Announcements

- Attendance for this week’s recitation is mandatory!
- A2 is due Wednesday
- Get started on A3 – do one method a day
Overview

- Big Demo!
- Interfaces
- Abstract Classes
- Normal Classes vs. Abstract Classes vs. Interfaces
Interfaces

/** A mutable collection of E values */
public interface Collection<E> {
    /** Return true if this collection contains elem */
    boolean contains(Object elem);

    /** Ensure that this contains elem.
     * Return true if the collection is changed by this. */
    boolean add(E elem);

    ...}

New keyword

Always public

Interfaces cannot by newed!

No fields! No constructors!

No implementations!!!
Implementing Interfaces

Classes implement interfaces

/** A range of integers that always includes 0 */
public class IntRange implements Collection<Integer> {
    private int min = 0; // Represents the range min..max
    private int max = 0; // min <= max
    /** Return true if elem is an integer in the range. */
    public boolean contains(Object elem) {...}
    /** Minimally extend the range to include elem.
     * Return true if the range had to be extended. */
    public boolean add(Integer elem) {...}
}
/** Returns whether the collection contains every integer between and including min and max. * Precondition: ints is not null */

public static boolean containsRange(
    Collection<Integer> ints,
    int min, int max) {
    for (int i = min; i <= max; i++)
        if (!ints.contains(i))
            return false;
    return true;
}

Because ints has type Collection<Integer>, you can use any method declared in the Collection<Integer> interface.

Interfaces are types

Works on any Collection!!! This includes your own DLinkedList from A3!
Extending Interfaces

/** A mutable indexed list of E values */
public interface List<E> extends Collection<E> {
    int size(); // return size of the list
    E get(int index); // return elem at index
    E set(int index, E elem); // change elem at index
    boolean add(int index, E elem); // insert elem at index
    E remove(int index); // remove and return elem at index
    ...
}

Implicitly includes all methods in Collection<E>
### Abstract Classes

Abstract classes cannot be *newed*

Indicates that subclasses are responsible for providing the implementation

Only abstract classes can have abstract methods

```java
/**
 * Provides default implementations for list methods */
 public abstract class AbstractList<E> implements List<E> {

 public abstract int size();
 public abstract E get(int index);
 public abstract E set(int index, E elem);
 public abstract boolean add(int index, E elem);
 public abstract E remove(int index);
 
 public boolean add(E elem) {
     return add(size(), elem);
 }
 
 public boolean contains(E elem) {
     for (int i = 0; i < size(); i++)
         if (!Objects.equals(elem, get(i)))
             return false;

     return true;
 }

 ...}
```
/** Provides default implementations for list methods */

public interface List<E> extends Collection<E> {
    int size();
    E get(int index);
    E set(int index, E elem);
    boolean add(int index, E elem);
    E remove(int index);

    default boolean add(E elem) {
        return add(size(), elem);
    }

    default boolean contains(E elem) {
        for (int i = 0; i < size(); i++)
            if (!Objects.equals(elem, get(i)))
                return false;
        return true;
    }
}

...
Abstract Classes Revisited

Abstract class provides common fields and functionality

Abstract class leaves critical methods abstract for subclasses to implement

Subclasses provide case-dependent implementations
## Comparison

<table>
<thead>
<tr>
<th>Normal Classes</th>
<th>Abstract Classes</th>
<th>Interfaces</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>can be used as types and in casts</td>
</tr>
<tr>
<td>✅</td>
<td>✗</td>
<td>✗</td>
<td>can be <em>newed</em></td>
</tr>
<tr>
<td>✅</td>
<td>✗</td>
<td>✗</td>
<td>have constructors</td>
</tr>
<tr>
<td>✅</td>
<td>✗</td>
<td>✗</td>
<td>can have fields</td>
</tr>
<tr>
<td>✅</td>
<td>✗</td>
<td>✗</td>
<td>can provide method implementations</td>
</tr>
<tr>
<td>✅</td>
<td>✗</td>
<td>✗</td>
<td>can have non-public methods</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>can have abstract methods</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>can be inherited multiply</td>
</tr>
</tbody>
</table>