Photo credit: Andrew Kennedy

JAVA GENERICS

Lecture 17
CS2110 – Spring 2017

Textbook and Homework

2

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

3

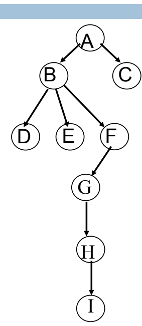
B.sharedAncestorOf(D, G): B

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



How to think about/implement sharedAncestorOf

4

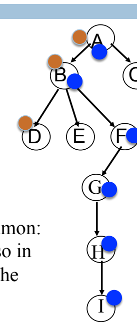
A.sharedAncestorOf(D, I): B

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!

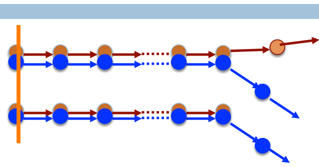


How to think about/implement sharedAncestorOf

5

case 1

case 2



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.

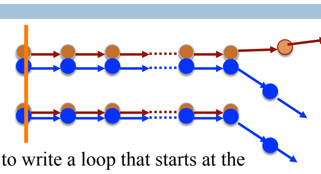
The shared ancestor is the last one that is on both lists!

How to think about/implement sharedAncestorOf

6

c1
c2

c1
c2



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

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)

```
i= 0;
while ( …) {
  n1= c1.get(i);
  n2= c2.get(i);
  …
  i= i+1;
```

Moral: Spend time with pencil and paper away from Eclipse to think problem through!

Java Collections

Early versions of Java lacked generics…

```
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);
  ...
}
```

Java Collections

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

```
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!

Using Java Collections

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

```
String [] a = ...
a[0]= ("Hello")
a[1]= ("World");
...
for (String s : a) {
  System.out.println(s);
}
```

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

Arrays → Generics

One can think of the array "brackets" as a kind of *parameterized* type: a type-level function that takes one type as input and yields another type as output

```
Object[] a= ...
String[] a= ...
Integer[] a= ...
Button[] a= ...
```

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

Proposals for adding Generics to Java

PolyJ Pizza/GJ LOOJ

Generic Collections

With generics, the Collection interface becomes...

```
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

13

With generics, no casts are needed...

```
Collection<String> c= ...
c.add("Hello")
c.add("World");
...
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)

14

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 byString.

Subtyping

15

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. */
List<String> l= new LinkedList<String>();
ArrayList<String> a= new ArrayList<String>();
Collection<String> c= a;
l= a
c= l;
```

Subtyping

16

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: ls is a List<String>
```

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

Array Subtyping

17

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

```
String[] as= new String[10];
Object[] ao= new Object[10];

ao= as;          //Type-checks: considered outdated design
ao[0]= 2110;     //Type-checks: Integer subtype Object
String s= as[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

18

```
/** 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= i+1)
     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.

A type parameter for a method

19

Try replacing Double by some "Type parameter" T, and Java will still complain that type T is unknown.

```
/** Replace all values x in list ts by y. */
                             T            T         T
public void replaceAll(List<Double> ts, Double x, Double y) {
  for (int i= 0; i < ts.size(); i= i+1)
     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

20

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

```
/** 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= i+1)
     if (Objects.equals(ts.get(i), x))
        ts.set(i, y);
}
```

Printing Collections

21

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

```
void print(Collection<Object> c) {
  for (Object x : c) {
    System.out.println(x);
  }
}
...
Collection<Integer> c= ...
c.add(42);
print(c);  /* Illegal: Collection<Integer> is not a
            * subtype of Collection<Object>! */
```

Wildcards: introduce wildcards

22

To get around this problem, Java's designers added *wildcards* to the language

```
void print(Collection<?> c) {
  for (Object x : c) {
    System.out.println(x);
  }
}
...
Collection<Integer> c= ...
c.add(42);
print(c);   /* Legal! */
```

One can think of Collection<?> as a "Collection of *some* unknown type of values".

Wildcards

23

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

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

24

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

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

25

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

```
void doIt(Collection<? extends Shape> c) {
  for (Shape s : c)
     s.draw();
}
...
Collection<Circle> c= ...
doIt(c);  /* Legal! */
Collection<Object> c= ...
doIt(c);  /* Illegal! */
```

Bounded Wildcards

26

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

```
void doIt(Iterable<? extends Collection<? super Float>> cs) {
  for(Collection<? super Float> c : cs)
    c.add(0.0f);
}
...
List<Set<Float>> l= ...
doIt(l);  /* Legal! */
Collection<List<Number>> c= ...
doIt(c);  /* Legal! */
Iterable<Iterable<Float>> i= ...
doIt(i);  /* Illegal! */
ArrayList<? extends Set<? super Number>> a= ...
doIt(a);  /* Legal! */
```

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

Generic Methods

27

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

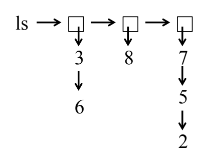
```
<T> void print(Collection<T> c) {// T is a type parameter
  for (T x : c) {
    System.out.println(x);
  }
}
...
Collection<Integer> c= ...
c.add(42);
print(c);  /* More explicitly: this.<Integer>print(c) */
```

But wildcards are preferred when just as expressive.

Catenating Lists

28

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.



Return this list

→ 3 → 6 → 8 → 7 → 5 → 2

Catenating Lists

29

The return type depends on what the input type is.

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

Interface Comparable

30

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

```
interface Comparable<T> {
   /* Return a negative number, 0, or positive number
    * depending on whether this is less than,
    * equal to, or greater than that */
  int compareTo(T that);
}
```

Integer, Double, Character, and String are all Comparable with themselves

Our binary bearch

31

Type parameter: anything T that implements Comparable<T>

```
/** Return h such that c[0..h] <= x < c[h+1..].
  * Precondition: c is sorted according to .. */
public static <T extends Comparable<T>>
                    int indexOf1(List<T> c, T x) {
   int h= -1;
   int t= c.size();
   // inv: h < t  &&  c[0..h] <= 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:

32

Type parameter: anything T that implements Comparable<T>

```
/** Return h such that c[0..h] <= x < c[h+1..].
  * Precondition: c is sorted according to .. */
public static <T extends Comparable<? super T>>
                  int indexOf1(List<T> c, T x) {
   int h= -1;
   int t= c.size();
   // inv: h < t  &&  c[0..h] <= 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.