So far...

Last lecture (among other things):
- The difference between OOP and "records of functions with shared private state" is dynamic-dispatch (a.k.a. late-binding) of self
- (Informally) defined method-lookup to implement dynamic-dispatch correctly: use run-time tags or code-pointers

Now:
- Purpose of static typing for (pure) OOP
- Subtyping and contrasting it with subclassing
- Static overloading
- Multimethods

Type-Safety in OOP

Remember the two main goals we had with static type systems:
- Prevent “getting stuck” which is how we encode language-level errors in our operational semantics
  - Without rejecting too many useful programs
  - Enforce abstractions so programmers can hide application-level things and enforce invariants, preconditions, etc.
    - Subtyping and parametric polymorphism do this in complementary ways, assuming no downcasts or other run-time type tests

Pure OOP has only method calls (and field accesses)
- A method-lookup is stuck if receiver has no method with right name/arity (no match)
- (If we add overloading,) a method-lookup is stuck if receiver has no “best” method (no best match)

Structural or Nominal

A straightforward structural type system for OOP would be like our type system with record types and function types
- An object type lists the methods that objects of that type have, plus the the types of the argument(s) and result(s) for each method
- Sound subtyping just as we learned
  - Width, permutation, and depth for object types
  - Contravariant arguments and covariant result for each method type in an object type

A nominal type system could give named types and explicit subtyping relationships
- Allow a subset of the subtyping (therefore sound) of the structural system (see lecture 11 for plusses/minuses)
- Common to reuse class names as type names and require subclasses to be subtypes...
**Subclassing is Subtyping**

Statically typed OOP languages often purposely “confuse” classes and types: \( C \) is a class and a type and if \( C \) extends \( D \) then \( C \) is a subtype of \( D \).

Therefore, if \( C \) overrides \( m \), the type of \( m \) in \( C \) must be a subtype of the type of \( m \) in \( D \).

Just like functions, method subtyping allows contravariant arguments and covariant results:
- If code knows it has a \( C \), it can call methods with “more” arguments and know there are “fewer” results.

**Subtyping and Dynamic Dispatch**

We defined dynamic dispatch in terms of functions taking `self` as an argument.

But unlike other arguments, `self` is covariant!!
- Else overriding method couldn’t access new fields/methods.
- Sound because `self` must be passed, not another value with the supertype.

This is the key reason encoding OOP in a typed \( \lambda \)-calculus requires ingenuity, fancy types, and/or run-time cost.

---

**More subtyping**

With single-inheritance and the class/type confusion, we don’t get all the subtyping we want.
- Example: Taking any object that has an \( m \) method from \( \mathbb{int} \) to \( \mathbb{int} \).

Interfaces help somewhat, but class declarations must still say they implement an interface.
- An interface is just a named type independent of the class hierarchy.

**Why subsume?**

Subsuming to a supertype allows reusing code expecting the supertype.

It also allows hiding if you don’t have downcasts, etc. Example:

```java
interface I { int distance(Point1 p); }
class Point1 implements I { ... I f() { self } ... }
```

But again objects are awkward for many binary methods.
- `distance` takes a `Point1`, not an `I`.
More subclassing

Breaking one direction of “subclassing = subtyping” allowed more subtyping (so more code reuse and/or information hiding)

Breaking the other direction (“subclassing does not imply subtyping”) allows more inheritance (so more code reuse)

Simple idea: If \( C \) extends \( D \) and overrides a method in a way that makes \( C \leq D \) unsound, then \( C \not\leq D \). This is useful:

```java
class P1 {
    Int x;
    Int get_x() { x }
    Bool compare(P1 p) { self.get_x() == p.get_x() }
}
class P2 extends P1 {
    Int y;
    Int get_y() { y }
    Bool compare(P2 p) { self.get_x() == p.get_x() &&
                             self.get_y() == p.get_y() }
}
```

But this is not always correct...

Subclass not a subtype

- Can still inherit implementation (need not reimplement `get_x`)
- We cannot always do this: what if `get_x` called `self.compare`? Possible solutions:
  - Re-typecheck `get_x` in subclass
  - Use a “Really Fancy Type System”

There may be little use in allowing subclassing that is not subtyping

Summary of subclass vs. subtype

- Typing is about interfaces, subtyping about broader interfaces
- Subclassing is about inheritance and code-sharing

Combining typing and inheritance restricts both

- Most OOP languages purposely confuse subtyping (about type-checking) and inheritance (about code-sharing), which is reasonable in practice
- But please use `subclass` to talk about inheritance and `subtype` to talk about static checking
Static Overloading

So far, we have assumed every method had a different name
▶ Same name implied overriding and required a subtype

Many OOP languages allow the same name for different methods with different argument types:

\[
\begin{align*}
A & \ f(B \ x) \ \{ \ldots \} \\
C & \ f(D \ x, \ E \ y) \ \{ \ldots \} \\
F & \ f(G \ x, \ H \ z) \ \{ \ldots \}
\end{align*}
\]

Complicates definition of method-lookup for \( e_1.m(e_2, \ldots, e_n) \)

Previously, we had dynamic-dispatch on \( e_1 \): method-lookup a function of the class of the object \( e_1 \) evaluates to (at run-time)

We now have static overloading: Method-lookup is also a function of the types of \( e_2, \ldots, e_n \) (at compile-time)

Multiple Dispatch

Static overloading saves keystrokes from shorter method-names
▶ We know the compile-time types of arguments at each call-site, so we could call methods with different names

Multiple (dynamic) dispatch (a.k.a. multimethods) is more interesting: Method-lookup a function of the run-time types of arguments

It’s a natural generalization: the “receiver” argument is no longer treated differently!

So \( e_1.m(e_2, \ldots, e_n) \) is just sugar for \( m(e_1,e_2,\ldots,e_n) \)
▶ It wasn’t before, e.g., when \( e_1 \) is self and may be a subtype

Example

class A { int f; }
class B extends A { int g; }
Bool compare(A x, A y) { x.f == y.f }
Bool compare(B x, B y) { x.f == y.f && x.g == y.g }
Bool f(A x, A y, A z) { compare(x,y) && compare(y,z) }

Neat: late-binding for both arguments to compare (choose second method if both arguments are subtypes of \( B \), else first method)

With power comes danger. Tricky question: Can we add “\&\& compare(x,z)” to body of \( f \) and have an equivalent function?
▶ With static overloading?
▶ With multiple dispatch?

Static Overloading Continued

Because of subtyping, multiple methods can match a call!

“Best-match” can be roughly “Subsume fewest arguments. For a tie, allow subsumption to immediate supertypes and recur”

Ambiguities remain (no best match):
▶ A \( f(B) \) vs. \( C \ f(B) \) (usually rejected)
▶ A \( f(I) \) vs. A \( f(J) \) for \( f(e) \) where \( e \) has type \( T, T \leq I, T \leq J \) and \( I,J \) are incomparable (possible with multiple interfaces or multiple inheritance)
▶ A \( f(B,C) \) vs. A \( f(C,B) \) for \( f(e_1,e_2) \) where \( B \leq C \), and \( e_1 \) and \( e_2 \) have type \( B \)

Type systems often reject ambiguous calls or use ad hoc rules to give a best match (e.g., “left-argument precedence”)
Pragmatics

Not clear where multimethods should be defined
  ▶ No longer “belong to a class” because receiver isn’t special

Multimethods are “more OOP” because dynamic dispatch is the essence of OOP

Multimethods are “less OOP” because without a distinguished receiver the analogy to physical objects is reduced

Nice paper in OOPSLA08: “Multiple Dispatch in Practice”

Revenge of Ambiguity

The ”no best match” issues with static overloading exist with multimethods and ambiguities arise at run-time

It’s undecidable if “no best match” will happen:

```plaintext
// B <= C
A f(B,C) {...}
A f(C,B) {...}
unit g(C a, C b) { f(a,b); /* may be ambiguous */ }
```

Possible solutions:
  ▶ Raise exception when no best match
  ▶ Define “best match” such that it always exists
  ▶ A conservative type system to reject programs that might have a “no best match” error when run