Java 2 Collections

- **Common collections:** bag, set, list, stack, queue, priority queue, map
  - General ways of organizing data and providing controlled access to that data
- **Common data structures:** array, linked list, tree, heap, hash table
  - Specific ways of sorting and conneting data in a program
- **Abstract data type:** a specifi way of providing a collection within a given programing language
  - public class ArrayList<T> implements List<T> { ... }
- Why build your own?
  - It's possible that you might need to build a customized collection one day
  - It's guaranteed that you will need to build and manage your own data structures
  - A solid understanding of how data structures work is required, fundamental knowledge
- An implementation pattern
  - Interface (specification: concept, abstract behavior) <------ Class (implementation: impelement the interface, provide physical storage, reusable but typesafe) <---------- Helper Class
    - Possibly other classes: support for physicl storage, iteration, exceptions, etc.
  - Possible naming conventions:
    - _____(collection name)Interface(typically not used)
    - _____(a clue to the data structure being used) _____(collection name)
- Why interfaces?
  - **Interfaces provide the best mechanism in Java for decoupling a specification from its (various) implementations**

```
public class Client {
  CollectionInterface<MyType> c = new ArrayCollection<MyType>();  //The only spot in the client that depends in any way on the collection implementation
}
```

  - An interface defines a contract between a client and a provider
  - **Big Ideas:** information hiding, encapsulation
- A Bag collection
  - A **bag** or multiset is a collection of elements where there is no particular order and duplicates are allowed; this is essentially what java.util.Collection describes
  - We will **specify the behavior** of this collection with an **interface:**

```
import java.util.Iterator;
public interface Bag<T> {
  boolean add(T element);
  boolean remove(T element);
  boolean contains(T element);
  int size();
  boolean isEmpty();
  Iterator<T> iterator();
}
```

- ArrayBag
  - We will **implement the behavior** of the collection with a **class**

```
import java.util.Iterator
public class ArrayBag<T> implements Bag<T> {
  private T[] elements;  //provide physical storage
  private int size;  //Add a convenience field
  public ArrayBag() {   //provide a constructor

  }

  public boolean add(T element);
  public boolean remove(T element);
  public boolean contains(T element);
  public int size();
  public boolean isEmpty();
```

```
  public Iterator<T> iterator();
}

import java.util.Iterator
public class ArrayBag<T> implements Bag<T> {

	private static final int DEFAULT_CAPACITY = 1;
	private T[] elements;
	private int size;

	public ArrayBag() {
		this(DEFAULT_CAPACITY);
	}

	//Use annotation
	@SuppressWarnings(“unchecked”)
	public ArrayBag(int capacity) {
		elements = (T[]) new Object[capacity]; //Java does not support generics for arrays// *Which generates a type-safety error that cannot be eliminated*
		size = 0;
	}

	public int size() {
		return size;
	}

	public boolean isEmpty() {
		return size() == 0;
	}

	//Refactoring the add()
	//Changing add()
	public boolean add(T element) {

	//ignore and return false
		if (isFull()) {
			resize(elements.length * 2);
		}

		elements[size] = element;
		size++;
		return true;
		//Time Comp: O(1) which means its independent of the size of the things in the bag.
		//Time Comp: O(N) when accommodating add requests greater than the array length, but most of the time it is O(1) amortized.

	}

	private void resize(int capacity) {
		T[] a = (T[]) new Object[capacity];

		*/ Two ways to do this
		for (int i = 0; i < size(); i++) {
			a[i] = elements[i];
		}
		/*
```

```
        //Semantically the same, but it is a native method to Java
        //Could be more efficient from platform to platform
        System.arraycopy(elements, 0, a, 0, elements.length);

        elements = a;
    }

    private boolean isFull() {
        return size == elements.length;
    }

    //Just a Linear Search//
    public boolean contains(T element) {
        return locate(element) >= 0;
        //Time Comp: O(N) where N refers to the size of the bag
    }

    //Another Linear Search, but Contains() is not useful//
    public boolean remove(T element) {
        int i = locate(element);
        if (i < 0) {
            return false;
        }

        if (i >= size) {
            return false;
        }

        //Replace found with the last element//
        elements[i] = elements[—size];
        elements[size] = null;

        if (size > 0 && size < elements.length / 4) {
            resize(elements.length / 2);
        }

        return true;

        //Time Comp: O(N) where N is the number of elements in the bag.
    }
```

- Starting an empty bag at capacity 1 and using dynamic resizing strategies allows us to maintain an array that is always between 25% and 100% full.
- The amount of memory is always constant.

```
    private int locate(T element) {

        for (int i = 0; i < size; i++) {
            if (elements[i].equals(element))
                return i;
        }
        return -1;
    }

    public Iterator<T> iterator() {
```

```
        }
}

import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {
        private T[] items;

        private int count;

        private int current;

        public ArrayIterator(T[] elements, int size) {
                items = elements;
                count = size;
                current = 0;
        }

        public boolean hasNext() {
                return (current < count);
        }

        public void remove() {
                throw new UnsupportedOperationException();
        }

        public T next() {
                if (!hasNext()) {
                        throw new NoSuchElementException();
                }

                return items[current++];
        }
}

public class ArrayBagTest {

        @Test public void addTest1() {
                Bag<Integer> bag = new
        ArrayBag<Integer>();
                boolean expected = true;
                boolean actual = bag.add(2);
                Assert.assertEquals(expected, actual);
        }

        //Note that we have no access to the fields size and elements from the test case methods.

        //Only testing the return value is not enough. We have to test the interactions among add and other
methods.

}
```

Refactoring

- Not necessary
  - Increases readability.

- increases maintainability

Approach

- Develop a method at a time
- Run tests
- Check time complexity