

Lectures 8-9

# Object-oriented programming languages

Iztok Sarnik, FAMNIT

April, 2023.

# Literature

- Textbooks:

John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapters 10-13)

Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapter 9)

- Examples from:

Emmanuel Chailoux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, O'REILLY & Associates, 2000 (Chapter 15)

# Outline

1. Introduction
2. Classes and objects
3. Inheritance
4. Dynamic binding
5. Subtyping and substitutivity
6. Abstract classes
7. Genericity
8. Popular OO programming languages

# Data abstraction

- Data abstraction
  - Enforces a clear separation between the abstract properties and the concrete details of a data type
  - Information hiding, representation independence, encapsulation, interface/implementation, ...
- With development of complicated computer apps, data abstraction has become essential to sw eng.
  - It reduces conceptual load by minimizing the amount of detail that the programmer must think about at one time
  - Fault containment by preventing the programmer from using a program component in inappropriate ways
  - Provides a degree of independence among program components that can be easily separated by functionality

# Data abstraction and encapsulation

- Class/object abstraction
  - Creates a simplified view of modeled class/object
  - Focusing attention to the most important features
  - Omitting the representation and implementation details
  - Data is hidden in object
  - External observer (of class) needs to see only abstract view of class and its instances, i.e., class interface
  - Interface consists of well-defined methods that manipulate instances of class
- Classes/objects encapsulate internal structure and behavior of object

# Object-oriented model

- An object responds to a set of operations on some hidden data
  - Object representation abstracts away all details about the implementation of object
  - All interactions with an object occur by means of messages or member-function calls
- Objects are grouped into classes that serve as object prototypes
  - Classes are sets of their instances (denotational view)
  - As prototypes, classes stand for types comprised of method signatures and data members
  - Classes inherit properties of their super-classes

# Example: int\_stack

- Stack of integers
  - Ocaml implementation
  - Stack is represented by list
- Encapsulation
  - Representation of stack and its implementation can change
  - Interface (abstraction) stays the same

```
# class int_stack =
  object
    val mutable l = ([] : int list)
    method push x = l <- x::l
    method pop = match l with
                  [] -> raise Empty |
                  a::l' -> l <- l'; a

    method clear = l <- []
    method length = List.length l
  end;;

# let is = new int_stack;;
val is : int_stack = <obj>
# is#push 1; is#push 2;
  is#push 3; is#push 4;;
- : unit = ()
# is#length;;
- : int = 4
# is#pop;;
- : int = 4
```

# History

- Simula, 1960, Norwegian Computing Center
  - The first object-oriented language
  - It included everything recent OO languages have
  - Object/classes, inheritance, subclasses, virtual procedures
- Smalltalk, 1970, Xerox PARC
  - Dynamically typed object-oriented language
  - Everything is object; still remains to be interesting design
  - Object/classes, messages, subclasses, metaclasses
  - Structural and computational reflection: can observe/influence its own structure and behaviour



# Mature OO programming languages

- C++, 1983, Bjarne Stroustrup, Bell Labs
  - Widely used statically typed object-oriented language
  - Extension of the C language; standardized 2014
  - Classes/objects, inheritance (multiple), polymorphism, templates, virtual function and classes, exceptions
  - Compatibility with C
- Java, 1995, James Gosling, Sun Microsystems
  - Concurrent, class-based, object-oriented
  - Compiled to bytecode and run on any Java virtual machine (JVM)
  - Single inheritance, abstract classes, interfaces, generics, introspection, file-system based modules
  - Portability, reliability, safety, dynamic linking, threads, simplicity, efficiency

# Outline

1. Introduction
2. Classes and objects
3. Inheritance
4. Dynamic binding
5. Subtyping and substitutivity
6. Abstract classes
7. Genericity
8. Popular OO programming languages

# Classes and objects

- Object model is close to simulation view to problem solving: make a simulation model of problem and run it
  - Objects represent components of model
  - Objects are interconnected by possibly very complex static structures
  - Object may ask other objects to run a task by sending it a message
  - Objects have behavior realized by communication with other interrelated objects
  - Abstractions used are much closer to human representation of problem

# Classes and objects

- Class can be viewed as **prototype object** from which its instances (elements of the class) are created
  - Class includes the definition of static structure and behavior of a prototype object
  - Behavior of class instances is implemented in methods
  - Methods use internal logic to communicate with other objects to solve the sub-problem
  - New classes can be constructed from existing classes
    - specialization and generalization
    - composition and decomposition

# Class definition in Ocaml

- Data members
  - Can have any type
  - Value or mutable

```
val name = expr  
val mutable name = expr  
name <- expression
```

- Methods
  - Methods have parameters p1,...,pn

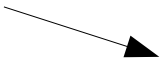
```
method name p1 . . . pn = expr
```

- Class can have parameters

```
class name p1 ... pn =  
  object  
  ...  
  instance variables  
  ...  
  methods  
  ...  
end
```

```
#class point x_init =  
  object  
    val mutable x = x_init  
    method get_x = x  
    method move d = x <- x + d  
  end;;  
class point :  
  int ->  
  object val mutable x : int method get_x : int method move : int -> unit end
```

Type!



# Object creation

```
# new point;;  
- : int -> point = <fun>  
# let p = new point 7;;  
val p : point = <obj>
```

- Most languages have operator `new( )`
  - Ocaml does not have constructors as C++, Java
  - Class parameters are accessible for the definition of class variables
  - Class acts as generator i.e. function that creates objects
- Initialization of objects in Ocaml
  - Code can be added before and after object creation
  - Before: `let` statement can be put before `object`
  - After: Ocaml uses special function as initializer
- How is the initialization of objects done in Java?

# Example:

```
# class printable_point x_init =
  let origin = (x_init / 10) * 10 in
  object (self)
    val mutable x = origin
    method get_x = x
    method move d = x <- x + d
    method print = print_int self#get_x
    initializer print_string "new point at ";
                  self#print; print_newline()
  end;;

class printable_point :
  int ->
  object
    val mutable x : int
    method get_x : int
    method move : int -> unit
    method print : unit
  end

# let p = new printable_point 17;;
new point at 10 val p : printable_point = <obj>
```

# Methods

- Sending a message
  - `o#message( )` (Ocaml)

```
gp_list_node::remove(); // C++
super.remove();        // Java
base.remove();         // C#
super remove.          // Smalltalk
[super remove]         // Objective-C
```

- Function call in C++, Java, Ocaml, ...
- Actual messages in Smalltalk, Erlang, ...
- A method must be defined in a class
- Types of actual parameters must match type of formal parameters

```
# class point (x_init,y_init) =
  object
    val mutable x = x_init
    val mutable y = y_init
    method get_x = x
    method get_y = y
    method moveto (a,b) = x <- a ; y <- b
    method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
    method to_string () =
      "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^ ")"
    method distance () = sqrt (float(x*x + y*y))
  end ;;
```

```
# let p1 = new point (0,0);;
val p1 : point = <obj>
# let p2 = new point (3,4);;
val p2 : point = <obj>
# p1#get_x;;
- : int = 0
# p2#to_string () ;;
- : string = "(3, 4)"
# if (p1#distance () ) = (p2#distance () )
  then print_string ("That's just chance\n")
  else print_string ("We could bet on it\n");;
We could bet on it
- : unit = ()
```



# Private methods

Ocaml

- Private and public methods

```
method private name = expr
```

- Private methods are accessible from object only

- Example of private methods

- Point remembers previous position

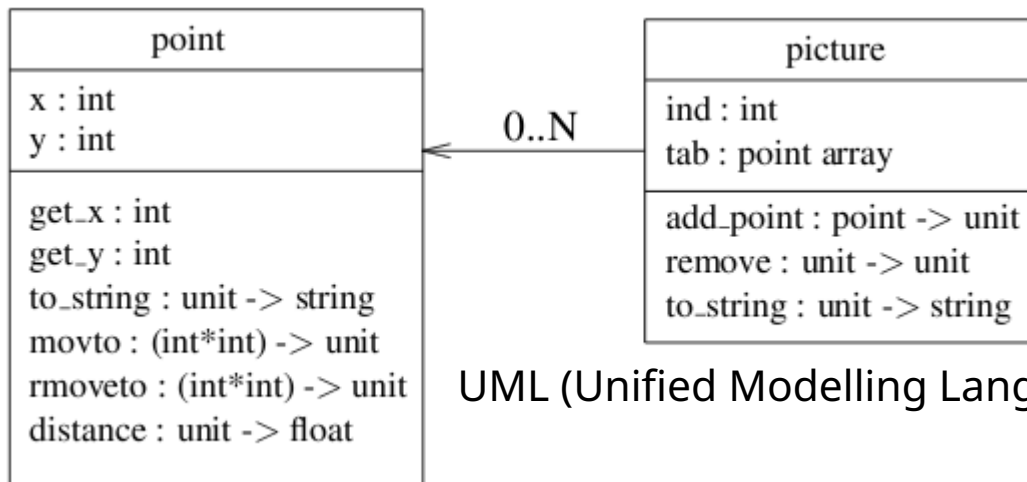
- mem\_pos()

```
# class point (x0,y0) =
  object(self)
    val mutable x = x0
    val mutable y = y0
    val mutable old_x = x0
    val mutable old_y = y0
    method get_x = x
    method get_y = y
    method private mem_pos () = old_x <- x ; old_y <- y
    method undo () = x <- old_x; y <- old_y
    method moveto (x1, y1) = self#mem_pos (); x <- x1; y <- y1
    method rmoveto (dx, dy) = self#mem_pos (); x <- x+dx; y <- y+dy
    method to_string () =
      "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^ ")"
    method distance () = sqrt (float(x*x + y*y))
  end ;;
```

# Structures of objects and aggregation

- An object can be a component of other object
  - Class defines other classes as components
  - Object includes references to other objects specifying a link among two or more objects
- Aggregation is one of two important abstractions used in OO languages
  - Aggregation (composition): “has - a”
  - Inheritance (specialization): “is - a”
- Example in Ocaml
  - Class `Picture` includes an array of type `Point`
  - `to_string()` calls methods `to_string()` of class `Point`

# Example



UML (Unified Modelling Language)

## Type

```
# class picture n =
  object
  val mutable ind = 0
  val tab = Array.create n (new point(0,0))
  method add p = tab.(ind)<-p ; ind <- ind + 1
  method remove () = if (ind > 0) then ind <-ind-1
  method to_string () =
    let s = ref "["
    in for i=0 to ind-1 do
      s:= !s ^ " " ^ tab.(i)#to_string () done ;
    (!s) ^ "]"
  end ;;
```

```
class picture :
  int ->
  object
  val mutable ind : int
  val tab : point array
  method add : point -> unit
  method remove : unit -> unit
  method to_string : unit -> string
end
```

# Outline

1. Introduction
2. Classes and objects
3. Inheritance
4. Dynamic binding
5. Subtyping and substitutivity
6. Abstract classes
7. Genericity
8. Popular OO programming languages

# Inheritance

- Inheritance realizes specialization abstraction
  - Aristotel: genus and differentia specifica.
  - Sub-class is a specialization of super-class
  - Sub-class inherits all properties of super-class
- Ocaml syntax for definition of inheritance

```
inherit name1 p1 . . . pn [ as name2 ]
```

- Parameters  $p_1, \dots, p_n$  are constructor parameters of super-class
- Super-class part of object can be referenced inside the class definition using variable `name2`

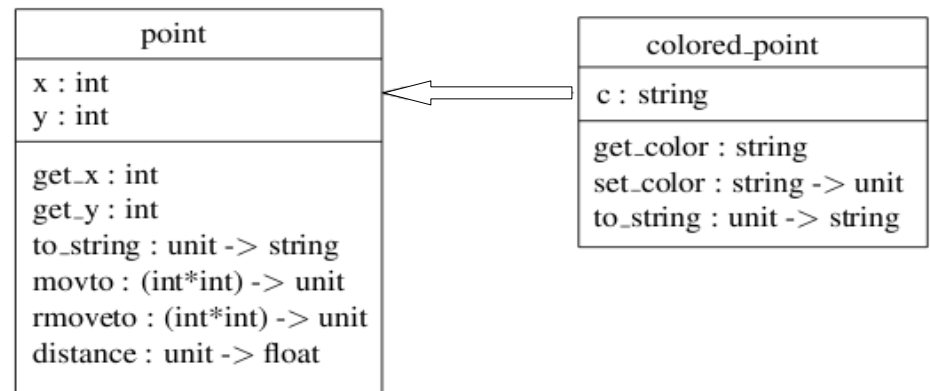
# Example of sub-class

```
# class colored_point (x,y) c =  
  object  
  inherit point (x,y)  
  val mutable c = c  
  method get_color = c  
  method set_color nc = c <- nc  
  method to_string () = "(" ^ (string_of_int x) ^  
    "," ^ (string_of_int y) ^ ")" ^  
    "[" ^ c ^ "]"  
  
end ;;
```

```
# let pc = new colored_point (2,3) "white";;  
val pc : colored_point = <obj>  
# pc#get_color;;  
- : string = "white"  
# pc#get_x;;  
- : int = 2  
# pc#to_string () ;;  
- : string = "(2,3)[white] "  
# pc#distance;;  
- : unit -> float = <fun>
```

Type:

```
class colored_point :  
  int * int ->  
  string ->  
  object  
  val mutable c : string  
  val mutable x : int  
  val mutable y : int  
  method distance : unit -> float  
  method get_color : string  
  method get_x : int  
  method get_y : int  
  method moveto : int * int -> unit  
  method rmoveto : int * int -> unit  
  method set_color : string -> unit  
  method to_string : unit -> string  
end
```



# Method overriding

- Method `to_string()` of class `colored_point()` overrides method `to_string()` of class `point()`
  - Overriding method can be more specific than the overridden method
  - Relationship between types of methods will be presented in section on types (of classes)
- Consequences of method overriding (through inheritance)
  - Dynamic binding of method name to correct code
    - Otherwise method code can be determined statically
  - Multiple inheritance
    - More than one methods of the same type can be inherited

# References to self and super

- While defining methods it is useful to have access
  - to the object itself, as well as
  - to the part of object that is described by super-class
- Reference to object itself
  - Keywords “self” in Smalltalk. Java and C++ ?
  - Ocaml allows definition of custom name to self
- Reference to super-object
  - Ocaml allows own definition of reference to super
  - Access to variable in case it is overloaded by some other variable or parameter
  - Calling object's own methods

```
super.remove();    // Java
base.remove();     // C#
super remove.      // Smalltalk
[super remove]    // Objective C
```



# References to self and super

- Method `to_string()`
  - Using `to_string()` of super-class
  - Using `self` to call method `get_color()`

```
# class colored_point (x,y) c =  
  object (self)  
  inherit point (x,y) as super  
  val mutable c = c  
  method get_color = c  
  method set_color nc = c <- nc  
  method to_string () = super#to_string () ^  
    "["^ self#get_color ^ "]"  
  
end ;;
```

# Object initialization

- Instance of a specialized class C is initialized by
  - Code of initializer-s of all super-classes of C
  - Initializer code of C
- Order of evaluation of initializer-s is from the most general towards more specific
- How is an object initialized in Java? C++?

# Example of object initialization

```
# class point (x_init,y_init) =
  object (self)
    val mutable x = x_init
    val mutable y = y_init
    method get_x = x
    method get_y = y
    method moveto (a,b) = x <- a ; y <- b
    method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
    method to_string () =
      "(" ^ (string_of_int x) ^ "," ^ (string_of_int y) ^ ")"
    method distance () = sqrt (float(x*x + y*y))
    initializer
      Printf.printf ">> Creation of point: %s\n"
        (self#to_string ());
  end ;;
```

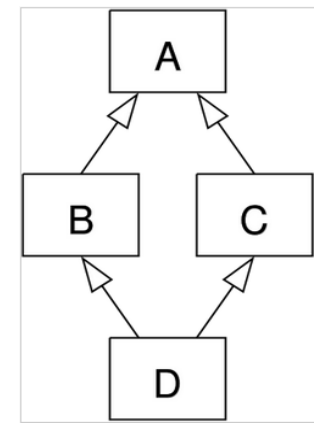
```
# class verbose_point p =
  object (self)
    inherit point p as super
    method to_string () = "point=" ^ (super#to_string ()) ^
      ",distance=" ^ string_of_float (self#distance ())
    initializer
      Printf.printf ">> Creation of verbose point: %s\n"
        (self#to_string ())
  end ;;
# new verbose_point (1,1);;
>> Creation of point: (1,1)
>> Creation of verbose point: point=(1,1), distance=1.414213
- : verbose_point = <obj>
```

# Multiple inheritance

- A concept modeled by a class may be a specialization of more than one concepts
  - A class may have more than one base classes
  - Some languages allow only single base class
    - Java, C#, Ruby, ...
  - Some language can use multiple base classes
    - C++, Ocaml, Perl, ...
- Common conceptual mistake
  - Car can be seen as a class that inherits from classes Wheels, Motor, etc.
  - Specialization should not be used to model composition, although it is technically possible

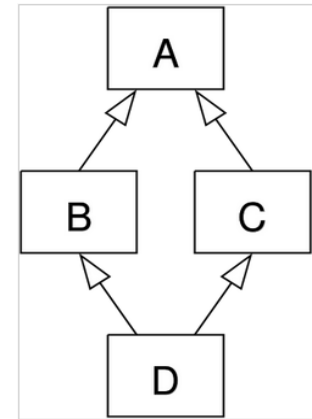
# Name clashes

- Method *m* is defined in A, B and C.
  - Method *m* is sent to instance of C.
  - Which instance to choose?
- Implicit resolution of name clashes
  - Language resolves name conflicts with a set of rules
  - Rules use syntactical order of superclass specification
    - Python, CLOS, Perl, ...: 1st definition in sup-class hierarchy (recursive traversal)
    - Ocaml: last definition in superclasses hierarchy (recursive traversal)
  - Dylan, Python, Perl: C3 method resolution order [OPSLA,96]



# Name clashes

- Explicit resolution of name clashes
  - The programmer must explicitly resolve name conflicts in the code in some way (C++, Ocaml, Eiffel, Python)
  - Method must be identified explicitly (unambiguously)
  - Ocaml uses the name to refer to a super-class
  - C++ uses the paths to the target class
    - Example: `D::B::m()`, `C::m()`
- Disallow name clashes
  - Programs are not allowed to contain name clashes
    - Java, C#, Swift, Go, Scala... but Java 8 (diamond problem with interfaces)



# C3 superclass linearization

- Defines the order in which a method is searched in the superclass hierarchy with multiple inheritance.
- **C3 Method Resolution Order (MRO)**
  - Recursive method (similar to topological sorting?)
    - Starts with the base (top) class; Descends level by level; Using linearizations of parents to compute linearizations of children; Nice merging algorithm (selects next class in ordering)
  - Used in: Dylan, Python, Perl, ...
- C3 superclass linearization results in **three important properties**:
  - Consistent extended precedence graph,
  - Preservation of local precedence order, and
  - Fitting the monotonicity criterion

# Example of multiple inheritance

```
# class colored_point (x,y) c =  
  object (self)  
    inherit point (x,y) as super  
    val mutable c = c  
    method get_color = c  
    method set_color nc = c <- nc  
    method to_string () = super#to_string () ^  
      "[" ^ self#get_color ^ "]"  
  end ;;
```

```
# class reference n d =  
  object  
    val mutable name = n  
    val mutable descr = d  
    method to_string () = "{" ^ name ^ "  
  end ;;
```

```
# class reference_point (x,y) c n d =  
  object (self)  
    inherit colored_point (x,y) c as cp  
    inherit reference n d as ref  
  end ;;  
  
# let rp1 = new reference_point (1,1) "red" "r1"  
"reference point";;  
  
val rp1 : reference_point = <obj>  
  
# rp1#get_x;;  
- : int = 1  
  
# rp1#to_string ();;  
- : string = "{r1}"
```

Why?



# Example of multiple inheritance

```
# class reference_point (x,y) c n d =  
  object (self)  
    inherit colored_point (x,y) c as cp  
    inherit reference n d as ref  
    method to_string () = cp#to_string () ^ ref#to_string ()  
  end ;;  
# let rp1 = new reference_point (1,1) "red" "rp1" "reference point";;  
val rp1 : reference_point = <obj>  
# rp1#to_string ();;  
- : string = "(1, 1)[red]{rp1}"
```

# C++ example of multiple inheritance

```
// multiple inheritance.cpp
#include
using namespace std;
class A
{
    public:
    int x;
    void getx()
    {
        cout << "enter value of x: "; cin >> x;
    }
};
class B
{
    public:
    int y;
    void gety()
    {
        cout << "enter value of y: "; cin >> y;
    }
};
```

```
class C : public A, public B //C is derived from classes A and B
{
    public:
    void sum()
    {
        cout << "Sum = " << x + y;
    }
};

int main()
{
    C obj1; //object of derived class C
    obj1.getx();
    obj1.gety();
    obj1.sum();
    return 0;
} //end of program
```

```
enter value of x: 5
enter value of y: 4
Sum = 9
```

# Diamond problem

- Problem with multiple inheritance

- 1) Name clashes

- 2) Multiple copies of object A part

- Name clashes

- There are more than one solutions to name clashes

- See few slides back...

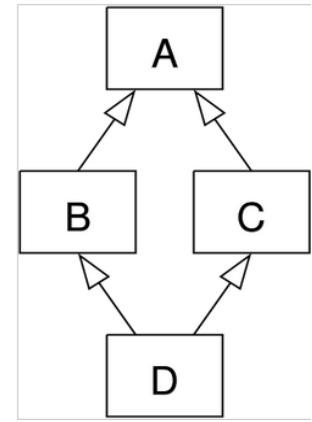
- Multiple copies of A

- D inherits from B and C

- B and C inherit data members and methods from A

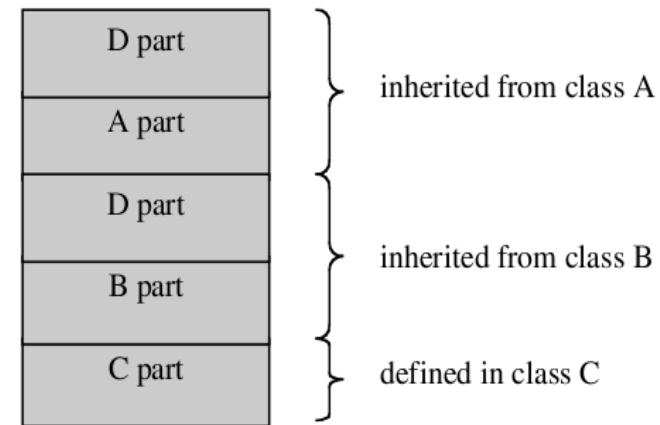
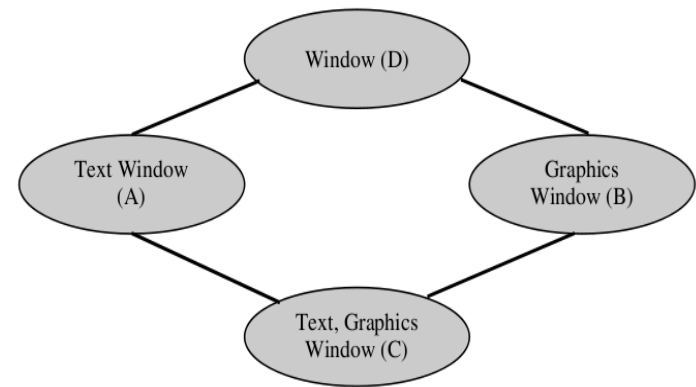
- Instance of D includes two copies of data members from A

- Solutions?



# Diamond problem

- Description of problem
  - Multiple instances of object D part
  - Since A and B inherit from D they both include instance of D
  - What to do?
- No proper solution!
  - Two copies of D may work well
    - Naming problem appears
    - C++ and Ocaml can follow each inheritance path separately
  - Put one copy of D in C instance
    - C++ virtual base classes
    - A and B instances refer to the same D part
    - Also a problem if A and B treat D part differently



# Outline

1. Introduction
2. Classes and objects
3. Inheritance
4. Dynamic binding
5. Subtyping and substitutivity
6. Abstract classes
7. Genericity
8. Popular OO programming languages

# Method binding

- When and how is method bound to its code?
- Dynamic and static binding
  - Dynamic binding takes place in run-time
    - Smalltalk, Objective-C, Ocaml, Python, Ruby (only), C++, C#, Go, Java, Eiffel (default); final classes can be optimized
  - Static binding happens at compile time
    - Simula, C++, C#, Scala, Ada95 (default)
  - Not exclusive; some PLs use both.
- Dynamic binding is more expensive than static.
  - Appropriate method is searched (accessed) in the method lookup table (abbr. MLT)
    - Smalltalk searches the table in run-time
    - C++, Ocaml, ... computes an index at compile time; on method lookup method address is accessed from MLT..

# Dynamic binding

- Consequences of inheritance/type extension
  - Derived class D has all the members—data and subroutines—of its base class C
  - Anything we might want to do to an object of class C we can also do to an object of class D
- When a message is sent to an object:
  - Function code (or method) is determined by the way that the object is implemented.
  - Object “chooses” how to respond to a message.
  - Diff. objects may implement same operation differently.
- Very useful PL feature
  - Sending the same message to a collection of objs from different classes.

# Dynamic binding

- Classical example of the use of dynamic binding
  - We have a collection of objects of class C
  - Collection can include also the instances of any class S such that S inherits from C
  - The same method, for instance `to_string()`, is called for each of the objects from the collection
- Example in Ocaml
  - We have defined classes `point`, `colored_point` and `verbose_point`
  - All of them have method `to_string()`



# Example 1

```
# class picture n =  
  object  
    val mutable ind = 0  
    val tab = Array.create n (new point(0,0))  
    method add p = tab.(ind)<-p ; ind <- ind + 1  
    method remove () = if (ind > 0) then ind <-ind-1  
    method to_string () =  
      let s = ref "["  
      in for i=0 to ind-1 do s:= !s ^ " " ^ tab.(i)#to_string () done ;  
        (!s) ^ "]"  
  end ;;
```

```
# let pic = new picture 3;  
>> Creation of point: (0,0)  
val pic : picture = <obj>  
# pic#add (new point (1,1));  
  pic#add ((new colored_point (2,2) "red") :> point);  
  pic#add ((new verbose_point (3,3)) :> point);;  
- : unit = ()  
# pic#to_string () ;;  
- : string = "[ (1,1) (2,2)[red] point=(3,3),distance=4.24264068712]"
```

# Example 2

```
# class colored_point (x,y) c =  
  object (self)  
    inherit point (x,y) as super  
    val mutable c = c  
    method get_color = c  
    method set_color nc = c <- nc  
    method to_string () = super#to_string () ^  
                          "["^ self#get_color ^ "]"  
  
  end ;;
```

```
# let p1 = new colored_point (1,1) "Blue";;  
val p1 : colored_point = <obj>  
# p1#to_string () ;;  
- : string = "(1,1) [Blue] "  
# let p2 = new colored_point_1 (1,1) "Blue";;  
val p2 : colored_point_1 = <obj>  
# p2#to_string () ;;  
- : string = "(1,1) [UNKNOWN] "
```

```
# class colored_point_1 coord c =  
  object  
    inherit colored_point coord c  
    val true_colors = ["white"; "black"; "red"; "green"; "blue"; "yellow"]  
    method get_color = if List.mem c true_colors then c else "UNKNOWN"  
  
  end ;;
```

Why?

# Implementation of objects in Ocaml

- Each object is represented by two parts
- Variable part
  - Includes object variables as in the case of records
  - Using reference or value model (some use both)
- Fixed part
  - Method lookup table (abbrev. MLT) stores methods that can be looked up dynamically
  - All methods, including the inherited methods, are stored in a MLT (one) of a class
  - Fixed part is the same for all instances of some class
  - There are different implementations of MLT (in diff PLs)

# Method lookup (binding)

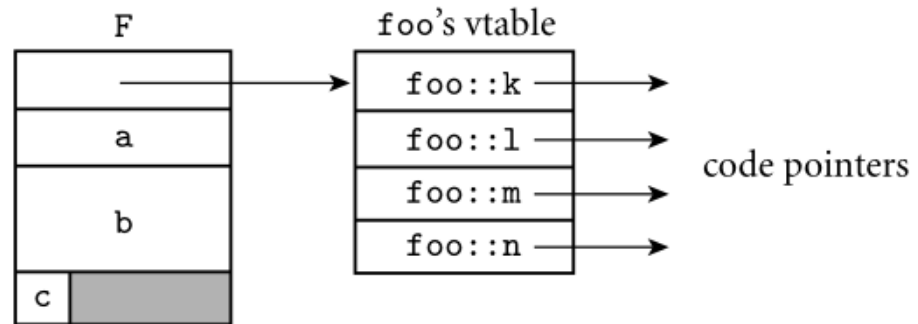
- Find a concrete method for a method call  $o.m(v_1, v_2, \dots)$
- Static method lookup
  - Compiler can determine the method using the type of object.
- Dynamic method lookup
  - Code generated by the compiler must find the right method.
  - Object is represented by a record that contains the pointer to the method lookup table of parent class.
    - References to newly defined methods are added
    - Overriding  $\Rightarrow$  ref to the old method is replaced with new one
    - The same index maintained for the same method in all method lookup tables of classes from a family (Ocaml)
  - Pointer to the type descriptor can be added to method lookup table to be able to check the type in run-time

# Implementation of objects in C++

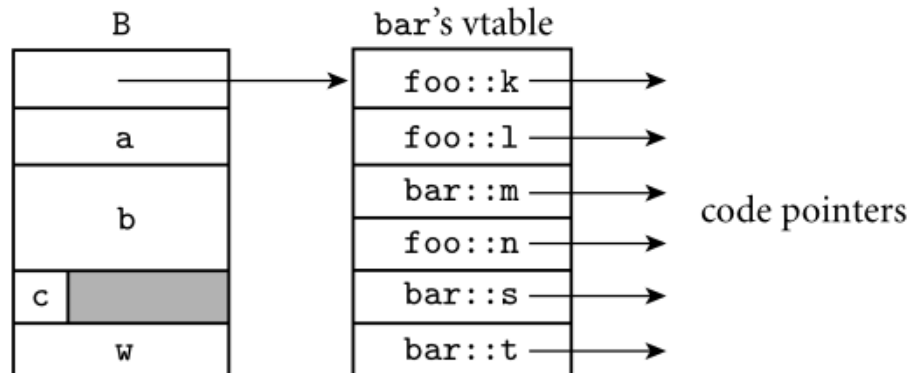
- Example in C++
  - Static and dynamic parts of objects
  - C++ checks types statically
  - Dynamic binding allowed for virtual methods only

# Implementation of objects in C++

```
class foo {  
    int a;  
    double b;  
    char c;  
public:  
    virtual void k( ...  
    virtual int l( ...  
    virtual void m();  
    virtual double n( ...  
    ...  
} F;
```



```
class bar : public foo {  
    int w;  
public:  
    void m() override;  
    virtual double s( ...  
    virtual char *t( ...  
    ...  
} B;
```



# Outline

1. Introduction
2. Classes and objects
3. Inheritance
4. Dynamic binding
5. Subtyping and substitutivity
6. Abstract classes
7. Genericity
8. Popular OO programming languages

# Subtypes

- Sub-typing is a relation on types that allows values of one type to be used in place of values of another.
  - Assuming that we are in a typed language,
- In Ocaml the type of class C is a record containing the types of all member functions of C
  - The type of class Point
- Formally, type of class includes only public functions

```
< distance : unit → float; get_x : int; get_y : int;  
  moveto : int * int → unit; rmoveto : int * int → unit;  
  to_string : unit → string >
```

```
class point :  
  int * int ->  
  object  
    val mutable x : int  
    val mutable y : int  
    method distance : unit -> float  
    method get_x : int  
    method get_y : int  
    method moveto : int * int -> unit  
    method rmoveto : int * int -> unit  
    method to_string : unit -> string  
  end
```



# Subtypes

In Ocaml class is function  
that constructs objects

- Type is different concept to class
  - Class is a factory, a prototype, that generates objects
  - Type is conceptually one level higher entity than its instances
- The subtype relationship is defined among types of classes
  - Type  $t'$  is a subtype of  $t$ , denoted by  $t' \leq t$ , if and only if  $\sigma_i \leq \tau_i$  for  $i \in \{1, \dots, n\}$   
 $t = \langle m_1 : \tau_1 ; \dots m_n : \tau_n \rangle$  and  
 $t' = \langle m_1 : \sigma_1 ; \dots ; m_n : \sigma_n ; m_{n+1} : \sigma_{n+1} ; \dots \rangle$
  - Type  $t'$  can have more components than  $t$  and types of common components must be in subtype relationship

# Subtyping functions

- Function call
  - If  $f : t \rightarrow s$ , and if  $a : t'$  and  $t' \leq t$  then  $(f a)$  is well typed, and has type  $s$
  - Intuitively, a function  $f$  expecting an argument of type  $t$  may safely receive an argument of a subtype  $t'$  of  $t$
  - Why? Instances of  $t'$  have either additional components, or, more specific components
- Function type
  - Type  $t' \rightarrow s'$  is a subtype of  $t \rightarrow s$ , denoted by  $t' \rightarrow s' \leq t \rightarrow s$ , if and only if  $s' \leq s$  and  $t \leq t'$
  - The relation  $s' \leq s$  is called covariance, and the relation  $t \leq t'$  is called contravariance.
  - Although surprising, relation can easily be justified

# Covariance/contravariance

- Justification of covariance/contravariance
  - Let  $f : t \rightarrow s$  and  $f' : t' \rightarrow s'$ .
  - **When the call of  $f$  can be replaced by a call of  $f'$  ?**
  - Argument of  $f$  can be used as the argument of  $f'$  if  $t \leq t'$ . This is contravariance.
  - Result of  $f'$  is acceptable in context of the result of  $f$  if  $s' \leq s$ . This is covariance.
  - Therefore, we have  $t' \rightarrow s' \leq t \rightarrow s$  iff  $t \leq t'$  and  $s' \leq s$ .

# Subtyping methods

- Assume two classes  $c1$  and  $c2$  both have a method  $m$  and
  - Method  $m$  has type  $t_1 \rightarrow s_1$  in  $c1$ , and type  $t_2 \rightarrow s_2$  in  $c2$
  - $m_{(1)}$  the method  $m$  of  $c1$  and  $m_{(2)}$  that of  $c2$
  - $c2 \leq c1$  and  $t_2 \rightarrow s_2 \leq t_1 \rightarrow s_1$
- Let  $g : s_1 \rightarrow \alpha$ , and  $g(o\#m(x))$  where  $o:c1$  and  $x:t_1$ 
  - $g$  defines the context, i.e., type  $s_1$
  - Other scenario:  $a = o\#m(x)$  where  $a:s_1$  and  $x:t_1$
- Covariance
  - Originally,  $o$  is an object of type  $c1$
  - $c2 \leq c1 \Rightarrow$  it is legal to use an object  $o$  of type  $c2$
  - $o\#m(x)$  is  $m_{(2)}$ , which returns a value of type  $s_2$
  - $g$  expects an argument of type  $s_1 \Rightarrow s_2 \leq s_1$  is OK

# Subtyping methods

- Contravariance
  - Method  $m$  requires a parameter value of type  $t_1$
  - We use  $o$  of type  $c_2 \Rightarrow m_{(2)}$  is invoked
  - It expects an argument of type  $t_2$ , so  $t_1 \leq t_2$  is OK
- Covariance/contravariance in PLs
  - Some languages use invariance for parameters (C++, C#)
  - Java, C++, C# support covariant return type
  - Contravariance of parameters used by Python (mypy) and Sather
    - Scala use contravar. on collection types (set, array, ...)
  - Contravariance seems unnecessary ...
    - $\text{employee}\#\text{follow}(\text{employee}) \leq \text{person}\#\text{follow}(\text{person})$  ??
    - The setup for methods is different to the setup for functions?
  - Covariance among parameters seems reasonable
    - Eiffel and Dart use this approach
    - This is not type safe

# Substitutivity

- The basic principle associated with subtyping is substitutivity
  - We write  $A \leq B$  to indicate that A is a subtype (sub-class) of B
  - If  $A \leq B$  then instance of A can appear in all contexts instance of B is expected
  - Function  $f : B \rightarrow C$  can be applied to any object of type A if  $A \leq B$ .
  - Instance of class  $A \leq B$  can be assigned to variable of class B
  - Collection of type B can include instances of type  $A \leq B$
  - Etc.

# Type coercion in Ocaml

- Most object-oriented programming languages support substitutivity
  - Ocaml uses explicit type coercion <:

```
(name : sub_type :> super_type)
(name :> super_type)
```

- Type coercion (**upcast**) :>
  - Object of type `colored_point` is treated as an instance if its superclass `point`
    - It remains to be `colored_point`!
  - After type coercion object still knows it is of original type!
    - ... and uses its methods!

```
# let cp = new colored_point (1,1)
"red";;
val cp : colored_point = <obj>
# let p = (cp :> point);;
val p : point = <obj>
# p#get_color ();;
Error: This expression has type point
It has no method get_color
# p#to_string ();;
- : string = "(1,1)[red]"
```

# Upcast/downcast in Java

- Java and C++ use type casting
  - (Implicit) **upcast**
    - Assignment statement, or explicit upcast
    - Can access overridden methods

```
Apple apple = new Apple();  
Fruit castedApple = apple;  
Fruit castedApple = (Fruit)apple;
```

- An explicit **downcast** type conversion
  - An object has to be of the downcasted type to be able to downcast it !

Leads to exception

```
Fruit fruit = new Fruit();  
Apple notApple = (Apple) Fruit;
```

Correct

```
Fruit fruit = new Apple();  
Apple castedApple = (Apple) fruit;
```



# Run-time (inclusion) polymorphism

- Dynamic binding and subtyping provide the means for expressing **run-time polymorphism**
- Let  $C$  be a base class that includes subclasses  $S_i$ 
  - Class  $C$  is the root of “family” of classes  $S_i$
  - Subtyping (substitutivity) allows treating instances of type  $S_i$  as instances of class  $C$
  - Dynamic binding assures that given method name and an instance of  $S_i$ , method will be linked to class  $S_i$
- Run-time (inclusion) polymorphism
  - Method  $m$  called for any instance  $o$  of some  $S_i$  depends on the implementation of  $m$  in  $S_i$
  - Sending the same message to objects of different “shapes” gives different responses.

# Outline

1. Introduction
2. Classes and objects
3. Inheritance
4. Dynamic binding
5. Subtyping and substitutivity
6. Abstract classes
7. Genericity
8. Popular OO programming languages

# Abstract classes

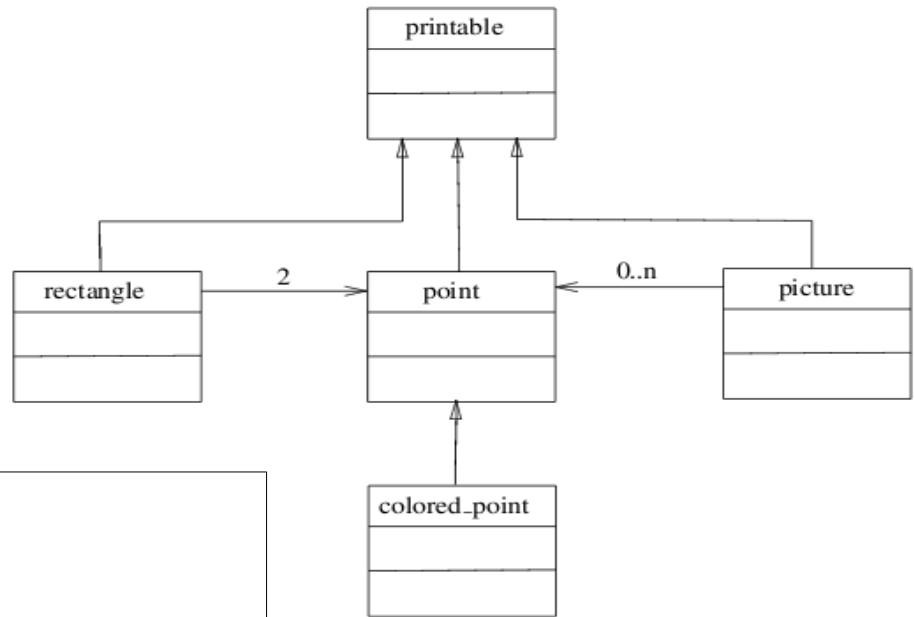
- In many OO programs, it is useful to define general concepts as a root of some family of classes
  - General concepts are implem. as abstract base classes
    - They are not implemented themselves
  - The only purpose of an abstract class is to serve as a base for other, concrete classes
    - They are the generalizations of concrete classes from a family
    - Examples: container, account, shape, or vehicle, etc.
- Abstract classes serve an organizational purpose
  - They have no instances
  - They define a common interface for the family

# Abstract classes

- Abstract class includes virtual methods
  - Virtual methods are solely defined by specifying their types (signatures)
  - Class must be **abstract** if it includes one virtual method
  - When a sub-class implements all virtual methods it becomes **concrete**
    - Otherwise sub-class must be defined abstract
  - Ocaml abstract classes
- Multiple inheritance of abstract classes
  - If multiple data members are defined then last definition is visible

```
class virtual name = object . . . end  
method virtual name : type
```

# Example



```
# class virtual printable () =
  object(self)
  method virtual to_string : unit -> string
  method print () = print_string (self#to_string () )
end ;;

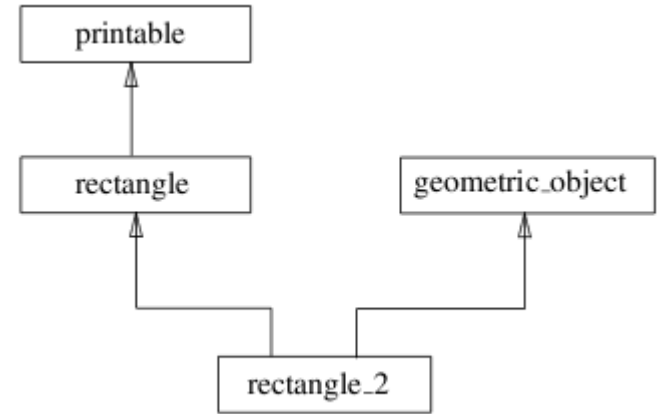
# class rectangle (p1,p2) =
  object
  inherit printable ()
  val llc = (p1 : point)
  val urc = (p2 : point)
  method to_string () = "["^llc#to_string ()^","^urc#to_string ()^"]"
End ;;

# let r = new rectangle (new point (2,3), new point (4,5));;
val r : rectangle = <obj>
# r#print () ;;
[(2,3),(4,5)]- : unit = ()
```

abstract method in a base class provides a “hook” for dynamic method binding

print () is implemented by using a method that is not yet implemented

# Example 2



```
# class virtual geometric_object () =
  object
  method virtual compute_area : unit -> float
  method virtual compute_peri : unit -> float
end;;

# class rectangle_2 (p2 : point * point) =
  object
  inherit rectangle p2
  inherit geometric_object ()
  method compute_area () =
    float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
  method compute_peri () =
    float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
end;;
```

**Multiple inheritance from  
abstract classes**

# Outline

1. Introduction
2. Classes and objects
3. Inheritance
4. Dynamic binding
5. Subtyping and substitutivity
6. Abstract classes
7. Genericity
8. Popular OO programming languages

# Genericity

- Generic programming
  - Programs based on types that are specified later
- Forms of genericity
  - Polymorphic functions, polymorphic types
  - Parameterized classes, functors (param. modules)
- Evolved from type theory and study of polymorphic lambda calculus
  - Approach is pioneered by ML (1972)
  - Definition of types, ADTs, classes **including types as parameters**



# Parameterized classes

- Parametric polymorphism is extended to classes
  - Ocaml defines it as extension of parametric types
    - Type variables are used in the definition of class
  - Syntax is close to definition of parametric types

```
class ['a,'b,...] name = object . . . end
```

- Used in many languages
  - Templates in C++
  - Generics in Java, C#, Go
  - Genericity in Delphi, Haskell, Scala

# Example

First approach:  
don't care about types.  
Bad idea?

Correct definition.

```
# class pair x0 y0 =  
  object  
    val x = x0  
    val y = y0  
    method fst = x  
    method snd = y  
  end ;;
```

Characters 6-106:

Some type variables are unbound in this type:

```
class pair :
```

```
'a ->
```

```
'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

The method fst has type 'a where 'a is unbound

```
# class ['a,'b] pair (x0:'a) (y0:'b) =  
  object  
    val x = x0  
    val y = y0  
    method fst = x  
    method snd = y  
  end ;;
```

```
class ['a, 'b] pair :
```

```
'a ->
```

```
'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

Parameterized class

```
# let p = new pair 2 'X';;  
val p : (int, char) pair = <obj>  
# p#fst;;  
- : int = 2  
# let q = new pair 3.12 true;;  
val q : (float, bool) pair = <obj>  
# q#snd;;  
- : bool = true
```

# Inheritance of parameterized classes

Definition of a class by inheriting from parameterized class

```
# class point_pair (p1,p2) =
  object
    inherit [point,point] pair p1 p2
  end;;
class point_pair :
  point * point ->
  object
    val x : point
    val y : point
    method fst : point
    method snd : point
  end
```

```
# class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
  object
    inherit ['a,'b] pair x0 y0
    method get1 z = if x = z then y else raise Not_found
    method get2 z = if y = z then x else raise Not_found
  end;;
class ['a, 'b] acc_pair :
  'a ->
  'b ->
  object
    val x : 'a
    val y : 'b
    method fst : 'a
    method get1 : 'a -> 'b
    method get2 : 'b -> 'a
    method snd : 'b
  end
# let p = new acc_pair 3 true;;
val p : (int, bool) acc_pair = <obj>
# p#get1 3;;
- : bool = true
```

Parameterized class can inherit from parameterized class

# Example: Stack in Ocaml

```
# class ['a] stack =
  object
    val mutable l = ([] : 'a list)
    method push x = l <- x::l
    method pop = match l with
      [] -> raise Empty |
      a::l' -> l <- l'; a

    method clear = l <- []
    method length = List.length l
  end;;

class ['a] stack :
object
  val mutable l : 'a list
  method clear : unit
  method length : int
  method pop : 'a
  method push : 'a -> unit
end
```

```
# let s = new stack;;
val s : '_a stack = <obj>
# s#push 1;;
- : unit = ()
# s#push 2;;
- : unit = ()
# s#pop;;
- : int = 2
# s#pop;;
- : int = 1
```

# Example: Stack in Scala

```
class Stack[A] {  
  private var elements: List[A] = Nil  
  def push(x: A): Unit =  
    elements = x :: elements  
  def peek: A = elements.head  
  def pop(): A = {  
    val currentTop = peek  
    elements = elements.tail  
    currentTop  
  }  
}
```

# Outline

1. Introduction
2. Classes and objects
3. Inheritance
4. Dynamic binding
5. Subtyping and substitutivity
6. Abstract classes
7. Genericity
8. Popular OO programming languages

# Scala

- Martin Odersky, EPFL, Lausanne, 2003
- Multi-paradigm: concurrent, functional, imperative, object-oriented
- Influenced by Lisp, Eiffel, Erlang, F#, Scheme, Haskell, Java, Ocaml, SML, ...
- Features
  - Immutability, Currying, polymorphism, higher-order functions, lazy evaluation, continuations, pattern matching, strong typing,
  - Pure OO language, every value is object, classes and traits, multiple inheritance, algebraic data types, type inference (Curry-style), co/contravariance, higher-order types, generic classes, ... JVM
- Concurrent and distributed
  - Concurrent: Actor model (from Erlang), asynchronous prog.
  - Distributed: Apache Spark

# Google Go

- Statically typed, compiled high-level PL, by Google
  - Robert Griesemer, Rob Pike, and Ken Thompson, 2007
- Main features
  - Designed for: multicore, networked machines and large codebases
  - C syntax, memory safety, garbage collection, structural typing, abstract data struct
- Concepts of Golang
  - Variables, Constants, For, If/Else, Switch, Arrays, Slices, Maps, Functions, Closures, Recursion, Pointers, Structs, Methods, Interfaces, Generics
  - Goroutines, Channels, Async. messages, Timeouts, Timers, Counters, Mutexes, Processes, Signals
  - In many ways close to Erlang (Ericson); that is Yahoo's choice



# C#

- Anders Hejlsberg, Microsoft, 2000
- .NET Framework implementation (initial name, Cool)
- Multi-paradigm programming language
  - Imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines
- Features
  - Strong types, type inference,
  - Data structures make high-level programming language
    - Arrays, collections, sets, dictionaries, sets, lists, queue/stack, bags, ...
- Development
  - C# 1.0 -> Java
  - C# 2.0 -> functional, generics, partial types, iterators, static classes,...
  - C# 3.0-7.0 -> Dynamic binding, named/optional arguments, asynchronous methods, compiler-as-service, exceptions, out variables, pattern-matching, query expressions, lambda expressions, ...
  - C# 8.0 -> Readonly members, default interface methods, pattern matching enhancements, static local functions, asynchronous streams, indices and ranges, ...

# F#

- Don Syme (BDFL), Microsoft Research, 2005
  - ML family, based on Ocaml
  - Influenced by C#, Python, Haskell, Scala, and Erlang.
  - Multi-paradigm programming language
    - Functional, imperative, modular and object-oriented programming
  - Some features
    - Strongly typed, type inference, eager evaluation, closures, lambda expressions, higher-order functions, pattern matching
  - Programming styles
    - Asynchronous, parallel, meta, agent
- Implementation
  - .NET Framework implementation of Ocaml core
  - Common Language Infrastructure (CLI), JavaScript and GPU code