CSCE 314
Programming Languages

Java Generics II

Dr. Hyunyoung Lee

Type System and Variance

Within the type system of a programming language, variance refers to how subtyping between complex types (e.g., list of Cats versus list of Animals, and function returning Cat versus function returning Animal) relates to subtyping between their components (e.g., Cats and Animals).

Covariance and Contravariance

Within the type system of a programming language, a typing rule or a type constructor is:

- covariant if it preserves the ordering ≤ of types, which orders types from more specific to more generic
- contravariant if it reverses this ordering
- invariant if neither of these applies.

Co/Contravariance Idea

- Read-only data types can be covariant;
- Write-only data types can be contravariant;
- Mutable data types which support read and write operations should be invariant.

Even though Java arrays are a mutable data type, Java treats array types covariantly, by marking each array object with a type when it is created.

Covariant Method Return Type

Return types of methods in Java can be covariant:

```
class Animal {
  . . .
  public Animal clone() { return new Animal(); }
}
class Panda extends Animal {
  . . .
  public Panda clone() { return new Panda(); }
}
```

This is safe - whenever we call Animal's clone(), we get at least an Animal, but possibly something more (a subtype of Animal)

```
Animal p = new Panda();
 . . .
Animal a = p.clone(); // returns a Panda, OK
```

Covariant Method Argument Type (Bad Idea)

Would this be a good idea?

```
class Animal {
    . . .
   public bool compare(Animal) { . . . }
}
class Panda extends Animal {
    . . .
   public bool compare(Panda) { . . . }
}
```

Covariant argument types are not type safe, and thus not supported in (most) mainstream languages including Java

```
Animal p = new Panda();
Animal a = new Animal();
 . . .
p.compare(a); // type error at run time
```

Contravariant Argument Types?

Contravariant argument types would be safe, but they have not been found useful and hence not supported in practical languages (not in C++ nor in Java).

```
class Animal { . . .
   public bool compare(Panda) { . . . }
}
class Panda extends Animal { . . .
   public bool compare(Animal) { . . . }
}
```

```
Animal p = new Panda();
Animal a = new Animal();
 . . .
p.compare(a); // not useful
```

Summary: When overriding methods in Java - invariant argument type, covariant return type.

Co- and contravariance of parameter and argument types are best captured with this subtyping rules between function types:

$$\frac{U1 <: U2 \quad T2 <: T1}{U2 \rightarrow T2 <: U1 \rightarrow T1}$$

Co- and Contravariance and Generic Types

The same phenomenon occurs with generic definitions.

Subtyping of generic types:

Assume B<T> <: A<T> and Integer <: Number

Which of these are true (safe)?

B<Integer> <: A<Integer>

B<Integer> <: B<Number>

B<Integer> <: A<Number>

Only the first (pointwise subtyping)

Other two are forms of covariance in arguments, and unsafe (unless restricted suitably)

Co/Contravariance and Generic Types (Cont.)

In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is not the case that G<Foo> is a subtype of G<Bar>.

```
List<String> ls = new ArrayList<String>(); // ok
List<Object> lo = ls; // not ok, consider the following
lo.add(new Object()); // still ok
String s = lo.get(0); // attempt to assign an Object to a String!
```

Co- and Contravariance

Generic class C<T> is covariant with respect to T if A <: B implies C<A> <: C<B>

Generic class C<T> is contravariant with respect to T if A <: B implies C<B> <: C<A>

Generic class C<T> is invariant with respect to T if C<A> <: C<B> holds only if A = B.

Generic class C<T> is bivariant with respect to T if C<A> <: C<B> for any A and B.

Example: Java Arrays

```
Object[] o = new String[]{"1", "2", "3"};   // OK
o[0] = new Integer(1); // throws here
```

First statement OK. Java arrays are (unsafely) covariant:

String <: Object ⇒ String[] <: Object[]

You can think of String[] as Array<String>

From the type checker's perspective, second OK

Integer <: Object

However, the second statement is an attempt to assign an Integer in place of String.

Full functionality of arrays does not safely allow co/contravariance. Reasons:

1. Covariant arrays would be safe for reading elements from them.
2. Contravariant arrays would be safe for writing elements into them.
3. Java arrays support both reading and writing.

When is Co/Contravariance Safe?

Covariance on type parameter T requires T <u>not appear</u> as a type of a <u>writable</u> field, or an <u>argument</u> type of a method.

Contravariance on type parameter T requires T <u>not appear</u> as a type of a <u>readable</u> field, or as a <u>return</u> type of a method.

Example: class below is safely contravariant on X, and covariant on Y

```
class Pair<X, Y> {
    private X fst;
    private Y snd;
    Pair(X a, Y b) { this.fst = a; this.snd = b; }
    void setFst(X a) { this.fst = a; }
    Y getSnd() { return this.snd; }
}
```

Using Co/Contravariant Pair

Assume L <: D <: A, and assume co- and contravariant subtyping

```
class Pair<X, Y> { // x is contravariant, y is covariant
    private X fst;
    private Y snd;
    Pair(X a, Y b) { this.fst = a; this.snd = b; }
    void setFst(X a) { this.fst = a; }
    Y getSnd() { return this.snd; }
}
void setAndGet(Pair<D, D> p, D d) {
    p.setFst(d);
    D x = p.getSnd();
}
setAndGet(new Pair<A, L>(), new D()); // This would be safe
setAndGet(new Pair<L, A>(), new D()); // This would be unsafe
// It would be safe to allow:  Pair<A, L> <: Pair<D, D>
```

Example:
Animal
Dog extends Animal
Labrador extends Dog

Labrador <: Dog <: Animal

Java's Approach for Safe Covariance

Parametric polymorphisms and subtype polymorphism leaves a bit of a gap in between.

Various co/contravariance solutions attempt to fill that gap.

Java's approach is wildcards.

Example:

```
public abstract class Shape {
   public abstract void drawOn (Canvas c);
}
public class Circle extends Shape {
   private int x, y, radius;
   public void drawOn (Canvas c) { . . . }
}
public class Line extends Shape {
   private int x1, y1, x2, y2;
   public void drawOn (Canvas c) { . . . }
}
```

Wildcards Bound From Above

Example: List<Line> and List<Shape>

```
public class Canvas {
    public void draw(List<Shape> shapes) {
        for (Shape s: shapes) s.drawOn(this);
    }
}
List<Line> ll;  . . . ;  canvas.draw(ll);  // error
// Now with wildcards
public class Canvas {
    public void draw(List<? extends Shape> shapes) {
        for (Shape s: shapes) s.drawOn(this);
    }
List<Line> ll;  . . . ;  canvas.draw(ll);  // OK
// ? reads "unknown", corresponds to a fresh type variable
// Shape is the upper bound of the wildcard
```

Wildcards Bound From Below

Contravariance expressed with wildcards:

```
<T> void copy(List<T> src, List<? super T> dest)
          { . . .  dest.add(src.get(i)) . . . }
List<Object> lo;
List<String> ls;
copy(ls, lo); // ok
copy(lo, ls); // not ok
```

Two alternative formulations of copy:

```
<T> void copy(List<? extends T> src, List<T> dest);

<T, S extends T> void copy(List<S> src, List<T> dest);
```

The "fresh type variable" model describes wildcards to some extent, but does not apply in all contexts.

Type Inference with Wildcards

```
<T> T choose (T a, T b) { . . . }

Set<Integer> intSet = . . .
List<String> stringList = . . .

choose(intSet, stringList);
```

Without wildcards, only valid typing for T in the call to choose is

```
Object
```

With wildcards, T can be typed as

```
Collection<?>
```

Typing and Subtyping of Wildcards

For any T, List<T> <: List<?>

Since element type is not known of an object of type List<?>,

- no objects can be written to it (nulls can)
- but Objects can be read from it

Wildcards in Variable Types

Wildcards are allowed in other contexts than in just method parameter types:

types of local variables, fields, arrays

Example:

```
Set<?> aSet = new HashSet<String>();
```

"Wildcard Capture" Rule

The type bound to a wildcard cannot (in general) be recovered.
Example:

```
public static <T> void addToSet(Set<T> s, T t) { . . . }
 . . .
Set<?> aSet = new HashSet<String>();
addToSet(aSet, "string"); // error
```

However "wildcard capture" allows this in some situations

```
<T> public static Set<T> foo(Set<T> s) { . . . }
 . . .
Set<?> aSet = new HashSet<String>();
Set<?> s = foo(aSet); // ok
```

In Set<?>, the type parameter of Set is bound to some type, and as foo is well-typed for Set<U> for any U, the call is fine (and T "captures" the wildcard).

The capturing is not visible outside of the method where it occurs

```
Set<String> bSet = foo(aSet); // error
```

Type Erasure

1. Backwards compatibility was a hard constraint for design
2. Generic Java programs are compiled to class files that run on unchanged JVM.
3. Generic types are used during type checking, then <u>erased</u> according to a set of translation rules:
   a. Type parameters are replaced with their bounds
   b. Parametrized types throw away the parameters
   c. Casts inserted where necessary

[Note: Compared to time prior generics, run-time casts still prevail (no improvement on performance) but they'll never throw an exception.]

Example of Erasure

Generic code

```
class A<T> { abstract T id(T x); }
class B extends A<Integer> { Integer id(Integer x) { return x; } }
```

Translated code

A<T> -> A

T -> Object (T had no bound, thus the bound is Object)

Create a <u>bridge</u> method to fix type incompatibilities created in the translation.

```
class A { abstract Object id(Object x); }
class B extends A {
    Integer id(Integer x) { return x; }
    Object id(Object x) { return id((Integer)(x)); } // bridge
}
```

Example of Erasure (Cont.)

Uses of generics must be translated too:

```
Integer foo(A<Integer> a) {
   return a.id(1);
}
```

translates to:

```
Integer foo(A a) {
   return (Integer)a.id(1);
}
```

Implications

Only one run-time representation for all instances of a generic class:

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass()); // true
```

Some natural code is unnaturally rejected:

```
class Foo<T> {
   public void bar(T x) {
      T t = new T();                          // error
      if(x instanceof T) {}                   // error
   }
   public static void static_bar(T t) {} // error
   public static List<T> l;                   // error
}
```

Last two: no type parameters allowed in static context.

Implications (Cont.)

Type erasure can lead to two methods having ambiguous signatures

```
class C<T> { T id (T x) { . . . } }
class D extends C<String> {
   Object id(Object x) { . . . }
}
```

Illegal, C.id and D.id have the same type erasure.

Two different bounds for the same parameter cannot have same erasures.

A class may not directly or indirectly implement two interfaces with same erasures:

```
class B implements I<Integer> {}
class C extends B implements I<String> {}
```

Another example:

```
class DecimalString implements
     Comparable<Number>, Comparable<String> { . . . }
```

Primitive Types

boolean, int, etc. not allowed as type arguments for the type parameters.

Fix: Autoboxing, auto-unboxing of primitive types:

```
ArrayList<Integer> list = new ArrayList<Integer>();

// used to be like this
list.add(0, new Integer(42));
int total = (list.get(0)).intValue();

// thanks to auto(un)boxing, one can say
list.add(0, 42);
total = list.get(0);
```

Still affects performance.

Type Erasure Summary

- Type erasure was chosen for it so that no changes are required to the instruction set of the JVM.
- Desire to retain backward compatibility affected design decisions in Java libraries as well.
- Some confusing issues - sometimes compromises are necessary in practical language design.
- Aside: C# generics very similar, but C# does not fully erase type information.