CS61B Lecture #25: Java Generics
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, starting with 1.5, the designers tried 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 (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

```java
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

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

- And then, likewise, `List<String>` refers to a new interface type as well.
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) {
    ...
}
/** 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

```java
List<String> empty = Collections.emptyList();
```

the parameters obviously don’t suffice, but the compiler deduces the parameter $T$ from context: it must be assignable to String.
Wildcards

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

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

```java
...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):

```java
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 \subseteq List and String \subseteq Object (using \subseteq for “is a subtype of”)... 

• ... So is List<String> \subseteq List<Object>?
Subtyping (II)

- Consider this fragment:

```java
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> \(\leq\) List<Object> would violate type safety: The compiler is wrong about the type of a value.

- So in general for \(T1<X> \leq T2<Y>\), must have \(X = Y\).

- But what about \(T1\) and \(T2\)?
Subtyping (III)

• Now consider

```java
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> \leq T2<X>` if `T1 \leq 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

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

- So in Java, the Bad line causes an `ArrayStoreException`—a (dynamic) runtime error instead of a (static) compile-time error.
- 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,

```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.
Type Bounds (II)

• 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> 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

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

?
Type Bounds (II)

- 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> 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

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

? -

- Consider

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

This would be illegal if L were forced to be a List<String>.
Type Bounds (III)

- And one more:
  ```java
  /** 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 to some supertype of T.

- Does L have to be able to contain the value key?

- Why does this make sense?
Type Bounds (III)

• And one more:

```java
/** 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 to some supertype of T.

• Does L have to be able to contain the value key?

• Why does this make sense?

• As long as the items in L can be compared to key, it doesn't really matter whether they might include key (not that this is often useful).
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 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.

```
  Foo<Integer> q = new Foo<Integer>();
  Integer r = q.mogrify(s);
  ```

```java
  Foo q = new Foo();
  Integer r =
  ```

  (Integer) q.mogrify((Integer) s);
  ```
Limitations

Because of Java’s 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 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 and unboxing have significant costs.