CS61B Lecture #24

Today: Java support for generic programming

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

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 definition of `ArrayList` in `java.util`:

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

- First occurrence of `Item` introduces a formal *type parameter*, whose "value" (a reference type) in effect gets substituted for all the other occurrences of `Item` when `ArrayList` is "called" (when a programmer writes, e.g., `ArrayList<String>` or `ArrayList<int[]>`).
- Not limited to one parameter:

```
Map<String,Table> database = new HashMap<String,Table>();
```

- Can also say that you don't care what a type parameter is (wildcards):

```
/** Number of items in C that are .equal to X. */
static int frequency (Collection<?> c, Object x) {...}
```

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`.

- 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>emptySet ()`, to tell the compiler that `T` is `Particle`.

Type Bounds

- 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.

- 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>` as long as `T` is a subtype of (extends or implements) `Q`.

Type Bounds (II)

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.

Use in Project #2

- Problem in Project #2 was to allow you to *extend* the information stored in points.
- But at the same time, implementations of Set2D have to know something about Points, too.
- So, we define the minimum that a Point must supply:

```
// Nested in Set2D, for convenience
public static abstract class BasePoint {
   public abstract double x ();
   public abstract double y ();
   etc.
}
```

- Then we say that Set2D works on any kind of Point that subtypes that:

```
public abstract class Set2D<Point extends Set2D.BasePoint> {
    ...
    public abstract boolean contains (Point p);
   etc.
}
```

Use in Project #2 (QuadTree)

- Now we can extend Set2D to a concrete class, QuadTree.
- QuadTree must be free to define its own kind of Point, but again want clients to be able to design more featureful Points.
- So we repeat the same trick:

```
public class QuadTree<Point extends QuadTree.QuadPoint> extends Se
    /** The supertype of all possible kinds of QuadTree member.
     *  Type arguments to QuadTree are subtypes of QuadPoint. */
    public static class QuadPoint extends Set2D.BasePoint {
        public double x () { ... }
        public double y () { ... }
        etc.
    }
   etc.
}
```

Use in Project #2 (QuadTree clients)

- Can build yourself a QuadTree containing just positions:

```
import util.QuadTree;
import util.QuadTree.QuadPoint;
...

   QuadTree<QuadPoint> tree = new QuadTree<QuadPoint> (...);
   ...
```

- Or you can add stuff to Points:

```
class MyPoint extends QuadPoint {
    int id () { ... }
    ...
}
```

```
QuadTree<MyPoint> tree
   = new QuadTree<MyPoint> (...);
```