Java Generics

Lecture 17
CS2110 – Spring 2017

Textbook and Homework

Generics: Appendix B
Generic types we discussed: Chapters 1-3, 15
Useful tutorial:
docs.oracle.com/javase/tutorial/extra/generics/index.html

How to think about/implement sharedAncestorOf

B.sharedAncestorOf(D, G): B
A.sharedAncestorOf(F, G): F
A.sharedAncestorOf(D, I): B

Use ParentOf for the last one?

A.getParentOf(I) to get H.
Then A.getParentOf(H) to get G.
Then ...
Searching the tree over and over and over!
Terrible!

How about getting the routes to D and I?

Route to D:
Route to I:

These two lists have at least one node in common:
A. They may have more. In this case, B is also in both lists. The last one that is in both lists is the shared ancestor!

How to think about/implement sharedAncestorOf

We have these two linked lists. They are the same for a number of nodes in the beginning and then perhaps they diverge. They may both continue, or one or both might end there.

The shared ancestor is the last one that is on both lists!

How to write a loop that starts at the beginning and moves through both lists in synched fashion?

Can’t use foreach.
Don’t want to use stuff to right because we don’t know cost of c1.get(i)

Use the iterators of c1 and c2!
We do it in Eclipse

Moral: Spend time with pencil and paper away from Eclipse to think problem through!
Early versions of Java lacked generics...

```java
interface Collection {
    /** Return true iff the collection contains o */
    boolean contains(Object o);
    /** Add ob to the collection; return true iff
     * the collection is changed. */
    boolean add(Object ob);
    /** Remove ob from the collection; return true iff
     * the collection is changed. */
    boolean remove(Object ob);
    ...
}
```

Lack of generics was painful because programmers had to manually cast.

```java
Collection c = ...
c.add("Hello");
c.add("World");
...
for (Object ob : c) {
    String s = (String) ob;
    System.out.println(s.length + " : " + s.length());
}
```

... and people often made mistakes!

Limitation seemed especially awkward because built-in arrays do not have the same problem!

```java
String [] a = ...
a[0]= "Hello"
...
for (String s : a) {
    System.out.println(s);
}
```

We should be able to do the same thing with object types generated by classes!

In late 1990s, Sun Microsystems initiated a design process to add generics to the language ...

```
PolyJ  Pizza/GJ  LOQI
```

With generics, the Collection interface becomes...

```java
interface Collection<T> {
    /** Return true iff the collection contains x */
    boolean contains(T x);
    /** Add x to the collection; return true iff
     * the collection is changed. */
    boolean add(T x);
    /** Remove x from the collection; return true iff
     * the collection is changed. */
    boolean remove(T x);
    ...
}
```
Using Java Collections

With generics, no casts are needed...

```
Collection<String> c = ... 
c.add("Hello"); 
... 
for (String s : c) {
    System.out.println(s.length + " : " + s.length());
}
```

... and mistakes (usually) get caught!

Type checking as part of syntax check (compile time)

The compiler can automatically detect uses of collections with incorrect types...

```
Collection<String> c = ... 
c.add("Hello") /* Okay */
c.add(1979); /* Illegal: static error */
```

Generally speaking,
Collection<String>
behaves like the parameterized type
Collection<T>
where all occurrences of T have been replaced by String.

Subtyping

Subtyping extends naturally to generic types.

```
interface Collection<T> { ... }
interface List<T> extends Collection<T> { ... }
class LinkedList<T> implements List<T> { ... }
class ArrayList<T> implements List<T> { ... }
/* The following statements are all legal. */
ArrayList<String> a = new ArrayList<String>();
Collection<String> c = a;
l = a;
c = l;
```

String is a subtype of object so...

... is LinkedList<String> a subtype of LinkedList<Object>?

```
LinkedList<String> ls = new LinkedList<String>();
LinkedList<Object> lo = new LinkedList<Object>();
lo = ls;  //Suppose this is legal
lo.add(2110);  //Type-checks: Integer subtype Object
String s = ls.get(0);  //Type-checks: is is a List<String>
```

But what would happen at run-time if we were able to actually execute this code?

Array Subtyping

Java’s type system allows the analogous rule for arrays:

```
String[] as = new String[10];
Object[] ao = new Object[10];
ao[0] = 2110;  //Type-checks: Integer subtype Object
String s = ao[0];  //Type-checks: as is a String array
```

What happens when this code is run? TRY IT OUT!

It throws an ArrayStoreException! Because arrays are built into Java right from beginning, it could be defined to detect such errors

A type parameter for a method

```
/** Replace all values x in list ts by y. */
public void replaceAll(List<Double> ts, Double x, Double y) {
    for (int i = 0; i < ts.size(); i++)
        if (Objects.equals(ts.get(i), x))
            ts.set(i, y);
}
```

We would like to rewrite the parameter declarations so this method can be used for ANY list, no matter the type of its elements.
Try replacing `Double` by some “Type parameter” `T`, and Java will still complain that type `T` is unknown.

```java
/** Replace all values x in list ts by y. */
public void replaceAll(List<Double> ts, Double x, Double y) {
  for (int i = 0; i < ts.size(); i++)
    if (Objects.equals(ts.get(i), x))
      ts.set(i, y);
}
```

Somehow, Java must be told that `T` is a type parameter and not a real type. Next slide says how to do this.

### A type parameter for a method

Placing `<T>` after the access modifier indicates that `T` is to be considered as a type parameter, to be replaced when method is called.

```java
/** Replace all values x in list ts by y. */
public <T> void replaceAll(List<T> ts, T x, T y) {
  for (int i = 0; i < ts.size(); i++)
    if (Objects.equals(ts.get(i), x))
      ts.set(i, y);
}
```

### Printing Collections

Suppose we want to write a method to print every value in a `Collection<T>`.

```java
void print(Collection<Object> c) {
  for (Object x : c) {
    System.out.println(x);
  }
}
```

```java
... Collection<Integer> c = ...
c.add(42);
print(c); /* Illegal: Collection<Integer> is not a * subtype of Collection<Object> */
```

### Wildcards: introduce wildcards

To get around this problem, Java’s designers added wildcards to the language.

```java
void print(Collection<?> c) {
  for (Object x : c) {
    System.out.println(x);
  }
}
```

```java
... Collection<Integer> c = ...
c.add(42);
print(c); /* Legal */
```

One can think of `Collection<?>` as a “Collection of some unknown type of values”.

### Wildcards

We can’t add values to collections whose types are wildcards...

```java
void doIt(Collection<?> c) {
  c.add(42); /* Illegal */
}
```

```java
... Collection<String> c = ...
doIt(c); /* Legal */
```

42 can be added to:
- `Collection<Integer>
- `Collection<Number>
- `Collection<Object`

but `c` could be a `Collection` of anything, not just supertypes of `Integer`.

### Bounded Wildcards

Sometimes it is useful to have some information about a wildcard. Can do this by adding bounds...

```java
void doIt(Collection<? super Integer> c) {
  c.add(42); /* Legal */
}
```

```java
... Collection<Object> c = ...
doIt(c); /* Illegal */
```

```
/* Illegal */
```

Now `c` can only be a `Collection` of some supertype of `Integer`, and 42 can be added to any such `Collection`.

“`<? super`” is useful when you are only giving values to the object, such as putting values into a `Collection`
**Bounded Wildcards**

"? extends" is useful for when you are only receiving values from the object, such as getting values out of a Collection.

```java
void doIt(Collection<? extends Shape> c) {
    for (Shape s : c)
        s.draw();
}
```

**Bounded Wildcards**

Wildcards can be nested. The following receives Collections from an Iterable and then gives floats to those Collections.

```java
void doIt(Iterable<? extends Collection<? super Float>> cs) {
    for (Collection<? super Float> c : cs)
        c.add(0.0f);
}
```

We skip over this in lecture. Far too intricate for everyone to understand. We won’t quiz you on this.

**Generic Methods**

Here’s the printing example again. Written with a method type-parameter.

```java
<T> void print(Collection<T> c) {
    // T is a type parameter,
    for (T x : c)
        System.out.println(x);
}
```

But wildcards are preferred when just as expressive.

**Catenating Lists**

Suppose we want to catenate a list of lists into one list. We want the return type to depend on what the input type is.

```java
List<List<Integer>> ls = ...
List<Integer> list = flatten(ls);
```

The return type depends on what the input type is.

```java
/**
 * Return the flattened version of ls.
 */
<T> List<T> flatten(List<? extends List<? extends T>> ls) {
    List<T> flat = new ArrayList<T>();
    for (List<T> l : ls)
        flat.addAll(l);
    return flat;
}
```

**Interface Comparable**

Interface Comparable<T> declares a method for comparing one object to another.

```java
interface Comparable<T> {
    /* Return a negative number, 0, or positive number */
    int compareTo(T that);
}
```

Integer, Double, Character, and String are all Comparable with themselves.
Our binary search

Type parameter: anything \( T \) that implements \( \text{Comparable}<T> \)

```java
/**
 * Return \( h \) such that \( c[0..h] \leq x < c[h+1..] \).
 * Precondition: \( c \) is sorted according to .. */
public static <T extends Comparable<T>>
    int indexOf(List<T> c, T x) {
    int h = -1;
    int t = c.size();
    // inv: \( h < t \land c[0..h] \leq x < c[t..] \)
    while (h + 1 < t) {
        int e = (h + t) / 2;
        if (c.get(e).compareTo(x) <= 0)
            h = e;
        else t = e;
    }
    return h;
}
```

Those who fully Grok generics write:

Type parameter: anything \( T \) that implements \( \text{Comparable}<T> \)

```java
/**
 * Return \( h \) such that \( c[0..h] \leq x < c[h+1..] \).
 * Precondition: \( c \) is sorted according to .. */
public static <T extends Comparable<? super T>>
    int indexOf(List<T> c, T x) {
    int h = -1;
    int t = c.size();
    // inv: \( h < t \land c[0..h] \leq x < c[t..] \)
    while (h + 1 < t) {
        int e = (h + t) / 2;
        if (c.get(e).compareTo(x) <= 0)
            h = e;
        else t = e;
    }
    return h;
}
```

Anything that is a superclass of \( T \).

Don’t be concerned with this!
You don’t have to fully understand this.