AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Collections

COMP 2210 – Dr. Hendrix

Collections

*In computer science, a **collection** or container is a grouping of some variable number of data items (possibly zero) that have some shared significance to the problem being solved and need to be operated upon together in some controlled fashion. Generally, the data items will be of the same type or, in languages supporting inheritance, derived from some common ancestor type. A collection is a concept applicable to abstract data types, and does not prescribe a specific implementation as a concrete data structure, though often there is a conventional choice; see container (type theory) for type theory discussion.*

`http://en.wikipedia.org/wiki/Collection_(computing)`

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 storing and connecting data in a program.

These terms are commonly used and misused interchangeably.

Abstract Data Type: A specific way of providing a collection within a given programming language

```
public class ArrayList<T> implements List<T> { . . . }
```

Java Collections Framework

The Java platform includes a collections framework. A collection is an object that represents a group of objects (such as the classic Vector class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

The primary advantages of a collections framework are that it:

- ***Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.*
- ***Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable**, programs can be tuned** by switching implementations.*
- ***Provides interoperability** between unrelated APIs by establishing a common language to pass collections back and forth.*
- ***Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.*
- ***Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.*
- ***Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.*

`http://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html`

Why build our own?

It's possible that you might need to build a customized collection one day.

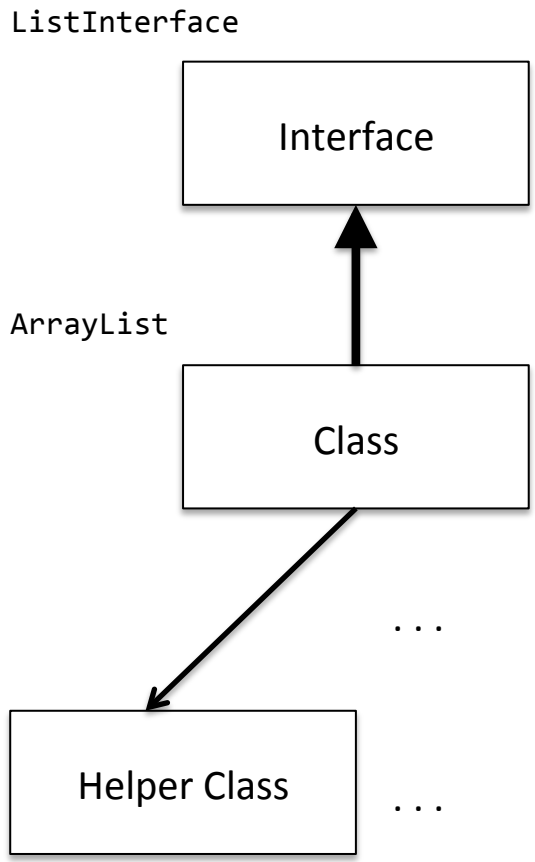
It's guaranteed that you will need to build and manage your own data structures.

A solid understanding of how data structures work is required, fundamental knowledge.

Building our own collections and data structures is an excellent way to understand:

- Algorithm efficiency
- Software design and implementation issues
- Engineering tradeoffs

An implementation pattern



Specification

concept

abstract behavior

Implementation

Implement the interface

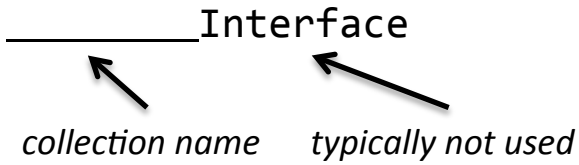
Provide physical storage

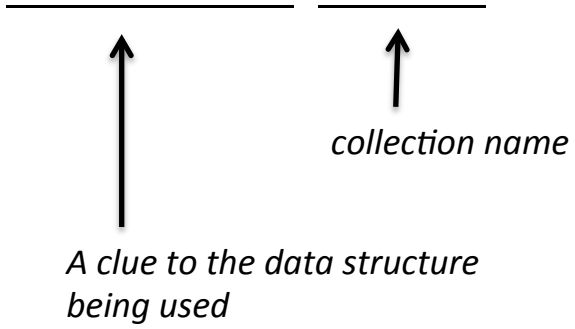
Reusable, but typesafe

Possibly other classes:

Support for physical storage, iteration, exceptions, etc.

Possible naming conventions:





Why interfaces?

Why specify an interface and then a class that implements it? Why not just write the class?

Interfaces provide the best mechanism in Java for decoupling a specification from its (various) implementations.

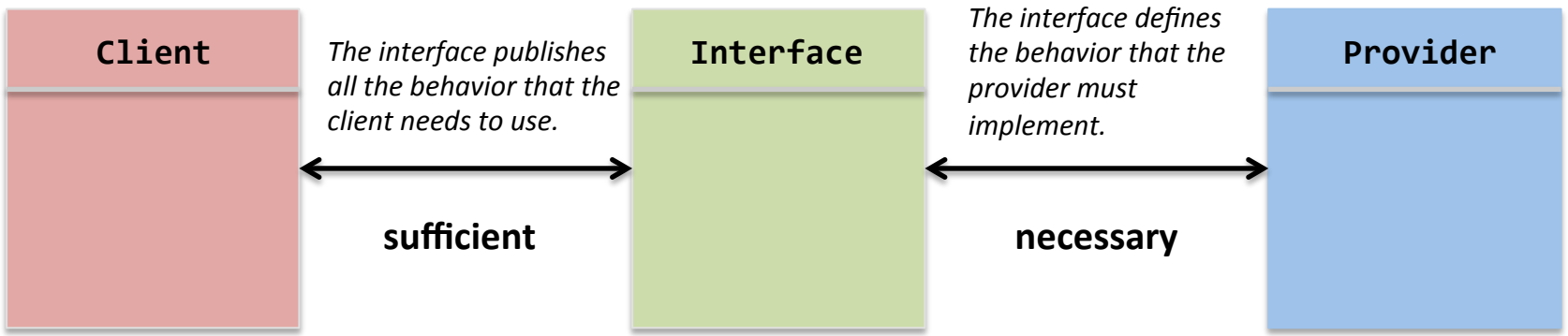
```
public class Client {

   CollectionInterface<MyType> c = new ArrayCollection<MyType>();

}
```

The only spot in the client that depends in any way on the collection implementation.

The client is only dependent on the abstract behavior described in the interface. Any class that provides this behavior will work.

Why interfaces?

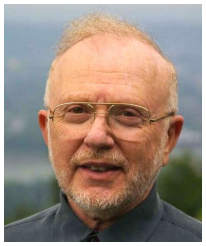
An interface defines a contract between a client and a provider.



This is the basic "if and only if" relationship that makes programming beyond a single person really work.

Why interfaces?

Big Ideas *Information hiding, encapsulation, abstract data types*



David Parnas

"… one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

On the Criteria To Be Used in Decomposing Systems into Modules. *CACM* 15(12), 1972.

"The connections between modules are the assumptions which the modules make about each other."

Information Distribution Aspects of Design Methodology. *Information Processing* (71), 1972.

Barbara Liskov

"When a programmer makes use of an abstract data object, he is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation. The behavior of an object is captured by the set of characterizing operations. Implementation information, such as how the object is represented in storage, is only needed when defining how the characterizing operations are to be implemented. The user of the object is not required to know or supply this information."

Programming with abstract data types. SIGPLAN Not. 9, 4 (March 1974)

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();
}
```

A subset of the JCF Collection interface

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;

    private int size;

    public ArrayBag() { . . . }

    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() { . . . }

}
```

Provide physical storage

Add a convenience field

Provide a constructor

Implement all interface methods

Choose an appropriate data structure that will efficiently support the collection methods.

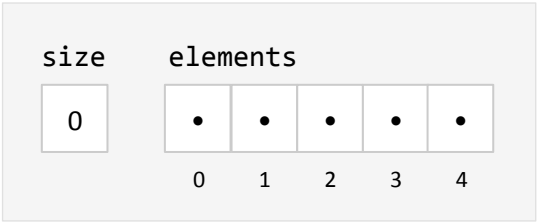
ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;

   private T[] elements;
   private int size;

   public ArrayBag() {
      this(DEFAULT_CAPACITY);
   }
```

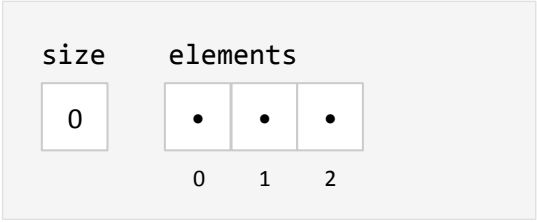
Design decision: Should this constructor be public or private?

```
   public ArrayBag(int capacity) {


   }
}
```

```
Bag bag = new ArrayBag();
```




```
bag = new ArrayBag(3);
```



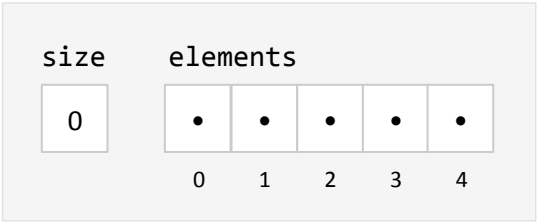
ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;

   @SuppressWarnings("unchecked")
   public ArrayBag(int capacity) {
      elements = (T[]) new Object[capacity];
      size = 0;
   }

}
```

This annotation will suppress the notification.

This will generate a type-safety warning that can't be eliminated.

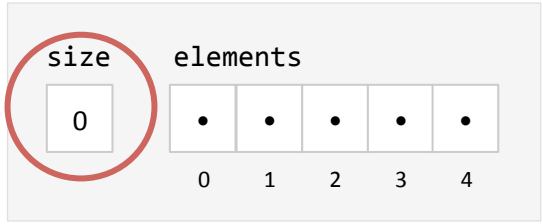
```
Bag bag = new ArrayBag(5);
```



ArrayBag – size and isEmpty

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;

   public int size() {
      return size;
   }

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

}
```

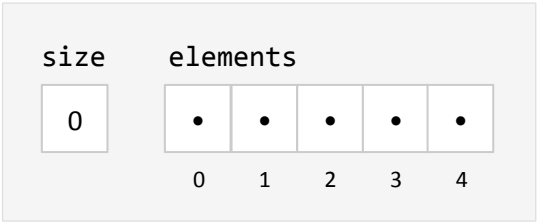
```
Bag bag = new ArrayBag(5);
```



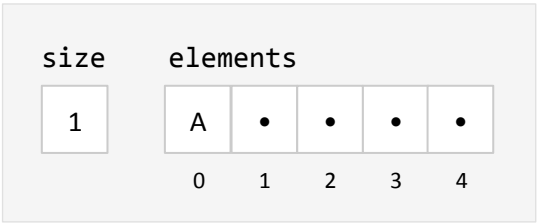
These can be fast and trivial with O(1) time complexity.

ArrayBag – add()

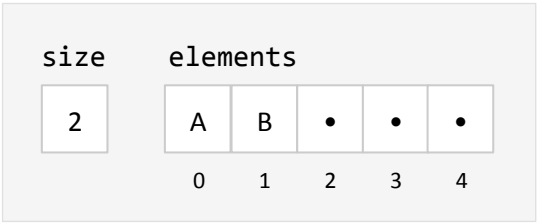
```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;




   }
}
```



bag.add("A");

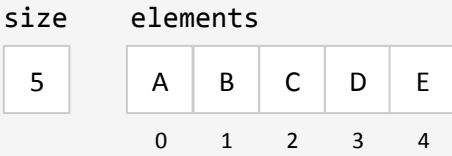


bag.add("B");



ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;




   }
}
```

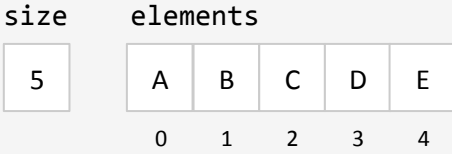


`bag.add("F");`

What happens at this point?

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;

   }
}
```



`bag.add("F");`

What happens at this point?

Options?

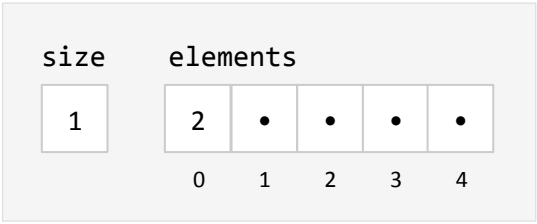
Ignore and return false

Throw an exception

Get a bigger array

ArrayBag – add() testing

```
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);
   }

   @Test public void addTest2() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      int expected = 1;
      bag.add(2);
      int actual = bag.size();
      Assert.assertEquals(expected, actual);
   }
}
```

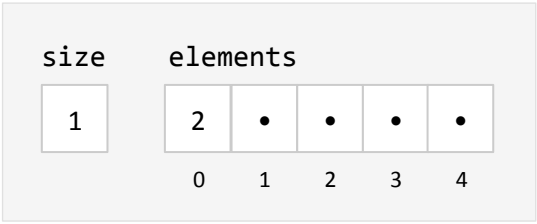


Note that we have no access to the fields size and elements from the test case methods.

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

ArrayBag – add() testing

```
public class ArrayBagTest {

   @Test public void addTest3() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      bag.add(2);
      boolean actual = bag.contains(2);
      Assert.assertEquals(expected, actual);
   }

   @Test public void addTest4() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      bag.add(2);
      boolean actual = bag.remove(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.

ArrayBag – add() efficiency

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

}
```

Time Complexity: O(1)

We can add a new element to the bag in constant time. That is, no matter how large the bag grows, it always takes the same amount of time to add a new element.

ArrayBag – add() refactoring

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {   isFull
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

}
```

REFACTORING

http://www.refactoring.com/

Extract Method:

"Turn [a] fragment into a method whose name explains the purpose of the method."

ArrayBag – add() refactoring

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (isFull()) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

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

REFACTORING

http://www.refactoring.com/

Extract Method:

"Turn [a] fragment into a method whose name explains the purpose of the method."

This isn't strictly necessary, but:

- It increases readability.
- It increases maintainability.

ArrayBag – so far

```
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();
}
```

We're taking a systematic approach to developing the ArrayBag class:

- Develop one method at a time.
- Run it against its full test suite (which will involve calls to other methods that may still be stubs).
- Analyze its time complexity, revise if appropriate.
- Consider refactoring, clean-up, and generality.

Note that a given method in this class can't be fully tested until all the methods have been written. Development and testing are necessarily iterative.

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

      for (int i = 0; i < ____________; i++) {
         if (elements[i].equals(element)) {
            return true;
         }
      }
      return false;

   }



}
```

This is just linear search.

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

      for (int i = 0; i < ___size___; i++) {
         if (elements[i].equals(element)) {
            return true;
         }
      }
      return false;

   }



}
```

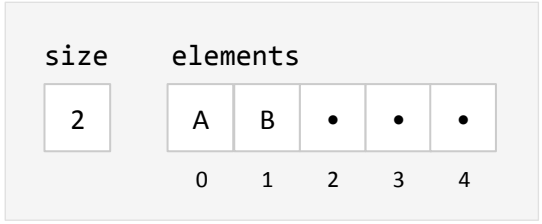
Q: What should go in the blank?

A. `elements.length`

B. `size`

C. `isFull()`

D. `DEFAULT_CAPACITY`




```
size = 2
elements.length = 5
```

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

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

   }



}
```

Testing …

```
@Test
public void testContainsPresentMiddleFull() {
   BagInterface<String> bag =
   new ArrayBag<String>(5);
   bag.add("A"); bag.add("B");
   bag.add("C"); bag.add("D");
   bag.add("E");
   boolean expected = true;
   boolean actual = bag.contains("C");
   Assert.assertEquals(expected, actual);
}
```

Time complexity …

O(N) *where N is the size of the bag, not the capacity of the array*

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
             (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

            located, so remove it

   }
}
```

attempt to locate element

Linear search again …

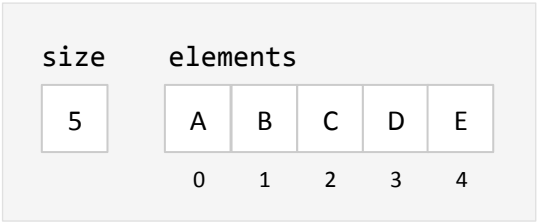
unable to locate

Linear search from contains:

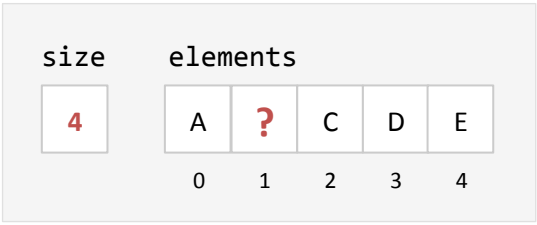
```
for (int i = 0; i < size; i++) {
   if (elements[i].equals(element)) {
      return true;
   }
}
return false;
```

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
             (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

      [        located, so remove it        ]

   }
}
```




```
bag.remove("B");
```



Must handle the array consistent with add() – left justified, no gaps.

Participation

Q: Which is the **correct and most efficient** option for removing element?

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
             (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

      [          located, so remove it          ]

   }
}
```

A. Just set to null

size	elements				
4	A	•	C	D	E
	0	1	2	3	4

B. Shift to the left

size	elements				
4	A	C	D	E	•
	0	1	2	3	4

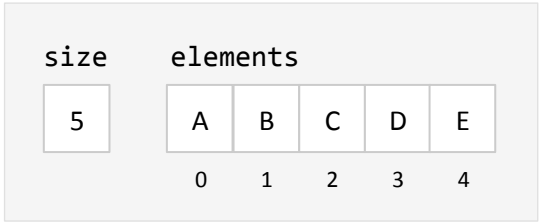
C. Replace with the last

size	elements				
4	A	E	C	D	•
	0	1	2	3	4

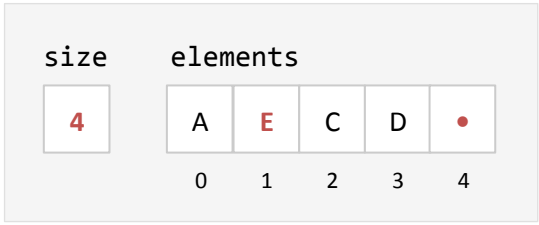
ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
             (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

located, so remove it




```
bag.remove("B");
```



ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

      elements[i] = elements[--size];
      elements[size] = null;
      return true;

   }
}
```

Time complexity: **O(N)**

N = number of elements in the bag, not the capacity of the array

O(N)

O(1)

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

      elements[i] = elements[--size];
      elements[size] = null;
      return true;

   }
}
```

Refactoring: Extract method

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
             (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

      elements[i] = elements[--size];
      elements[size] = null;
      return true;

   }
}
```

Refactoring: Extract method

Refactor this for two reasons:
(1) Textbook "extract method" – it's linear search.
(2) Linear search is used in two different methods – contains and remove.

Note:
The remove() method needs the location of the element, but contains() doesn't. So, remove() can't use the linear search from contains(), but contains() can use the linear search from remove().

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }

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

Refactoring: Extract method

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean contains(T element) {

      return locate(element) >= 0;
   }


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

Refactoring: Extract method

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

ArrayBag – iterator()

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public Iterator<T> iterator() {
      return new ArrayIterator(elements, size);
   }

}
```

```
public class ArrayIterator<T>
implements Iterator<T>
```

Nested class

Has access to private fields; don't have to expose them in any way.

Top-level class

Can be used by different collection classes.

ArrayBag – iterator()

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  // the array of elements to be iterated over.
  private T[] items;

  // the number of elements in the array.
  private int count;

  // the current position in the iteration.
  private int current;

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

ArrayBag – 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 boolean hasNext() {
     return (current < count);
  }


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


}
```

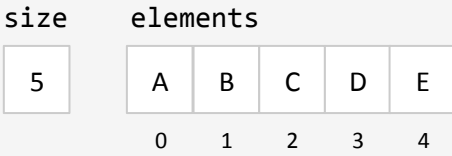
The remove method is listed as an "optional operation" in the Iterator API.

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

     return items[current++];
  }


}
```

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {
      if (size == elements.length) {
         return false;
      }
      elements[size] = element;
      size++;
      return true;
   }


}
```



`bag.add("F");`

What happens at this point?

Options?

Ignore and return false

Throw an exception

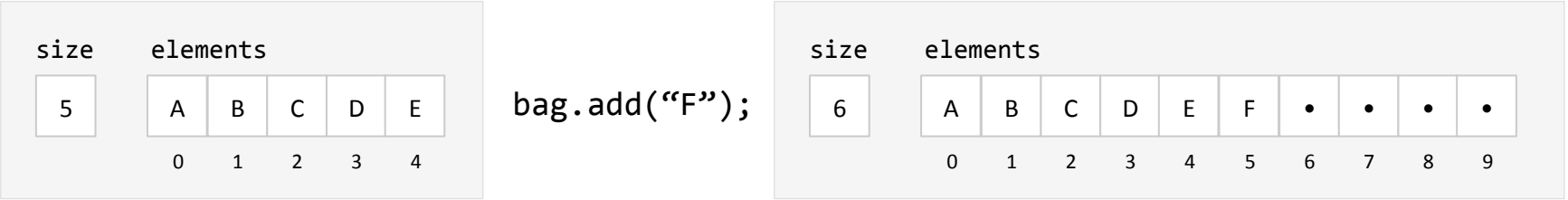
Get a bigger array

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (size == elements.length) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

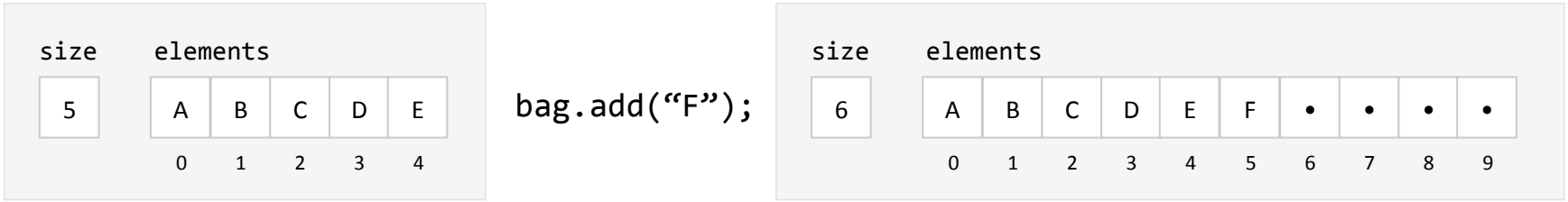
Strategy:

When the array becomes full, double the capacity.



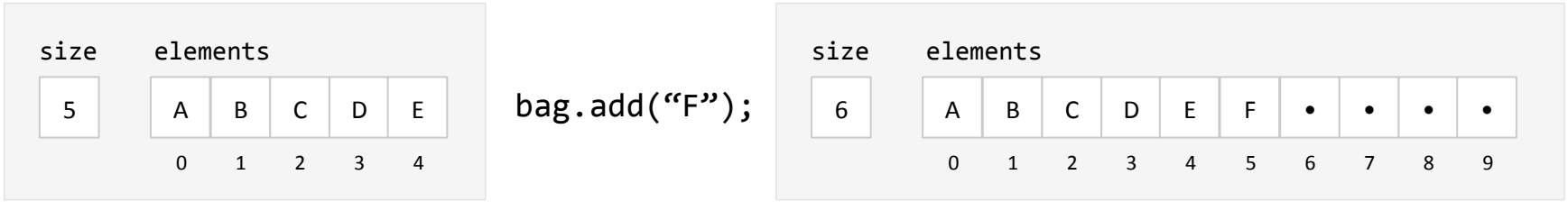
Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   private void resize(int capacity) {
      T[] a = (T[]) new Object[capacity];
      for (int i = 0; i < size(); i++) {
         a[i] = elements[i];
      }
      elements = a;
   }
}
```



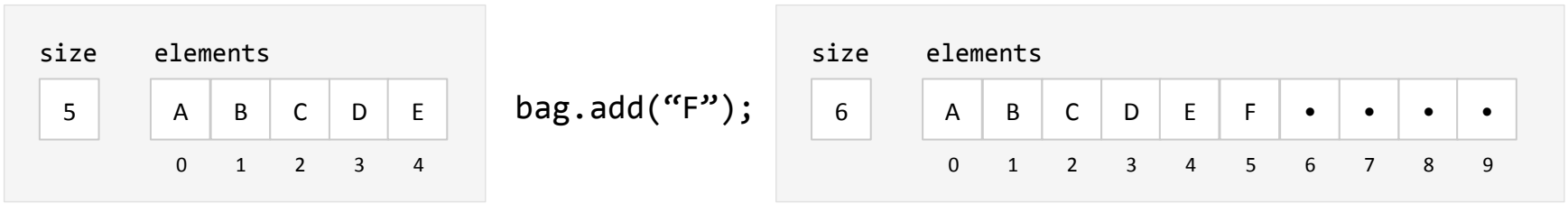
Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   private void resize(int capacity) {
      T[] a = (T[]) new Object[capacity];
      System.arraycopy(elements, 0, a, 0, elements.length);
      elements = a;
   }


}
```



Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   private void resize(int capacity) {
      T[] a = Arrays.<T>copyOf(elements, capacity);
      elements = a;
   }



}
```



Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (size == elements.length) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #1: **O(N)**

Although we won't have to expand the array very often, it will be linear cost when we do. So, in a strict sense, the worst case is O(N).

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (size == elements.length) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

We can *amortize* the cost of expanding the capacity of the array over a sequence of N calls to add().

add() 1:	1	∑ = ~2N
add() 2:	1	÷
add() 3:	1	N
		= ~1
add() N-1:	1	
add() N:	N	

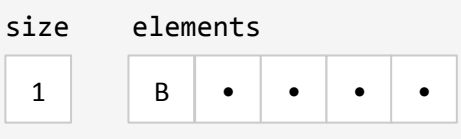
Dynamic resizing – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      if (size > 0 && size < elements.length / 4) {
         resize(elements.length / 2);
      }
      return true;
   }
}
```

Strategy:

When the array becomes less than 25% full, reduce the capacity by half.

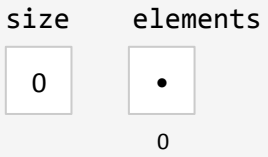
bag.remove("A");



ArrayBag – constructor

```
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);
   }

}
```

```
Bag bag = new ArrayBag();
```



Starting an empty bag at capacity 1 and using the dynamic resizing strategies just described allows us to maintain the following invariant: the array is always between 25% and 100% full.

Thus, the amount of memory needed for the array is a constant times N, that is, O(N).

We can guarantee that our implementation only needs a linear amount of memory.

Participation

Q: Assuming that the ArrayBag class implements the dynamic resizing strategy just described, what is the capacity of the internal array after the following sequence of statements has executed?

```
Bag<String> sb = new ArrayBag<String>();

sb.add("A"); sb.add("B"); sb.add("C"); sb.add("D"); sb.add("E");

sb.remove("A"); sb.remove("B"); sb.remove("C"); sb.remove("D");
```

A. 10
B. 8
C. 4 ⬅
D. 2