldberg and David Robson)

Contents

- Fundamentals 139
- Expressions 143
- Control 146
- Class descriptions 149
- Subclasses 153
- Metaclasses 159
- Parts of the standard object hierarchy 164

Fundamentals

Highlights of Smalltalk

- Orthogonality—five basic concepts: object, message, class, instance, method.
- A customizable programming environment and a constantly evolving system.

An **object** has private memory and a set of public operations.

Examples of objects: numbers, strings, queues, dictionaries, rectangles, files, text editors —all these are components of the Smalltalk system.

☒ A Rectangle is an object with two private data items: Points at the opposing corners.

Examples of operations: ask a Rectangle to find the location of its centre, or to move itself.

A **message** requests an object to perform an operation:

- the message says which operation,
- the receiver says how to do it.

The set of all message patterns of an object is its interface with the rest of the system.

☒ A Rectangle may be asked to return its corners, centre, area, to move itself, rotate itself, draw itself, and so on.

It is good programming practice to design a complete set of operations, with high potential for re-usability.

A **class** is a set of objects of some kind, and each of them is called an **instance** of this class.

Every object must be an instance of a class. In fact, even every class is an instance of a class! Such a class of classes is called a metaclass.

Messages are public.

Instance variables of an object are its private memory. The value of an instance variable is an object, and it can only be available to another object through an operation.

☒ A Rectangle cannot see the co-ordinates of its Points (represented as instance variables). It must *ask* a Point to return its co-ordinates.

Methods describe how to perform operations.

☒ To find its centre, a Rectangle could:
(1) ask its Points to produce the co-ordinates (pixel numbers),
(2) initiate the calculations (asking the respective integers to perform them),
(3) ask the class of Points to produce a new Point from the co-ordinates just computed.

Methods are in 1-1 correspondence with messages. Primitive built-in methods (such as arithmetics or I/O) cannot be changed.

System classes in Smalltalk include:

- arithmetics,
- collections (more will be shown later),
- control structures (blocks, conditions, iterations),
- environment (methods in source form and in compiled form),
- viewing and editing (this includes simple graphics),
- input/output.

Our larger example will be FinancialHistory, with operations like the following:

- create a new FinancialHistory with some initial amount,
- record spending some amount, and on what,
- record receiving some amount, and from where,
- find how much money is available,
- find how much was spent for a given reason,
- find how much was received from a given source.

Expressions

- **Literals** (literal constants):

  number character string symbol

  ```
  1951   $a $Z   'ami'   #bill
  ```

  (a symbol is unique in the system)

  an array of literals

  ```
  #(9 'Bill' $8 (0 'a' ()) $# 'idle')
  ```

- **Variables** are untyped:

  an identifier is a handle on an object.

- **Assignment**:

  ```
  quantity ← 17
  centre ← aPoint x: xCoord y: yCoord
  ```

- **Pseudo-variables** refer to objects. They cannot be the target of assignment. Some system pseudo-variables are system-wide and do not change:

  ```
  nil true false
  ```

Other system pseudo-variables refer to the object itself (in its own methods) and to the superclass:

```
self super
```


- **Messages**:

```
3 + 4                          a binary message
indx > lim                     a binary message
theta sin                      a unary message
list addFirst: newItem         a keyword message
list remItem                   a unary message
ages at: 'Jim' put: 27         (set an element)
addresses at: 'Bill'           (get an element)
thisRectangle centre
myFinances spend: 100.00 for: 'food'
myFinances totalSpentFor: 'food'
```

A message describes an operation by specifying:

- the receiver,
- the selector(s),
- the argument(s).

A unary message has only a selector: sin, remLast
A binary message, e.g. + >
A keyword has parts marked by colons. Each part precedes one argument, e.g. addFirst:, at:put:

Precedence of messages is

unary > binary > keyword

and left-to-right within these groups. For example,

```
3 + 4 * 5
```

gives 35. Another example:

```
results at: 17 put: t1 sin + t2 sin * 2
```

is equivalent to the Pascal assignment

```
results[17] := (sin(t1)+sin(t2))*2
```

Returning a value: the receiver sends back an object which then may be assigned.

```
sum ← 3 + 4
```

(the message + 4 sent to 3 which sends back 7)

```
aLot ← myFinances totalSpentFor: 'food'
```

Even if the value is irrelevant, as in

```
myFinances spend: 100.00 for: 'food'
```

something is still sent back. Default: the receiver, denoted as

```
self
```


Control

A block is a deferred (not evaluated) sequence of actions, written in brackets and separated with periods:

```
[ action1. action2. ..... actionN]
```

☒ Example:

```
[ indx ← indx + 1.
  array1 at: indx put: 0]
```

A block is an object (of course!), so it can be assigned to a variable. It is executed only upon request. The assignment

```
toPay ← [myFinances spend: 10.00 for: 'haircut'.
         myFinances spend: 800.00 for: 'rent']
```

will not record any expenses. To execute this block, we write this:

```
toPay value
```

By the way, the expression `[] value` returns `nil`.

Control mechanisms use blocks.

• sequence (obviously!)

• counted repetition

```
4 timesRepeat: [indx ← indx + 1]
```

• conditional execution

```
N odd ifTrue: [parity ← 1]
      ifFalse: [parity ← 0]
```

or, equivalently,

```
parity ← N odd ifTrue: [1] ifFalse: [0]
```

or

```
parity ← N odd ifFalse: [0] ifTrue: [1]
```

A one-branch conditional operation, e.g.,

```
cond ifTrue: block
```

is equivalent to

```
cond ifTrue: block ifFalse: []
```


• condition-driven loop

```
[indx <= high] whileTrue:
    [ array1 at: indx put: 0.
      indx ← indx + 1]
```

or, equivalently,

```
[indx > high] whileFalse:
    [ array1 at: indx put: 0.
      indx ← indx + 1]
```

Block arguments (iterators)

```
sum ← 0.
#(2 3 5 7 11) do:
    [:prime | sum ← sum + (prime * prime)]
```

This will produce 208.

```
#(2 3 5 7 11) collect:
    [:prime | prime * prime]
```

This will produce #(4 9 25 49 121).

Another method of getting #(4 9 25 49 121):

```
collector ← [:prime | prime * prime].
collector value: #(2 3 5 7 11).
```

Class descriptions

• A protocol description lists all the messages.
• An implementation description lists the methods.
Both descriptions are available for editing through the system browser.

☒ Example: class FinancialHistory, the protocol.

transaction recording

receive: amount from: source

Record that amount has been received from source.

spend: amount for: reason

Record that amount has ben spent for reason.

inquiries

cashOnHand

Return the total amount currently on hand.

totalReceivedFrom: source

Return the total amount received from source so far.

totalSpentFor: reason

Return the total amount spent for reason so far.

initialization

initialBalance: amount

Begin a financial history with amount as the inital deposit.

class name	instance variable names
FinancialHistory	cashOnHand
	incomes
	expenditures

instance methods

transaction recording

```
receive: amount from: source
    incomes at: source
        put: (self totalReceivedFrom: source ) + amount.
    cashOnHand ← cashOnHand + amount
spend: amount for: reason
    expenditures at: reason
        put: (self totalSpentFor: reason ) + amount.
    cashOnHand ← cashOnHand - amount
```

inquiries

```
cashOnHand
    ↑ cashOnHand
totalReceivedFrom: source
    (incomes includesKey: source)
        ifTrue: [↑ incomes at: source] ifFalse: [↑ 0]
totalSpentFor: reason
    (expenditures includesKey: reason)
        ifTrue: [↑ expenditures at: reason] ifFalse: [↑ 0]
```

initialization

```
initialBalance: amount
    cashOnHand ← amount.
    incomes ← Dictionary new.
    expenditures ← Dictionary new
```


Kinds of variables (they are always typeless, and are declared only with a name).

1. Instance variables are an object's private data. They may also be collections—indexed instance variables.
2. Temporary variables occur inside methods.
3. Class variables are shared by all objects of a class.
4. Global variables are shared by all objects.
5. Pool variables are shared by the instances of a subset of all variables.
   Global variables are in a pool called Smalltalk. For example, to give such a variable a value:

```
Smalltalk at: #GlobVar0 put: nil
```

All class variables are in one pool.

☒ In the class FinancialHistory:

instance variable names	cashOnHand
	incomes
	expenditures
class variable names	SalesTaxRate
shared pools	FinancialConstants

Returning a value from a method (again)
Default value is the receiver, other values to be returned must be indicated by an up-arrow ↑.
See the method for totalSpentFor: in the class FinancialHistory.

Names that refer to message arguments within a method are pseudovariables, matched with the argument values. Suppose we send the message

```
myFinances spend: 10.00 for: 'haircut'
```

amount refers to 10.00, and reason to 'haircut'.
The pseudovariable self refers to the receiver of the message (see the method spend:for:).

Another example:

```
factorial
  self = 0 ifTrue: [↑ 1].
  self < 0
      ifTrue: [self error: 'factorial invalid']
      ifFalse: [↑ self * (self - 1) factorial]
```

Temporary variables are typeless, local in a method, declared (only names) at the beginning of a method.

☒ Another version of the method spend:for:

```
spend: amount for: reason
| previousExpenditures |
   previousExpenditures ← self totalSpentFor: reason.
   expenditures at: reason
      put: previousExpenditures + amount.
   cashOnHand ← cashOnHand - amount
```

Subclasses

- strict hierarchy
- inheritance and overriding
  - the name of the subclass must be new
  - variables may be added
  - methods may be added
  - methods may override methods defined in the superclass

Class DeductibleHistory is a subclass of FinancialHistory.

class name: DeductibleHistory
instance variable names: deductibleExpenditures
superclass: FinancialHistory
instance methods

transaction recording

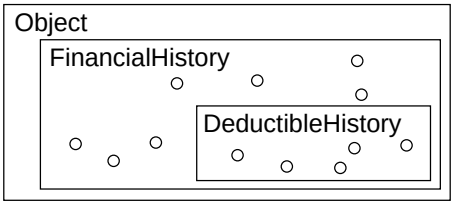
```
spendDeductible: amount for: reason
    self spend: amount for: reason.
    deductibleExpenditures ←
        deductibleExpenditures + amount
spend: amount for: reason
       deducting: deductibleAmount
    self spend: amount for: reason.
    deductibleExpenditures ←
        deductibleExpenditures + deductibleAmount
```

inquiries

```
totalDeductions
    ↑ deductibleExpenditures
```

initialization

```
initialBalance: amount
    super InitialBalance: amount.
    deductibleExpenditures ← 0
```



Search for a matching method

- Go up the hierarchy of classes, stop search in the class Object—it is an error not to find a matching method.
- If a method contains a message to self, start search from self's class (this instance's class), regardless of where the method is located.

class name: One
superclass: Object
instance methods

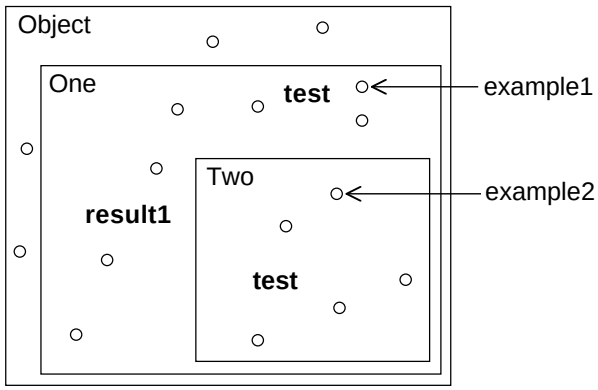
```
test
    ↑ 1
result1
    ↑ self test
```

class name: Two
superclass: One
instance methods

```
test
    ↑ 2
```

Now, we create two objects:

```
example1 ← One new.
example2 ← Two new
```

expression	result
example1 test	1
example1 result1	1
example2 test	2
example2 result1	2

For One new result1, self is referred to in this class. Starting the search in One, we find test — it returns 1.

For Two new result1, we start search in Two, and find test — it returns 2.

The pseudovariable super refers to the receiver of the message, as self does. The difference is that, when a message is sent to super, search begins in the superclass of the class that contains the method.

class name	superclass
Three	Two

instance methods

result2
↑ self result1

result3
↑ super test

class name	superclass
Four	Three

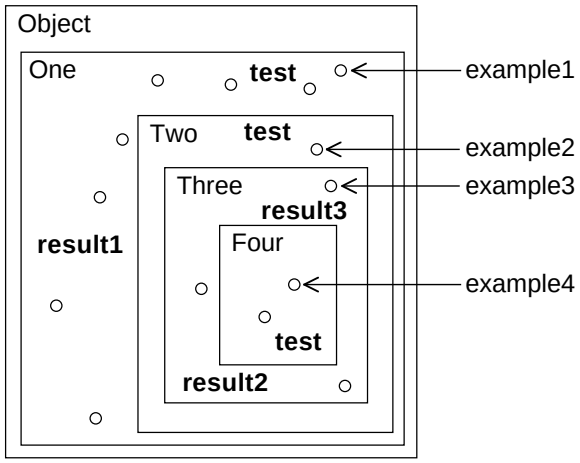
instance methods

test
↑ 4

Now, we create two more objects:

example3 ← Three new.

example4 ← Four new



expression	result
example3 test	2
example3 result2	2
example3 result3	2
example4 test	4
example4 result1	4
example4 result2	4
example4 result3	2

Metaclasses

Everything is an object, each object is an instance of a class,. So, a class must be an instance, too! We call a class of classes a metaclass.

When a class is created, a metaclass is automatically created for it. It is described together with its class, and inheritance also works in parallel.

A metaclass contains methods such as instance creation or instance initialization.

Examples instance of creation:

Time now Date today

Examples of instance initialization:

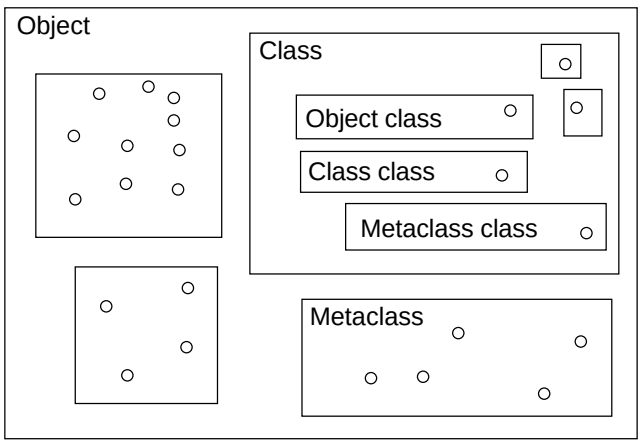
Point x: 100 y: 150

Rectangle origin: (Point x: 50 y: 150)
corner: (Point x: 250 y: 300)

All metaclasses are instances of the class called Metaclass, and they are nameless. You access them indirectly, for example:

Rectangle class

Here is a graphical illustration of these rather complicated dependencies:



And here is a new version of the class FinancialHistory, with class methods included.

class name
FinancialHistory
superclass
Object

instance variable names
cashOnHand
incomes
expenditures

class methods

instance creation

initialBalance: amount
↑ super new setInitialBalance: amount

new
↑ super new setInitialBalance: 0

instance methods

transaction recording

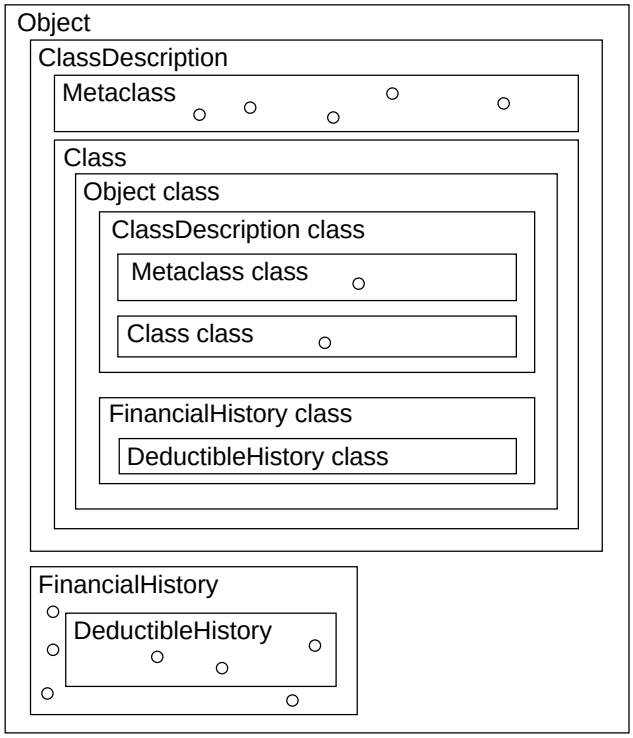
"as previously"

inquiries

"as previously"

private

setInitialBalance: amount
cashOnHand ← amount.
incomes ← Dictionary new.
expenditures ← Dictionary new



class name
DeductibleHistory
superclass
FinancialHistory

instance variable names
deductibleExpenditures

class variable names
minimumDeductions

class methods

instance creation

initialBalance: amount
| newHistory |
newHistory ← super initialBalance: amount.
newHistory initializeDeductions.
↑ newHistory

new
| newHistory |
newHistory ← super initialBalance: 0.
newHistory initializeDeductions.
↑ newHistory

class initialization

initialize
minimumDeductions ← 2300

instance methods

transaction recording

"as previously"

inquiries

isItemizable
↑ deductibleExpenditures >= minimumDeductions

totalDeductions
↑ deductibleExpenditures

private

initializeDeductions
deductibleExpenditures ← 0

- Magnitude
  - Character
  - Date
  - Time
  - Number
    - Float
    - Fraction
    - Integer
      - LargeNegativeInt
      - LargePositiveInt
      - SmallInteger
  - LookupKey
    - Association
- Link
  - Process
- Collection
  - SequenceableCollection
    - LinkedList
      - Semaphore
    - ArrayedCollection
      - Array
      - Bitmap
        - DisplayBitmap
      - RunArray
      - String
        - Symbol
      - Text
      - ByteArray
    - Interval
    - OrderedCollection
      - SortedCollection
  - Bag
  - MappedCollection
  - Set
    - Dictionary
      - IdentityDictionary
- Stream
  - PositionableStream
    - ReadStream
    - WriteStream
      - ReadWriteStream
        - ExternalStream
          - FileStream
  - RandomStream
- File
- FileDirectory
- FilePage
- UndefinedObject
- Boolean
  - False
  - True
- ProcessorScheduler
- Delay
- SharedQueue
- Behavior
  - ClassDescription
    - Class
    - Metaclass
- Point
- Rectangle
- BitBlockTransfer
  - CharacterScanner
  - Pen
- DisplayObject
  - DisplayMedium
    - Form
      - Cursor
      - DisplayScreen
  - InfiniteForm
  - OpaqueForm
  - Path
    - Arc .....

Summary

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....