CS61B Lecture 25

Friday, March 20, 2020

Java Generics

We have been using generics for quite some time now, but it's probably a good idea to examine how they actually work.

The Old Days

Java library types like `List`s didn't used to be parameterized -- All `List`s were lists of `Object`s, and you'd have to cast items back to their original type. You would do something like:

```
for (int i = 0; i < L.size(); i++) {
    String s= (String) L.get(i);
}
```

That is, you must explicitly cast result of `get` to let the compiler know its type. Also, when calling `L.add`, there was no check that you put only objects of a certain type into the list.

The Java designers decided this was a problem, and in version 1.5, introduced parameterized types to the language as we do today, like `List<String>`, like C++.

Unfortunately, it is not as simple as one might think, as it introduces a whole set of complexities and semantics into the language. They also had to work with the additional restriction of doing this without breaking compatibility with existing Java programs.

Basic Parameterization

What's the idea? It's not particularly difficult. From the definitions of `ArrayList` and `Map` in `java.util`, we see that these act sort of like functions, but instead of taking in a pointer or primitive as a value, they take in a type.:

```
public class ArrayList<Item> implements List<Item> {
    public Item get(int i) { ... }
    public boolean add(Item x) { ... }
    ...
}

public interface Map<Key, Value> {
    Value get(Key x);
    ...
}
```

The first occurrences of `Item`, `Key` and `Value` introduce formal type parameters, whose "values" (reference types) get substituted for all the other occurrences of `Item`, `Key` or `Value` whenever `ArrayList` or `Map` is "called".

Other occurrences of `Item`, `Key` and `Value` are uses of the formal types, just like uses of a formal parameter in the body of a function.

Type Instantiation

Instantiating a generic type is analogous to calling a function. Consider again:

```
public class ArrayList<Item> implements List<Item> {
    public Item get(int i) { ... }
    public boolean add(Item x) { ... }
    ...
}
```

When we "call" it with `ArrayList<String>`, we are, in effect, creating a new type:

```
public class StringArrayList implements List<String> {
    public String get(int i) { ... }
    public boolean add(String x) { ... }
    ...
}
```

And likewise, `List<String>` "creates" a new interface type as well. We say "in effect" because Java does not actually create a new type, but it's the basic idea of it.

Parameters on Methods

Not only can classes be parameterized by type, but also functions. Here is an example of this from `java.util.Collections`:

```
/** A read-only list containing just ITEM. */
static <T> List<T> singleton(T Item) { ... }

/** An unmodifiable empty list. */
static <T> List<T> emptyList() { ... }
```

The compiler figures out `T` in the expression `singleton(x)` by looking at the type of `x`. This is a simple example of **type inference**.

In the call `List<String> empty = Collections.emptyList()`, the parameters obviously don't suffice, but the compiler deduces the parameter `T` from context and assigns it to `String`.

Java has gotten progressively smarter and smarter: in the newest versions, it is possible to have local variables where you do not need to specify the type. It is not covered in this course, but it is covered under the "var" keyword.

Wildcards

Consider the definition of something that counts the number of items something occurs in a collection of items. We could write this as:

```
sttic <T> int frequency(Collection<T> c, Object x) {
    int n = 0;
    for (T y : c) {
        if (x.equals(y)) {
            n++;
        }
    }
    return 1;
}
```

In this particular case, we don't *care* what `T` is, because we don't need to declare anything of type `T` in the body. Thus, we could instead write:

```
    ...
    for (Object y : c) {
    ...
```

Wildcard type parameters say that you don't care what type a parameter is -- any subtype of `Object` will do:

```
static int frequency(Collection<?> c, Object x) { ... }
```

Subtyping

This generics business raises an entirely new set of questions regarding the relationship between subtypes. We know that `ArrayList` is a subtype of `List`, and `String` is a subtype of `Object`. Is `List<String>` then a subtype of `List<Object>`?

It turns out that's not quite true... Consider this fragment if this were true:

```
List<String> LS = new ArrayList<String>();
List<Object> LO = LS;   // OK?
int[] A = { 1, 2 };
LO.add(A);              // Legal, since A is an Object
String S = LS.get(0);   // OOPS! A.get(0) is NOT a String, even though
                        // the spec of List<String>.get says it is.
```

Having `List<String>` as a subtype of `List<Object>` would violate type safety: the compiler is wrong about the type of a value. So in general, for `T1<x>` to be the subtype of `T2<y>`, `x` must be identical to `y`. But what about `T1` and `T2`?

Now consider:

```
ArrayList<String> ALS = new ArrayList<String>();
List<String> LS = ALS;
```

In this case, everything is fine:

- The object's dynamic type is `ArrayList<String>`.
- Therefore, the methods expected for `LS` must be a subset of those for `ALS`.
- Since the type parameters are the same, the signatures of those methods will be the same.
- Therefore, all legal calls on methods of `LS` (according to the compiler) will be valid for the actual object pointed to by `LS`.

In general, `T1<X>` is a subtype of `T2<X>` if `T1` is a subtype of `T2`.

Java Inconsistencies

You would think that in an ordered programming language, this rule would be carried consistently throughout. In Java, this is not the case.

For the same reason that `ArrayList<String>` should not be a subtype of `ArrayList<Object>`, you should also expect that `String[]` is not a subtype of `Object[]`. And yet it is! One can get into trouble with:

```
String[] AS = new String[3];
Object[] AO = AS;
AO[0] = new int[] {1, 2};
```

So in Java, the last line causes an `ArrayStoreException` a (dynamic) runtime error instead of a (static) compile-time error. Why is this the case? Because basically, there would otherwise be no way to implement `ArrayList`, and other classes.

Type Bounds

Sometimes, your program needs to ensure that a particular type parameter is replaced only by a subtype (or super-type) of a particular type (sort of like specifying "the type of a type"). For example,

```
class NumericSet<T extends Number> extends HashSet<T> {
    /** My minimal element */
    T min() { ... }
}
```

This requires that all type parameters to `NumericType` must be subtypes of `Number` (the **type bound**). I can either extend to implement the type bound as appropriate.

Here is another example from the `Collections` library:

```
/** Set all elements of L to X. */
static <T> void fill(List<? super T> L, T x) { ... }
```

This code means that `L` can be a `List<Q>` as long as `T` is a subtype of (extends or implements `Q`). Why didn't the library designers just define this as the code below?

```
/** Set all elements of L to X. */
static <T> void fill(List<T> L, T x) { ... }
```

Well, consider the case:

```
static void blankIt(List<Object> L) {
    fill(L, " ");
}
```

This would be illegal if `L` were forced to be a `List<String>`.

> **Python's way**

> Here is an example of the complexity that generics add. You might contemplate in all of this how Python handles this. Well, in fact, Python doesn't bother with any of this because it doesn't do compile-time checks and all: it uses an example of duck typing: if it walks like a duck, and quacks like a duck, then it is a duck.

And here's one final example also from the `Collections` library:

```
static <T> int binarySearch(List<? extends Comparable<? super T>> L, T key) {
... }
```

Here, the items of `L` have to have a type that is comparable to `T` or to some super-type of `T`.

Should `L` be able to contain `key`? It doesn't appear so: because it could be a list of some entirely different type, just so long as that type is declared comparable to `T` or some super-type of it. Sounds pretty weird, and a niche use case that has few (if any) practical benefits, but it is allowed.

Dirty Secrets Behind the Scenes

Java's design for parameterized types was constrained by a desire for backward compatibility: Java had been out for a while by the time version 1.5 rolled around, and the designers did not want the existing code or compilers to be obsolete. Thus, when you write:

```
class Foo<T> {
    T x;
    T mogrify(T y) { ... }
    Foo<Integer> q = new Foo<Integer>();
    Integer r = q.mogrify(s);
}
```

Java is instead doing:

```
class Foo { // This is called the "raw type".
    Object x;
    Object mogrify(Object y) { ... }
    Foo q = new Foo();
    Integer r = (Integer) q.mogrify((Integer) s);
}
```

It supplies the casts automatically, and also throws in some additional checks. If it cannot guarantee that all those casts will work, Java will throw a warning about "unsafe" constructs, which you might have seen in Homework 6.

Limitations

Because of those design choices, there are some limitations to generic programming:

- Since all kinds of `Foo` or `List` are really the same:
  - `L.instanceOf(List<String>)` will be true even when `L` is a `List<Integer>`.
  - Inside the class, you cannot create new instances of, arrays of or check the instance of the generic type `T`:
    - `new T()`
    - `new T[]`
    - and `x.instanceOf(T)` are all not legal

- Primitive types are not allowed as type parameters: we cannot have an `ArrayList<int>`, for example, which is fortunately less of an issue with autoboxing and unboxing. Unfortunately, it has a significant cost on efficiency.

> **Language Design**
>
> Just so you know, C# and Python, which came later than Java, both did not have this problem. They decided to bite the bullet and make the type parameters visible. As a result, that alleviated this particular problem.
>
> For the most part, however, this is not a big deal. You aren't really supposed to use `instanceOf` in significant capacities. The restrictions on creating new instances and arrays of generics are perhaps more serious, but can also be alleviated: instead of `new T[]`, you can say `new Object[]`, and because of the previously discussed problem, will usually work. And instead of `new T()`, you can also use factory methods to produce your objects instead of constructors, which you saw examples of in Project 0, and will see more of in Project 2.