Photo credit: Andrew Kennedy

JAVA GENERICS

Lecture 14
CS2110 – Fall 2016

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

Java Collections

3

Early versions of Java lacked generics…

```
interface Collection {
  /* Return true if the collection contains o */
  boolean contains(Object o);

  /* Add o to the collection; return true if
   *the collection is changed. */
  boolean add(Object o);

  /* Remove o fromthe collection; return true if
   * the collection is changed. */
  boolean remove(Object o);
  ...
}
```

Java Collections

4

The lack of generics was painful when using collections, because programmers had to insert manual casts into their code...

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

… and people often made mistakes!

Using Java Collections

5

This limitation was 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);
}
```

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

Arrays → Generics

6

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

7

PolyJ Pizza/GJ LOOJ

Generic Collections

8

With generics, the Collection interface becomes...

```
interface Collection<T> {
  /* Return true if the collection contains x */
  boolean contains(T x);

  /* Add x to the collection; return true if
   *the collection is changed. */
  boolean add(T x);

  /* Remove x fromthe collection; return true if
   * the collection is changed. */
  boolean remove(T x);
  ...
}
```

Using Java Collections

9

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!

Static Type checking

10

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 substituted with String.

Subtyping

11

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

12

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>
//UH OH: What does s point to, and what is its type?!?!?!
```

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

Array Subtyping

13

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?

It throws an ArrayStoreException!

Printing Collections

14

Suppose we want to write a helper 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

15

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

16

Note that we cannot 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

17

Sometimes it is useful to know *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 for when you are only *giving* values to the object, such as putting values into a Collection

Bounded Wildcards

18

"? 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

19

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! */
```

Generic Methods

20

Returning to the printing example, another option would be to use a method-level 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

21

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

```
<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);
```

Replacing Elements

22

Suppose we need two parameters to have similar types.

```
<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);
}
```

Note that we are both *receiving* values from ts and *giving* values to ts, so we can't use a wildcard.

Interface Comparable

23

The Comparable<T> interface 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

Binary Search

24

Suppose we want to look up a value in a sorted list

```
<T extends Comparable<? super T>>//bounded type parameter
int indexOf(List<T> sorted, T x) { // no null values
  int min = 0;
  int max = l.size();
  while (min < max) {
     int guess = (min + max) / 2;
     int comparison = x.compareTo(l.get(guess));
     if (comparison < 0)
        max = guess;
     else if (comparison == 0)
        return guess;
     else
        min = guess + 1;
  return -1;
}
```