Review

- OOP features: interfaces, multiple inheritance, mixins, double dispatch, subtyping

- Today:
  - Generics vs. subtyping
  - Bounded polymorphism

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:

  ```java
  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); }
  }
  ```
Generics are great for collections

```java
class LinkedList<T> {
    T head;
    LinkedList<T> tail;
    T getHead() { return head; }
    LinkedList<T> getTail() { return tail; }
}

class HashMap<K, V> {
    V get(K key) {
        ...
    }
    void put(K key, V value) {
        ...
    }
}
```

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)

```java
class LamePair {
    Object x;
    Object y;
    LamePair(Object _x, Object _y){ x=_x; y=_y; }
    LamePair swap() { return new LamePair(y,x); }
}
```

```java
// 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:

```java
(* {x:real, y:real} -> real *)
fun distToOrigin ((x,y):real) = Math.sqrt(x*x + y*y)
val five = distToOrigin {x=3.0,y=4.0,color="red"}
```

```java
(* (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 \( T_1 \) that is a subtype of \( T_2 \)
  - This is 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:

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

Java implementation straightforward assuming `Point` has a `distance` method

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

Subtyping?

```java
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?

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

- We could change the method to be
  ```java
  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 we cannot implement `inCircle` properly because method body should have no knowledge of type `T`
Bounds

• What we want:

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

{ ... }
```

• Caller uses it generically, but must instantiate `T` with a 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 list elements from input

Bounded polymorphism in Java

• The actual Java syntax

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

• 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

Implementation

How to implement generics and bounded polymorphism?

1. Modify the JVM (breaks debuggers) [C#]
2. Translate to Java without generics
   A. Keep enough information around to do run-time type checks for casts (performance penalty on method invocations) [Poly]
   B. Erasure: forget about the type variables (unsound casts) [Java]

Erasure

Main goal: don’t incur any runtime overhead

• Replace all type variables with their bounds
  – default bound is `Object`
• Insert casts where necessary
• (Generate "bridge" methods where necessary)
Erasing type variables

T is replaced by Object:

```java
// What the programmer writes:
class LinkedList<T> {
    T head;
    LinkedList<T> tail;
    T getHead() { return head; }
}

// What the compiler generates:
class LinkedList {
    Object head;
    LinkedList tail;
    Object getHead() { return head; }
}
```

Summary

- Generics = parametric polymorphism
- Subtyping = subtype polymorphism
- Generics+subtyping = bounded polymorphism
- Subtyping is a poor substitute for generics
- Generics are a poor substitute for subtyping
- Bounded polymorphism is more than the sum of its parts