OOP in Smalltalk

- Object-Oriented Programming (OOP) is a programming discipline that relies on objects to impose a modular structure on programs.
- OOP can be practised using simple languages with no support for information hiding. *Has to rely on programmers' discipline in accessing private information.*
- OOP can also be practised using languages with support for encapsulation. *Has to rely on programmers' discipline in proper object decomposition.*
- OOP is *best* practised using object-oriented languages, which require that every program must be constructed from objects.
- What exactly is an object-oriented language?
  - *Objects* and *classes* are obviously fundamental concepts. An object class is a set of objects that share the same operations and state information.
  - Objects are *1st class values*. Thus any operations may take an object as an argument, and may return an object as a result.
  - *Inheritance* is also a key concept. It is the ability to organize object classes into a hierarchy of subclasses and superclasses so that operations of a given class are applicable to object instances of a subclass.

- **Smalltalk** is
  - an object-oriented language:
    - Simple syntax and concept.
    - Interpreted with explicit storage allocation and automatic deallocation.
  - a programming paradigm:
    - Problem decomposition into objects and classes.
    - Computation as object interactions via message exchange.
    - Inheritance for software reuse.
  - a programming system:
    - A lot of "system objects" defined that make programming possible and convenient.
  - a programming environment and a GUI:
    - Window and mouse based, using high resolution bit-map display and keyboard.
    - Tools for presenting the underlying structure of the system to the user.
    - A mechanism for implementing such tools.
- In **Smalltalk**, everything is an *object*. There is no way to create, within the language, an entity which is not an object.
- Even object classes are objects, and control structures are just operations of appropriate classes.

- All actions are produced by passing *messages*. A message is a request for some operation (a method) to be performed by the object receiving the message (the *receiver*).
- A message can contain some arguments as well. All messages result in a response (or reply) being returned to the senders of the messages.
- The **Smalltalk** messaging mechanism can be compared with a procedure call in a conventional language as follows.
  - They are *similar* in that the point of control is moved from the sender to the receiver; the sender is suspended until a response is received from the receiver.
  - They are different in that the receiver of a message is *not* determined when the code is created (at compile-time), but is determined only when the message is actually sent (at run-time).
- This *dynamic* "binding" mechanism is the feature which gives **Smalltalk** its polymorphic capabilities.
- In **Smalltalk**, objects are *instances* of classes, which attempt to capture the important general properties (state information and methods) of all objects that are instances of that class.
- All instances of a class share the same responses to messages by invoking the same methods, but they will have different different *states* in general.

- Classes can inherit properties (state information and methods) from other classes; this process is repeated to form a *tree* of classes (the class hierarchy), rooted at the `Object` class.
- A subclass can *inherit* properties from its superclass; a subclass can also *override* properties of its superclass.
- Subclasses are used to *refine* the functionality of the superclass, for either the external protocol or the internal implementation.
- All classes are subclasses of the class `Object`.
- Clearly, an *expression syntax* is required to express how objects interact with one another. An expression is a sequence of characters which describe an object, called the value of the expression.
- Four types of expressions are permitted.
  - Literals describe constant objects (numbers, characters, strings, *etc*).
  - Variable names describe accessible variables. The value of a variable name is the current value of the variable with that name.
  - Message expressions describe messages to receivers. The value of an expression is determined by the method it invokes.
  - Block expressions describe deferred activities. Blocks are often used for control structures.

- There are four kinds of literals.
  - Numbers: `3`, `-5`, `0.56`, `1.3e5`, `16rA5`
  - Characters: `$A`, `$1`, `$$`, `$'`, `$+`
  - Strings: `'test string'`, `'string with '' embedded single quote'`
  - Symbols: `#initialize`, `#W80`, `#temp`
  - arrays: `#(40 41 42)`, `#((1 2) ('first' 'second'))`, `#(1 'one' $1 (5 'five' $5))`
- A variable, many of which are *named*, represents an object. The variable's name is an expression referring to that object.
- Variable names are identifiers made up of letters and digits with an initial letter:
  e.g. `someObject`, `Integer`, `Smalltalk`
- A capitalization convention is used consistently throughout the standard image (or the kernel) of **Smalltalk**.
  - Private variables (instance variables and temporary variables) start with an initial lower-case letter.
  - Shared variables (class variables, global variables, pool variables) start with an initial upper-case letter.
- Message selectors (see later) should also start with a lower-case letter.

- For both variable names and message selectors made up from more than one word, *capitalize* the first letter of each word in the identifier.
  e.g. `anExampleIdentifier`
- A variable name can refer to different objects at different times; assignment can be made to variable names using the ":=" operator.

  e.g. `newIndex := 1`, `this := that`
  `someString := 'Part Two'`, `aSymbol := #jim`,
  `anArray := #(1 2 3 4 5)`
- Assignments return values (like other expressions) so that several assignments can be made together.

  e.g. `this := that := theOther`
  `stop := end := 0`
- Variables are *not* typed in that a variable can be successively bound to objects from different classes.
- There are various kinds of variables in **Smalltalk**.
  - Temporary variables exist only for the duration of some activities (*e.g.* the execution of a method).
  - Instance variables have scope limited to a single object. They are defined in the class of an object and represent the *private* state of the object.

  - Class variables are shared by all instances of a single class.
  - Global variables are shared by all instances of all classes (*i.e.* all objects).
  - Pool variables are shared by all instances of some of the classes. Pool variables are *very rarely* used.
  - Pseudo-variables are pre-defined by the **Smalltalk** system. They cannot be changed.
    e.g. `nil`, `true`, `false`, `self`, `super`
- Typical use of shared variables are as follows.
  - "Constant" values used by a group of objects, but which might need to be changed occasionally.
  - Private communication between objects.
- Shared variables provide another mechanism for communication (as well as message passing). The use of shared variables is frequently an indication that a solution has not been well thought out.
- A message expression consists of the object which is the *receiver* of the message, a *message selector* which specifies the message being sent, and zero or more *argument* objects.
- When the method which implements a message has completed execution, it will return an object.
- If, in addition, the message changes the state of the receiver, then we say that a side effect has occurred.
- Note that assignment is *not* a message expression.

- There are three kinds of messages.
  - Unary messages have no arguments. The message selector can be any simple identifier. *e.g.* `anArray size`, `theta sin`, `4 even`
  - Binary messages have one argument, and the selector is composed of one or two non-alphanumeric characters. *e.g.* `3 + 4`, `index <= limit`
  - A keyword is an identifier followed by a colon. Keyword messages are ones in which the selector consists of one or more keywords, each with an argument. The arguments can be expressions representing any objects. *e.g.* `index max: limit`

    ```
    anArray at: first put: 'aardvark'
    ```

    Note that the selector is `at:put:` in the last example.
- In order to disambiguate expressions, it is necessary to establish a precedence in the order of message exchanges:

  Parenthesized $\geq$ Unary $\geq$ Binary $\geq$ Keyword

- Messages of the same precedence are executed from left to right.

  e.g. `1+2*3` $\mapsto$ 9, `1+(2*3)` $\mapsto$ 7

- Blocks represent *deferred* sequence of actions. They are represented by a sequence of expressions (separated by periods), surrounded by square brackets.
  e.g. `[ans := Float pi/2]`
- When a block expression is encountered, the statements enclosed are *not* executed immediately.
- The value of a block is an object (an instance of the `BlockContext` class) which can execute these expressions later, when required. A "block context" can be assigned to a variable.
  e.g. `testBlock := [this > that]`
- When the message `value` is received, the block will execute *in the context in which it was defined*, not necessarily the current context.

  e.g.
  ```
  testBlock := [this > that].
  testBlock value
          |||
  [this > that] value.
  ```

EXERCISE: Give an example to illustrate the last point.

- The result returned from a block is the value of the last expression executed.
  e.g. `[3 even. 3 odd] value` $\mapsto$ `true`

- A very simple control structure using blocks is the `timesRepeat:` message to integers. Integer objects respond to a `timesRepeat:` message by sending the `value` message to the block argument of the `timesRepeat:` message as many times as the integer's own value indicates.
  e.g. `4 timesRepeat: [amount := amount + 1]`
- In order to do conditional selection, **Smalltalk** provides two classes: `True` and `False`, which are both subclasses of the `Boolean` class. The pseudo-variables `true` and `false` denote instances of the two classes respectively.
- The message `ifTrue:ifFalse:` is a method defined in both of these classes. Both arguments to the message must be blocks.
- The `true` object responds to the `ifTrue:ifFalse:` message by sending the message `value` to the first argument block while the `false` object reponds by sending the message `value` to the second argument block.
- The value returned from `ifTrue:ifFalse:` is the value of the last block executed.

e.g.
```
number := 17.
    (number \\ 2) = 0
        ifTrue: [parity := 'even']
        ifFalse: [parity := 'odd'].
```

Small talk has no if then else, every thing is done by message passing and object.

- Another way to implement the last example is as follows.

```
number := 17.
parity := (number \\ 2) = 0
              ifTrue: ['even']
              ifFalse: ['odd'].
```

- The other keyword messages `ifFalse:ifTrue:`, `ifTrue:`, and `ifFalse:` behave in a similar fashion.
- Simple conditional loops can be constructed using the `whileTrue:` and `whileFalse:` messages. Both the receiver and the argument of the messages are expected to be blocks.
- For the `whileTrue:` message, the receiver block is sent the `value` message; if the response is `true`, then the argument block is sent the `value` message. This is repeated until the receiver block answers `false`.

  e.g. To initialize an array `list`:

```
list := Array new: 10.
index := 1.
[index <= list size] whileTrue:
    [list at: index put: 0.
     index := index + 1].
```

receiver block

EXERCISE: Re-implement the example using `whileFalse:`.

- The `whileTrue:` and `whileFalse:` messages always return the `nil` object.
- In some cases, all the "useful" work may be done in the receiver block. In this case, the argument block may be omitted and the unary messages `whileTrue` and `whileFalse` are used instead.
- Blocks can have one or more arguments. If a block has one argument, then it responds to the message `value:` by binding its argument to the argument object of the message before executing the content of the block.

  e.g.
  ```
  sizeAdder := [:array | total := total+array size].
  total := 0.
  sizeAdder value: #(a b c).
  sizeAdder value: #(1 2).
  ```
  $\mapsto$ `total` contains 5

- Blocks with two arguments respond to the `value:value:` message and so on.
- The `do:` message has a one argument block as argument and can be sent to a `Collection` object, such as an `Array`. The collection responds by sending a `value:` message to the block for each element in the collection and binding the block argument to each element of the collection in turn as the `value:` messages are sent.

e.g. `sum := 0.`
`#(2 3 5 7 11) do:`
`[:prime | sum := sum + (prime*prime)]`
$\mapsto$ `sum` contains 208

- Classes and instances revisited.
  - Each **Smalltalk** object is an instance of some class.
  - Each class is also an object and is an instance of a metaclass.
  - Since classes are objects, they can also respond to messages. There are two kinds of methods: *instance methods* and *class methods*, in a class definition.
  - While instance methods of a class are responsible for messages sent to the instances of the class, class methods of a class can be thought of the instance methods of the metaclass of the class!
- For programmers' convenience, methods are grouped together method categories (or protocols) according to usage (such as initialization, accessing, testing, instance creation, private, ...)
- When a message is sent to an object, a *search* for the corresponding method is made.
- The search first looks in the instance protocols of the object's own class. If the corresponding method is located there, then the method is executed, and the appropriate response is returned.

- If the appropriate method is not found, then a search is made in the instance protocols of the immediate superclass of the object's class. This process is repeated up the class hierarchy until either the method is located or there are no further superclass (*i.e.* the `Object` class is reached).
- In the latter case, the system sends the `doesNotUnderstand:` message to the receiver; this method is implemented in the `Object` class to display an error message.
- Messages can of course be sent to classes too. In this case, the system searches up the class protocols of the class hierarchy.
- The most common purpose of sending a message to a class is to create a new instance of that class.
- The followings are needed in defining (or implementing) a new class:
  - Class name: It must start with an upper case letter.
  - Super class: The name of the immediate superclass of the new class.
  - Named instance variable names: They must start with lower case letters. Each instance of the class will have its own values for these variables.
  - Class variable names: They must start with upper case letters. These variables are shared by all instances of the class.

- Instance methods: Each instance message must be implemented by an instance method.
- Class methods: Each class message much be implemented by a class method.

◯ Putting all things together, we illustrate the taught concepts using a simple example class: `Patron`.

A `Patron` Class in **Smalltalk**

Class Definition

```
Object subclass: #Patron
  instanceVariableNames: 'name id classroom loanList'
  classVariableNames: 'LoanLimit'
  poolDictionaries: ''
  category: 'Library'
```

Class Methods

class initialization

```
initialize
  "Initialize the loan limit."
  LoanLimit := 2
```

instance creation

```
name: nameString id: anInteger classroom: classString
  "Create an instance of myself with the given name, id and
classroom and with no items on loan."
  |newPatron|
  newPatron := self new.
  newPatron
    name: nameString
    id: anInteger
    classroom: classString.
  ^newPatron
```

A Patron Class in **Smalltalk** (cont'd)

Instance Methods

initialize-release

```
name: nameString id: anInteger classroom: classString
  "Initialize myself by setting my name, id number and classroom
to be the three arguments, nameString, anInteger and
classString respectively and by setting my loan list to be an
empty set."
  name := nameString.
  id := anInteger.
  classroom := classString.
  loanList := Set new
```

accessing

```
name
  "Answer my name."
  ^name

id
  "Answer my id."
  ^id

classroom
  "Answer my classroom."
  ^classroom

loanList
  "Answer the set of items currently on loan to me."
  ^loanList
```

A `Patron` Class in **Smalltalk** (cont'd)

```
circulation
loan: anItem
  "If I have not reached the loan limit, then add the first
argument, anItem, to my loan list and answer myself.
Otherwise, answer nil."
  (loanList size < LoanLimit)
    ifTrue: [loanList add: anItem. ^self]
    ifFalse: [^nil]

return: anItem
  "Remove the first argument, anItem, from my loan list."
  loanList remove: anItem
```

- ◯ The variable "`self`" refers to the object itself.
- ◯ The "`new`" method is implemented in the `Object` class.
- ◯ The methods "`add:`" and "`remove:`" are part of the protocol of the `Set` class.
- ◯ One of the fundamental requirements of an OO language is the ability to abstract the concept of a class so that different objects could share a protocol and the implementation of the messages.
- ◯ A useful generalization of this abstraction is the concept of *class inheritance*.
- ◯ A subclass inherits all state information (variables) and methods from its superclass.

- A subclass *differs* from its superclass in that:
  - it may have *additional variables*,
  - it may have *extra methods*, and
  - it may *override* existing methods with new method implementation.
- To illustrate the subclass concept, we assume now that our librarian would like to differentiate between student and teacher patrons, who have different loan limit.
- To represent this situation in **Smalltalk**, we can use two classes: `Teacher` and `Student` instead of the single class `Patron`.
- On the other hand, except for the message "`loan:`," the objects from the two classes should behave in exactly the same fashion. So we will maintain a class called `Patron` as the superclass of both `Teacher` and `Student`, and implement all common methods in `Patron`.
- The last implementation of class `Patron` will be modified as follows.
  - Remove the class variable "`LoanLimit`."
  - Remove the class initialization method "`initialize`" which sets the loan limit.
  - Remove the circulation method "`loan:`."

- First, define the class `Teacher`.

A Teacher Class in **Smalltalk**

Class Definition

```
Patron subclass: #Teacher
  instanceVariableNames: ''
  classVariableNames: 'LoanLimit'
  poolDictionaries: ''
  category: 'Library'
```

Class Methods

class initialization

```
initialize
  "Initialize the loan limit."
  LoanLimit := 50
```

Instance Methods

circulation

```
loan: anItem
  "If I have not reached the loan limit, then add the first
argument, anItem, to my loan list and answer myself.
Otherwise, answer nil."
  (loanList size < LoanLimit)
    ifTrue: [loanList add: anItem. ^self]
    ifFalse: [^nil]
```

- We can define the class `Student` in exactly the same way, except in the message `initialize`, use the literal 2 instead of 50.
- Notice, however, that the method `loan:` is identical in classes `Teacher` and `Student`.

- We can share this code by returning the method `loan:` to the class `Patron` and replacing the original references to the class variable `LoanLimit` of class `Patron` by an accessing message to retrieve the correct loan limit for each object.

```
loan: anItem
  "If I have not reached the loan limit, then add the first
argument, anItem, to my loan list and answer myself.
Otherwise, answer nil."
  (loanList size < self loanLimit)
    ifTrue: [loanList add: anItem. ^self]
    ifFalse: [^nil]
```

- In this case, we must add the accessing instance method `loanLimit` to each of `Teacher` and `Student`.

```
loanLimit
  "Answer the maximum number of items that can be on loan
to me."
  ^LoanLimit
```

- Notice that, although this method is the same for class `Teacher` and `Student`, it cannot be moved to class `Patron` since the class variable `LoanLimit` is not defined in `Patron`.
- Since we do not intend to and shall never instantiate an object of class `Patron` directly, we call `Patron` an *abstract superclass*.
- An abstract superclass is a class with one or more subclasses, which does not have sufficient functionality to make instances of it useful.

- The intention of an abstract superclass is to capture some common aspects of its subclasses, rather than having the functionality duplicated in several (sub)classes.
- `Object` is an abstract superclass for all classes in the **Smalltalk** system. Instances of `Object` are seldom useful.

An excursion on `self` *and* `super` *...*

SELF and SUPER in Smalltalk

- When a method contains a message whose receiver is `self`, the search for the method for that message begins in the instance's class, regardless of which class contains the method containing `self`.

e.g.

Class Name One
Superclass Object
Instance Methods

```
test
  ^1

result1
  ^self test
```

若self为class Two，则返回class Two的test值

Class Name Two
Superclass One
Instance Methods

```
test
  ^2

example1 := One new.
example2 := Two new.

example1 test          ↦ 1
example1 result1       ↦ 1
example2 test          ↦ 2
example2 result1       ↦ 2
```

- The pseudo-variable `self` can be used to implement recursive functions.

- When a message is sent to `super`, the search begins in the superclass of the class containing the method.
- The use of `super` allows a method to access methods defined in a superclass even if the methods have been overridden in subclasses.

e.g. Continuing from the last example ...

```
Class Name                    Three
Superclass                    Two
Instance Methods
result2
  ^self result1

result3
  ^super test

Class Name                    Four
Superclass                    Three
Instance Methods
test
  ^4

example3 := Three new.
example4 := Four new.

example3 test          ↦  2
example4 result1       ↦  4
example3 result2       ↦  2
example4 result2       ↦  4
example3 result3       ↦  2
example4 result3       ↦  2
```

(class 2)

在class 3的superclass里找test，不要在class 3里找，even this is called by class 4.

example3 test ↦ 2 class Two 的test

example4 result1 ↦ 4 → class Three → class Two → class One → class four 的test

example3 result2 ↦ 2 → class Two 的 result 1 → class One → class Two 的 test (因为class 3有test)

example4 result2 ↦ 4 → class Four 的test

Final: give example of why we use superclass and how we use

- Back from the excursion (or daydreaming) ...
- Notice that if an object from class `Patron` is created and the message `loan:` is sent to it, then the message expression "`self loanLimit`" would be executed.
- Since the `loanLimit` method is not in the protocol of `Patron`, the superclass of `Patron`, namely `Object`, will be required to respond and it will respond with an error message "`message not understood: loanLimit`."
- To prevent such system errors from happening, we should include a method for `loanLimit` in the class `Patron` as well.

```
loanLimit
  "Answer my loan limit.  This number is only known to my
subclasses."
  self subclassResponsibility
```

- The message `subclassResponsibility` is in the protocol of `Object` and results in the more *informative* error message: "`My subclass should have overriden one of my messages.`"

EXERCISE: Read chapters 1 and 2 of the Booch book.

final(限定符): reference environment