CSCE 314
Programming Languages

Java Generics I

Dr. Hyunyoung Lee

Java Generics: History

- Pizza: 1996-97, extended Java with generics, function pointers, class cases and pattern matching.
- GJ: 1998 derivative of Pizza; Generics the only extension.
- Java 1.5: 2004. Modeled after GJ.
- PolyJ: 1997, would have required changes to JVM.
- NextGen: 1998, avoids oddities of type erasure, still compatible with JVM and existing binaries. Extension of GJ.

Java Generics: Motivation

Typesafe polymorphic containers

what happens
without type cast?

Without generics:

```
List l = new LinkedList();
l.add(new Integer(0));
Integer x = (Integer) l.iterator().next(); // need type cast
String s = (String) l.iterator().next(); // bad cast exception
```

With generics:

```
List<Integer> l = new LinkedList<Integer>();
l.add(new Integer(0));
Integer x = l.iterator().next(); // no need for type cast
String x = l.iterator().next(); // compile-time error
```

Parameterized Classes

```
public class Pair<T, U> { //T, U: type variables, also formal type parameters
  private T a; private U b;
  public Pair(T t, U u) { a = t; b = u; }
  public T getFirst() { return a; }
  public U getSecond() { return b; }
}
Pair<Integer, String> p = new Pair<Integer, String>(0, "");
// Pair<Integer, String> is an invocation/instance of the generic type
// declaration with actual type arguments.
```

Compare to Haskell:

```
data Pair a b = Pair a b
getFirst :: Pair a b -> a
getFirst (Pair x y) = x
getSecond :: Pair a b -> b
getSecond (Pair x y) = y
```

"Parametric polymorphism"
Generics in object oriented programming : Polymorphism in functional programming

(Mental) Model of Parametrized Classes

```
public class Pair<T, U> {
  private T a; private U b;
  public Pair(T t, U u) { a = t; b = u; }
  public T getFirst() { return a; }
  public U getSecond() { return b; }
}
Pair<Integer, String> p = new Pair<Integer, String>(0, "");
public class Pair<Object, Object> {
  private Object a; private Object b;
  public Pair(Object t, Object u) { a = t; b = u; }
  public Object getFirst() { return a; }
  public Object getSecond() { return b; }
}
```

Parametrized Methods

Example

```
public class ArrayUtil {
  . . .
  public static <E> void print(E[] a) {  // generic method
     for (E e : a) System.out.print(e.toString() + " ");
     System.out.println();
  }
}
Rectangle[] rects = . . . ;  String[] strs = . . . ;
ArrayUtil.print(rects);
ArrayUtil.print(strs);
```

Explicit instantiation allowed as well:

```
ArrayUtil.<Rectangle>print(rects);
ArrayUtil.<String>print(strs);
```

Parametric Polymorphism

Why does this work?

```
public static <E> void print(E[] a) {
  for (E e : a) System.out.print(e.toString() + " ");
  System.out.println();
}
```

In Haskell, this did not work:

```
print :: [a] -> IO ()
print ls = mapM_ (putStr . show) ls
```

But this did:

```
print :: Show a => [a] -> IO ()
print ls = mapM_ (putStr . show) ls
```

Parametric Polymorphism (Cont.)

Java, too, needs constraints to type parameters

Without constraints, only operations that are supported for all types can be applied to values whose types are type parameters.

If no constraints, the constraint extends Object is assumed:

```
public static <E extends Object> void print(E[] a)
{
   for (E e : a) System.out.print(e.toString() + " ");
   System.out.println();
}
```

"E extends Object" justifies toString

Example of Constraints

~~Erroneous:~~ OK

public static <E extends Comparable<E>>

~~public static <E>~~

```
void print(List<E> a, E threshold) {
  for (E e : a)
    if (e.compareTo(threshold) < 0) // ~~type error !!~~ OK
      System.out.print(e.toString() + " ");
  System.out.println();
}
```

Comparable interface itself is really parametrized (to be discussed)

Type Parameter with Multiple Bounds

Multiple bounds for a single type parameter allowed:

```
class SortedList<T extends Comparable & Serializable> { . . . }
```

In <T extends T1 & T2 & . . . & Tn>,

* extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces)
* If one of the bounds is a class, it must be specified first

Compare with multiple class constraints in Haskell:

```
smallToString :: (Show a, Ord a) => a -> [a] -> [String]
smallToString x xs = map show (filter (< x) xs)

*Main> smallToString 5 [1,2,3,4,5,6]
["1","2","3","4"]
```

Unnamed Type Parameters – Wildcards

```
static void printAll (List<?> l) {
  for (Object o : l) System.out.println(o);
}
```

Wildcards are both a convenience feature (more concise syntax), and to add support for co/contravariance for type parameters (discussed later).

Plain Bounded Quantification

Example (in C++ like syntax)

```
class Point { public int x; public int y; }
class ColorPoint extends Point { public int color; }
```

This establishes:

ColorPoint <: Point

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. "F-bounded polymorphism for object-oriented programming." Functional Programming Languages and Computer Architecture, pages 273-280, ACM, 1989.

Subtyping Example

```
class Point { public int x; public int y; }
class ColorPoint extends Point { public int color; }
Point move(Point a, int dx, int dy)
{ a.x += dx; a.y += dy; return a; }
. . .
Point p = new Point();
p.x = 0; p.y = 0;
p = move(p, 1, 2);

ColorPoint cp = new ColorPoint();
cp.x = 0; cp.y = 0; cp.color = 0;
cp = move(cp, 1, 2); // Type error!
p = move(cp, 1, 2);  // OK!
```

With just subtyping, the exact type of cp is lost when passed to and returned from move()

Bounded Quantification

Subtype polymorphism in itself is not sufficient

A possible fix to this losing of type accuracy is to use parametric polymorphism (instead of subtype polymorphism):

```
<T> T move(T a, int dx, int dy)
 { a.x += dx; a.y += dy; return a; }
```

Access to members x and y, and operations on them must be justified by some constraints

```
<T extends Point> T move(T a, int dx, int dy)
 { a.x += dx; a.y += dy; return a; }
```

Simple constraint: T itself does not occur in the constraint (in the literature, this is sometimes referred to as bounded quantification [Cardelli, Wegner 85])

Constraint is fixed for all instantiations, which turns out to be too limited

Problems with Bounded Quantification

```
interface Movable { Movable move(int x, int y); }
```

Define function translate that takes any Movable object, and returns another one of the same type, moved one unit along both axis. translate should have (roughly) the type:

$$\forall T <: \text{Movable}.\ \ T \to T$$

First attempt (does not work):

```
<T extends Movable> T translate(T m) { return m.move(1, 1); }
// type error! move is (essentially) of type
(Movable, int, int) -> Movable
```

So, we again hit the problem of subtyping losing information!

Binary Method Problem

Assume the following interface:

```
interface EqComparable { bool eq(EqComparable o); }
```

Task: Define a function noteq to compute the negation of eq

This is what bounded quantification enables:

```
<T extends EqComparable> bool noteq(T t, T u) { return !t.eq(u); }
```

Now define a class MyInt which is EqComparable

```
class MyInt implements EqComparable {
   bool eq(MyInt o) { ... }  // not a valid override
   bool eq(EqComparable o) { ... } // meaningless comparison
}
```

Detour: Rules for Method Overriding (1)

1. The argument list should be exactly the same as that of the overridden method.
2. The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
3. The access level cannot be more restrictive than that of the overridden method. For example: if the superclass method is declared public then the overridding method in the subclass cannot be either private or protected.

Detour: Rules for Method Overriding (2)

4. Constructors cannot be overridden.
5. A method declared final cannot be overridden.
6. A method declared static cannot be overridden but can be re-declared.
7. If a method cannot be inherited, then it cannot be overridden.

Detour: Rules for Method Overriding (3)

8. A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
9. A subclass in a different package can only override the non-final methods declared public or protected.
10. An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.

F-Bounded Quantification

F-bounded quantification allows the type parameter being constrained to appear in its own bound: <T extends A<T>>

The unsuccessful translate example:

```
<T extends Movable> T translate(T m) { return m.move(1, 1); }
```

can now be written as:

```
interface Movable<T> { T move(int x, int y); }

<T extends Movable<T>> T translate<T>(T m) { return m.move(1, 1); }
```

Binary Method Problem Revisited

- Assume the following interface:

```
interface EqComparable { bool eq(EqComparable); }
<T extends EqComparable> bool noteq(T t, T u){ return !t.eq(u); }
```

- Define a class MyInt which is EqComparable

```
class MyInt implements EqComparable {
  bool eq(MyInt) { . . . } // not a valid override
  bool eq(EqComparable) { . . . } // meaningless comparison
}
```

- Now, using F-bounds:

```
interface EqComparable<T> { bool eq(T); }
<T extends EqComparable<T>> bool noteq<T>(T t, T u)
{  return !t.eq(u); }
class MyInt implements EqComparable<MyInt> {
   bool eq(MyInt) { . . . }
}
```

Another Example: generic sort

```
<T extends Comparable<T>> void sort(T[] v)
{
    for(int i = 0; i < v.length; i++)
        for(int j = 0; j < i; j++)
            if (v[j].compareTo(v[i]) < 0) swap(v, i, j);
}
```

Summary

- Minor generalizations of F-bounded polymorphism are the backbone of Java, C#, and Eiffel generics
- The essential feature in F-bounds is that bounds are generic too — they change along the argument that is being tested for
- Subtyping is one way of expressing constraints, there are others (Haskell type classes, C++ concepts, ML signatures)