Objects, Encapsulation, Inheritance

John Mitchell

Reading (two lectures)
Chapter 10, except section 10.4
Chapter 11, sections 11.1, 11.2, 11.3.1 and 11.4, 11.5, 11.6 only
Chapter 12, sections 12.1, 12.2, 12.3 and 12.5 only
Chapter 13, sections 13.1 and 13.2 only

Warning

• This is a new course organization
  — The book covers selected object-oriented languages in historical order
  — Lectures this year are organized by concept instead of by language
  — Why the change?
    • Older languages seem old; let’s try something new
    • Add comparisons with newer languages

Outline

• Central concepts in object-oriented languages
  — Dynamic lookup, encapsulation, subtyping, inheritance
• Objects as activation records
  — Simula: implementation as activation records with static scope
• Pure dynamically-typed object-oriented languages
  — Object implementation and run-time lookup
  — Class-based languages (Smalltalk)
  — Prototype-based languages (Self, JavaScript)
• Statically-typed object-oriented languages (second lecture)
  — C++ — using static typing to eliminate search
    — problems with C++ multiple inheritance
  — Java — using Interfaces to avoid multiple inheritance

Object-oriented programming

• Primary object-oriented language concepts
  — dynamic lookup
  — encapsulation
  — inheritance
  — subtyping
• Program organization
  — Work queue, geometry program, design patterns
• Comparison
  — Objects as closures?

Objects

• An object consists of
  — hidden data
    instance variables, also called fields, data members, ...
    hidden functions also possible
  — public operations
    methods or member functions
  — can also have public variables
  in some languages

• Object-oriented program:
  — Send messages to objects
What’s interesting about this?

- Universal encapsulation construct
  - Data structure
  - File system
  - Database
  - Window
  - Integer
- Metaphor usefully ambiguous
  - Sequential or concurrent computation
  - Distributed, sync. or async. communication

Object-Orientation

- Programming methodology
  - Organize concepts into objects and classes
  - Build extensible systems
- Language concepts
  - Dynamic lookup
  - Encapsulation
  - Subtyping allows extensions of concepts
  - Inheritance allows reuse of implementation

Dynamic Lookup

- In object-oriented programming,
  object \( \rightarrow \) message (arguments)
  code depends on object and message
- In conventional programming,
  operation (operands)
  meaning of operation is always the same

Example

- Add two numbers \( x \rightarrow \text{add}(y) \)
  different add if \( x \) is integer, string
- Conventional programming \( \text{add}(x, y) \)
  function \( \text{add} \) has fixed meaning

Important distinction:
- Overloading is resolved at compile time
- Dynamic lookup is a run time operation

Language concepts

- “Dynamic lookup”
  - Different code for different objects
  - Integer “+” different from string “+”
- Encapsulation
- Subtyping
- Inheritance

Encapsulation

- Builder of a concept has detailed view
- User of a concept has “abstract” view
- Encapsulation separates these two views
  - Implementation code: operate on representation
  - Client code: operate by applying fixed set of operations provided by implementer of abstraction

Object
Language concepts

- “Dynamic lookup”
  - different code for different object
  - integer “+” different from real “+”
- Encapsulation
  - Implementer of a concept has detailed view
  - User has “abstract” view
  - Encapsulation separates these two views
- Subtyping
- Inheritance

Subtyping and Inheritance

- Interface
  - The external view of an object
- Subtyping
  - Relation between interfaces
- Implementation
  - The internal representation of an object
- Inheritance
  - Relation between implementations

Object Interfaces

- Interface
  - The messages understood by an object
- Example: point
  - x-coord: returns x-coordinate of a point
  - y-coord: returns y-coordinate of a point
  - move: method for changing location
- The interface of an object is its type

Subtyping

- If interface A contains all of interface B, then A objects can also be used B objects.

<table>
<thead>
<tr>
<th>Point</th>
<th>Colored_point</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-coord</td>
<td>x-coord</td>
</tr>
<tr>
<td>y-coord</td>
<td>y-coord</td>
</tr>
<tr>
<td>move</td>
<td>move</td>
</tr>
<tr>
<td></td>
<td>change_color</td>
</tr>
</tbody>
</table>

Colored_point interface contains Point
Colored_point is a subtype of Point

Inheritance

- Implementation mechanism
- New objects may be defined by reusing implementations of other objects

Example

```java
class Point
    private float x, y
    public point move(float dx, float dy);

class Colored_point
    private float x, y; color c
    public point move(float dx, float dy);
    point change_color(color newc);
```

- Subtyping
  - Colored points can be used in place of points
  - Property used by client program
- Inheritance
  - Colored points can be implemented by reusing point implementation
  - Technique used by implementer of classes
OO Program Structure

- Group data and functions
- Class
  - Defines behavior of all objects that are instances of the class
- Subtyping
  - Place similar data in related classes
- Inheritance
  - Avoid reimplementing functions that are already defined

Example: Geometry Library

- Define general concept: shape
- Implement two shapes: circle, rectangle
- Functions on implemented shapes
  - center, move, rotate, print
- Anticipate additions to library

Shapes

- Interface of every shape must include
  - center, move, rotate, print
- Different kinds of shapes are implemented differently
  - Square: four points, representing corners
  - Circle: center point and radius

Subtype hierarchy

- General interface defined in the shape class
- Implementations defined in circle, rectangle
- Extend hierarchy with additional shapes

Code placed in classes

<table>
<thead>
<tr>
<th></th>
<th>center</th>
<th>move</th>
<th>rotate</th>
<th>print</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>c_center</td>
<td>c_move</td>
<td>c_rotate</td>
<td>c_print</td>
</tr>
<tr>
<td>Rectangle</td>
<td>r_center</td>
<td>r_move</td>
<td>r_rotate</td>
<td>r_print</td>
</tr>
</tbody>
</table>

- Dynamic lookup
  - \texttt{circle} \rightarrow \texttt{move(x,y)} calls function \texttt{c\_move}
- Conventional organization
  - Place \texttt{c\_move}, \texttt{r\_move} in \texttt{move} function

Example use: Processing Loop

- Remove shape from work queue
- Perform action
- Control loop does not know the type of each shape
Outline

- Central concepts in object-oriented languages
  - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records
  - Simula – implementation as activation records with static scope
- Pure dynamically-typed object-oriented languages
  - Object implementation and run-time lookup
  - Class-based languages (Smalltalk)
  - Prototype-based languages (Self, JavaScript)
- Statically-typed object-oriented languages
  - C++ – using static typing to eliminate search
  - Java – using Interfaces to avoid multiple inheritance

Simula: objects as activation records

- Simula 67: First object-oriented language
- Designed for simulation
  - Later recognized as general-purpose prog language
- Extension of Algol 60
- Standardized as Simula (no “67”) in 1977
- Inspiration to many later designers
  - Smalltalk
  - C++
  - ...

Brief history

- Norwegian Computing Center
  - Designers: Dahl, Myhrhaug, Nygaard
  - Simula-1 in 1966 (strictly a simulation language)
  - General language ideas
    - Influenced by Hoare’s ideas on data types
    - Added classes and prefixing (subtyping) to Algol 60
  - Nygaard
    - Operations Research specialist and political activist
    - Wanted language to describe social and industrial systems
    - Allow “ordinary people” to understand political (?) changes
  - Dahl and Myhrhaug
    - Maintained concern for general programming

Objects in Simula

- Class
  - A procedure that returns a pointer to its activation record
- Object
  - Activation record produced by call to a class
- Object access
  - Access any local variable or procedures using dot notation: object.var
- Memory management
  - Objects are garbage collected
    - user destructors considered undesirable

Example: Circles and lines

- Problem
  - Find the center and radius of the circle passing through three distinct points, p, q, and r
- Solution
  - Draw intersecting circles Cp, Cq around p,q and circles Cq, Cr around q, r (Picture assumes Cq + Cr)
  - Draw lines through circle intersections
  - The intersection of the lines is the center of the desired circle.
  - Error if the points are collinear.

Approach in Simula

- Methodology
  - Represent points, lines, and circles as objects.
  - Equip objects with necessary operations.
- Operations
  - Point
    - equality(anotherPoint) : boolean (needed to construct circles)
  - Line
    - parallelTo(anotherLine) : boolean (to see if lines intersect)
    - meets(anotherLine) : REF(Point)
  - Circle
    - intersects(anotherCircle) : REF(Line)
Simula Point Class

```simula67
class Point(x,y); real x,y;
begin
    boolean procedure equals(p);
        real p;
    if p /= none then
        equals := abs(x - p.x) + abs(y - p.y) < 0.00001;
    end ***Point***
end
```

```
p := new Point(1.0, 2.5);
q := new Point(2.0, 3.5);
if p.distance(q) > 2 then ...
```

Simula line class

```simula67
class Line(a,b,c); real a,b,c;
begin
    boolean procedure parallelto(l);
        real l;
    if l /= none then
        parallelto := ...
    end ***Line***
end
```

```simula67
ref(Point)
```

Representation of objects

```
p <- access link
real x 1.0
real y 2.5

proc equals

proc distance

Object is represented by activation record with access link to find global variables according to static scoping
```

Derived classes in Simula

- A class decl may be prefixed by a class name
  - class A
  - A class B
  - A class C
  - B class D
- An object of a “prefixed class” is the concatenation of objects of each class in prefix
  - d := new D(...)
Simula Summary

- **Class**
  - "procedure" that returns ptr to activation record
  - initialization code always run as procedure body
- **Objects**: closure created by a class
- **Encapsulation**
  - protected and private not recognized in 1967
  - added later and used as basis for C++
- **Subtyping**: determined by class hierarchy
- **Inheritance**: provided by class prefixing

Outline

- **Central concepts in object-oriented languages**
  - Dynamic lookup, encapsulation, subtyping, inheritance
- **Objects as activation records**
  - Simula – implementation as activation records with static scope
  - Pure dynamically-typed object-oriented languages
    - Object implementation and run-time lookup
    - Class-based languages (Smalltalk)
    - Prototype-based languages (Self, JavaScript)
- **Statically-typed object-oriented languages**
  - C++ – using static typing to eliminate search
  - Java – using Interfaces to avoid multiple inheritance

Smalltalk

- **Major language that popularized objects**
- **Developed at Xerox PARC**
  - Smalltalk-76, Smalltalk-80 were important versions
- **Object metaphor extended and refined**
  - Used some ideas from Simula, but very different language
  - Everything is an object, even a class
  - All operations are "messages to objects"
  - Very flexible and powerful language
    - Similar to "everything is a list" in Lisp, but more so
    - Example: object can detect that it has received a message it doesn’t understand, can try to figure out how to respond.

Motivating application: Dynabook

- **Concept developed by Alan Kay**
- **Small portable computer**
  - Revolutionary idea in early 1970s
    - At the time, a minicomputer was shared by 10 people, stored in a machine room.
  - What would you compute on an airplane?
- **Influence on Smalltalk**
  - Language intended to be programming language and operating system interface
  - Intended for "non-programmer"
  - Syntax presented by language-specific editor

Smalltalk language terminology

- **Object** Instance of some class
- **Class** Defines behavior of its objects
- **Selector** Name of a message
- **Message** Selector together with parameter values
- **Method** Code used by a class to respond to message
- **Instance variable** Data stored in object
- **Subclass** Class defined by giving incremental modifications to some superclass

Example: Point class

- **Class definition written in tabular form**

<table>
<thead>
<tr>
<th>class name</th>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>super class</td>
<td>Object</td>
</tr>
<tr>
<td>class var</td>
<td>pi</td>
</tr>
<tr>
<td>instance var</td>
<td>x y</td>
</tr>
<tr>
<td>class messages</td>
<td></td>
</tr>
<tr>
<td>messages and methods</td>
<td></td>
</tr>
<tr>
<td>instance messages</td>
<td></td>
</tr>
</tbody>
</table>

(...names and code for methods...)
Class messages and methods

Three class methods
- **newX:x value Y:y value**
- ^self new x:x value y:y value

newOrigin ^self new x: 0 y: 0
initialize ^pi <- 3.14159

- **Explanation**
  - selector is mix-fix newX:Y:
  - e.g., Point newX:3 Y:2
  - symbol ^ marks return value
  - new is method in all classes, inherited from Object
  - ^ marks scope for local decl
  - initialize method sets pi, called automatically
  - ^ is syntax for assignment

Instance messages and methods

Five instance methods
- **newX:x Y:y**
- ^x <- newX
  ^y <- newY

moveX: dx Y: dy ^
  ^x <- dx + x
  ^y <- dy + y

initialize ^
  ^pi <- 3.14159

- **Explanation**
  - set x,y coordinates, e.g., pt x:5 y:3
  - move point by given amount
  - return hidden inst var x
  - return hidden inst var y
  - draw point on screen

Run-time representation of point

```
Point object
<table>
<thead>
<tr>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
```

Point class
```
x
```
```
y
```
```
Method dictionary
```
newX:Y: ^
  ^... code ...
```
```
move ^
  ^... code ...
```
```
to superclass Object
```

```
Detail: class method shown in dictionary, but lookup procedure distinguishes class and instance methods
```

Inheritance

- **Define colored points from points**

```
class name | ColorPoint
super class | Point
class var   | color
instance var| color
```
```
class messages and methods
```
```
newX:x Y:y C:cv ^
  ^... code ...
```
```
instance messages and methods
```
```
color | ^*color
```
```
draw | (^*code ...)
```
```
new method
```
```
new instance variable
```
```
override Point method
```

Encapsulation in Smalltalk

- **Methods are public**
- **Instance variables are hidden**
  - Not visible to other objects
    - pt x is not allowed unless x is a method
    - But may be manipulated by subclass methods
  - This limits ability to establish invariants
  - Example:
    - Superclass maintains sorted list of messages with some selector, say `insert`
    - Subclass may access this list directly, rearrange order
Smalltalk Summary

- **Class**
  - creates objects that share methods
  - pointers to template, dictionary, parent class
- **Objects**: created by a class, contains instance variables
- **Encapsulation**
  - methods public, instance variables hidden
- **Subtyping**: implicit, no static type system
- **Inheritance**: subclasses, self, super
  Single inheritance in Smalltalk-76, Smalltalk-80

Self programming language

- Prototype-based pure object-oriented language.
- Designed by Randall Smith (Xerox PARC) and David Ungar (Stanford University)
  - Successor to Smalltalk-80
  - "Self: The power of simplicity" appeared at OOPSLA ’87
  - Initial implementation done at Stanford; then project shifted to Sun Microsystems Labs
  - Vehicle for implementation research
- **Self 4.3 available from Oracle web site:**

Design Goals

- **Conceptual economy**
  - Everything is an object
  - Everything done using messages
  - No classes
  - No variables
- **Concreteness**
  - Objects should seem “real”
  - GUI to manipulate objects directly

“A Language for Smalltalk runtime structures”

How successful?

- **Self is a carefully designed language**
- **Few users: not a popular success**
  - No compelling application, until JavaScript
  - Influenced development of object calculi w/o classes
- **However, many research innovations**
  - Very simple computational model
  - Enormous advances in compilation techniques
  - Influenced the design of Java compilers

Language Overview

- **Dynamically typed**
- **Everything is an object**
- **All computation via message passing**
- **Creation and initialization: clone object**
- **Operations on objects:**
  - send messages
  - add new slots
  - replace old slots
  - remove slots
Objects and Slots

Object consists of named slots.
- Data
  • Such slots return contents upon evaluation; so act like instance variables
- Assignment
  • Set the value of associated slot
- Method
  • Slot contains Self code
- Parent
  • Point to existing object to inherit slots

Messages and Methods

- When message is sent, object searched for slot with name.
  - If none found, all parents are searched.
    - Runtime error if more than one parent has a slot with the same name.
  - If slot is found, its contents evaluated and returned.
    - Runtime error if no slot found.

Messages and Methods

```
obj x
obj x: 4
clone...
parent*
print...
parent*
x: 3
x: ←
```

Mixing State and Behavior

```
parent*
+ add points
parent*
x 4
y 17
x: ←
y: ←
```

Object Creation

- To create an object, we copy an old one
- We can add new methods, override existing ones, or even remove methods
- These operations also apply to parent slots

Changing Parent Pointers

```
frog
jump...

prince
dance...
eatCake...
```

```
p
p jump.
p eatFly.
p parent: prince.
p dance.
parent*
parent*:
name Charles
name: ←
```
Changing Parent Pointers

```
p jump.
p eatFly.
p parent: prince.
p dance.
```

Disadvantages of classes?

• Classes require programmers to understand a more complex model.
  – To make a new kind of object, we have to create a new class first.
  – To change an object, we have to change the class.
  – Infinite meta-class regression.
• But: Does Self require programmer to reinvent structure?
  – Common to structure Self programs with traits: objects that simply collect behavior for sharing.

Recall JavaScript Prototypes

• Every JavaScript object has a prototype
  – Object literals linked to Object.prototype
  – Otherwise, prototype based on constructor
    ```javascript
    function Foo() {
      this.x = 1;
    }
    obj = new Foo;
    ```
• Changing the JavaScript prototype
  – The prototype property is immutable
  – Changes to prototype property inherited immediately

Outline

• Central concepts in object-oriented languages
  – Dynamic lookup, encapsulation, subtyping, inheritance
• Objects as activation records
  – Simula – Implementation as activation records with static scope
• Pure dynamically-typed object-oriented languages
  – Object implementation and run-time lookup
  – Class-based languages (Smalltalk)
  – Prototype-based languages (Self, JavaScript)
• Statically-typed object-oriented languages (next lecture)
  – C++ – using static typing to eliminate search
    – problems with C++ multiple inheritance
  – Java – using Interfaces to avoid multiple inheritance