61B Lecture #25: Java Generics

The Old Days

types such as `List` didn't used to be parameterized. All
ists of `Object`s.

e things like this:

```
ist L) {
 s; s = "";
nt i = 0; i < L.size(); i += 1)
(s.compareTo((String) L.get(i)) < 0) s = (String) L.get(i);
 s;
```

had to explicitly cast result of `L.get(i)` to let the
w what its static type was.

alling `L.add(x)`, there was no check that you put only
it.

with 1.5, the designers tried to alleviate these per-
ems by introducing *parameterized types*, like `List<String>`.

ly, it is not as simple as one might think.

Basic Parameterization

finitions of `ArrayList` and `Map` in `java.util`:

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


erface Map<Key, Value> {
et(Key x);
```

) occurrences of `Item`, `Key`, and `Value` introduce formal
ters, whose "values" (which are reference types) get
for all the other occurrences of `Item`, `Key`, or `Value`
ist or `Map` is "called" (as in `ArrayList<String>`, or
nt[]>, or `Map<String, List<Particle>>`).

rences of `Item`, `Key`, and `Value` are *uses* of the formal
ke uses of a formal parameter in the body of a function.

Type Instantiation

a generic type is analogous to calling a function.

in

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

ite `ArrayList<String>`, we get, in effect, a new type,
e

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

ewise, `List<String>` refers to a new interface type as

Parameters on Methods

ethods) may also be parameterized by type. Example of
a.util.Collections:

```
-only list containing just ITEM. */
 List<T> singleton(T item) { ... }
difiable empty list. */
 List<T> emptyList() { ... }
```

figures out T in the expression `singleton(x)` by look-
pe of `x`. This is a simple example of *type inference*.

```
g> empty = Collections.emptyList();
```

ers obviously don't suffice, but the compiler deduces
er `T` from context: it must be assignable to `String`.

Wildcards

e definition of something that counts the number of
ning occurs in a collection of items. Could write:

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

n;
```

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

```
bject y : c) {
```

e parameters say that you don't care what a type pa-
e., it's any subtype of `Object`):

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

Subtyping (II)

s fragment:

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

st<String> $\preceq$ List<Object> would violate *type safety:*
is wrong about the type of a value.

l for T1<X> $\preceq$ T2<Y>, must have X $=$ Y.

ut T1 and T2?

Subtyping (I)

e relationships between the types

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

at ArrayList $\preceq$ List and String $\preceq$ Object (using $\preceq$ type of")...

<String> $\preceq$ List<Object>?

A Java Inconsistency: Arrays

guage design is not entirely consistent when it comes to

reason that ArrayList<String> $\not\preceq$ ArrayList<Object>,
pect that String[] $\not\preceq$ Object[].

a *does* make String[] $\preceq$ Object[].

explained above, one gets into trouble with

```
S = new String[3];
Obj = AS;
new int[] { 1, 2 };  // Bad
```

the Bad line causes an ArrayStoreException—a (dy-
ne error instead of a (static) compile-time error.

is way? Basically, because otherwise there'd be no way
, e.g., ArrayList.

Subtyping (III)

r

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

everything's fine:

t's dynamic type is ArrayList<String>.

e, the methods expected for LS must be a subset of
ALS.

the type parameters are the same, the signatures of
hods will be the same.

e, all the legal calls on methods of LS (according to the
will be valid for the actual object pointed to by LS.

1<X> $\preceq$ T2<X> if T1 $\preceq$ T2.

Type Bounds (II)

mple:

```
e elements of SRC into DEST. */
void copy(List<? super T> dest, List<T> src) { ... }
```

est can be a List<Q> for any Q as long as T is a subtype
or implements) Q.

t define this as

```
e elements of SRC into DEST. */
void copy(List<T> dest, List<T> src) { ... }
```


Type Bounds (I)

your program needs to ensure that a particular type pa-
eplaced only by a subtype (or supertype) of a particular
like specifying the "type of a type.").

```
ricSet<T extends Number> extends HashSet<T> {
minimal element */
) { ... }
```

t all type parameters to NumbericSet must be subtypes
he "type bound"). T can either extend or implement the
propriate.

Type Bounds (III)

e:

```
 sorted list L for KEY, returning either its position (if
t), or k-1, where k is where KEY should be inserted.  */
 int binarySearch(List<? extends Comparable<? super T>> L,
                  T key)
```

ems of L have to have a type that is comparable to T's upertype of T.

to be able to contain the value key?

is make sense?

Type Bounds (II)

mple:

```
he elements of SRC into DEST. */
 void copy(List<? super T> dest, List<T> src) { ... }
```

lest can be a List<Q> for any Q as long as T is a subtype or implements) Q.

t define this as

```
he elements of SRC into DEST. */
 void copy(List<T> dest, List<T> src) { ... }
```

e perfect sense to copy a List<String> into a List<Object>, ld be disallowed with this declaration.

Type Bounds (III)

e:

```
 sorted list L for KEY, returning either its position (if
t), or k-1, where k is where KEY should be inserted.  */
 int binarySearch(List<? extends Comparable<? super T>> L,
                  T key)
```

ems of L have to have a type that is comparable to T's upertype of T.

to be able to contain the value key?

is make sense?

he items in L can be compared to key, it doesn't really her they might include key (not that this is often use-

irty Secrets Behind the Scenes

n for parameterized types was constrained by a desire d compatibility.

en you write

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

ives you

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

pplies the casts automatically, and also throws in some ecks. If it can't guarantee that all those casts will work, arning about "unsafe" constructs.

Limitations

's design choices, there are some limitations to generic

ds of Foo or List are really the same,

ceof List<String> will be true when L is a List<Integer>.

g., class Foo, you cannot write new T(), new T[], or x of T.

es are not allowed as type parameters.

ArrayList<int>, just ArrayList<Integer>.

ly, automatic boxing and unboxing makes this substitu-

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

ately, boxing and unboxing have significant costs.