~ Lecture V ~

Object-oriented languages : Concepts and origins SIMULA and Smalltalk

References:

- ⋆ **Chapters 10 and 11** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.
- ♦ **Chapters 8, and 12(§§2 and 3)** of *Programming languages: Design and implementation* (3RD EDITION) by T. W. Pratt and M. V. Zelkowitz. Prentice Hall, 1999.

- ♦ **Chapter 7** of *Programming languages: Concepts & constructs* by R. Sethi (2ND EDITION). Addison-Wesley, 1996.
- ♦ **Chapters 14 and 15** of *Understanding programming languages* by M Ben-Ari. Wiley, 1996.
- ⋆ B. Stroustrup. What is "Object-Oriented Programming"? (1991 revised version). Proc. 1st European Conf. on Object-Oriented Programming. (Available on-line from ⟨http://public.research.att.com/~bs/papers.html⟩.)

Objects in ML !?

```
exception Empty ;
fun newStack(x0)
  = let val  stack = ref [x0]
    in ref{  push = fn(x)
               => stack := ( x :: !stack )    ,
             pop = fn()
               => case !stack of
                    nil => raise Empty
                  | h::t => ( stack := t; h )
    }end ;
exception Empty
val newStack = fn :
      'a -> {pop:unit -> 'a, push:'a -> unit} ref
```



```
val BoolStack = newStack(true) ;

val BoolStack = ref {pop=fn,push=fn}
  : {pop:unit -> bool, push:bool -> unit} ref

val IntStack0 = newStack(0) ;

val IntStack0 = ref {pop=fn,push=fn}
  : {pop:unit -> int, push:int -> unit} ref

val IntStack1 = newStack(1) ;

val IntStack0 = ref {pop=fn,push=fn}
  : {pop:unit -> int, push:int -> unit} ref
```

```
IntStack0 := !IntStack1 ;

val it = () : unit

#pop(!IntStack0)() ;

val it = 1 : int

#push(!IntStack0)(4) ;

val it = () : unit
```



```
map ( #push(!IntStack0) ) [3,2,1] ;

val it = [(),(),()] : unit list

map ( #pop(!IntStack0) ) [(),(),(),()] ;

val it = [1,2,3,4] : int list
```

NB:

- $\boxed{!}$ The *stack discipline* for activation records fails!
- $\boxed{?}$ Is ML an object-oriented language?

  $\boxed{!}$ Of course not!

  $\boxed{?}$ Why?

Basic concepts in object-oriented languages[a]

Four main language concepts for object-oriented languages:

1. Dynamic lookup.
2. Abstraction.
3. Subtyping.
4. Inheritance.

[a]Notes from **Chapter 10** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.

Dynamic lookup

- *Dynamic lookup* means that when a message is sent to an object, the method to be executed is selected dynamically, at run time, according to the implementation of the object that receives the message. In other words, the object "chooses" how to respond to a message.

  The important property of dynamic lookup is that different objects may implement the same operation differently, and so may respond to the same message in different ways.

- Dynamic lookup is sometimes confused with overloading, which is a mechanism based on *static types* of operands. However, the two are very different. $\boxed{?}$ Why?

There is a family of object-oriented languages that is based on the *"run-time overloading"* view of dynamic lookup. The most prominent design of this form is CLOS (= Common Lisp Object System), which features *multiple dispatch*.

Abstraction

- *Abstraction* means that implementation details are hidden inside a program unit with a specific interface. For objects, the interface usually consists of a set of methods that manipulate hidden data.
- Abstraction based on objects is similar in many ways to abstraction based on abstract data types: Objects and abstract data types both combine functions and data, and abstraction in both cases involves distinguishing between a public interface and private implementation.

  Other features of object-oriented languages, however, make abstraction in object-oriented languages more flexible than abstraction with abstract data types.

Subtyping

- *Subtyping* is a relation on types that allows values of one type to be used in place of values of another. Specifically, if an object `a` has all the functionality of another object `b`, then we may use `a` in any context expecting `b`.
- The basic principle associated with subtyping is *substitutivity*: If `A` is a subtype of `B`, then any expression of type `A` may be used without type error in any context that requires an expression of type `B`.

- The primary advantage of subtyping is that it permits uniform operations over various types of data.

  For instance, subtyping makes it possible to have heterogeneous data structures that contain objects that belong to different subtypes of some common type.
- Subtyping in an object-oriented language allows functionality to be added without modifying general parts of a system.

Inheritance

- *Inheritance* is the ability to reuse the definition of one kind of object to define another kind of object.
- The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code and that, when one class is implemented by inheriting form another, changes to one affect the other. This has a significant impact on code maintenance and modification.

Inheritance is not subtyping[a]

Subtyping is a relation on interfaces,
inheritance is a relation on implementations.

One reason subtyping and inheritance are often confused is that some class mechanisms combine the two. A typical example is C++, in which `A` will be recognized by the compiler as a subtype of `B` only if `B` is a public base class of `A`. Combining subtyping and inheritance is an elective design decision.

[a]Interested students may consult the POPL'90 paper *Inheritance is not subtyping* by W. R. Cook, W. L. Hill and P. S. Canning available on-line from ⟨http://www.cs.utexas.edu/users/wcook/publications.htm⟩.

History of objects
SIMULA and Smalltalk

- Objects were invented in the design of SIMULA and refined in the evolution of Smalltalk.
- SIMULA: The first object-oriented language.
  The object model in SIMULA was based on procedures activation records, with objects originally described as procedures that return a pointer to their own activation record.
- Smalltalk: A dynamically typed object-oriented language.
  Many object-oriented ideas originated or were popularised by the Smalltalk group, which built on Alan Kay's then-futuristic idea of the Dynabook.

SIMULA

- Extremely influential as the first language with classes objects, dynamic lookup, subtyping, and inheritance.
- Originally designed for the purpose of *simulation* by O.-J. Dahl and K. Nygaard at the Norwegian Computing Center, Oslo, in the 1960s.
- SIMULA was designed as an extension and modification of Algol 60. The main features added to Algol 60 were: class concepts and reference variables (pointers to objects); pass-by-reference; input-output features; coroutines (a mechanism for writing concurrent programs).

- A generic event-based simulation program

```
Q := make_queue(initial_event);
repeat
  select event e from Q
  simulate event e
  place all events generated by e on Q
until Q is empty
```

naturally requires:

  - A data structure that may contain a variety of kinds of events. $\leadsto$ subtyping
  - The selection of the simulation operation according to the kind of event being processed. $\leadsto$ dynamic lookup
  - Ways in which to structure the implementation of related kinds of events. $\leadsto$ inheritance

Objects in SIMULA

Class: A procedure returning a pointer to its activation record.

Object: An activation record produced by call to a class, called an instance of the class. $\leadsto$ a SIMULA object is a closure

- SIMULA implementations place objects on the heap.
- Objects are deallocated by the garbage collector (which deallocates objects only when they are no longer reachable from the program that created them).

SIMULA

Object-oriented features

- *Objects*: A SIMULA object is an activation record produced by call to a class.
- *Classes*: A SIMULA class is a procedure that returns a pointer to its activation record. The body of a class may initialise the objects it creates.
- *Dynamic lookup*: Operations on an object are selected from the activation record of that object.

- *Abstraction*: Hiding was not provided in SIMULA 67 but was added later and used as the basis for C++.
- *Subtyping*: Objects are typed according to the classes that create them. Subtyping is determined by class hierarchy.
- *Inheritance*: A SIMULA class may be defined, by class prefixing, as an extension of a class that has already been defined including the ability to redefine parts of a class in a subclass.

SIMULA 67 did not distinguish between public and private members of classes.

A later version of the language, however, allowed attributes to be made "protected", which means that they are accessible for subclasses (but not for other classes), or "hidden", in which case they are not accessible to subclasses either.

SIMULA

Further object-related features

- *Inner*, which indicates that the method of a subclass should be called in combination with execution of superclass code that contains the `inner` keyword.
- *Inspect* and *qua*, which provide the ability to test the type of an object at run time and to execute appropriate code accordingly. (`inspect` is a class (type) test, and `qua` is a form of type cast that is checked for correctness at run time.)

SIMULA

Sample code[a]

```
CLASS POINT(X,Y); REAL X, Y;
  COMMENT***CARTESIAN REPRESENTATION
BEGIN
  BOOLEAN PROCEDURE EQUALS(P); REF(POINT) P;
    IF P =/= NONE THEN
      EQUALS := ABS(X-P.X) + ABS(Y-P.Y) < 0.00001;
  REAL PROCEDURE DISTANCE(P); REF(POINT) P;
    IF P == NONE THEN ERROR ELSE
      DISTANCE := SQRT( (X-P.X)**2 + (Y-P.Y)**2 );
END***POINT***
```

[a] See Chapter 4(§1) of *SIMULA begin* (2nd edition) by G. Birtwistle, O.-J. Dahl, B. Myhrhug, and K. Nygaard. Chartwell-Bratt Ltd., 1980.


```
CLASS LINE(A,B,C); REAL A,B,C;
  COMMENT***Ax+By+C=0 REPRESENTATION
BEGIN
  BOOLEAN PROCEDURE PARALLELTO(L); REF(LINE) L;
    IF L =/= NONE THEN
      PARALLELTO := ABS( A*L.B - B*L.A ) < 0.00001;

  REF(POINT) PROCEDURE MEETS(L); REF(LINE) L;
    BEGIN REAL T;
      IF L =/= NONE and ~PARALLELTO(L) THEN
        BEGIN
          ...
          MEETS :- NEW POINT(...,...);
        END;
    END;***MEETS***
```

```
  COMMENT*** INITIALISATION CODE
  REAL D;
  D := SQRT( A**2 + B**2 )
  IF D = 0.0 THEN ERROR ELSE
    BEGIN
      D := 1/D;
      A := A*D;  B := B*D;  C := C * D;
    END;
END***LINE***
```


SIMULA

Subclasses and inheritance

SIMULA syntax for a class C1 with subclasses C2 and C3 is

```
CLASS C1
  <DECLARATIONS1>;
C1 CLASS C2
  <DECLARATIONS2>;
C1 CLASS C3
  <DECLARATIONS3>;
```

When we create a C2 object, for example, we do this by first creating a C1 object (activation record) and then appending a C2 object (activation record).

Example:

```
POINT CLASS COLOREDPOINT(C); COLOR C;
BEGIN
  BOOLEAN PROCEDURE EQUALS(Q); REF(COLOREDPOINT) Q;
    ...;
END***COLOREDPOINT**

REF(POINT) P;  P :- NEW POINT(1.0,2.5);

REF(COLOREDPOINT) CP;  CP :- NEW COLOREDPOINT(2.5,1.0,RED);
```

NB: SIMULA 67 did not hide fields. Thus,

```
CP.C := BLUE;
```

changes the color of the point referenced by CP.

SIMULA

Object types and subtypes

- All instances of a class are given the same *type*. The name of this type is the same as the name of the class.
- The class names (types of objects) are arranged in a *subtype* hierarchy corresponding exactly to the subclass hierarchy.

Examples:

1.
```
CLASS A;    A CLASS B;
REF(A) a;   REF(B) b;
a :- b; COMMENT***legal since B is a subclass of A
...
```

```
b :- a; COMMENT***also legal, but checked at run
           ***time to make sure that a points
           ***to a B object, so as to avoid a
           ***type error
```

2. `inspect a`
   ```
   when B do b :- a
   otherwise ...
   ```

3. An error in the original SIMULA type checker surrounding the relationship between subtyping and inheritance:

   `CLASS A;    A CLASS B;`

   SIMULA subclassing produces the subtype relation `B<:A`.

   `REF(A) a;   REF(B) b;`

   SIMULA also uses the semantically incorrect principle

that, if `B<:A` then `REF(B)<:REF(A)`.

This code . . .

```
PROCEDURE ASSIGNa( REF(A) x )
  BEGIN  x :- a  END;
ASSIGNa(b);
```

. . . will statically type check, but may cause a type error at run time.

P.S. The same type error occurs in the original implementation of Eiffel.[a]

[a] Interested students may consult *A proposal for making Eiffel type-safe* by W. Cook in *The Computer Journal*, 32(4):305-311, 1989. Available on-line from ⟨`http://www.cs.utexas.edu/users/wcook/publications.htm`⟩.

Smalltalk

- Developed at XEROX PARC in the 1970s.
- Major language that popularised objects; very flexible and powerful.
- The object metaphor was extended and refined.
  - Used some ideas from SIMULA; but it was a completely new language, with new terminology and an original syntax.
  - Abstraction via private *instance variables* (data associated with an object) and public *methods* (code for performing operations).
  - Everything is an object; even a class. All operations are messages to objects.

Smalltalk

Motivating application: Dynabook

- Concept developed by Alan Kay.
- Influence on Smalltalk:
  - Objects and classes as useful organising concepts for building an entire programming environment and system.
  - Language intended to be the operating system interface as well as the programming language for Dynabook.
  - Syntax designed to be used with a special-purpose editor.
  - The implementation emphasised flexibility and ease of use over efficiency.

Smalltalk
Terminology

- *Object*: A combination of private data and functions. Each object is an *instance* of some class.
- *Class*: A template defining the implementation of a set of objects.
- *Subclass*: Class defined by inheriting from its superclass.
- *Selector*: The name of a message (analogous to a function name).
- *Message*: A selector together with actual parameter values (analogous to a function call).
- *Method*: The code in a class for responding to a message.
- *Instance variable*: Data stored in an individual object (instance class).

Smalltalk
Classes and objects

class name	`Point`
super class	`Object`
class var	`pi`
instance var	`x, y`
class messages and methods	
⟨...names and codes for methods ...⟩	
instance messages and methods	
⟨...names and codes for methods ...⟩	

Definition of `Point` class

A class message and method for point objects

```
newX:xvalue Y:yvalue ||
   ^ self new x: xvalue y: yvalue
```

A new point at coordinates $(3, 4)$ is created when the message

```
newX:3 Y:4
```

is sent to the `Point` class.

For instance:

```
p <- Point newX:3 Y:4
```


Some instance messages and methods

```
x || ^x
y || ^y
moveDx: dx Dy: dy ||
    x <- x+dx
    y <- y+dy
```

Executing the following code

```
p moveDX:2 Y:1
```

the value of the expressions `p x` and `p y` is the object `5`.

Smalltalk
Inheritance

class name	`ColoredPoint`
super class	`Point`
class var	
instance var	`color`
class messages and methods	
`newX:xv Y:yv C:cv`	⟨. . . code . . .⟩
instance messages and methods	
`color`	`||^color`
`draw`	⟨. . . code . . .⟩

Definition of `ColoredPoint` class

- `ColoredPoint` inherits instance variables `x` and `y`, methods `x`, `y`, `moveDX:Dy:`, *etc.*
- `ColoredPoint` adds an instance variable `color` and a method `color` to return the `color` of a `ColoredPoint`.
- The `ColoredPoint draw` method *redefines* (or *overrides*) the one inherited from `Point`.
- An option available in Smalltalk is to specify that a superclass method should be undefined on a subclass.

Example: Consider

```
newX:xv Y:yv C:cv ||
   ^ self new x:xv y:yv color:cv
cp <- ColoredPoint newX:1 Y:2 C:red
cp moveDx:3 Dy:4
```

The value of `cp x` is the object `4`, and the value of the expression `cp color` is the object `red`.

Note that even though `moveDx:Dy:` is an inherited method, defined originally for points without color, the result of moving a `ColoredPoint` is again a `ColoredPoint`.

Smalltalk
Abstraction

Smalltalk rules:

- *Methods are public.*

  Any code with a pointer to an object may send any message to that object. If the corresponding method is defined in the class of the object, or any superclass, the method will be invoked. This makes all methods of an object visible to any code that can access the object.
- *Instance variables are protected.*

  The instance variables of an object are accessible only to methods of the class of the object and to methods of its subclasses.

Smalltalk
Dynamic lookup

The run-time structures used for Smalltalk classes and objects support *dynamic lookup* in two ways.

1. Methods are selected through the receiver object.
2. Method lookup starts with the method dictionary of the class of the receiver and then proceeds upwards through the class hierarchy.

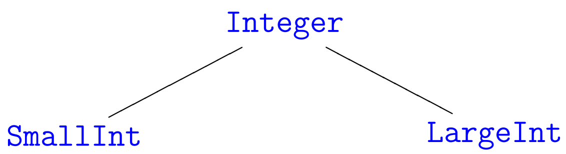
Smalltalk
Self and super

- The special symbol `self` may be used in the body of a Smalltalk method. The special property of `self` is that it always refers to the object that contains this method, whether directly or by inheritance.
- The special symbol `super` is similar to `self`, except that, when a message is sent to `super`, the search for the appropriate method body starts with the superclass of the object instead of the class of the object. This mechanism provides a way of accessing a superclass version of a method that has been overridden in the subclass.

Example: A factorial method

```
factorial ||
   self <= 1
      ifTrue: [^1]
      ifFalse: [^ (self-1) factorial * self]
```

in the `Integer` class for



Smalltalk
Interfaces as object types

Although Smalltalk does not use any static type checking, there is an implicit form of type that every Smalltalk programmer uses in some way.

The *type* of an object in Smalltalk is its interface, *i.e.* the set of messages that can be sent to the object without receiving the error “message not understood”.

The interface of an object is determined by its class, as a class lists the messages that each object will answer. However, different classes may implement the same messages, as there are no Smalltalk rules to keep different classes from using the same selector names.

Smalltalk
Subtyping

Type `A` is a *subtype* of type `B` if any context expecting an expression of type `B` may take any expression of type `A` without introducing a type error.

Semantically, in Smalltalk, it makes sense to associate *subtyping* with the *superset* relation on class interfaces.

? Why?

- In Smalltalk, the interface of a subclass is often a subtype of the interface of its superclass. The reason being that a subclass will ordinarily inherit all of the methods of its superclass, possibly adding more methods.
- In general, however, subclassing does not always lead to subtyping in Smalltalk.
  1. Because it is possible to delete a method from a superclass in a subclass, a subclass may not produce a subtype.
  2. On the other hand, it is easy to have subtyping without inheritance.

Smalltalk
Object-oriented features

- *Objects*: A Smalltalk object is created by a class.

  At run time, an object stores its instance variables and a pointer to the instantiating class.
- *Classes*: A Smalltalk class defines variables, class methods, and the instance methods that are shared by all objects of the class.

  At run time, the class data structure contains pointers to an instance variable template, a method dictionary, and the superclass.

- *Abstraction*: Abstraction is provided through protected instance variables. All methods are public but instance variables may be accessed only by the methods of the class and methods of subclasses.
- *Subtyping*: Smalltalk does not have a compile-time type system. Subtyping arises implicitly through relations between the interfaces of objects. Subtyping depends on the set of messages that are understood by an object, not the representation of objects or whether inheritance is used.
- *Inheritance*: Smalltalk subclasses inherit all instance variables and methods of their superclasses. Methods defined in a superclass may be redefined in a subclass or deleted.