CS61B Lecture #24

Today:   Java support for generic programming

Readings for today:  A Java Reference, Chapter 10.

Readings for Monday:  Data Structures, §6.4.
The Old Days

• Java library types such as List didn’t used to be parameterized. All Lists were lists of Objects.

• So you’d write things like this:

```java
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 Strings 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`:

```java
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 occurrence of** `Item`, `Key`, and `Value` **introduce formal type parameters**, whose “value” (a reference type) in effect gets 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>>`).

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

  ```java
  /** 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:

  ```java
  /** 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):

  ```java
  /** 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,

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

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

- Another example:

  ```java
  /** 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

```java
class Foo<T> {
    T x;
    T mogrify (T y) { ... }
}
```

Java gives really gives you

```java
class Foo {
    Object x;
    Object mogrify (Object y) { ... }
}
```

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:

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

  - Unfortunately, boxing/unboxing have significant costs.