CSE 341: Programming Languages

Winter 2005

Lecture 25— Named Types; Class vs. Types; Interfaces

Named Types

In Java/C++/C#/..., types don't look like `[t10 m1:(t11,...), ..., tn0 mn(tn1,...)]`.

Instead they look like `C` where `C` is a class or interface.

But everything we just learned about subtyping still applies!

Yet the only subtyping is (the transitive closure of) declared subtypes (e.g., `class C extends D implements I,J`).

Given *types* `D`, `I`, and `J`, ensure objects produced by *class* `C`'s constructors can have subtypes (more methods, contra/co, etc.)

Named vs. Unnamed

For preventing "message not understood", unnamed ("structural") types worked fine.

But many languages have named ("nominal") types.

Which is better is a tired old argument, but fortunately it has some interesting intellectual points (unlike emacs vs. vi).

First, frame the question more narrowly: Should subtyping be nominal or structural? (Named types don't preclude structural subtyping, e.g. casting between two otherwise-unrelated interfaces.)

Some Fair Points

For structural subtyping:

- Allows more code reuse, while remaining sound.
- Does not require refactoring or adding "`implements` clauses" later when you discover you could share some implementation.
- A simpler system (type names are just an abbreviation and convenient way to write recursive types)

For nominal subtyping:

- Reject more code, which catches bugs and treating unmeaningful method-name clashes as significant.
- Confusion with classes saves keystrokes and "doing everything twice"?
- Fewer subtypes makes type-checking (??) and efficient code-generation easier.

The Grand Confusion

For convenience, many languages *confuse* classes and types:

- `C` is a class and a type
- If `C` extends `D`, then:
  - instances of the class `C` inherit from the class `D`
  - expressions of type `C` can be subsumed to have type `D`

Do you usually want this confusion? Probably.

Do you always want "subclass implies subtype"?

- No: Recall `distTo` for `Point` and `3DPoint`.

Do you always want "subtype implies subclass"?

- No: Two classes with `display` methods may have no inheritance relationship.

Untangling Classes and Types

- Classes define object behavior; subclassing inherits behavior
- Subtyping defines substitutability
- You often want subclasses to be subtypes; most languages give you no choice.

Now some other common features make more sense:

- "Abstract" methods:
  - Expand the supertype without providing behavior to subclass
  - Superclass does not implement behavior, so no constructors allowed (an additional static check because the *class* is abstract)
  - The static-check is the only fundamental justification (trivial to provide a method that raises an exception).
- Interfaces...

Interfaces

A Java interface is just a (named) object type.

By implementing an interface, you get subsumption but no behavior.

- Same thing with "multiple inheritance" when $n - 1$ superclasses have all abstract methods. Should be called "multiple subsumance", but *subsumance* is not a word. :)
- None of the semantic issues we previously discussed with multiple inheritance arise with interfaces.
- But there are issues we didn't discuss before because they're about typing, and we'll skip now:
  - Lack of least supertypes
  - Ambiguity if "subsumption is not a run-time no-op" (*coercive*)