1 Interfaces

Many programmers find Ruby’s dynamic typing and duck typing to be very convenient features. One downside of their combination is that programs may produce run time errors when a method is invoked on an object but that object does not support that method. How might we prevent those kinds of errors?

In a statically typed language, like Java, one way to prevent such run-time errors would be to identify which methods (and what arguments) we want an object to support. Then the compiler would never permit code to invoke a method that an object isn’t guaranteed to support. For example, for a set data type, we’d probably want to have a method add that adds an element to a set, as well as a method contains that checks whether an element is a member of a set. Both methods would take a single argument—the element in question. (A set might also include other operations, such as union, intersection, difference, subset, singleton, and various iterators like fold.)

In fact, Java provides a way of doing this—with interfaces. An interface is a list of methods, with each method’s argument types and return type. Here’s a set interface:

```java
interface MySet<E> {
    void add(E e);
    boolean contains(E e);
}
```

An interface is a type, and if class C implements interface I, then any code expecting an I will happily accept an instance of C. Here’s an implementation of MySet in terms of a list:

```java
class MyListSet<E> implements MySet<E> {
    private LinkedList<E> l;
    MyListSet() {
        l = new LinkedList<E>();
    }
    public void add(E e) {
        l.add(e);
    }
    public boolean contains(E e) {
        return l.contains(e);
    }
}
```

If a piece of code declares object o to have type I, then the code will be allowed to use only the methods declared in I. Since a class type-checks only if it actually provides (directly or via inheritance) all the methods of all the interfaces it claims to implement, no “undefined method” errors will ever occur. So interfaces are essentially a statically-typed analogue to duck typing. (However, interfaces must be named, and a class has an interface type only if the class definition explicitly says it implements the interface. These don’t have analogues in duck typing.) Recall that when a Java class implements an interface, it does not inherit any code; interfaces are only about static type checking.

In a dynamically-typed language, there is little reason to have interfaces. We can already pass any object to any method and call any method on any object. It is up to the programmer to keep track—either “in his or her head”, or better, in documentation—what objects can respond to what messages. The essence of dynamic typing is not writing down this stuff. So Ruby does not need interfaces.
2 Signatures

SML has a feature similar to interfaces, called signatures. Here's an SML signature for sets:

signature MY_SET = sig
  type 'a set
  val empty: 'a set
  val add: ('a set * 'a) -> 'a set
  val contains: ('a set * 'a) -> bool
end

The name of the signature is MY_SET. By convention, SML signatures are named with all caps. Keyword sig just indicates the beginning of the signature definition. This signature contains two functions, add and contains. Their types differ a little from the types in the Java interface, because each needs to take in a set as an argument. (Objects implicitly have this, the receiver, instead of that argument.) Because that argument has to be given a type, the signature has to declare a type for sets. Here, that type is 'a set. The signature also needs a constructor for empty sets; that wasn’t necessary in Java.

Signatures are implemented by structures. Here’s an SML structure that implements MY_SET in terms of (linked) lists:

structure MyListSet :> MY_SET = struct
  type 'a set = 'a list
  val empty = []
  fun add(s,e) = e::s
  fun contains(s,e) = List.exists (fn x => x=e) s
end

The name of the structure is MyListSet. By convention, SML structures are named with camel case. Keyword struct just indicates the beginning of the structure definition. Syntax :> MY_SET indicates that ListSet implements MY_SET, which the type checker will verify.

Signatures are like interfaces in that (i) they identify the names of functions and the types of their arguments, (ii) they do not contains any actual implementations, and (iii) they help to guarantee that no “function undefined” errors occurs at runtime.

We’ve been using structures and signatures all along. List is a structure with signature LIST, for example.

3 Multiple inheritance

Java classes may implement multiple interfaces. For example, java.util.LinkedList<E> implements Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, and Queue<E>. However, classes may not subclass multiple classes, thus inheriting code from separate places in the class hierarchy. Multiple inheritance would actually be useful. Here are two classic examples (also available in Ruby in the accompanying code file):

• Consider a Pt class with subclasses Pt3D (adding a z-dimension) and ColorPt (adding a color field). To create a ColorPt3D class, it would seem natural to have two immediate superclasses, Pt3D and ColorPt.

• Consider a Person class with subclasses Artist and Cowboy. To create an ArtistCowboy (someone who is both), it would seem natural again to have two immediate superclasses.
With single inheritance, the class hierarchy forms a tree, where if A extends B then A is a child of B in the tree. With multiple inheritance, the class hierarchy need not be a tree. It can have diamonds—four classes where one is a (not necessarily immediate) subclass of two others that have a common (not necessarily immediate) superclass. By immediate we mean directly extends (child-parent relationship); whereas we could say transitive for the more general ancestor-descendant relationship.

So what should the semantics of multiple inheritance be? Naively we might say that a class has all the fields and methods of all its superclasses. However, with multiple superclasses, there can be conflicts for the fields and methods inherited from the different classes. For example, both Artist and Cowboy could have a method draw that have very different behaviors (producing a painting versus producing a gun). What should happen when draw is invoked on an instance of ArtistCowboy?

The draw method for ArtistCowboy is an example where we would like somehow to have both methods in the subclass, or potentially to override one or both of them. At the very least we need expressions using super to indicate which superclass is intended. But this is not necessarily the only conflict. Suppose the Person class has a pocket field that Artists and Cowboys use for different things. Then perhaps an ArtistCowboy should have two pockets, even though the creation of the notion of pocket was in the common ancestor Person.

But if you look at our ColorPt3D example, you would reach the opposite conclusion. Here both Pt3D and ColorPt inherit the x and y from a common ancestor, but we certainly don’t want a ColorPt3D to have two x methods or two @x instance variables.

These issues are some of the reasons language with multiple inheritance (e.g., C++, Perl, and Python) need fairly complicated rules for how subclassing, method lookup, and field access work. For example, C++ has two different ways of deriving a subclass from a superclass. One way creates copies of all fields from all superclasses; the other way makes only one copy of fields that were initially declared by the same common ancestor. That solution would not work well in Ruby, because it doesn’t have explicit field declarations. There are also alternatives for how to deal with method conflicts. The language could pick a fixed order (like the leftmost superclass shadows the others) or perhaps require the subclass to override all method names that conflict (or at least manually specify which draw method should dominate, for example). Perl and Python use an algorithm called C3 linearization to resolve method calls.

Because interfaces do not actually define methods—they only name them and give them types—none of the problems discussed above about multiple inheritance arise in Java. If two interfaces have a method-name conflict, it doesn’t matter; a class can still implement them both. If two interfaces disagree on a method’s type, then no class can possibly implement them both, and the type-checker will catch that.

4 Abstract methods

An abstract method is a method that must exist in all instances of a class but is not defined by the class itself. Hence any subclass that is actually used to create objects must override this method or inherit from a class that does. Consider a class that has only abstract methods: it’s essentially an interface. Conflicts aren’t a problem since the method definition does not actually exist—a non-abstract method can trivially “win” over an abstract method. So languages with abstract methods and multiple inheritance (e.g., C++) do not need interfaces. On the other hand, Java has abstract classes but only single inheritance. So Java needs interfaces.

5 Mixins

Ruby has mixins, which is a language feature somewhere between multiple inheritance and interfaces. Mixins provide method implementations to classes that include them. But mixins are not classes themselves, so they do not have constructors or a separate notion of fields, nor can they be directly instantiated. Mixins are

http://haahr.tempdomainname.com/dylan/linearization-oopsla96.html
a way of getting code reuse that is more powerful than interfaces, but avoiding the complications of multiple inheritance. Ruby did not invent mixins. Its standard library uses them heavily, though.

To define a Ruby mixin, we use the keyword `module` instead of `class`. (Modules do more than just serve as mixins, hence the seeming discrepancy in the keyword.) For example, (and actually not an especially stylish example,) here is a mixin for adding color methods to a class:

```ruby
module Color
  attr_accessor :color
  def darken
    self.color = "dark " + self.color
  end
end
```

This mixin defines three methods: `color`, `color=`, and `darken`. A class definition can include these methods by using the `include` keyword and the name of the mixin. For example:

```ruby
class ColorPt < Pt
  include Color
end
```

This defines a subclass of `Pt` that also has the three methods defined by `Color`. Such classes can have other methods defined/overridden too; here we just chose not to add anything additional. This is not necessarily good style for a couple reasons. First, our `initialize` (inherited from `Pt`) does not create the `@color` field, so we're relying on clients to call `color=` before they call `color` or they will get `nil` back. So overriding `initialize` is probably a good idea. Second, mixins that use instance variables are a bit questionable stylistically. As you might expect in Ruby, the instance variables they use will be part of the object the mixin is included in. So if there is a name conflict with some intended-to-be separate instance variable defined by the class, the two separate pieces of code will mutate the same data.

**Mixin semantics.** Now that we have mixins, we also have to revisit the method invocation semantics. Here is what Ruby does: If `obj` is an instance of class `C` and message `m` is sent to `obj`,

- First look in the class `C` for a definition of `obj`.
- Next look in mixins included in `C`. Later includes shadow earlier ones.
- Next look in `C`'s superclass.
- Next look in `C`'s superclass' mixins.
- Next look in `C`'s super-superclass.
- Etc.

**Mixins that use base class methods.** Many of the elegant uses of mixins do the following strange-sounding thing: They define methods that call other methods on `self` that are not defined by the mixin. Instead the mixin assumes that all classes that include the mixin define this method. For example, consider this mixin that lets us “double” instances of any class that has `+` defined:

```ruby
module Doubler
  def double
  end
end
```

They seem to have been introduced first in a LISP extension called Flavors.
If we include `Doubler` in some class `C` and call `double` on an instance of the class, we will call the + method on the instance, getting an error if it is not defined. But if + is defined, everything works out. So now we can easily get the convenience of doubling just by defining + and including the `Doubler` mixin. For example:

```ruby
class AnotherPt
  attr_accessor :x, :y
  include Doubler
  def + other # add two points
    ans = AnotherPt.new
    ans.x = self.x + other.x
    ans.y = self.y + other.y
    ans
  end
end
```

Now instances of `AnotherPt` have `double` methods that do what we want. We could even add `double` to classes that already exists:

```ruby
class String
  include Doubler
end
```

**Example mixin: Comparable.** Standard library mixin `Comparable` provides methods `==`, `!=`, `>`, `>=`, `<`, and `<=`, all of which assume the class defines `<>` (the *spaceship* operator). What `<>` needs to do is return a negative number if its left argument is less than its right, 0 if they are equal, and a positive number if the left argument is greater than the right. So now a class doesn’t have to define all these comparisons — it just defines `<>` and includes `Comparable`. Consider this example for comparing names:

```ruby
class Name
  attr_accessor :first, :middle, :last
  include Comparable
  def initialize(first, last, middle="")
    @first = first
    @last = last
    @middle = middle
  end
  def <=> other
    l = @last <=> other.last # <=> defined on strings
    return l if l != 0
    f = @first <=> other.first
    return f if f != 0
    @middle <=> other.middle
  end
end
```

Defining methods in `Comparable` is easy, but we certainly wouldn’t want to repeat the work for every class that wants comparisons. For example, the > method is just:
Example mixin: Enumerable. The Enumerable mixin is where many of the useful block-taking methods that iterate over some data structure are defined. Examples are any?, map, and inject. They are all written assuming the class has the method each defined. A class can define each, include the Enumerable mixin, and automatically get all these convenient methods. For example, the Array class just defines each includes Enumerable. Here is another example for a range class we might define:

```ruby
class MyRange
  include Enumerable
  def initialize(low, high)
    @low = low
    @high = high
  end
  def each
    i = @low
    while i <= @high
      yield i
      i = i + 1
    end
  end
end
```

Now we can write code like `MyRange.new(4,8).inject { |x, y| x+y }`.

One curiosity in Enumerable is that the map method always returns an instance of Array. After all, it doesn’t “know how” to produce an instance of any class, but it does know how to produce an array containing one element for everything produced by each. We could define it in the Enumerable mixin like this:

```ruby
def map
  arr = []
  each { |x| arr.push x }
  arr
end
```

Mixins vs. multiple inheritance. Mixins are not as powerful as multiple inheritance because we have to decide upfront what to make a class and what to make a mixin. Given Artist and Cowboy classes, we still have no natural way to make an ArtistCowboy. And it’s unclear which of Artist or Cowboy or both we might want to define in terms of a mixin.

---

3 We wouldn’t actually define this because Ruby already has very powerful range classes.