



Photo credit: Andrew Kennedy

JAVA GENERICS

Lecture 17
CS2110 – Spring 2017

Java Collections

2

Early versions of Java lacked generics...

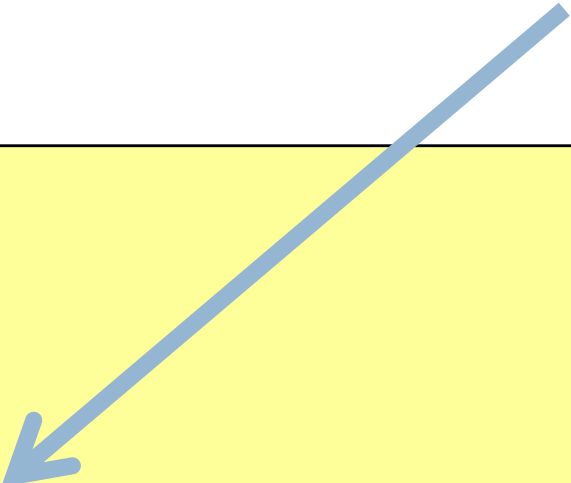
```
interface Collection {  
    /** Return true iff the collection contains ob */  
    boolean contains(Object ob);  
  
    /** Add ob to the collection; return true iff  
     * the collection is changed. */  
    boolean add(Object ob);  
  
    /** Remove ob from the collection; return true iff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

Java Collections

3

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 + " : " + s.length());
}
```



... and people often made mistakes!

Using Java Collections

4

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

5

Array of Strings, ArrayList of strings ---same concept **with a different syntax**

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

```
Object[] oa= ... // array of Objects
String[] sa= ... // array of Strings
ArrayList<Object> oA= ... // ArrayList of Objects
ArrayList<String> oA= ... // ArrayList of Strings
```

Proposals for adding Generics to Java

6

Andrew Meyers

Turing Award winner Barbara Liskov

Nate Foster



PolyJ

Pizza/GJ

LOOJ

Generic Collections

7

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

8

With generics, no casts are needed...

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

... and mistakes (usually) get caught!

Type checking (at compile time)

9

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

```
// This is Demo0  
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

10

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;
```

Array Subtyping

11

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

```
// This is Demo1
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

Array Subtyping

12

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

```
// This is Demo1
String[] as= new String[10];
Object[] ao= new Object[10];

ao= as;
ao[0]= 2110;
String s= as[0];
```

Is this legal? TRY IT OUT!

Subtyping

13

String[] is a subtype of Object[]

...is `ArrayList<String>` a subtype of `ArrayList<Object>`?

```
// This is Demo1
ArrayList<String> ls= new ArrayList<String>();
ArrayList<Object> lo= new ArrayList<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>
```

TRY IT OUT!

The answer is NO. `ArrayList<String>` is **NOT** a subtype of `ArrayList<Object>`

A type parameter for a method

14

Demo 2

```
/** 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

15

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. */
public void replaceAll(List<DoubleT> ts, DoubleT x, DoubleT 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

16

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

17

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

18

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

19

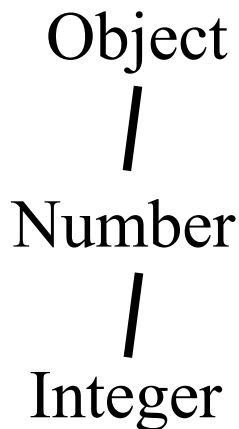
We can't add values to collections whose types are wildcards ...

```
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 Collection of anything, not just supertypes of Integer



How to say that ? Can be a supertype of Integer?

Bounded Wildcards

20

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 `Integer` or 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

21

“**? extends**” is useful 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! */
```

Object

Shape

Rectangle

Square

Bounded Wildcards

22

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

23

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

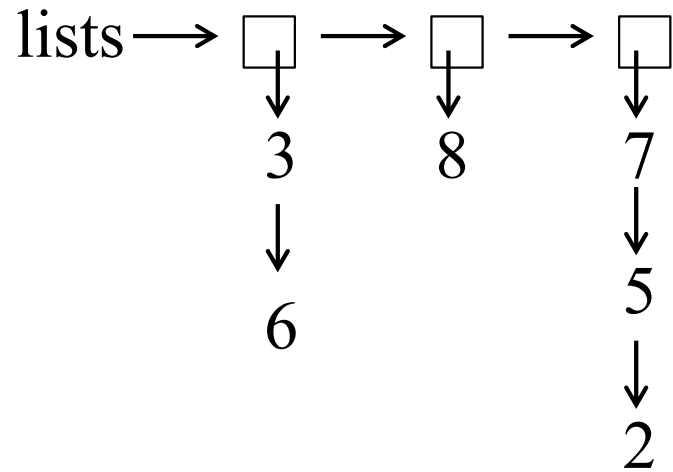
```
<T> void print(Collection<T> c) { // T is a type parameter
    for (T x : c) {
        System.out.println(x);
    }
}
...
Collection<Integer> c = ...
c.add(42);
print(c); /* More explicitly: this.<Integer>print(c) */
```

But wildcards are preferred when just as expressive.

Concatenating Lists

24

Suppose we want to concatenate 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

Concatenating Lists

25

The return type depends on what the input type is.

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

26

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 search

27

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:

28

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>>
    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.