CSCI 241
Flipped Lecture 13a
The Set ADT
A Very Brief Intro to Generics
Goals

- Know why Java has generics, and how to use and implement generic classes.
- Know how the Comparable interface works.
Set ADT

/** A collection that contains no duplicate * elements. */

interface Set {
    /** Return true if the set contains ob */
    boolean contains(Object ob);

    /** Add ob to the set; return true iff
    * the collection is changed. */
    boolean add(Object ob);

    /** Remove ob from the set; return true iff
    * the collection is changed. */
    boolean remove(Object ob);
    ...
}
Before Generics

/** A collection that contains no duplicate elements. */
interface Set {
    /** 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);
    ...
}

Can contain anything that extends Object (any class at all)
• But not primitive types: int, double, float, boolean, ...
The Problem

Set c = ...
c.add("Hello")
c.add("World");
...
for (Object ob : c) {
    String s = (String) ob;
    // do things with s
}

Notice: Arrays don’t have this problem!

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

Object[] oa = ... // array of Objects
String[] sa = ... // array of Strings
ArrayList<Object> oA = ... // ArrayList of Objects
ArrayList<String> oA = ... // ArrayList of Strings
The Solution: Generics

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

Now the Set interface written like this:
The Solution: Generics

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

Now the Set interface written like this:

```java
interface Set<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);
    ...
}
```
The Solution: Generics

The Set interface is now written like this:

```java
interface Set<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);
    ...
}
```
The Solution: Generics

The Set interface is now written like this:

```java
interface Set<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);
    ...
}
```

Key idea: I don’t need to know what T is to implement these!
The Solution: Generics

Key idea: I don’t need to know what T is to implement these!

Set<String> c = ...
c.add("Hello")    /* Okay */
c.add(1979);      /* Illegal: compile error! */

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

The bummer: T must extend Object - no primitive types. Can’t do:

```java
Collection<int> c = ...;
```

Have to use:

```java
Collection<Integer>
```

Java often seamlessly converts int to Integer and back.

```java
Integer x = 5;  // works
```

```java
int x = new Integer(5);  // works
```

“Autoboxing/unboxing”
Now you get it:

Cup<T>
Set ADT

/** A collection that contains no duplicate elements. */

interface Set<T> {

    /** Return true if the set contains ob */
    boolean contains(T ob);

    /** Add ob to the set; return true iff the collection is changed. */
    boolean add(T ob);

    /** Remove ob from the set; return true iff the collection is changed. */
    boolean remove(T ob);

    ...
}

class ArraySet<T> implements Set<T> {
    T[] a;
    int size;
    /** Return true iff the collection contains x */
    boolean contains(T x) {
        for (int i = 0; i < size; i++) {
            if (a[i].equals(x)) {
                return true;
            }
        }
        return false;
    }
    return false;
}/** Add x to the collection; return true iff
 * the collection is changed. */
boolean add(T x) {
    if (!contains(x)) {
        a[size] = x; // let’s hope a is big enough...
        size++;
        return true;
    }
    return false;
}
Inner Classes

• We often use an *inner class* to store Node objects of trees, graphs, lists, etc.

• Example: a Node class defined inside the LinkedList class, only used within the class.

```java
public class LinkedList {
    private Node head;

    public class Node {
        private int value;
        private Node next;
    }

    // methods, etc
}
```
The Comparable interface
is (almost) defined like this:

```java
interface Comparable {
    int compareTo(Object o);
}
```
The Comparable interface

```java
class Student implements Comparable {
    // fields and methods
    public int compareTo(Object o) {
        // compare students by last
        // then first name
    }
}

Need to sort students? Built-in Arrays.sort() calls compareTo instead of using <: 

Student[] my241Section = [...] 
Arrays.sort(my241Section);
The Comparable interface

class Orange implements Comparable;
class Apple implements Comparable;
Orange o = new Orange();
Apple a = new Apple();

We can compare apples to oranges!

a.compareTo(o);
The Comparable interface

is (actually) defined like this:

```java
interface Comparable<T> {
    int compareTo(T o);
}
```
The Comparable Interface

```java
interface Comparable<T> {
    int compareTo(T o);
}
```

class Orange implements Comparable<Orange>;
class Apple implements Comparable<Apple>;
Orange o = new Orange();
Apple a = new Apple();
a.compareTo(o);

Won’t compile because Apple doesn’t have:
    compareTo(Orange o)
It only has:
    compareTo(Apple o)
Fancier Generics

What if I care a little bit what T is?

SortableCollection<String> c = ...
c.sort(); ← requires T to be Comparable!
What if I care a little bit what T is?

SortableCollection<String> c = ...  
c.sort(); ← requires T to be Comparable<T>!

interface SortableCollection<T extends Comparable<T>>
{
    ...
}

Fancier Generics