This lecture

How does subtyping for Java/C# relate to the subtyping in the last lecture?

Many of the same principles but Java/C#:

- Use class and interface names for types
- Support static overloading instead of contravariant arguments

What we have learned

- A record subtype can have more fields than its supertype
- A mutable record field cannot have its type change via subtyping
- An immutable record field can be covariant for subtyping (depth)
- Function subtyping uses contravariant argument types and covariant result types

Now can use this to understand how we could type-check OOP…

An object is…

- Objects are basically records holding fields and methods
  - Fields are mutable
  - Methods are immutable functions that also have access to this/self
- So we could design a type system using types very much like our record types from last lecture
  - Subtypes can have extra fields
  - Subtypes can have extra methods
  - Subtypes can have methods with contravariant arguments and covariant result compared to same method in supertype
    - Sound only because method "slots" are immutable!

Java is more restrictive

Java's object types don't look like:

```
{fields: x:real, y:real, ...
  methods: distToOrigin : () -> real, ...
}
```

Instead:

- Reuse class names as types
  - Type has everything implied by the class definition
- Add more types with interface definitions
- Have only the subtyping explicitly stated via extends and implements

Cannot get "field missing" or "method missing" errors because this approach allows a subset of the subtyping that would be sound

In Java…

- A subclass can add fields but not remove them (width)
- A subclass can add methods but not remove them (width)
- A subclass can override a method with a covariant return type
  - (Java didn't use to allow this)
  - Depth on immutable slot + function subtyping
  - But doesn't allow contravariant arguments (see later slides)
- A class can implement more methods than an interface requires (width)
  - Also allow covariant return types
Example (constructors and public omitted)

```java
class Pt {
    double x, y;
    double distance(Pt z) { … }  
    Pt shift(double dx, double dy) { … } 
}

interface Colorable {
    Color getColor();
    void setColor(Color c);
}

class ColorPt extends Pt implements Colorable {
    Color color;
    Color getColor () { return this.color; }
    void setColor(Color c) { this.color = c; }
    ColorPt shift(double dx, double dy) {
        Pt p = super.shift();
        return new ColorPt(p.x,p.y,this.color);
    }
}
```

Example so far

- An instance of `ColorPt` is substitutable for any value of type `Pt` or type `Colorable`
  - Adds field `color`
  - Gives `shift` a more specific return type
  - Adds methods w.r.t. `ColorPt` and w.r.t. `Colorable`

- What about changing the types of fields or method arguments?
  - Not possible in Java
  - For fields: to stay sound
  - For methods: because Java has static overloading instead
  - In both cases, "it type-checks" but "it" actually adds new fields/methods with the same name (kind of confusing)

More example (again omitting constructors)

```java
class ColorPt extends Pt implements Colorable {
    Color color;
    Color getColor () { return this.color; }
    void setColor(Color c) { this.color = c; }
    ColorPt shift(double dx, double dy) {
        Pt p = super.shift();
        return new ColorPt(p.x,p.y,this.color);
    }
}
```

class Color extends Object {
    String s;
}

class FancyColor extends Color {
    double shade;
}

class MyColorPt extends ColorPt {
    T1 color;
    T2 getColor () { … }
    void setColor(T3 c) { … }
}

Field shadowing

```java
class MyColorPt extends ColorPt {
    T1 color;
    …
}
```

- What we have learned: Mutable fields must have the same type in subclass and superclass, so no "overriding" possible
  - Changing to `Object` or `FancyColor` would be unsound
- Java: A field declared in the subclass can have the same name as an inherited field, but it is a new, different field
  - Field in subclass shadows
  - Can access other field with `super.color`
  - No dynamic dispatch: inherited methods use old field
- So: `T1` can be any type, `Object`, `Color`, `FancyColor`, `Pizza`
  - A different field with shadowing rules, not a subtyping issue

Method overriding / overloading

```java
class MyColorPt extends ColorPt {
    T2 getColor () { … }
    void setColor(T3 c) { … }
}
```

- What we have learned: If we replace a method with one of a different type, need contravariant arguments, covariant result
  - So `T2` could be `Color` or `FancyColor` (true in Java too)
  - So `T3` could be `Color` or `Object` (not `FancyColor`!)
- Java: A method declared with different argument types is a different method with the same name
  - So `T3` can be any type
  - If `T3` is `Color`, then we are overriding, for any other type, we are adding a new method
  - Simply no syntax for overriding with contravariant args 😞

Static overloading

- So a Java class can have multiple methods with the same name
  - Called overloading
- Must revisit the key question in OOP:
  - What does `e0.m(e1,…,en)` mean?
  - As before:
    - Evaluate `e0`, …, `en` to `v0`, …, `vn`
    - Look up `class` of `v0` (dynamic dispatch)
- But now the class may have more than one `m`
  - Java: Pick the "best" one using the static types of `e1`, …, `en`
    - The (run-time) class of `v1`, …, `vn` is irrelevant
    - "Best" is complicated, roughly "least amount of subtyping"
Static overloading examples

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)

So...

- Java's rules for subclassing and overriding are sound because they allow less than they could based on record and function subtyping
- Static overloading saves you the trouble of making up different method names
  - Often convenient, but the exact rules are complicated
  - This is not multimethods
    - So still have to code up double dispatch manually
    - Multimethods look up method using class of all args
- Biggest unnecessary restriction in Java is having subtyping only via subclasses and interfaces...

Names vs. structure

- From a "method not understood" perspective, no reason we couldn't make ThreeActPlay <: StringPair

class StringPair {
  String first;
  String second;
  void setFirst(String x){ ... }
}
class ThreeActPlay {
  String first;
  String second;
  String third;
  void setFirst(String x){ ... }
}

- Silly example, but key idea behind duck-typing: Is the type of an object "what it can do" or "its place in the class hierarchy"
  - Interfaces the former, but require explicit implements clause

Classes vs. Types

- A class defines an object's behavior
  - Subclassing inherits behavior and changes it via extension and overriding
- A type describes an object's field and method types
  - A subtype is substitutable in terms of its field/method types
- These are separate concepts! Try to use the terms correctly!
  - Java/C# confuse them by requiring subclasses to be subtypes
  - A class name is both a class and a type
  - This confusion is convenient in practice

What if?

- If subclasses did not have to be subtypes, then a ThreeDPoint could override distance to take a ThreeDPoint argument
  - Not allowed via subtyping (arguments are contravariant)
  - But only works if other methods in superclass do not assume the type
  - (Such a method allowed in Java via overloading)
- If subtypes did not have to be subclasses, then could have a Launchable type for any class with a method void launch()
  - This is what interfaces are for
  - Classes still have to explicitly "opt-in" to implementing Launchable
  - Allows more subtyping, which allows more code reuse, but means you have to keep track of when you are launching a Missile versus a MarketingCampaign

Abstract methods again

- Abstract methods are about the type of the class name
  - All values of the type have the method
  - So subclasses with instances must implement the method
- Abstract methods have nothing to do with defining behavior
  - This is why Ruby doesn't have them
self/this *is special*

- Recall our Racket encoding of OOP-style
  - "Objects" have a list of fields and a list of functions that take *self* as an explicit extra argument

- So if *self/this* is a function argument, is it contravariant?
  - No, it's *covariant*: a method in a subclass can use fields and methods only available in the subclass: essential for OOP

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

- Sound because calls always use the "whole object" for *self*
- This is why coding up your own objects manually works much less well in a statically typed languages