GENERICS AND THE JAVA COLLECTIONS FRAMEWORK

Lecture 16
CS2110 – Fall 2015
Textbook and Homework

Generics: Appendix B
Generic types we discussed: Chapters 1-3, 15
Useful tutorial:
docs.oracle.com/javase/tutorial/extras/generics/index.html
Early versions of Java lacked generics...

```java
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 from the collection; return true if
     * the collection is changed. */
    boolean remove(Object o);
    ...
}
```
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());
}
```
Using Java Collections

This limitation was especially awkward because built-in arrays do not have the same problem!

```java
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

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
With generics, the Collection interface becomes...

```java
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 from the collection; return true if * the collection is changed. */
    boolean remove(T x);
    ...
}
```
Using Java Collections

With generics, no casts are needed...

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

**Terminology:** a type like `Collection<String>` is called an *instantiation* of the *parameterized type* `Collection`. 
Static Type checking

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, an instantiation like `Collection<String>` behaves like the parameterized type `Collection<T>` where all occurrences of `T` have been substituted with `String`. 
Subtyping extends naturally to generic types.

```java
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

String is a subtype of object so...

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

```java
LinkedList<String> ls = new LinkedList<String>();
LinkedList<Object> lo = new LinkedList<Object>();

lo = ls; //OK, if subtypes
lo.add(2110); //OK: Integer subtype Object
String s = ls.last(); //OK: elements of ls are strings
```

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 :-/

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

ao = as;                  // OK, if subtypes
ao[0] = 2110;            // OK: Integer subtype Object
String s = as[0];        // OK: elements of s are strings
```

What happens when this code is run?

It throws an ArrayStoreException!
Suppose we want to write a helper 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

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

One can think of Collection<?> as a “Collection of unknown” values.
Wildcards

Note that we cannot add values to collections whose types are wildcards...

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

More generally, can’t use any methods of Collection<T> where the T occurs in a “negative” position, like a parameter.
Sometimes it is useful to know some information about a wildcard. Can do this by adding bounds...

```java
void doIt(Collection<? extends Shape> c) {
    c.draw(this);
}
...
Collection<Circle> c = ...
doIt(c); /* Legal! */
```
Bounded Wildcards

Sometimes it is useful to know some information about a wildcard. Can do using bounds...

```java
void doIt(Collection<? extends Collection<?>> c) {
    for(Collection<?> ci : c) {
        for(Object x : ci) {
            System.out.println(x);
        }
    }
}
...
Collection<String> ci = ...
Collection.Collection<String>> c = ...
c.add(ci);
doit(c); /* Legal! */
```
Returning to the printing example, another option would be to use a method-level type parameter...

```java
<T> void print(Collection<T> c) {
    for (T x : c) {
        System.out.println(x);
    }
}
...
Collection<Integer> c = ...
c.add(42);
c.add(42);
print(c)  /* More explicitly: this.<Integer>print(c) */
```
Appending an Array

Suppose we want to write a method to append each element of an array to a collection.

```java
<T> void m(T[] a, LinkedList<T> l) {
    for (int i = 0; i < a.length; i++) {
        l.add(a[i]);
    }
}
...
List<Integer> c = ...
Integer[] a = ...
m(a, l);
```
Printing with Cutoff

Suppose we want to print all elements that are “less than” a given element, generically.

```java
<T> void printLessThan(Collection<T> c, T x) {
    for (T y : c) {
        if ( /* y <= x ??? */ )
            System.out.println(y);
    }
}
```
Interface Comparable

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

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

Suppose we want to print all elements that are “less than” a given element, generically.

```java
<T extends Comparable<T>>
    void printLessThan(Collection<T> c, T x) {
        for (T y : c) {
            if (y.compareTo(x) <= 0)
                System.out.println(y);
        }
    }
```
Iterators: How “foreach” works

The notation `for(Something var: collection) { ... }` is syntactic sugar. It compiles into this “old code”:

```java
Iterator<E> _i= collection.iterator();
while (_i.hasNext()) {
    E var= _i.Next();
    . . . Your code . . .
}
```

The two ways of doing this are identical but the foreach loop is nicer looking.

You can create your own iterable collections
java.util.Iterator<E> (an interface)

public boolean hasNext();
- Return true if the enumeration has more elements

public E next();
- Return the next element of the enumeration
- Throw NoSuchElementException if no next element

public void remove();
- Remove most recently returned element by next() from the underlying collection
- Throw IllegalStateException if next() not yet called or if remove() already called since last next()
- Throw UnsupportedOperationException if remove() not supported
Efficiency Depends on Implementation

- Object `x = list.get(k);`
  - \(O(1)\) time for `ArrayList`
  - \(O(k)\) time for `LinkedList`

- `list.remove(0);`
  - \(O(n)\) time for `ArrayList`
  - \(O(1)\) time for `LinkedList`

- `if (set.contains(x)) ...`
  - \(O(1)\) expected time for `HashSet`
  - \(O(\log n)\) for `TreeSet`