CS61B Lecture #23

Today: Java support for generic programming

Readings for today: *A Java Reference*, Chapter 10.

Readings for Wednesday: *Data Structures*, §6.4.

The Old Days

- Java library types such as `List` didn't used to be parameterized. All `List`s were lists of `Object`s.
- So you'd write things like this:

```
for (int i = 0; i < L.size (); i += 1)
   { String s = (String) L.get (i); ... }
```

- That is, must explicitly cast result of `L.get (i)` to let the compiler know what it is.
- Also, when calling `L.add(x)`, was no check that you put only `String`s into it.
- So, newest release attempts to alleviate these perceived problems by introducing *parameterized types*, like `List<String>`.
- Unfortunately, it is not as simple as one might think.

Basic Parameterization

- From the definitions of `ArrayList` and `Map` in `java.util`:

```
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);
   ...
}
```

- First (blue) occurrences of `Item`, `Key`, and `Value` introduce formal *type parameters*, whose "values" (which are reference types) get substituted for all the other occurrences of `Item`, `Key`, or `Value` when `ArrayList` or `Map` is "called" (as in `ArrayList<String>`, or `ArrayList<int[]>`, or `Map<String, List<Particle>>`).
- 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 write `ArrayList<String>`, we get, in effect, a new type, somewhat like

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

- And in turn, `List<String>` refers to a new type as well.

Parameters on Methods

- Functions (methods) may also be parameterized by type. Example of use from `java.util.Collections`:

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

In this case, compiler figures out T without help when you call `singleton(x)` by looking at the type of `x`. This is a simple example of *type inference*.

- Another example (from `java.util.Collections`):

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

Here, a call to `emptyList()` would *not* contain enough information, so instead we write, e.g., `Collections.<Particle>emptyList()`, to tell the compiler that `T` is `Particle`.

Wildcards

- Consider the definition of something that counts the number of times something occurs in a collection of items. Could write this as

```
/** Number of items in C that are equal to X. */
static <T> int frequency (Collection<T> c, Object x) {
    int n; n = 0;
    for (T y : c) {
        if (x.equals(y)
            n += 1;
    }
    return n;
}
```

- But we don't really care what `T` is; we don't need to declare anything of type `T` in the body, because we could write instead

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

- *Wildcard type parameters* say that you don't care what a type parameter is (i.e., it's any subtype of `Object`):

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


Subtyping (I)

- What are the relationships between the types

```
List<String>,  List<Object>, ArrayList<String>, ArrayList<Object>?
```

- We know that `ArrayList` $\preceq$ `List` and `String` $\preceq$ `Object` (using $\preceq$ for "is a subtype of")...
- ...So is `List<String>` $\preceq$ `List<Object>`?

Subtyping (II)

- Consider this fragment:

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

- So, having `List<String>` $\preceq$ `List<Object>` would violate *type safety:* The compiler is wrong about the type of a value.
- So in general for `T1<X>` $\preceq$ `T2<T>`, must have `X` $=$ `Y`.
- But what about `T1` and `T2`?

Subtyping (III)

- Now consider

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

- In this case, everything's fine:
  - The object's dynamic type is `ArrayList<String>`.
  - Therefore, the methods expected for `LS` must be a subset of those for `ALS`.
  - And since the type parameters are the same, the signatures of those methods will be the same.
  - Therefore, all the 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>` $\preceq$ `T2<X>` if `T1` $\preceq$ `T2`.

A Java Inconsistency: Arrays

- The Java language design is not entirely consistent when it comes to subtyping.
- For the same reason that `ArrayList<String>` $\not\preceq$ `ArrayList<Object>`, you'd also expect that `String[]` $\not\preceq$ `Object[]`.
- And yet, Java *does* make `String[]` $\preceq$ `Object[]`.
- And, just as explained above, one gets into trouble with

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

- So in Java, the Bad line causes an `ArrayStoreExceptipm`.
- Why do it this way? Basically, because otherwise there'd be no way to implement, e.g., `ArrayList`.

Type Bounds (I)

- Sometimes, your program needs to ensure that a particular type parameter is replaced only by a subtype (or supertype) 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() { ... }
    ...
}
```

Requires that all type parameters to `NumbericSet` must be subtypes of `Number` (the "type bound"). `T` can either extend or implement the bound, as appropriate.

Type Bounds (II)

- Another example:

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

means that `L` can be a `List<Q>` for any `Q` as long as `T` is a subtype of (extends or implements) `Q`.

- Why didn't the library designers just define this as

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

?

Type Bounds (III)

And one more:

```
/** Search sorted list L for KEY, returning either its position (if
 *  present), or k-1, where k is where KEY should be inserted.  */
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`'s or some supertype of `T`. Does `L` have to be able to contain the value `key`? Why does this make sense?

Dirty Secrets Behind the Scenes

- Java's design for parameterized types was constrained by a desire for backward compatibility.
- Actually, when you write

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

Java gives really gives you

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

That is, it supplies the casts automatically, and also throws in some additional checks. If it can't guarantee that all those casts will work, gives you a warning about "unsafe" constructs.

Limitations

Because of Java's design choices, are some limitations to generic programming:

- Since all kinds of `Foo` or `List` are really the same,
  - `L instanceof List<String>` will be true when `L` is a `List<Integer>`.
  - Inside, e.g., class `Foo`, you cannot write `new T ()`, `new T[]`, or `x instanceof T`.
- Primitive types are not allowed as type parameters.
  - Can't have `ArrayList<int>`, just `ArrayList<Integer>`.
  - Fortunately, automatic boxing and unboxing makes this substitution easy:

```
int sum (ArrayList<Integer> L) {
    int N;  N = 0;
    for (int x : L) { N += x; }
    return N;
}
```

  - Unfortunately, boxing/unboxing have significant costs.