# CSE341: Programming Languages

## Lecture 25
## Subtyping for OOP; Comparing/Combining Generics and Subtyping

Dan Grossman
Spring 2017

## *Now…*

Use what we learned about subtyping for records and functions to understand subtyping for class-based OOP
- Like in Java/C#

Recall:
- Class names are also types
- Subclasses are also subtypes
- Substitution principle: Instance of subclass should usable in place of instance of superclass



## *An object is…*

- Objects: mostly records holding fields and methods
  - Fields are mutable
  - Methods are immutable functions that also have access to `self`
- So *could* design a type system using types very much like record types
  - Subtypes could have extra fields and methods
  - Overriding methods could have contravariant arguments and covariant results compared to method overridden
    - Sound only because method "slots" are immutable!



## *Actual Java/C#…*

Compare/contrast to what our "theory" allows:

1. Types are class names and subtyping are explicit subclasses
2. A subclass can add fields and methods
3. A subclass can override a method with a covariant return type
   - (No contravariant arguments; instead makes it a non-overriding method of the same name)

(1) Is a subset of what is sound (so also sound)

(3) Is a subset of what is sound and a different choice (adding method instead of overriding)



## *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 methods' argument/result 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
  - Confusion is convenient in practice



## *Optional: More details*

Java and C# are sound: They do not allow subtypes to do things that would lead to "method missing" or accessing a field at the wrong type

Confusing (?) Java example:
- Subclass can declare field name already declared by superclass
- Two classes can use any two types for the field name
- Instance of subclass have two fields with same name
- "Which field is in scope" depends on which class defined the method

## **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 is *covariant*: a method in a subclass can use fields and methods only available in the subclass: essential for OOP

```
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



## *What are generics good for?*

Some good uses for parametric polymorphism:

- Types for functions that combine other functions:

```
fun compose (g,h) = fn x => g (h x)
(* compose : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c) *)
```

- Types for functions that operate over generic collections

```
val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
val swap : ('a * 'b) -> ('b * 'a)
```

- Many other idioms
- General point: When types can "be anything" but multiple things need to be "the same type"



## *Generics in Java*

- Java generics a bit clumsier syntactically and semantically, but can express the same ideas
  - Without closures, often need to use (one-method) objects
  - See also earlier optional lecture on closures in Java/C
- Simple example without higher-order functions (optional):

```
class Pair<T1,T2> {
  T1 x;
  T2 y;
  Pair(T1 _x, T2 _y){ x = _x; y = _y; }
  Pair<T2,T1> swap() {
     return new Pair<T2,T1>(y,x);
  }
  …
}
```



## *Subtyping is not good for this*

- Using subtyping for containers is much more painful for clients
  - Have to downcast items retrieved from containers
  - Downcasting has run-time cost
  - Downcasting can fail: no static check that container holds the type of data you expect
  - (Only gets more painful with higher-order functions like **map**)

```
class LamePair {
  Object x;
  Object y;
  LamePair(Object _x, Object _y){ x=_x; y=_y; }
  LamePair swap() { return new LamePair(y,x); }
}
// error caught only at run-time:
String s = (String)(new LamePair("hi",4).y);
```



## *What is subtyping good for?*

Some good uses for subtype polymorphism:

- Code that "needs a Foo" but fine to have "more than a Foo"
- Geometry on points works fine for colored points
- GUI widgets specialize the basic idea of "being on the screen" and "responding to user actions"



## *Awkward in ML*

ML does not have subtyping, so this simply does not type-check:

```
(* {x:real, y:real} -> real *)
fun distToOrigin ({x=x,y=y}) =
   Math.sqrt(x*x + y*y)

val five = distToOrigin {x=3.0,y=4.0,color="red"}
```

Cumbersome workaround: have caller pass in getter functions:

```
(* ('a -> real) * ('a -> real) * 'a -> real *)
fun distToOrigin (getx, gety, v) =
   Math.sqrt((getx v)*(getx v)
             + (gety v)*(gety v))
```

  - And clients still need different getters for points, color-points

## Wanting both

- Could a language have generics and subtyping?
  - Sure!
- More interestingly, want to combine them
  - "Any type **T1** that is a subtype of **T2**"
  - Called bounded polymorphism
  - Lets you do things naturally you cannot do with generics or subtyping separately



## Example

Method that takes a list of points and a circle (center point, radius)
- Return new list of points in argument list that lie within circle

Basic method signature:

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

Java implementation straightforward assuming **Point** has a **distance** method:

```
List<Point> result = new ArrayList<Point>();
for(Point pt : pts)
  if(pt.distance(center) < r)
    result.add(pt);
return result;
```



## Subtyping?

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

- Would like to use **inCircle** by passing a **List<ColorPoint>** and getting back a **List<ColorPoint>**
- Java rightly disallows this: While **inCircle** would "do nothing wrong" its type does not prevent:
  - Returning a list that has a non-color-point in it
  - Modifying **pts** by adding non-color-points to it



## Generics?

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

- We could change the method to be

```
<T> List<T> inCircle(List<T> pts,
                     Point center,
                     double r) { … }
```

  - Now the type system allows passing in a **List<Point>** to get a **List<Point>** returned or a **List<ColorPoint>** to get a **List<ColorPoint>** returned
  - But cannot implement **inCircle** properly: method body should have *no* knowledge of type **T**



## Bounds

- What we want:

```
<T> List<T> inCircle(List<T> pts,
                     Point center,
                     double r) where T <: Point
  { … }
```

- Caller uses it generically, but must instantiate **T** with some subtype of **Point** (including **Point**)
- Callee can assume **T <: Point** so it can do its job
- Callee must return a **List<T>** so output will contain only elements from **pts**



## Real Java

- The actual Java syntax:

```
<T extends Pt> List<T> inCircle(List<T> pts,
                                Pt center,
                                double r) {
   List<T> result = new ArrayList<T>();
   for(T pt : pts)
     if(pt.distance(center) < r)
       result.add(pt);
   return result;
}
```

- Note: For backward-compatibility and implementation reasons, in Java there is actually always a way to use casts to get around the static checking with generics ☹
  - With or without bounded polymorphism