1 Encoding objects as records

An object is essentially a record, as we now understand records from our made-up language MF. An object’s fields are just like a record’s fields, in that they’re mutable. Here’s the key new idea, though: an object’s methods are like first-class functions that have been stored inside a field of a record. For example, a record pt for a 2D point object could be:

```haskell
val pt = {x = 1.0,
          y = 2.0,
          distToOrigin = fn self =>
                    Math.sqrt(self.x*self.x+self.y*self.y)}
```

Note how distToOrigin takes self as an argument; callers of the method must of course take care to pass in the appropriate object:

```haskell
val d = pt.distToOrigin(pt)
```

One way to ensure that is to make self an implicit argument that the compiler always inserts on behalf of the programmer. And that’s what OOP languages normally do. With that convention, we can rewrite our example as:

```haskell
val pt = {x = 1.0,
          y = 2.0,
          distToOrigin = fn _ =>
                    Math.sqrt(self.x*self.x+self.y*self.y)}
val d = pt.distToOrigin()
```

In Java, the “fields” for object methods are immutable: if an object’s method m is implemented with some code, then there is no way to mutate m to refer to different code. An instance of a subclass could have different code for m, but that instance would be a different object. (In Ruby, the “fields” for object methods are mutable: we’ve seen how to replace methods dynamically. But since Ruby doesn’t have types, it doesn’t have subtyping. So we’ll mostly ignore Ruby in this lecture.)

2 Subtyping for objects

It turns out that record and function subtyping already provides the key ideas needed for statically-typed object-oriented languages to prevent missing-field and missing-method errors. Recall that width subtyping is sound for records, that depth subtyping is sound for immutable values but unsound for mutable values, and that functions may be covariant in their return types and contravariant in their argument types. So:

- A subtype of an object type can have extra fields. [width subtyping]
- Mutable fields of an object type must have the same type in any subtypes of that object type. [depth subtyping on mutable fields]
- A subtype can have extra methods. [width subtyping]
A method of a subtype can have different argument and return types than in the supertype. The argument types can be supertypes, and the return types can be subtypes. [depth subtyping on immutable fields]

For example:

type point = {x:real, y:real, move_right: real->point}
val pt:point =
{x = 1.0,
y = 2.0,
move_right = fn dx =>
    (self.x = self.x + dx; self)}

type color_point = {x:real, y:real, c:string, move_right:real->color_point}
val cpt:color_point =
{x = 1.0,
y = 2.0,
c = "green"
move_right = fn dx =>
    (self.x = self.x + dx; self)}

According to our subtyping rules, we have that point <: color_point. Note that move_right in point returns a Point, but that move_right in color_point returns a color_point.

3 Subtyping in Java

Java conflates types with classes. That is, declaring a class also declares a type with the same name. And declaring a subclass also declares a subtype. Because of this conflation, object types in Java don’t look like record types, even though objects fundamentally are records.

So if a program declares a class Foo, then type Foo includes in it all field types and methods types implied by the class definition, including superclasses and superinterfaces (which contribute methods but not fields). And if C extends D or if C implements D, either immediately or transitively, then C <: D. This approach soundly prevents “field missing” and “method missing” errors. In particular:

- A subclass can add fields but not remove them. [width subtyping]
- A subclass can add methods but not remove them. [width subtyping]
- A subclass can override a method with a covariant return type [function subtyping]
- All the above are also true for classes that implement interfaces.
- However, a subclass cannot override a method with a contravariant argument type. Java is simply overly conservative in this choice. [function subtyping]

Subtyping only holds in Java when it is explicitly stated by extends and implements. For example, in the following code, Point is not a subtype of Rank.

class Rank { int x; }
class Point { int x; int y; }

But in MF, {x:int, y:int} <: {x:int} by width subtyping. So Java’s choice to conflate types with classes does prevent some subtyping that would otherwise be sound. Whether we want such subtyping is debatable:
it could enable code reuse, but **Rank** and **Point** represent different real-world concepts. So it might be good for the type-checker give an error if a **Point** is used where a **Rank** is expected.

Some vocabulary: Subtyping based on the names of types and their declared relationships, as in Java, is called *nominal subtyping*. Subtyping based on the structure of (anonymous) types, as in MF, is called *structural subtyping*.

**Field shadowing.** From our discussion of depth subtyping in the previous lecture, we know that a subtype cannot soundly have a field that has a different type than the type declared for that field in the supertype. This code, for example, manages to put a rhinoceros inside of a zebra-only zoo:

```java
class Animal
class Zebra extends Animal { void showStripes() {...} }
class Rhino extends Animal { }
class Zoo { Animal a; }
class ZebraZoo extends Zoo { Zebra a; }
class RhinoZoo extends Zoo { Rhino a; }

void installRhino(Zoo z) { z.a = new Rhino(); }

z = new ZebraZoo();
z.a = new Zebra();
installRhino(z); // A Rhino just sat on a Zebra!
```

Now if we now tried to invoke `z.a.showStripes`, we’d get a missing-method error. So it is surprising that Java actually allows the code above.

You might think that makes Java unsound, but in fact, it doesn’t. Instead, **Zebra a** adds a *different field* to instances of **ZebraZoo**. It just happens to have the same name as another field from the superclass. But they’re really two completely distinct fields, with their own individual space allocated in memory. The new field shadows the old field. So it’s okay to allow the new **a** field have a different type.

Unlike methods, fields don’t use dynamic dispatch. So any use of **a** in methods of **Zoo** will refer to the **a** field of **Zoo**, and any use of **a** in methods of **ZebraZoo** will refer the **a** field in **ZebraZoo**—not **Zoo**. To access **Zoo**’s **a** field, instances of **ZebraZoo** must write `super.a`.

**Static overloading.** From our discussion of function subtyping in the previous lecture, we know that a function’s argument can soundly be contravariant. However, Java does not permit overriding a method with contravariant arguments. Instead, if a class declares a method with name **m**, return type **t0** and argument types **t1**, **t2**, ..., **tn** (in that order), then:

- If the superclass already has a method name **m** with argument types **t1**, **t2**, ..., **tn** (in that order), then **m** is overridden in the subclass. The type-checker requires **t0** to be the same as (or a subtype of—which is covariant subtyping on return types) **m**’s return type in the superclass method.
- Otherwise, a new method added to the subclass. The original method is not overridden. So a class may have many methods with the same name.

Having multiple methods with the same name can be convenient so that you do not have to think up different names for similar behavior. For example, a **Rational** class could have methods like:

```java
void add(Rational r) {...}
void add(int i) {...}
void add(double d) {...}
```
But you must take care in Java to make sure you are overriding or not overriding when you intend to. It can be confusing to change or reorder argument types accidentally and, as a result, have different methods with the same name.

More importantly, we need to define which method is called when there are multiple methods with the right name. That is, we must revise our definition of the most fundamental issue in object-oriented programming: what is the semantics of $e_0.m(e_1,\ldots,e_n)$? As before, we start as follows:

- Evaluate $e_0,\ldots,e_n$ to objects $v_0,\ldots,v_n$.
- Use the (run-time) class of $v_0$ to look up $m$ (dynamic dispatch).
- But now there may be multiple choices named $m$. Which one should be picked?

Java picks the most specific choice using the static types of $e_1,\ldots,e_n$, not the run-time class of $v_0,\ldots,v_n$. This semantics is called static overloading, because name $m$ means multiple things, its meaning is determined by the static types of the arguments. Were the choice to be determined by the run-time class of the objects, we would instead have double dispatch (in the case of a single argument) or multimethods (in the general case of many arguments). Java does not have multimethods.

The exact definition of “most specific” choice is remarkably complicated. So we’ll just give some examples. Simple cases like the \texttt{add} methods for \texttt{Rational} are easy: type-checking the argument should produce an \texttt{int}, \texttt{double}, or \texttt{Rational} and the most specific choice is obvious.

Here is a more complicated example in which five methods are named $m$:

```java
class Color extends Object { String s; }
class FancyColor extends Color { double shade; }

class MyClass {
    void m(Object x) {...} // A
    void m(Color x) {...} // B
    void m(FancyColor x) {...} // C
    void m(Color x, FancyColor y) {...} // D
    void m(FancyColor x, Color y) {...} // E
}
MyClass obj = new MyClass(...);
Color c1 = new Color(...);
FancyColor c2 = new FancyColor(...);
Color c3 = new FancyColor(...); // subtyping!
obj.m(c1);  // B
obj.m(c2);  // C
obj.m(c3);  // B static overloading!
obj.m(c1,c2); // D
obj.m(c1,c3); // type error: no method matches
obj.m(c2,c2); // type error: no best match (tie)
```

The comments next to the methods and method calls indicate which method will get used by which call:

- \texttt{obj.m(c1)} calls $m$ with one argument of type \texttt{Color}. Both methods A and B would be sound, but B is more specific, so it is picked.
- \texttt{obj.m(c2)} is similar: it calls $m$ with one argument of type \texttt{FancyColor}. Even though A, B, and C all would be sound, C is the most specific.

\[^{1}\text{http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12.2.5}\]
• `obj.m(c3)` is handled exactly like `obj.m(c1)` because `c1` and `c3` both have type `Color`. It is irrelevant that the result of evaluating `c3` is an instance of the `FancyColor` class. Because of static overloading, Java stills pick method `B`.

• `obj.m(c1,c3)` calls `m` with two arguments of type `Color`. This does not type-check because none of the methods can take such arguments: the two-argument methods both require at least one argument to be a `FancyColor`. Again, the fact that evaluating `c3` would produce an instance of `FancyColor` class is irrelevant.

• `obj.m(c2,c2)` does not type-check for a fundamentally different reason: Methods `D` and `E` would both be sound, but they “tie”. Neither is more specific than the other. The type-checker rejects the call, rather than break the tie. The programmer can break the tie by rewriting the call as either `obj.m((Color)c2,c2)`, which would call `D`, or as `obj.m(c2,(Color)c2)`, which would call `E`. Or another method `m` that takes two `FancyColor` arguments could be added to `MyClass`.

4 Covariance of receiver

(The following discussion is true in general for statically-typed object-oriented languages, not just Java.)

At run-time, inside a method of class `C`, it’s guaranteed by dynamic dispatch that the run-time class of the receiver (named `this` in Java and C#, and `self` in Smalltalk and Ruby) is a subtype of `C`. So when type-checking `C`, the type checker soundly assumes that the receiver has type `C`. For example, in Java class `B` below, method `m` can type-check only if `this` has type `B`, not `A`.

```java
class A {
    int m() { return 0; }
}
class B extends A {
    int x=0;
    int m() { return x; }
}
```

But our encoding of objects as records in MF would represent `m` as a field of a record:

```plaintext
a:A = {m = fn self => 0}
b:B = {x = 0, m = fn self => self.x}
```

The type of `self` in `m` would be different in each object: in `a.m`, it would be `A`; whereas in `b.m`, it would be `B`. So the type of `self` is covariant.

“How can this be?”, you ask incredulously. In the previous lecture, your instructor insisted that argument types must be contravariant! It’s because the run-time actually looks at the run-time class of `this` to determine which method to invoke. The semantics of dynamic dispatch guarantee that the method invoked is exactly the right method for the class of `this`. So it’s impossible to “accidentally” end up executing a method that assumes something false about the class of `this`.

In general, arguments can be *statically dispatched*, meaning the method is chosen is based on the static types of the argument, or *dynamically dispatched*, meaning the method is chosen based on the run-time class of the argument. Statically dispatched arguments (like all the normal arguments of a method in Java or Ruby) must be contravariant, whereas dynamically dispatched arguments (like `this` and `self`) must be covariant.

5 Classes vs. types

Classes and types are different things! Many languages, including Java and C#, purposely conflate them because it’s convenient for programmers. But as a student of programming languages, you should keep these
concepts separate.

- A **class** defines an object’s behavior. **Subclassing** modifies run-time behavior through extension and overriding.

- A **type** defines what fields an object has and what messages it can respond to. **Subtyping** determines compile-time behavior through defining when one value is soundly substitutable for another.

So you should be careful with your vocabulary. It’s best to avoid saying things like “override a method in the *supertype*” (should be *superclass*), or “using subtyping to assign a value of a *subclass*” (should be *subtype*). That said, this confusion is understandable in languages like Java in which every class declaration simultaneously introduces a class and a type with the same name.