COMP6700/2140 Generic Methods

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

March 2017

Generic Methods and Type Wild Cards

1. Methods with Parametrised Types
2. Why the standard "IsA"-rule doesn't work?
3. How to define polymorphic generic methods: PECS Principle
4. Java Generics: have they screwed this up?

Lecturettes

<?>

COMP 6700

Polymorphic Methods

Writing a polymorphic method with an ordinary argument and/or return value is a trivial exercise: "program to an interface" principle is in its most standard form:

```
public class GenMethods {

    static double square(Number x) {
        double y = x.doubleValue();
        return 1.0 * y*y;
    }

    public static void main(String[] args) {
        double y = 10.0;
        int x = 10;
        System.out.printf("squared %d is %.2f, and squared %.2f is %.2f%n",
           x, square(x), y, square(y));
    }
}
```

The actual parameter can be of any subclass ("IsA"-relationship):

```
% java GenMethods
squared 10 is 100.00, and squared 10.00 is 100.00
```

Generic Polymorphic Method, Take One

Over to methods which are passed (return) a parameterised type argument (return value):

```
static Number sum(List<Number> numbers) {
    double sum = 0;
    for (Number n: numbers) sum += n.doubleValue();
    return sum;
}
public static void main(String[] args) {
    List<Double> dnumbers = Arrays.asList(new Double[] {1.0, 1.0, 3.1, 5.2, 9.5});
    System.out.println(sum(dnumbers));
    List<Integer> inumbers = Arrays.asList(new Integer[] {1, 1, 3, 5, 9});
    System.out.println(sum(inumbers));
}
```

Compilation will fail:

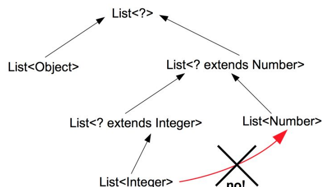
```
... error: incompatible types: List<Double> cannot be converted to List<Number>
        System.out.println(sum(dnumbers));
                               ^
...
```

because despite `Integer` and `Double` *are* subclasses of `Number`, neither `List<Integer>`, nor `List<Double>` are subclasses of `List<Number>`. How come?

Inheritance with Type Parameter

The reason is that the inheritance relation between parameterised types cannot be deduced from the inheritance relations between the parameter type values. In other words:

> *Just because* `Integer` *is a subclass of* `Number`*, it does not follow that* `List<Integer>` *is a subclass of* `List<Number>`*.*



Taxonomy of the generic *List* types

The parameterised types are *invariant* — for any two distinct types `T1` and `T2`, `List<T1>` is neither subclass to `List<T2>`, neither it is its superclass. Anything goes inside `List<Object>`, only strings go inside `List<String>`.

Generic Polymorphic Method, Take Two: Type Wildcard

To make substitution ("program to interface") principle work with parameterised types, one should make them *covariant* like in arrays: if `Sub` is a subtype of `Super`, then `Sub[]` is a subtype of `Super[]`.

A symbolic description of `G<T>` like *"G of T"* (`G` can be anything like *List*, *Iterable*, etc) should be replaced on *"G of some subtype of T"*, and Java has the notation for this:

```
G<? extends T>
```

Subtype here is defined in such a way that every type is its own subtype. The type declaration `? extends T` is known as *bounded wildcard parameter*. The symbol `?` is what Java syntax (since 1.5) calls the *wildcard* (don't confuse it with the same term in regular expressions). The `sum(List numbers)` method can be changed now:

```
static  Number sum(List<? extends Number> numbers) {
    double sum = 0;
    for (Number n: numbers)
        sum += n.doubleValue();
    return sum;
}
```

It will compile (ensuring type safety) and execute correctly for both `List<Integer>` and `List<Double>` (the main above).

PECS Principle

`<? extends E>` type declaration is used in generic methods to declare parameterised types which **supply** the data on which the method will perform computations. In other words, the "extend"-wildcard is used for objects which are read.

This is the first half of the *PECS* rule: **Producer Extends** — the producer is the object (usually a container type like *Collection* or *List*) with data which are **supplied** to the method code.

When an object **consumes** — uses to perform computations – the data, its generic type declaration must be made with "super"-wildcard (again, super-type also includes the type itself):

> *`G<? super T>` — "G of some super-type of T"*

This is the second part: **Consumer Supers** ("supers" is used as a verb here — in English, everything can be turned into a verb — which simply means "being a super-type to"). To see it in action, let's extend the previous example with a method `uniqueNumbers()` and its use (in main):

```
static <E> void uniqueNumbers(List<E> numbers, List<E> uniques) {
    for (E n: numbers) if (!uniques.contains(n)) uniques.add(n);
}
// added to main
List<Number> uniques = new ArrayList<>();
uniqueNumbers(inumbers, uniques);
uniqueNumbers(dnumbers, uniques);
```

PECS Principle

The compilation fails because the types of actual parameters do not match those in the method declaration:

```
error: method uniqueNumbers in class GenMethods cannot be applied to given types;
     uniqueNumbers(inumbers, uniques);
     ^
error: method uniqueNumbers in class GenMethods cannot be applied to given types;
     uniqueNumbers(dnumbers, uniques);
     ^
```

The reason is that (in both cases) "inferred type does not conform to equality constraints" (*ie*, when the method is invoked, the compiler cannot determine the value of `E`). When the uniques list type is declared with "Consumer Super" rule, everything starts working:

```
static <E> void uniqueNumbers(List<E> numbers, List<? super E> uniques) {
    for (E n: numbers)
        if (!uniques.contains(n)) uniques.add(n);
}
```

since `E` is inferred to be *Number*, and there is the full type compliance at run time. (The first argument `List<E> numbers` can be declared as `List<? extends E> numbers` but this does not change neither the compilation, not execution.) The full code is in GenMethods.java.

PECS Principle: to both parties

Let `list` be a `List<T>` of elements of the type `T` which can be compared, *ie* `T implements Comparable<T>`. With this property, it is well defined to ask "what is the maximum element in the list `list`". The method `max` declaration will involve a *recursive type bound*:

```
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

(the implementation body is pretty straightforward, it involves an iterator since one has to compare each successive pair of the elements). **But** if to apply the PECS rule, the *Comparable* should be treated as consumer (*Comparable* and *Comparator* **are always consumers**!), and the `list` is obviously considered as a producer. Therefore, the more general declaration should be:

```
public static <T extends Comparable<? super T>> T max( List<? extends T> list);
```

This is an example (a rare one, outside APIs) of type parameter containing the recursive wildcard bound of both types.

With the declaration like this, the local iterator used in the body must be also declared as `Iterator<? extends T> it = list.iterator();` to pass the compilation.

Type Erasure and Reification

The question: How type parameters are dealt with in JVM (at the level of bytecode)? has a surprisingly simple answer: “They ain't!” Type parameters are not types themselves: When compiled, the generic type information is totally erased: the Java uses the **type erasure** technique to interoperate freely with legacy code that does not use generics.

```
class A<T> {
    private T field;
    public A (T param) { this.field = param; }
    public T getField() { return this.field; }
}
```

when compiled, becomes “just” `A.class`, and

```
A<String> aS = new A<String>("I get it!");
A<Integer> aI = new A<Integer>(100);
as.getClass() == aI.getClass(); // returns true
```

Unlike generic containers, the old “quasi-generic” type arrays `T[]` do retain their type information (the value of `T` of the arrays elements), and they enforce this type constraint at run-time. Arrays are **reified** types unlike generics. Reifiable types also include primitives, non-parameterised types, parameterised types with unbound wildcards (eg, `List<?>`), raw types and arrays whose elements are also reifiable. The full story of reifiable and erased types in Java is an abstruse one.

“Did we screw it up?” (simple answer)

“Hum... I dunno, but it’s useful, JFC and stuff”. Yet —

1. When students first see the API with thousands of classes, they despair. I used to be able to tell them, “That’s OK, at least the language itself is very simple.” But that was before this:

```
static <T extends Object & Comparable<? super T>> T
                    Collections.max(Collection<? extends T> coll);
```

   (Cay Hortsmann’s interview Java Champion, Feb 2008))

2. “I am completely and totally humbled. Laid low. I realise now that I am simply not smart at all. I made the mistake of thinking that I could understand generics. I simply cannot. I just can’t. This is really depressing. It is the first time that I’ve ever not been able to understand something related to computers, in any domain, anywhere, period.”
3. “I’m the lead architect here, have a PhD in physics, and have been working daily in Java for 10 years and know it pretty well. The other guy is a very senior enterprise developer (wrote an email system that sends 600 million emails/year with almost no maintenance). If we can’t get [generics], it’s highly unlikely that the ‘average’ developer will ever in our lifetimes be able to figure this stuff out.” (both citations by Josh Bloch at *JavaPolis 2007*.)

“Did we screw it up?” (difficult answer)

Yes, they did! Otherwise, how else would this be possible:

```
public class Unbelievable {
    static Integer i;
    //static int i; // use this instead of previous and ... surprise!
    public static void main(String[] args) {
        if (i == 42)
            System.out.println("Unbelievable");
        else
            System.out.println("Unbelievable still");
    }
}
```

Guess what will be printed, and then compile and run to test yourself! Why is this happening? Again, the ugly beast of `null` (see Exceptions Lecture) rises its head.

“We simply cannot afford another *wildcards*” (Joshua Bloch’s talk at *JavaPolis2007* while talking about the prospects of introducing *closures* into the language; Closures are what now become λ-expressions).

Remark: despite a few (moderate) warts exhibited by lambdas, it turns out they are less controversial language feature than generics. Kudos by Brian Goetz and other guys!

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 6.3, 6.4, 6.5
- Oracle's Java Tutorial
  - Generic Methods
  - Wildcards
  - Restrictions on Generics
- Joshua Bloch's "Effective Java", Ch. 5, esp. *Item 28* (I followed this exposition here)
- Maurice Naftalin and Philip Wadler's book "Java Generics and Collections" (one of the most complete and consistent exposition of this abstruse subject)