14 Classes

This chapter was written by Leo White and Jason Hickey.

Programming with objects directly is great for encapsulation, but one of the main goals of object-oriented programming is code reuse through inheritance. For inheritance, we need to introduce *classes*. In object-oriented programming, a class is essentially a recipe for creating objects. The recipe can be changed by adding new methods and fields, or it can be changed by modifying existing methods.

14.1 OCaml Classes

In OCaml, class definitions must be defined as toplevel statements in a module. The syntax for a class definition uses the keyword `class`:

```
# open Base;;
# class istack = object
    val mutable v = [0; 2]

    method pop =
      match v with
      | hd :: tl ->
        v <- tl;
        Some hd
      | [] -> None

    method push hd =
      v <- hd :: v
  end;;
class istack :
  object
    val mutable v : int list
    method pop : int option
    method push : int -> unit
  end
```

The `class istack : object ... end` result shows that we have created a class `istack` with *class type* `object ... end`. Like module types, class types are completely separate from regular OCaml types (e.g., `int`, `string`, and `list`) and, in particular, should not be confused with object types (e.g., `< get : int; .. >`). The class type describes the class itself rather than the objects that the class creates. This particular

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

class type specifies that the `istack` class defines a mutable field `v`, a method `pop` that returns an `int option`, and a method `push` with type `int -> unit`.

To produce an object, classes are instantiated with the keyword `new`:

```
# let s = new istack;;
val s : istack = <obj>
# s#pop;;
- : int option = Some 0
# s#push 5;;
- : unit = ()
# s#pop;;
- : int option = Some 5
```

You may have noticed that the object `s` has been given the type `istack`. But wait, we've stressed *classes are not types*, so what's up with that? In fact, what we've said is entirely true: classes and class names *are not* types. However, for convenience, the definition of the class `istack` also defines an object type `istack` with the same methods as the class. This type definition is equivalent to:

```
# type istack = < pop: int option; push: int -> unit >;;
type istack = < pop : int option; push : int -> unit >
```

Note that this type represents any object with these methods: objects created using the `istack` class will have this type, but objects with this type may not have been created by the `istack` class.

14.2 Class Parameters and Polymorphism

A class definition serves as the *constructor* for the class. In general, a class definition may have parameters that must be provided as arguments when the object is created with `new`.

Let's implement a variant of the `istack` class that can hold any values, not just integers. When defining the class, the type parameters are placed in square brackets before the class name in the class definition. We also add a parameter `init` for the initial contents of the stack:

```
# class ['a] stack init = object
    val mutable v : 'a list = init

    method pop =
      match v with
      | hd :: tl ->
        v <- tl;
        Some hd
      | [] -> None

    method push hd =
      v <- hd :: v
  end;;
class ['a] stack :
  'a list ->
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
object
  val mutable v : 'a list
  method pop : 'a option
  method push : 'a -> unit
end
```

Note that the type parameter `['a]` in the definition uses square brackets, but for other uses of the type they are omitted (or replaced with parentheses if there is more than one type parameter).

The type annotation on the declaration of `v` is used to constrain type inference. If we omit this annotation, the type inferred for the class will be "too polymorphic": `init` could have some type `'b list`:

```
# class ['a] stack init = object
    val mutable v = init

    method pop =
      match v with
      | hd :: tl ->
        v <- tl;
        Some hd
      | [] -> None

    method push hd =
      v <- hd :: v
  end;;
Lines 1-13, characters 1-6:
Error: Some type variables are unbound in this type:
         class ['a] stack :
           'b list ->
           object
             val mutable v : 'b list
             method pop : 'b option
             method push : 'b -> unit
           end
       The method pop has type 'b option where 'b is unbound
```

In general, we need to provide enough constraints so that the compiler will infer the correct type. We can add type constraints to the parameters, to the fields, and to the methods. It is a matter of preference how many constraints to add. You can add type constraints in all three places, but the extra text may not help clarity. A convenient middle ground is to annotate the fields and/or class parameters, and add constraints to methods only if necessary.

14.3 Object Types as Interfaces

We may wish to traverse the elements on our stack. One common style for doing this in object-oriented languages is to define a class for an `iterator` object. An iterator provides a generic mechanism to inspect and traverse the elements of a collection.

There are two common styles for defining abstract interfaces like this. In Java, an

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

iterator would normally be specified with an interface, which specifies a set of method types:

```
// Java-style iterator, specified as an interface.
interface <T> iterator {
  T Get();
  boolean HasValue();
  void Next();
};
```

In languages without interfaces, like C++, the specification would normally use *abstract* classes to specify the methods without implementing them (C++ uses the "= 0" definition to mean "not implemented"):

```
// Abstract class definition in C++.
template<typename T>
class Iterator {
 public:
  virtual ~Iterator() {}
  virtual T get() const = 0;
  virtual bool has_value() const = 0;
  virtual void next() = 0;
};
```

OCaml supports both styles. In fact, OCaml is more flexible than these approaches because an object type can be implemented by any object with the appropriate methods; it does not have to be specified by the object's class *a priori*. We'll leave abstract classes for later. Let's demonstrate the technique using object types.

First, we'll define an object type `iterator` that specifies the methods in an iterator:

```
# type 'a iterator = < get : 'a; has_value : bool; next : unit >;;
type 'a iterator = < get : 'a; has_value : bool; next : unit >
```

Next, we'll define an actual iterator for lists. We can use this to iterate over the contents of our stack:

```
# class ['a] list_iterator init = object
    val mutable current : 'a list = init

    method has_value = not (List.is_empty current)

    method get =
      match current with
      | hd :: tl -> hd
      | [] -> raise (Invalid_argument "no value")

    method next =
      match current with
      | hd :: tl -> current <- tl
      | [] -> raise (Invalid_argument "no value")
  end;;
class ['a] list_iterator :
  'a list ->
  object
    val mutable current : 'a list
    method get : 'a
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
    method has_value : bool
    method next : unit
  end
```

Finally, we add a method `iterator` to the `stack` class to produce an iterator. To do so, we construct a `list_iterator` that refers to the current contents of the stack:

```
# class ['a] stack init = object
    val mutable v : 'a list = init

    method pop =
      match v with
      | hd :: tl ->
        v <- tl;
        Some hd
      | [] -> None

    method push hd =
      v <- hd :: v

    method iterator : 'a iterator =
      new list_iterator v
  end;;
class ['a] stack :
  'a list ->
  object
    val mutable v : 'a list
    method iterator : 'a iterator
    method pop : 'a option
    method push : 'a -> unit
  end
```

Now we can build a new stack, push some values to it, and iterate over them:

```
# let s = new stack [];;
val s : '_weak1 stack = <obj>
# s#push 5;;
- : unit = ()
# s#push 4;;
- : unit = ()
# let it = s#iterator;;
val it : int iterator = <obj>
# it#get;;
- : int = 4
# it#next;;
- : unit = ()
# it#get;;
- : int = 5
# it#next;;
- : unit = ()
# it#has_value;;
- : bool = false
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

14.3.1 Functional Iterators

In practice, most OCaml programmers avoid iterator objects in favor of functional-style techniques. For example, the alternative `stack` class that follows takes a function `f` and applies it to each of the elements on the stack:

```
# class ['a] stack init = object
    val mutable v : 'a list = init

    method pop =
      match v with
      | hd :: tl ->
        v <- tl;
        Some hd
      | [] -> None

    method push hd =
      v <- hd :: v

    method iter f =
      List.iter ~f v
  end;;
class ['a] stack :
  'a list ->
  object
    val mutable v : 'a list
    method iter : ('a -> unit) -> unit
    method pop : 'a option
    method push : 'a -> unit
  end
```

What about functional operations like `map` and `fold`? In general, these methods take a function that produces a value of some other type than the elements of the set.

For example, a `fold` method for our `['a] stack` class should have type `('b -> 'a -> 'b) -> 'b -> 'b`, where the `'b` is polymorphic. To express a polymorphic method type like this, we must use a type quantifier, as shown in the following example:

```
# class ['a] stack init = object
    val mutable v : 'a list = init

    method pop =
      match v with
      | hd :: tl ->
        v <- tl;
        Some hd
      | [] -> None

    method push hd =
      v <- hd :: v

    method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =
      (fun f init -> List.fold ~f ~init v)
  end;;
class ['a] stack :
  'a list ->
  object
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
    val mutable v : 'a list
    method fold : ('b -> 'a -> 'b) -> 'b -> 'b
    method pop : 'a option
    method push : 'a -> unit
  end
```

The type quantifier `'b.` can be read as "for all `'b`." Type quantifiers can only be used *directly after* the method name, which means that method parameters must be expressed using a `fun` or `function` expression.

14.4 Inheritance

Inheritance uses an existing class to define a new one. For example, the following class definition inherits from our `stack` class for strings and adds a new method `print` that prints all the strings on the stack:

```
# class sstack init = object
    inherit [string] stack init

    method print =
      List.iter ~f:Stdio.print_endline v
  end;;
class sstack :
  string list ->
  object
    val mutable v : string list
    method pop : string option
    method print : unit
    method push : string -> unit
  end
```

A class can override methods from classes it inherits. For example, this class creates stacks of integers that double the integers before they are pushed onto the stack:

```
# class double_stack init = object
    inherit [int] stack init as super

    method push hd =
      super#push (hd * 2)
  end;;
class double_stack :
  int list ->
  object
    val mutable v : int list
    method pop : int option
    method push : int -> unit
  end
```

The preceding `as super` statement creates a special object called `super` which can be used to call superclass methods. Note that `super` is not a real object and can only be used to call methods.

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

14.5 Class Types

To allow code in a different file or module to inherit from a class, we must expose it and give it a class type. What is the class type?

As an example, let's wrap up our `stack` class in an explicit module (we'll use explicit modules for illustration, but the process is similar when we want to define a `.mli` file). In keeping with the usual style for modules, we define a type `'a t` to represent the type of our stacks:

```
module Stack = struct
  class ['a] stack init = object
    ...
  end

  type 'a t = 'a stack

  let make init = new stack init
end
```

We have multiple choices in defining the module type, depending on how much of the implementation we want to expose. At one extreme, a maximally abstract signature would completely hide the class definitions:

```
module AbstractStack : sig
   type 'a t = < pop: 'a option; push: 'a -> unit >

   val make : 'a list -> 'a t
end = Stack
```

The abstract signature is simple because we ignore the classes. But what if we want to include them in the signature so that other modules can inherit from the class definitions? For this, we need to specify types for the classes, called *class types*.

Class types do not appear in mainstream object-oriented programming languages, so you may not be familiar with them, but the concept is pretty simple. A class type specifies the type of each of the visible parts of the class, including both fields and methods. Just as with module types, you don't have to give a type for everything; anything you omit will be hidden:

```
module VisibleStack : sig

  type 'a t = < pop: 'a option; push: 'a -> unit >

  class ['a] stack : object
    val mutable v : 'a list
    method pop : 'a option
    method push : 'a -> unit
  end

  val make : 'a list -> 'a t
end = Stack
```

In this signature, we've chosen to make everything visible. The class type for `stack` specifies the types of the field `v`, as well as the types of each of the methods.

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

14.6 Open Recursion

Open recursion allows an object's methods to invoke other methods on the same object. These calls are looked up dynamically, allowing a method in one class to call a method from another class, if both classes are inherited by the same object. This allows mutually recursive parts of an object to be defined separately.

This ability to define mutually recursive methods from separate components is a key feature of classes: achieving similar functionality with data types or modules is much more cumbersome and verbose.

For example, consider writing recursive functions over a simple document format. This format is represented as a tree with three different types of node:

```
type doc =
  | Heading of string
  | Paragraph of text_item list
  | Definition of string list_item list

and text_item =
  | Raw of string
  | Bold of text_item list
  | Enumerate of int list_item list
  | Quote of doc

and 'a list_item =
  { tag: 'a;
    text: text_item list }
```

It is quite easy to write a function that operates by recursively traversing this data. However, what if you need to write many similar recursive functions? How can you factor out the common parts of these functions to avoid repetitive boilerplate?

The simplest way is to use classes and open recursion. For example, the following class defines objects that fold over the document data:

```
class ['a] folder = object(self)
  method doc acc = function
  | Heading _ -> acc
  | Paragraph text -> List.fold ~f:self#text_item ~init:acc text
  | Definition list -> List.fold ~f:self#list_item ~init:acc list

  method list_item: 'b. 'a -> 'b list_item -> 'a =
    fun acc {tag; text} ->
      List.fold ~f:self#text_item ~init:acc text

  method text_item acc = function
  | Raw _ -> acc
  | Bold text -> List.fold ~f:self#text_item ~init:acc text
  | Enumerate list -> List.fold ~f:self#list_item ~init:acc list
  | Quote doc -> self#doc acc doc
end
```

The `object (self)` syntax binds `self` to the current object, allowing the `doc`, `list_item`, and `text_item` methods to call each other.

By inheriting from this class, we can create functions that fold over the document

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

data. For example, the count_doc function counts the number of bold tags in the document that are not within a list:

```
class counter = object
  inherit [int] folder as super

  method list_item acc li = acc

  method text_item acc ti =
    let acc = super#text_item acc ti in
    match ti with
    | Bold _ -> acc + 1
    | _ -> acc
end

let count_doc = (new counter)#doc
```

Note how the super special object is used in text_item to call the [int] folder class's text_item method to fold over the children of the text_item node.

14.7 Private Methods

Methods can be declared *private*, which means that they may be called by subclasses, but they are not visible otherwise (similar to a *protected* method in C++).

For example, we may want to include methods in our folder class for handling each of the different cases in doc and text_item. However, we may not want to force subclasses of folder to expose these methods, as they probably shouldn't be called directly:

```
class ['a] folder2 = object(self)
  method doc acc = function
  | Heading str -> self#heading acc str
  | Paragraph text -> self#paragraph acc text
  | Definition list -> self#definition acc list

  method list_item: 'b. 'a -> 'b list_item -> 'a =
    fun acc {tag; text} ->
      List.fold ~f:self#text_item ~init:acc text

  method text_item acc = function
  | Raw str -> self#raw acc str
  | Bold text -> self#bold acc text
  | Enumerate list -> self#enumerate acc list
  | Quote doc -> self#quote acc doc

  method private heading acc str = acc
  method private paragraph acc text =
    List.fold ~f:self#text_item ~init:acc text
  method private definition acc list =
    List.fold ~f:self#list_item ~init:acc list

  method private raw acc str = acc
  method private bold acc text =
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
    List.fold ~f:self#text_item ~init:acc text
  method private enumerate acc list =
    List.fold ~f:self#list_item ~init:acc list
  method private quote acc doc = self#doc acc doc
end

let f :
  < doc : int -> doc -> int;
    list_item : 'a . int -> 'a list_item -> int;
    text_item : int -> text_item -> int >  = new folder2
```

The final statement that builds the value `f` shows how the instantiation of a `folder2` object has a type that hides the private methods.

To be precise, the private methods are part of the class type, but not part of the object type. This means, for example, that the object `f` has no method `bold`. However, the private methods are available to subclasses: we can use them to simplify our `counter` class:

```
class counter_with_private_method = object
  inherit [int] folder2 as super

  method list_item acc li = acc

  method private bold acc txt =
    let acc = super#bold acc txt in
    acc + 1
end
```

The key property of private methods is that they are visible to subclasses, but not anywhere else. If you want the stronger guarantee that a method is *really* private, not even accessible in subclasses, you can use an explicit class type that omits the method. In the following code, the private methods are explicitly omitted from the class type of `counter_with_sig` and can't be invoked in subclasses of `counter_with_sig`:

```
class counter_with_sig : object
  method doc : int -> doc -> int
  method list_item : int -> 'b list_item -> int
  method text_item : int -> text_item -> int
end = object
  inherit [int] folder2 as super

  method list_item acc li = acc

  method private bold acc txt =
    let acc = super#bold acc txt in
    acc + 1
end
```

14.8 Binary Methods

A *binary method* is a method that takes an object of `self` type. One common example is defining a method for equality:

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
# class square w = object(self : 'self)
    method width = w
    method area = Float.of_int (self#width * self#width)
    method equals (other : 'self) = other#width = self#width
  end;;
class square :
  int ->
  object ('a)
    method area : float
    method equals : 'a -> bool
    method width : int
  end
# class circle r = object(self : 'self)
    method radius = r
    method area = 3.14 *. (Float.of_int self#radius) **. 2.0
    method equals (other : 'self) = other#radius = self#radius
  end;;
class circle :
  int ->
  object ('a)
    method area : float
    method equals : 'a -> bool
    method radius : int
  end
```

Note how we can use the type annotation `(self: 'self)` to obtain the type of the current object.

We can now test different object instances for equality by using the `equals` binary method:

```
# (new square 5)#equals (new square 5);;
- : bool = true
# (new circle 10)#equals (new circle 7);;
- : bool = false
```

This works, but there is a problem lurking here. The method `equals` takes an object of the exact type `square` or `circle`. Because of this, we can't define a common base class `shape` that also includes an equality method:

```
# type shape = < equals : shape -> bool; area : float >;;
type shape = < area : float; equals : shape -> bool >
# (new square 5 :> shape);;
Line 1, characters 1-24:
Error: Type square = < area : float; equals : square -> bool; width :
     int >
        is not a subtype of shape = < area : float; equals : shape ->
     bool >
        Type shape = < area : float; equals : shape -> bool >
        is not a subtype of
          square = < area : float; equals : square -> bool; width :
     int >
        The first object type has no method width
```

The problem is that a `square` expects to be compared with a `square`, not an arbitrary shape; likewise for `circle`. This problem is fundamental. Many languages solve it

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

either with narrowing (with dynamic type checking), or by method overloading. Since OCaml has neither of these, what can we do?

Since the problematic method is equality, one proposal we could consider is to just drop it from the base type `shape` and use polymorphic equality instead. However, the built-in polymorphic equality has very poor behavior when applied to objects:

```
# Poly.(=)
    (object method area = 5 end)
    (object method area = 5 end);;
- : bool = false
```

The problem here is that two objects are considered equal by the built-in polymorphic equality if and only if they are physically equal. There are other reasons not to use the built-in polymorphic equality, but these false negatives are a showstopper.

If we want to define equality for shapes in general, the remaining solution is to use the same approach as we described for narrowing. That is, introduce a *representation* type implemented using variants, and implement the comparison based on the representation type:

```
# type shape_repr =
    | Square of int
    | Circle of int;;
type shape_repr = Square of int | Circle of int
# type shape =
  < repr : shape_repr; equals : shape -> bool; area : float >;;
type shape = < area : float; equals : shape -> bool; repr :
    shape_repr >
# class square w = object(self)
    method width = w
    method area = Float.of_int (self#width * self#width)
    method repr = Square self#width
    method equals (other : shape) =
       match (self#repr, other#repr) with
       | Square x, Square x' -> Int.(=) x x'
       | _ -> false
  end;;
class square :
  int ->
  object
    method area : float
    method equals : shape -> bool
    method repr : shape_repr
    method width : int
  end
```

The binary method `equals` is now implemented in terms of the concrete type `shape_repr`. When using this pattern, you will not be able to hide the `repr` method, but you can hide the type definition using the module system:

```
module Shapes : sig
  type shape_repr
  type shape =
    < repr : shape_repr; equals : shape -> bool; area: float >

  class square : int ->
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
    object
      method width : int
      method area : float
      method repr : shape_repr
      method equals : shape -> bool
    end
end = struct
  type shape_repr =
  | Square of int
  | Circle of int
  ...
end
```

Note that this solution prevents us from adding new kinds of shapes without adding new constructors to the `shape_repr` type, which is quite restrictive. We can fix this, however, by using OCaml's rarely-used but still useful *extensible variants*.

Extensible variants let you separate the definition of a variant type from the definition of its constructors. The resulting type is by definition open, in the sense that new variants can always be added. As a result, the compiler can't check whether pattern matching on such a variant is exhaustive. Happily, exhaustivity is not what we need here.

Here's how we'd rewrite the above example with extensible variants.

```
# type shape_repr = ..;;
type shape_repr = ..
# type shape =
  < repr : shape_repr; equals : shape -> bool; area : float >;;
type shape = < area : float; equals : shape -> bool; repr :
    shape_repr >
# type shape_repr += Square of int;;
type shape_repr += Square of int
# class square w = object(self)
    method width = w
    method area = Float.of_int (self#width * self#width)
    method repr = Square self#width
    method equals (other : shape) =
       match (self#repr, other#repr) with
       | Square x, Square x' -> Int.(=) x x'
       | _ -> false
  end;;
class square :
  int ->
  object
    method area : float
    method equals : shape -> bool
    method repr : shape_repr
    method width : int
  end
```

One oddity of the representation type approach is that the objects created by these classes are in one-to-one correspondence with members of the representation type, making the objects seem somewhat redundant.

But equality is an extreme instance of a binary method: it needs access to all the information of the other object. Many other binary methods need only partial

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

information about the object. For instance, consider a method that compares shapes by their sizes:

```
# class square w = object(self)
    method width = w
    method area = Float.of_int (self#width * self#width)
    method larger (other : shape) = Float.(self#area > other#area)
  end;;
class square :
  int ->
  object
    method area : float
    method larger : shape -> bool
    method width : int
  end
```

The `larger` method can be used on a `square`, but it can also be applied to any object of type `shape`.

14.9 Virtual Classes and Methods

A *virtual* class is a class where some methods or fields are declared but not implemented. This should not be confused with the word `virtual` as it is used in C++. A `virtual` method in C++ uses dynamic dispatch, while regular, nonvirtual methods are statically dispatched. In OCaml, *all* methods use dynamic dispatch, but the keyword `virtual` means that the method or field is not implemented. A class containing virtual methods must also be flagged `virtual` and cannot be directly instantiated (i.e., no object of this class can be created).

To explore this, let's extend our shapes examples to simple, interactive graphics. We will use the Async concurrency library and the Async_graphics[1] library, which provides an asynchronous interface to OCaml's built-in Graphics library. Concurrent programming with Async will be explored later in Chapter 17 (Concurrent Programming with Async); for now you can safely ignore the details. You just need to run `opam install async_graphics` to get the library installed on your system.

We will give each shape a `draw` method that describes how to draw the shape on the `Async_graphics` display:

```
open Core
open Async
open Async_graphics

type drawable = < draw: unit >
```

14.9.1 Create Some Simple Shapes

Now let's add classes for making squares and circles. We include an `on_click` method for adding event handlers to the shapes:

[1] http://github.com/lpw25/async_graphics/

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
class square w x y = object(self)
  val mutable x: int = x
  method x = x

  val mutable y: int = y
  method y = y

  val mutable width = w
  method width = width

  method draw = fill_rect x y width width

  method private contains x' y' =
    x <= x' && x' <= x + width &&
      y <= y' && y' <= y + width

  method on_click ?start ?stop f =
    on_click ?start ?stop
      (fun ev ->
         if self#contains ev.mouse_x ev.mouse_y then
           f ev.mouse_x ev.mouse_y)
end
```

The square class is pretty straightforward, and the `circle` class below also looks very similar:

```
class circle r x y = object(self)
  val mutable x: int = x
  method x = x

  val mutable y: int = y
  method y = y

  val mutable radius = r
  method radius = radius

  method draw = fill_circle x y radius

  method private contains x' y' =
    let dx = x' - x in
    let dy = y' - y in
      dx * dx + dy * dy <= radius * radius

  method on_click ?start ?stop f =
    on_click ?start ?stop
      (fun ev ->
         if self#contains ev.mouse_x ev.mouse_y then
           f ev.mouse_x ev.mouse_y)
end
```

These classes have a lot in common, and it would be useful to factor out this common functionality into a superclass. We can easily move the definitions of `x` and `y` into a superclass, but what about `on_click`? Its definition depends on `contains`, which has a different definition in each class. The solution is to create a *virtual* class. This class will declare a `contains` method but leave its definition to the subclasses.

Here is the more succinct definition, starting with a virtual shape class that implements on_click and on_mousedown:

```
class virtual shape x y = object(self)
  method virtual private contains: int -> int -> bool

  val mutable x: int = x
  method x = x

  val mutable y: int = y
  method y = y

  method on_click ?start ?stop f =
    on_click ?start ?stop
      (fun ev ->
         if self#contains ev.mouse_x ev.mouse_y then
           f ev.mouse_x ev.mouse_y)

  method on_mousedown ?start ?stop f =
    on_mousedown ?start ?stop
      (fun ev ->
         if self#contains ev.mouse_x ev.mouse_y then
           f ev.mouse_x ev.mouse_y)
end
```

Now we can define square and circle by inheriting from shape:

```
class square w x y = object
  inherit shape x y

  val mutable width = w
  method width = width

  method draw = fill_rect x y width width

  method private contains x' y' =
    x <= x' && x' <= x + width &&
    y <= y' && y' <= y + width
end

class circle r x y = object
  inherit shape x y

  val mutable radius = r
  method radius = radius

  method draw = fill_circle x y radius

  method private contains x' y' =
    let dx = x' - x in
    let dy = y' - y in
      dx * dx + dy * dy <= radius * radius
end
```

One way to view a virtual class is that it is like a functor, where the “inputs” are the declared—but not defined—virtual methods and fields. The functor application is implemented through inheritance, when virtual methods are given concrete implementations.

14.10 Initializers

You can execute expressions during the instantiation of a class by placing them before the object expression or in the initial value of a field:

```
# class obj x =
    let () = Stdio.printf "Creating obj %d\n" x in
    object
      val field = Stdio.printf "Initializing field\n"; x
  end;;
class obj : int -> object val field : int end
# let o = new obj 3;;
Creating obj 3
Initializing field
val o : obj = <obj>
```

However, these expressions are executed before the object has been created and cannot refer to the methods of the object. If you need to use an object's methods during instantiation, you can use an initializer. An initializer is an expression that will be executed during instantiation but after the object has been created.

For example, suppose we wanted to extend our previous shapes module with a `growing_circle` class for circles that expand when clicked. We could inherit from `circle` and use the inherited `on_click` to add a handler for click events:

```
class growing_circle r x y = object(self)
  inherit circle r x y

  initializer
    self#on_click (fun _x _y -> radius <- radius * 2)
end
```

14.11 Multiple Inheritance

When a class inherits from more than one superclass, it is using *multiple inheritance*. Multiple inheritance extends the variety of ways that classes can be combined, and it can be quite useful, particularly with virtual classes. However, it can be tricky to use, particularly when the inheritance hierarchy is a graph rather than a tree, so it should be used with care.

14.11.1 How Names Are Resolved

The main trickiness of multiple inheritance is due to naming—what happens when a method or field with some name is defined in more than one class?

If there is one thing to remember about inheritance in OCaml, it is this: inheritance is

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

like textual inclusion. If there is more than one definition for a name, the last definition wins.

For example, consider this class, which inherits from square and defines a new draw method that uses draw_rect instead of fill_rect to draw the square:

```
class square_outline w x y = object
  inherit square w x y
  method draw = draw_rect x y width width
end
```

Since the inherit declaration comes before the method definition, the new draw method overrides the old one, and the square is drawn using draw_rect. But, what if we had defined square_outline as follows?

```
class square_outline w x y = object
  method draw = draw_rect x y w w
  inherit square w x y
end
```

Here the inherit declaration comes after the method definition, so the draw method from square will override the other definition, and the square will be drawn using fill_rect.

To reiterate, to understand what inheritance means, replace each inherit directive with its definition, and take the last definition of each method or field. Note that the methods and fields added by an inheritance are those listed in its class type, so private methods that are hidden by the type will not be included.

14.11.2 Mixins

When should you use multiple inheritance? If you ask multiple people, you're likely to get multiple (perhaps heated) answers. Some will argue that multiple inheritance is overly complicated; others will argue that inheritance is problematic in general, and one should use object composition instead. But regardless of who you talk to, you will rarely hear that multiple inheritance is great and that you should use it widely.

In any case, if you're programming with objects, there's one general pattern for multiple inheritance that is both useful and reasonably simple: the *mixin* pattern. Generically, a *mixin* is just a virtual class that implements a feature based on another one. If you have a class that implements methods A, and you have a mixin M that provides methods B from A, then you can inherit from M—"mixing" it in—to get features B.

That's too abstract, so let's give some examples based on our interactive shapes. We may wish to allow a shape to be dragged by the mouse. We can define this functionality for any object that has mutable x and y fields and an on_mousedown method for adding event handlers:

```
class virtual draggable = object(self)
  method virtual on_mousedown:
    ?start:unit Deferred.t ->
    ?stop:unit Deferred.t ->
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
    (int -> int -> unit) -> unit
  val virtual mutable x: int
  val virtual mutable y: int

  val mutable dragging = false
  method dragging = dragging

  initializer
    self#on_mousedown
      (fun mouse_x mouse_y ->
         let offset_x = x - mouse_x in
         let offset_y = y - mouse_y in
         let mouse_up = Ivar.create () in
         let stop = Ivar.read mouse_up in
         dragging <- true;
         on_mouseup ~stop
           (fun _ ->
              Ivar.fill mouse_up ();
              dragging <- false);
         on_mousemove ~stop
           (fun ev ->
              x <- ev.mouse_x + offset_x;
              y <- ev.mouse_y + offset_y))
end
```

This allows us to create draggable shapes using multiple inheritance:

```
class small_square = object
  inherit square 20 40 40
  inherit draggable
end
```

We can also use mixins to create animated shapes. Each animated shape has a list of update functions to be called during animation. We create an `animated` mixin to provide this update list and ensure that the functions in it are called at regular intervals when the shape is animated:

```
class virtual animated span = object(self)
  method virtual on_click:
    ?start:unit Deferred.t ->
    ?stop:unit Deferred.t ->
    (int -> int -> unit) -> unit
  val mutable updates: (int -> unit) list = []
  val mutable step = 0
  val mutable running = false

  method running = running

  method animate =
    step <- 0;
    running <- true;
    let stop =
      Clock.after span
      >>| fun () -> running <- false
    in
    Clock.every ~stop (Time.Span.of_sec (1.0 /. 24.0))
      (fun () ->
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
          step <- step + 1;
          List.iter ~f:(fun f -> f step) updates
       )

  initializer
    self#on_click (fun _x _y -> if not self#running then self#animate)
end
```

We use initializers to add functions to this update list. For example, this class will produce circles that move to the right for a second when clicked:

```
class my_circle = object
  inherit circle 20 50 50
  inherit animated Time.Span.second
  initializer updates <- [fun _ -> x <- x + 5]
end
```

These initializers can also be added using mixins:

```
class virtual linear x' y' = object
  val virtual mutable updates: (int -> unit) list
  val virtual mutable x: int
  val virtual mutable y: int

  initializer
    let update _ =
      x <- x + x';
      y <- y + y'
    in
    updates <- update :: updates
end

let pi = (Float.atan 1.0) *. 4.0

class virtual harmonic offset x' y' = object
  val virtual mutable updates: (int -> unit) list
  val virtual mutable x: int
  val virtual mutable y: int

  initializer
    let update step =
      let m = Float.sin (offset +. ((Float.of_int step) *. (pi /.
    64.))) in
      let x' = Float.to_int (m *. Float.of_int x') in
      let y' = Float.to_int (m *. Float.of_int y') in
      x <- x + x';
      y <- y + y'
    in
    updates <- update :: updates
end
```

Since the `linear` and `harmonic` mixins are only used for their side effects, they can be inherited multiple times within the same object to produce a variety of different animations:

```
class my_square x y = object
  inherit square 40 x y
```

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

```
  inherit draggable
  inherit animated (Time.Span.of_int_sec 5)
  inherit linear 5 0
  inherit harmonic 0.0 7 ~-10
end

let my_circle = object
  inherit circle 30 250 250
  inherit animated (Time.Span.minute)
  inherit harmonic 0.0 10 0
  inherit harmonic (pi /. 2.0) 0 10
end
```

14.11.3 Displaying the Animated Shapes

We finish our shapes module by creating a `main` function to draw some shapes on the graphical display and running that function using the Async scheduler:

```
let main () =
  let shapes = [
     (my_circle :> drawable);
     (new my_square 50 350 :> drawable);
     (new my_square 50 200 :> drawable);
     (new growing_circle 20 70 70 :> drawable);
  ] in
  let repaint () =
    clear_graph ();
    List.iter ~f:(fun s -> s#draw) shapes;
    synchronize ()
  in
    open_graph "";
    auto_synchronize false;
    Clock.every (Time.Span.of_sec (1.0 /. 24.0)) repaint

let () = never_returns (Scheduler.go_main ~main ())
```

Our `main` function creates a list of shapes to be displayed and defines a `repaint` function that actually draws them on the display. We then open a graphical display and ask Async to run `repaint` at regular intervals.

Finally, build the binary by linking against the `async_graphics` package, which will pull in all the other dependencies:

```
(executable
  (name      shapes)
  (modules   shapes)
  (libraries async_graphics))
```

```
$ dune build shapes.exe
```

When you run the binary, a new graphical window should appear (on macOS, you will need to install the X11 package first, which you will be prompted for). Try clicking on the various widgets, and gasp in awe at the sophisticated animations that unfold as a result.

https://doi.org/10.1017/9781009129220.016 Published online by Cambridge University Press

The graphics library described here is the one built into OCaml and is more useful as a learning tool than anything else. There are several third-party libraries that provide more sophisticated bindings to various graphics subsystems:

Lablgtk[2] A strongly typed interface to the GTK widget library.

LablGL[3] An interface between OCaml and OpenGL, a widely supported standard for 3D rendering.

js_of_ocaml[4] Compiles OCaml code to JavaScript and has bindings to WebGL. This is the emerging standard for 3D rendering in web browsers.