Outline

1. Classes and Objects
2. Virtual and Private Methods
3. Subtyping and Coercions
4. Class Interfaces
5. Inheritance
6. Parameterized Classes
7. Polymorphic Methods
8. Recursive Classes
9. Functional Objects
10. Object Cloning
11. Friends
12. Module System vs Class System
13. OOP of OCaml vs OOP of Java/C++
Classes can be defined using the `class` and `object` keywords.

```ocaml
class point =
  object
    val mutable x = 0
    method get_x = x
    method move d = x ← x + d
  end;

class point :
  object val mutable x : int method get_x
    : int method move : int → unit end
```

- Class bodies are evaluated at creation time.
Classes can be initialized using the `new` keyword.

```ocaml
let x0 = ref 0;;

class point =
  object
    val mutable x = incr x0; !x0
    method get_x = x
    method move d = x ← x + d
  end;;

new point#get_x;;
- : int = 1

new point#get_x;;
- : int = 2
```
Parameters can be specified after the class name.

```ocaml
class adjusted_point x_init =
  let origin = (x_init / 10) * 10 in
  object
    val mutable x = origin
    method get_x = x
    method get_offset = x - origin
    method move d = x <- x + d
  end;;
```

- Class bodies can contain expressions.
If a reference to `self` is to be used, it must be explicitly bound.

- Binding will occur at invocation time.

```ocaml
let ints = ref [];;

class my_int =
  object (self)
    method n = 1
    method register = ints := self :: !ints
  end;;
```

Error: This expression has type `<n : int; register : 'a; ..>` but an expression was expected of type `'b`

Self type cannot escape its class

Error caused by external reference to `self`.

Mohammad Alam and Damith A. Karunaratne
OCaml Objects and Classes
Referencing Self

If a reference to `self` is to be used, it must be explicitly bound.

- Binding will occur at invocation time.

```ocaml
let ints = ref [];;

class my_int =
    object (self)
        method n = 1
        method register = ints := self :: !ints
    end;;

Error: This expression has type < n : int; register : 'a; .. >
but an expression was expected of type 'b
Self type cannot escape its class
```

- Error caused by external an reference to `self`. 

Initializers can be used to evaluate an expression immediately after the object is built.

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

let p = new printable_point 17;;
new point at 10
```

- Initializers cannot be overridden and are evaluated sequentially.
Objects can be created without a class.

\[
\text{let } \text{minmax } x \ y = \\
\text{if } x < y \text{ then object method } \text{min} = x \text{ method } \text{max} = y \text{ end} \\
\text{else object method } \text{min} = y \text{ method } \text{max} = x \text{ end};
\]

\[
\text{val minmax : } 'a \rightarrow 'a \rightarrow < \text{max} : 'a; \text{min} : 'a > = <\text{fun}>
\]

- (+) Immediate objects can appear within expressions.
- (-) No type abbreviation.
- (-) Cannot inherit from immediate objects.
In OCaml, one can define **virtual methods** as well as **virtual instance variables**.

```ocaml
class virtual abstract_point =
  object
    val mutable virtual x : int
    method get_x = x
    method virtual move : int \rightarrow unit
  end;;

class point x_init =
  object
    inherit abstract_point
    val mutable x = x_init
    method move d = x ← x + d
  end;;
```

- A class containing virtual methods or virtual instance variables must be flagged **virtual**, and cannot be **instantiated**.
A **private method** do not appear in the *object interface*.

```ocaml
class restricted_point x_init =
  object (self)
    val mutable x = x_init
    method get x = x
    method private move d = x ← x + d
    method bump = self # move 1
  end;;
```

*Type: \(<\text{get : int}; \text{bump : unit}>\) (in the context of type expression)*

- Private methods are inherited, i.e., they are visible in subclasses.
- Private methods can be made public in a subclass.
- Private methods can also be virtual and is defined like, *method private virtual identifier : type*. 
Subtyping and Coercions

**Subtype:** A type $t_1$ is a subtype of $t_2$, written $t_1 <: t_2$, if values of type $t_1$ can be used where values of type $t_2$ are expected. Example,

```plaintext
type animal = < eat : unit >
type dog = < eat : unit; bark : unit >
```

Then subtyping relation dog $<$: animal holds.

There are two forms of Subtyping:

1. **width subtyping:** It means if a subtype $t_1$ implements all the methods (and possibly more) of $t_2$ with the same method types.

2. **depth subtyping:** It is defined as: “If each method type $s_i$ is a subtype of method type $t_i$, then the object type $< f_1..n : s_1..n >$ is a subtype of the object type $< f_1..n : t_1..n >$. [Hic08]”
**Coercions**, in OCaml is never implicit or automatic. In a coercions \((obj : t_1 :> t_2)\), there are two necessary conditions:

1. the expression \(obj\) should have type \(t_1\); and
2. type \(t_1\) must be a subtype of \(t_2\).

There are two ways to perform coercion:

1. Single coercion: \((object :> object\_type)\)
2. Double coercion: \((object : object\_type :> object\_type)\)

For example, if \(colored\_point\) is a subtype of \(points\) then coercion can be done as following:

\[
\text{let colored\_point\_to\_points \(cp = (cp : colored\_point :> points);;\)}
\]

But, the following is not support in OCaml, as it is *narrowing coercion*.

\[
\text{(p : points :> colored\_point);;}
\]
The fully explicit coercion (Double coercion) is more precise and is sometimes unavoidable.

Single coercion \((e : t_1 \rightarrow t_2)\) may fail if:

- the type \(t_2\) is recursive, or
- the type \(t_2\) has polymorphic structure

The solution is to use fully explicit coercion \((e : t_1 : t_2)\).
Class interfaces are inferred from class definitions.

- Interfaces can be defined directly to restrict the type of a class.

```ocaml
class type restricted_point_type =
    object
        method get_x : int
        method bump : unit
    end;;
```

```ocaml
class type restricted_point_type =
    object
        method bump : unit
        method get_x : int
    end
```

```ocaml
fun (x : restricted_point_type) → x;;
```

- : restricted_point_type → restricted_point_type = <fun>

- Concrete instance variables and concrete private methods can be hidden.

- Public methods and and and virtual members cannot be hidden.
Interfaces can be defined in module signatures in order to restrict the inferred signature of a module.

\[
\text{module type POINT = sig}
\begin{align*}
\text{class restricted_point : int } \rightarrow \\
\text{object}
& \quad \text{method get\_x : int} \\
& \quad \text{method bump : unit}
\end{align*}
\text{end}
\end{align*}
\text{end};
\]

\[
\text{module type POINT =}
\begin{align*}
\text{sig}
& \quad \text{class restricted\_point :}
\end{align*}
\begin{align*}
& \quad \text{int } \rightarrow \text{object method bump : unit method get\_x : int}
\end{align*}
\text{end}
\]
Inheritance

Through inheritance, one may do the following:

- add new fields and new private methods
- add new public methods
- override fields or methods, but the type can’t be changed

```ocaml
class animal species =
  object
    method eat = Printf.printf "A %s eats.\n" species
    method speak = Printf.printf "A %s speaks.\n" species
  end;;

class pet ~species ~owner ~name =
  object
    inherit animal species
    val owner = owner
    method name : string = name
    method eat =
      Printf.printf "A %s eats.\n" name;
    super#eat
  end;;
```

Mohammad Alam and Damith A. Karunaratne
OCaml Objects and Classes
Subtyping and inheritance are not related. Inheritance is a syntactic relation between classes while subtyping is a semantic relation between types. For example, in the previous example the class ‘pet’ could have been defined directly, without inheriting from the class ‘animal’; the type of pet would remain unchanged and thus still be a subtype of animal.
Multiple Inheritance

- Inheritance could be done from multiple independent classes.
- Inheritance could be done from multiple virtual classes.
- Inheritance could be done from combination of independent and virtual classes. Virtual classes also inherits the same way.

```ocaml
class floatNumber =
  object
    inherit comparable
    inherit number
  ;;
end;;
```

- Previous definitions of a method can be reused by binding the related ancestor.
- The name super is a pseudo value identifier that can only be used to invoke a super-class method.
Repeated inheritance is allowed in OCaml, i.e., a class can inherit another, along multiple paths (diamond problem) as well as directly. Example,

```ocaml
class a =
    object
        method x = 1
    end;;
class b =
    object
        inherit a
        method x = 2
        inherit a
    end;;
(new b)#x;;
```
Example (cont..)

```ocaml
class a =
  object
    val mutable x = 0
    method set y = x ← y
    method get = x
  end;;
class b =
  object
    inherit a as super1
    inherit a as super2
    method test =
      super1#set 10;
      super2#get
  end;;
```

In the above example, the mutable field x is duplicated not included textually.
OCaml policy for **multiple inheritance**:

1. *textual inclusion*, if methods are visible, then follows overriding rule: if a method is defined more than once, the last definition is used.

2. *Duplication*, if fields and methods are hidden (private), then follows copy rule.

Problem with textual inclusion is that, it might be necessary to know the text of all repeated superclasses.
Parameterized Classes

Reference cells can be implemented as objects but parameterized classes are needed (for non-immediate objects).

```ocaml
class ref x_init =
  object
    val mutable x = x_init
    method get = x
    method set y = x ← y
  end;;
```

Error: Some type variables are unbound in this type:
```
class ref :
  'a -> object
    val mutable x :
      'a
    method get :
      'a
    method set :
      'a -> unit
  end
```
The method get has type `'a` where `'a` is unbound.
Reference cells can be implemented as objects but parameterized classes are needed (for non-immediate objects).

```ocaml
class ref x_init =
    object
        val mutable x = x_init
        method get = x
        method set y = x ← y
    end;;

Error: Some type variables are unbound in this type:
class ref :
    'a →
    object
        val mutable x : 'a
        method get : 'a
        method set : 'a → unit
    end

The method get has type 'a where 'a is unbound
```
Class type parameters are listed between \([\text{ and }]\).

```
class \([a]\) ref x_init =
    object
        val mutable x = (x_init : a)
        method get = x
        method set y = x ← y
    end;;

class \([\text{\char'39}a]\) ref :
    \text{\char'39}a → object val mutable x : \text{\char'39}a method get : \text{\char'39}a method set : \text{\char'39}a → unit end
```
The type parameters in the declaration can be constrained within the class definitions body.

```
class 'a circle (c : 'a) =
  object
    constraint 'a = #point
    val mutable center = c
    method center = center
    method set_center c = center ← c
    method move = center#move
  end;;
```
Parameterized classes can be "specialized".

```ocaml
class ['a] colored_circle c =
  object
    constraint 'a = #colored_point
    inherit ['a] circle c
    method color = center #color
  end;
```

- The type parameter must be specified when inheriting.
- Parameterized classes are polymorphic in their contents.
Parameterized classes on their own don’t accommodate polymorphic methods.

```ocaml
class ['a] intlist (l : int list) =
    object
        method empty = (l = [])
        method fold f (accu : 'a) = List.fold_left f accu l
    end;;

let l = new intlist [1; 2; 3];;

val l : '_a intlist = <obj>
```
Polymorphic Methods Cont..

l#fold (fun x y → x+y) 0;;

- : int = 6

l;;

- : int intlist = <obj>

l#fold (fun s x → s ^ string_of_int x ^ ” ”) ””;;
Polymorphic Methods Cont..

```ocaml
l#fold (fun x y → x+y) 0;;
- : int = 6

l;;

- : int intlist = <obj>

l#fold (fun s x → s ^ string_of_int x ^ "") "";;

Error: This expression has type int but an expression was expected of type string
```

- The objects are not polymorphic as the use of fold method fixes the type.
Instead of making the class polymorphic, make the method polymorphic.

```ocaml
class intlist (l : int list) =
  object
    method empty = (l = [])
    method fold : 'a. ('a → int → 'a) → 'a → 'a =
      fun f accu → List.fold_left f accu l
  end;;
```

Mohammad Alam and Damith A. Karunaratne

OCaml Objects and Classes
let l = new intlist [1; 2; 3];

l#fold (fun x y → x+y) 0;;

- : int = 6

l#fold (fun s x → s ^ string_of_int x ^ ” ”) ””;;

- : string = ”1 2 3 ”
Recursive classes are used for objects whose types are mutually recursive. Example,

```
class window =
  object
    val mutable top_widget = (None : widget option)
    method top_widget = top_widget
  end
and widget (w : window) =
  object
    val window = w
    method window = window
  end;;
```

Here, the types are recursive but the classes are independent.
Functional Objects

 Allows classes to have instance variables without assignments.

 - Made possible through the \{\langle\ldots\rangle\} construct.

```ocaml
class functional_point y =
  object
    val x = y
    method get_x = x
    method move d = \{\langle x = x + d \rangle\}
  end;;

class functional_point :
  int \rightarrow
  object ('a) val x : int method get_x : int method move : int \rightarrow
  'a end
```

Mohammad Alam and Damith A. Karunaratne
OCaml Objects and Classes
The move method is similar to a binary method.

- A binary method takes an argument of the same type as self.
- The move method takes a “copy” of the current self with some updates.
Objects can be cloned using the **Oo.copy** library.

- Instance variables are copied, but their contents are shared.
  - eg. If the instance variable is a reference cell, the value will be shared.

```ocaml
let p = new point 5;;
val p : point = <obj>

let q = Oo.copy p;;
val q : point = <obj>

p = q, p = p;;
```

Two objects are equal iff they are physically equal.
Object Cloning

Objects can be cloned using the **Oo.copy** library.

- Instance variables are copied, but their contents are shared.
  - eg. If the instance variable is a reference cell, the value will be shared.

```ocaml
let p = new point 5;;

val p : point = <obj>

let q = Oo.copy p;;

val q : point = <obj>

p = q, p = p;;

- : bool * bool = (false, true)
```

- Two objects are equal iff they are physically equal.
Cloning vs. Overriding

Cloning and overriding of objects function the same when used within objects.

```ocaml
class copy =
    object
        method copy = {< >}
    end;;

class copy =
    object (self)
        method copy = Oo.copy self
    end;;
```
In OCaml, the only way to share the representation between two different objects is to expose it to the whole world. For instance, using *binary methods*.

A solution to this problem is to use the friends concept, where, all friends (classes or functions) defined within the same module share the same abstract view (signature) but knows the concrete representation. Thus, the concrete representation can be abstracted using signature. Example,

```ocaml
module type CURRENCY = sig
  type t
  class cur : float →
    object ('a)
      method v : t
      method plus : 'a → 'a
      method prod : float → 'a
    end
  end
end;;
```
module Currency = struct
  type t = float
  class cur x =
    object (_: 'a)
      val v = x
      method v = v
      method plus(z: 'a) = {< v = v +. z#v >}
      method prod x = {< v = x *. v >}
    end
  end;;
OCaml has two very similar mechanisms for modularity and abstraction: the **module system** and the **object system**.

- Before 3.12, the main difference was, one (object) is first class values and the other (module) is not. But, in 3.12 modules are first class values:
  - *(module module_expr : package_type)* converts the module (structure or functor) denoted by module_expr to a value that encapsulates the module.
  - *(val expr : package_type)* evaluates the expression expr to a value of type package_type (unpacking into a module).

For Example,

```ocaml
module type DEVICE = sig ··· end
module SVG = struct ··· end
module PDF = struct ··· end
let devices : (string, module DEVICE) Hashtbl.t = Hashtbl.create 5
let _ = Hashtbl.add devices "SVG" (module SVG : DEVICE)
let _ = Hashtbl.add devices "PDF" (module PDF : DEVICE)
module Device = (val (Hashtbl.find devices device_name) : DEVICE)
```
Modules can **contain type definitions** and objects cannot. This enables modules to provide privacy outside of the module boundaries. This advantage is used to implement the friends concept, as specified earlier.

When dynamic scoping behavior is required, one would prefer object implementation rather than module.

```ocaml
class dog name =
  object (self)
    method name = name
    method eat = Printf.printf "\%s eats.\n" self # name
    method bark = Printf.printf "\%s barks!\n" self # name
    method bark_eat = self # bark; self # eat
  end;;
class hound n =
  object (self)
    inherit dog n
    method bark = Printf.printf "\%s howls!\n" self # name
  end;;
```

Mohammad Alam and Damith A. Karunaratne

**OCaml Objects and Classes**
The relation between object, class and type in **Objective Caml** is very different from that in main stream object-oriented languages like **Java** or **C++**.

- In Java/C++ the class name is the type of the object; whereas, in OCaml object type is **set of public methods and their types**. Class types are abbreviated, it is the class name, but in the context of type expressions it stands for the object type.

- Types and classes in Objective Caml are independent of each other, i.e., **two unrelated classes** may produce objects of the **same type**, and there is no way at the type level to ensure that an object comes from a specific class.

- This also affects the coercion. In Java/C++, if a class hierarchy is defined as: animal ← pet ← petDog, then no other coercion than this hierarchy is allowed in Java/C++ but this is not the case in OCaml.

- In **Java/C++** subtyping and subclassing are the same but not in OCaml.

- In Java/C++ object cannot be created without classes. In OCaml one can create **(immediate) objects** without going through the classes. These objects can be created inside an expression.
OCaml instance variables are private cannot be made public; in Java/C++ instance variables can be defined as both.

In OCaml a subclass can make a private superclass method, public. In Java a subclass can only inherit a package-private member.

In OCaml inheritance from multiple independent class is allowed, even the same superclass directly/indirectly. It is not the same for Java.

Class type are similar to interface in Java. Java classes can implement multiple interface but OCaml classes can implement one type only. But OCaml type can inherit other type classes.

In OCaml the class system is statically typed and in Java/C++ it is dynamically typed. That is why a pointer of a superclass can be used to refer to a subclass object dynamically, in Java/C++.

In OCaml one can initialize as instance variable directly (without initializer) as well as indicate an instance variable virtual; not in Java/C++.
References

Jason Hickey.
Introduction to objective caml.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon.
The Objective Caml System: Documentation and Users Manual.
INRIA, June 2010.

The caml language.
[Online; Accessed: November 2, 2010].