CS61B Lecture #25: Java Generics

The Old Days

- Java library types such as `List` didn't used to be parameteri
  `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 c
  know what it is.
- Also, when calling `L.add(x)`, was no check that you put only
  into it.
- So, starting with 1.5, the designers tried to alleviate the
  ceived problems by introducing *parameterized types*, like `Lis`
- 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 *type parameters*, whose "values" (which are reference typ substituted for all the other occurrences of `Item`, `Key`, o when `ArrayList` or `Map` is "called" (as in `ArrayList<Stri` `ArrayList<int[]>`, or `Map<String, List<Particle>>`).
- Other occurrences of `Item`, `Key`, and `Value` are uses of the types, just like uses of a formal parameter in the body of a f

Type Instantiation

- *Instantiating* a generic type is analogous to calling a functio
- 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 ne somewhat like

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

- And then, likewise, `List<String>` refers to a new interface well.

Parameters on Methods

- Functions (methods) may also be parameterized by type. Exa
  use 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)`
  ing at the type of `x`. This is a simple example of *type infere*

- In the call

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

  the parameters obviously don't suffice, but the compiler
  the parameter `T` from context: it must be assignable to `Lis`

Wildcards

- Consider the definition of something that counts the nu
times something occurs in a collection of items. Could wr
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
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 t
rameter 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>`, `ArrayLis`

- We know that `ArrayList` $\preceq$ `List` and `String` $\preceq$ `Object` (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
String S = LS.get(0);        // OOPS! A.get(0) is
                             // but spec of List<S
                             // says that it is.
```

- So, having `List<String>` $\preceq$ `List<Object>` would violate *type*
  The compiler is wrong about the type of a value.
- So in general for `T1<X>` $\preceq$ `T2<Y>`, 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 su
    those for `ALS`.
  - And since the type parameters are the same, the signat
    those methods will be the same.
  - Therefore, all the legal calls on methods of `LS` (accordin
    compiler) will be valid for the actual object pointed to by
- 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 c subtyping.
- For the same reason that `ArrayList<String>` $\not\preceq$ `ArrayList` 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 `ArrayStoreException`.
- Why do it this way? Basically, because otherwise there'd be to implement, e.g., `ArrayList`.

Type Bounds (I)

- Sometimes, your program needs to ensure that a particular t
rameter is replaced only by a subtype (or supertype) of a pa
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 s
of `Number` (the "type bound"). `T` can either extend or implem
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 sub
  (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 (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 sub
  (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) { ... }
```

  ? -

- Consider

```
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:

```
/** Search sorted list L for KEY, returning either its posi
 *  present), or k-1, where k is where KEY should be insert
static <T> int binarySearch(List<? extends Comparable<? sup
                            T key)
```

- Here, the items of `L` have to have a type that is comparab
  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:

```
/** Search sorted list L for KEY, returning either its posi
 *  present), or k-1, where k is where KEY should be insert
static <T> int binarySearch(List<? extends Comparable<? sup
                            T key)
```

- Here, the items of `L` have to have a type that is comparab
  or to some supertype of `T`.
- Does `L` have to be able to contain the value `key`?
- Why does this make sense?
- Again, we might have

```
static int findX(List<Object> L) {
   return binarySearch(L, "X");
}
```

Dirty Secrets Behind the Scenes

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

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

  Java really gives you

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

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

Limitations

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

- Since all kinds of `Foo` or `List` are really the same,
  - `L instanceof List<String>` will be true when `L` is a `List<`
  - Inside, e.g., class `Foo`, you cannot write `new T()`, `new T` `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 s tion easy:

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

  - Unfortunately, boxing and unboxing have significant cost