

DATA STRUCTURE

Set Storage

Nguyễn Văn Khiêm

nguyenvankhiem@hcmuaf.edu.vn

Objectives

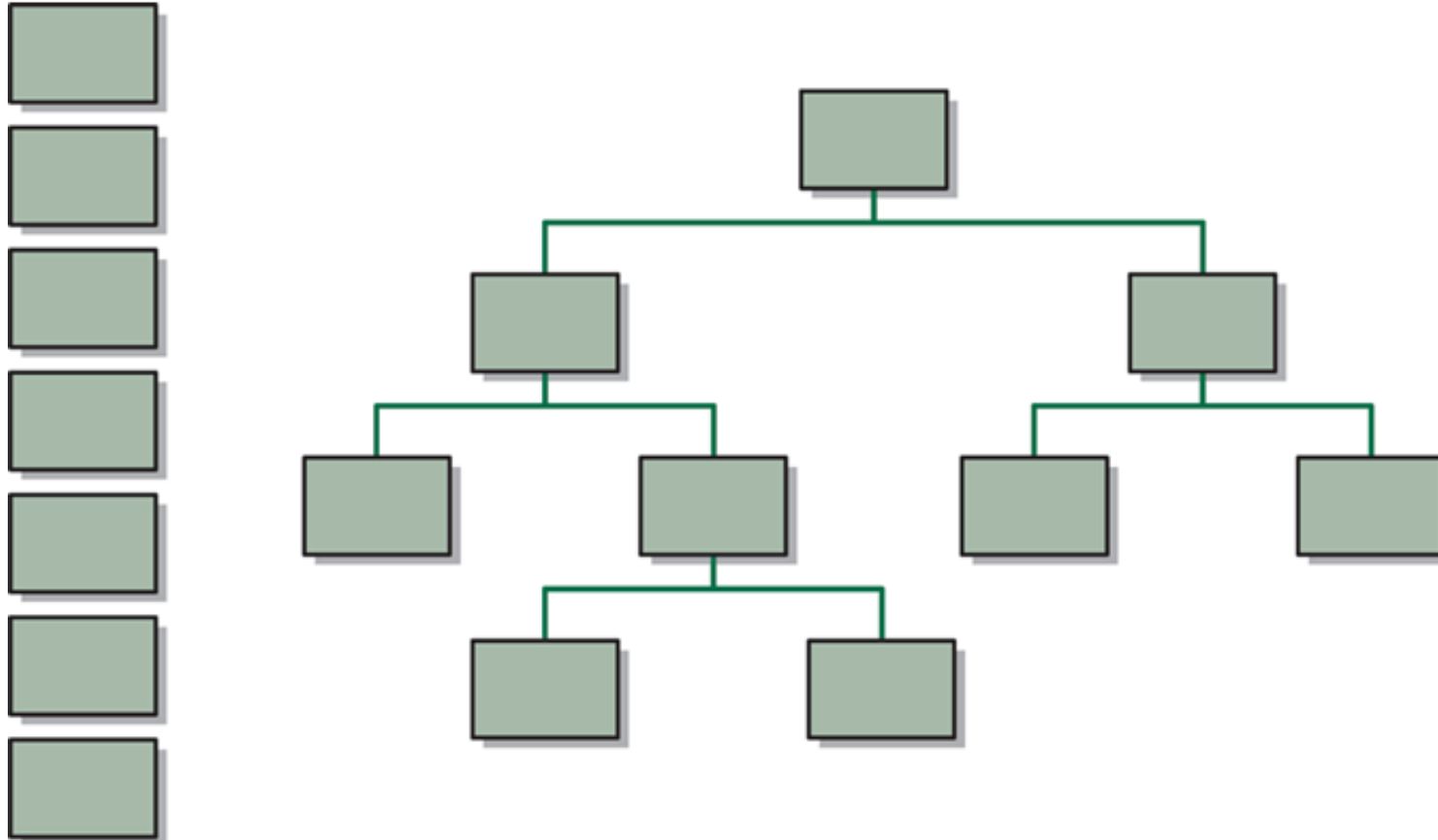
- ▶ Define the concept and terminology related to collections
- ▶ Explore the basic structure of the Java Collections API
- ▶ Discuss the abstract design of collections
- ▶ Define a set collection
- ▶ Use a set collection to solve a problem
- ▶ Examine an array implementation of a set

Collections

- ▶ A *collection* is an object that gathers and organizes other object (elements)
- ▶ Many types of fundamental collections have been defined: stack, queue, list, tree, graph, etc
- ▶ They can be broadly categorized as *linear* (organizes the elements in a straight line) or *nonlinear*

FIGURE 3.1

A linear and a nonlinear collection



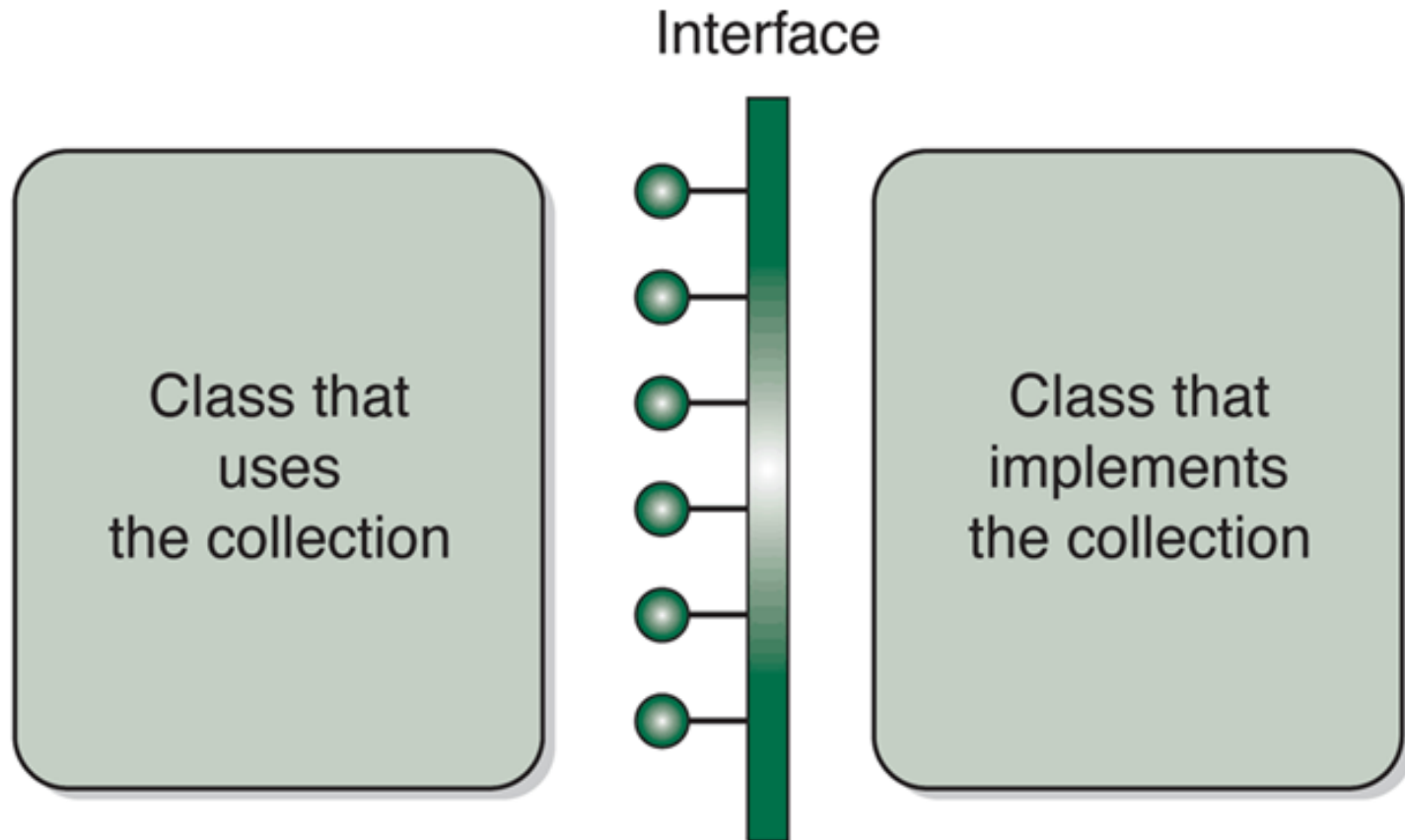
Collections

- ▶ The elements within a collection are usually organized based on:
 - The order in which they were added to a collection, or
 - Some relationship among the elements themselves
- ▶ For example, a list of people may be kept in alphabetical order by name or in the order in which they were added to the list
- ▶ Which type of collection you use depends on what you are trying to accomplish

Abstraction

- ▶ An abstraction hides certain details at certain times
- ▶ It provides a way to deal with the complexity of a large system
- ▶ A collection, like any well-defined object, is an abstraction
- ▶ We want to separate the *interface* of the collection (how we interact with it) from the underlying details of how we choose to implement it

FIGURE 3.2 A well-defined interface masks the implementation of the collection



Issues with Collections

- ▶ For each collection we examine, we will consider:
 - How does the collection operate conceptually?
 - How do we formally define its interface?
 - What kinds of problems does it help us solve?
 - What ways might we implement it?
 - What are the benefits and costs of each implementation?

Terms

- ▶ Some terms:
 - *Data type* – a group of values and the operations defined on those values
 - *Abstract data type* – a data type whose values and operations are not inherently defined in a programming language
 - *Data structure* – the programming constructs used to implement a collection

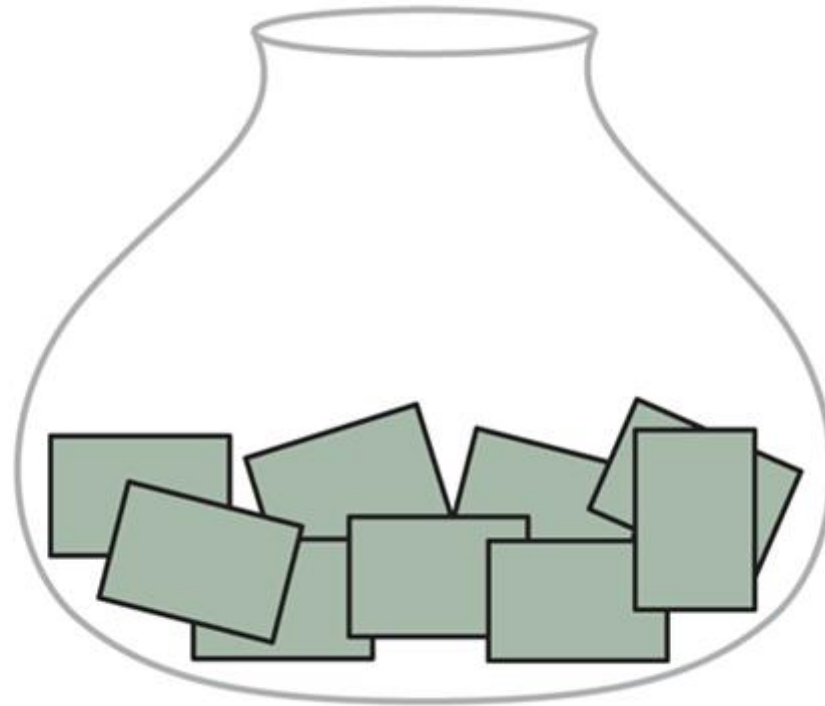
The Java Collections API

- ▶ The Java Collection API is a set of classes that represent some specific collection types, implemented in various ways
- ▶ It is part of the large class library that can be used by any Java program
- ▶ API stands for Application Programming Interface
- ▶ As we explore various collections, we will examine the appropriate classes in the Java Collection API

A Set Collection

- ▶ Lets look at an example of a collection
- ▶ A set collection groups elements without regard to their relationship to each other
- ▶ It's as if you threw them all in a box
- ▶ You can reach into a box and pull out an element, and are equally likely to get any one
- ▶ It is a nonlinear collection, but could be implemented with a linear data structure

FIGURE 3.3 The conceptual view of a set collection



Collection Operations

- ▶ Every collection has a set of operations that define how we interact with it
- ▶ They usually include ways for the user to:
 - Add and remove elements
 - Determine if the collection is empty
 - Determine the collection's size
- ▶ They also may include:
 - *Iterators*, to process each element in the collection
 - Operations that interact with other collections

FIGURE 3.4

The operations on a set collection

Operation	Description
add	Adds an element to the set.
addAll	Adds the elements of one set to another.
removeRandom	Removes an element at random from the set.
remove	Removes a particular element from the set.
union	Combines the elements of two sets to create a third.
contains	Determines if a particular element is in the set.
equals	Determines if two sets contain the same elements.
isEmpty	Determines if the set is empty.
size	Determines the number of elements in the set.
iterator	Provides an iterator for the set.
toString	Provides a string representation of the set.

Java Interfaces

- ▶ The programming construct in Java called an *interface* is a convenient way to define the operations on a collection
- ▶ A Java interface lists the set of abstract methods (no bodies) that a class implements
- ▶ It provides a way to establish a formal declaration that a class will respond to a particular set of messages (method calls)

Listing 3.1

```
▶ //*****
▶ // SetADT.java           Authors: Lewis/Chase
▶ //
▶ // Defines the interface to a set collection.
▶ //*****

▶ package setStorage;

▶ import java.util.Iterator;

▶ public interface SetADT<T>
▶ {
▶     // Adds one element to this set, ignoring duplicates
▶     public void add (T element);

▶     // Removes and returns a random element from this set
▶     public T removeRandom ();

▶     // Removes and returns the specified element from this set
▶     public T remove (T element);

▶     // Returns the union of this set and the parameter
▶     public SetADT<T> union (SetADT<T> set);
```

Listing 3.1 (cont.)

- ▶ `// Returns true if this set contains the parameter`
- ▶ `public boolean contains (T target);`

- ▶ `// Returns true if this set and the parameter`
`contain exactly`
- ▶ `// the same elements`
- ▶ `public boolean equals (SetADT<T> set);`

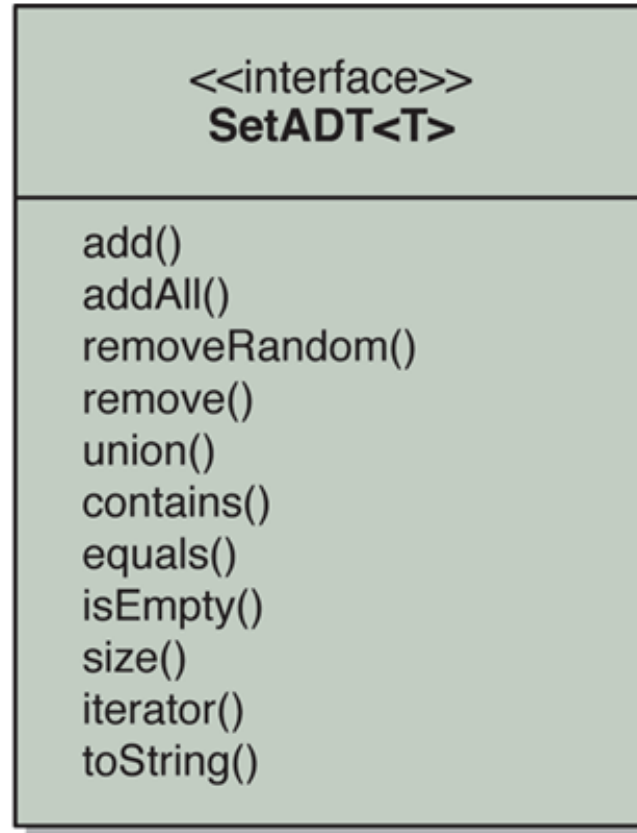
- ▶ `// Returns true if this set contains no elements`
- ▶ `public boolean isEmpty();`

- ▶ `// Returns the number of elements in this set`
- ▶ `public int size();`

- ▶ `// Returns an iterator for the elements in this set`
- ▶ `public Iterator<T> iterator();`

- ▶ `// Returns a string representation of this set`
- ▶ `public String toString();`
- ▶ `}`

FIGURE 3.5 UML description of the SetADT<T> interface



Iterators

- ▶ An iterator is an object that allows the user to acquire and use each element in a collection in turn
- ▶ The program design determines:
 - The order in which the elements are delivered
 - The way the iterator is implemented
- ▶ In the case of a set, there is no particular order to the elements, so the iterator order will be arbitrary (random)

Iterators

- ▶ Collections that support iterators often have a method called `iterator` that returns an `Iterator` object
- ▶ `Iterator` is actually an interface defined in the Java standard class library
- ▶ `Iterator` methods:
 - `hasNext` – returns true if there are more elements in the iteration
 - `next` – returns the next element in the iteration

Set Exceptions

- ▶ Collections must always manage problem situations carefully
- ▶ For example: attempting to remove an element from an empty set
- ▶ The designer of the collection determines how it might be handled
- ▶ Our implementation provides an `isEmpty` method, so the user can check beforehand
- ▶ And it throws an exception if the situation arises, which the user can catch

Using a Set: BINGO

- ▶ The game of BINGO can be used to demonstrate the use of a set collection
- ▶ Each player has a bingo card with numeric values associated with the letters B-I-N-G-O
- ▶ Letter/number combinations (on bingo balls) are picked at random, which the player marks on their card if possible
- ▶ The first player to get five squares in a row wins

FIGURE 3.6 A bingo card

B	I	N	G	O
9	25	34	48	69
15	19	31	59	74
2	28	FREE	52	62
7	16	41	58	70
4	20	38	47	64

BINGO

- ▶ A set is an appropriate collection for BINGO, allowing the caller to pick numbers at random
- ▶ We create an object of class BingoBalls to represent one letter/number combination
- ▶ The main program creates the balls, stores them in a set, and draws them at random

BINGO

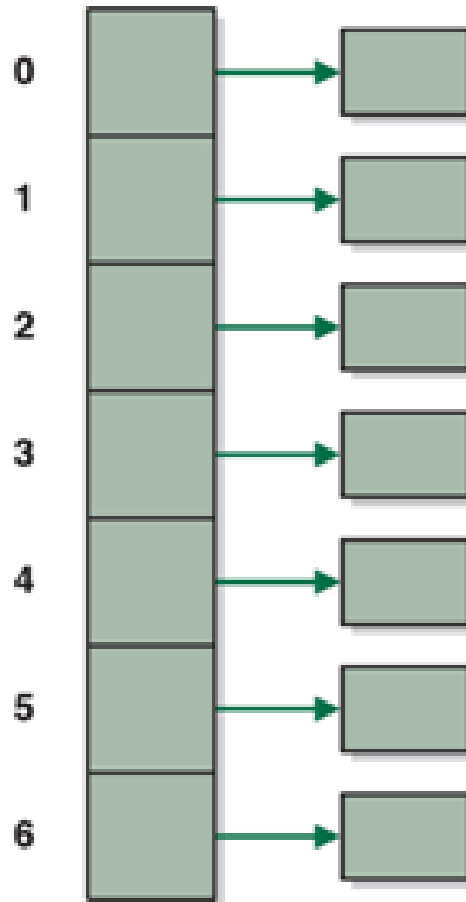
- ▶ Note that the class that represents the set is called `ArraySet`
- ▶ Any implementation of a set collection could have been used at that point
- ▶ Notice that because of our use of generics, we place `BingoBalls` in the set and that is what we get back

Implementing a Set

- ▶ The discussion of the BINGO game used a set collection without any regard to how that collection was implemented
- ▶ We were using the set collection for its functionality – how it is implemented is fundamentally irrelevant
- ▶ It could be implemented in various ways
- ▶ Let's examine how we can use an array to implement it

FIGURE 3.8

An array of object references



Managing Capacity

- ▶ An array has particular number of cells when it is created – its capacity
- ▶ So the array's capacity is also the set's capacity
- ▶ What do we do when the set full and a new element is added?
 - We could throw an exception
 - We could return some kind of status indicator
 - We could automatically expand the capacity

Managing Capacity

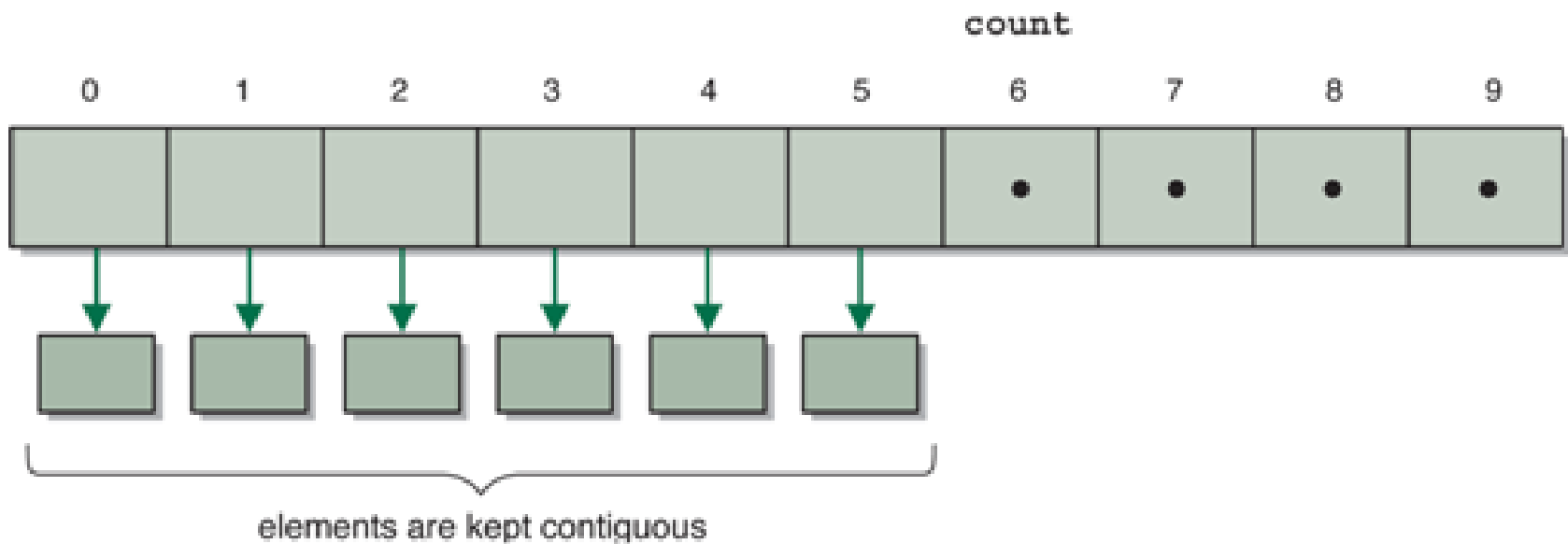
- ▶ The first two options require the user of the collection to be on guard and deal with the situation as needed
- ▶ The third option is best, especially in light of our desire to separate the implementation from the interface
- ▶ The capacity is an implementation problem, and shouldn't be passed along to the user unless there is a good reason to do so

The `ArraySet` Class

- ▶ In the Java Collection API, class names indicate both the underlying data structure and the collection
- ▶ We adopt the same naming convention
- ▶ Thus, the `ArraySet` class represents an array implementation of a set collection
- ▶ Set elements are kept contiguously at one end of the array
- ▶ An integer (`count`) represents:
 - The number of elements in the set
 - The next empty index in the array

FIGURE 3.9

An array implementation of a set



ArraySet - Constructors

```
▶ //-----  
▶ //Creates an empty set using the default capacity.  
▶ //-----  
▶ public ArraySet()  
▶ {  
▶     count = 0;  
▶     contents = (T[]) (new Object[DEFAULT_CAPACITY]);  
▶ }  
  
▶ //-----  
▶ //Creates an empty set using the specified capacity.  
▶ //-----  
▶ public ArraySet (int initialCapacity)  
▶ {  
▶     count = 0;  
▶     contents = (T[]) (new Object[initialCapacity]);  
▶ }
```

ArraySet – size and isEmpty

```
▶ //-----  
▶ // Returns true if this set is empty and  
▶ // false otherwise.  
▶ //-----  
▶ public boolean isEmpty()  
▶ {  
▶     return (count == 0);  
▶ }  
▶  
▶ //-----  
▶ // Returns the number of elements currently  
▶ // in this set.  
▶ //-----  
▶ public int size()  
▶ {  
▶     return count;  
▶ }
```

ArraySet - add

```
▶ //-----  
▶ // Adds the specified element to the set if it's  
▶ // not already present. Expands the capacity  
▶ // of the set array if necessary.  
▶ //-----  
▶ public void add (T element)  
▶ {  
▶     if (!contains(element))  
▶     {  
▶         if (size() == contents.length)  
▶             expandCapacity();  
▶  
▶         contents[count] = element;  
▶         count++;  
▶     }  
▶ }
```

ArraySet - expandCapacity

```
▶ //-----  
▶ //  Creates a new array to store the contents of the  
▶ //  set with twice the capacity of the old one.  
▶ //-----  
▶ private void expandCapacity()  
▶ {  
▶     T[] larger = (T[]) (new Object[contents.length*2]);  
▶  
▶     for (int index=0; index < contents.length; index++)  
▶         larger[index] = contents[index];  
▶  
▶     contents = larger;  
▶ }
```

ArraySet - addAll

```
▶ //-----  
▶ // Adds the contents of the  
▶ // parameter to this set.  
▶ //-----  
▶ public void addAll (SetADT<T> set)  
▶ {  
▶     Iterator<T> scan = set.iterator();  
  
▶     while (scan.hasNext())  
▶         add (scan.next());  
▶ }
```

ArraySet - removeRandom

```
▶ //-----  
▶ // Removes a random element from the set and returns it.  
▶ // Throws an EmptySetException if the set is empty.  
▶ //-----  
▶ public T removeRandom() throws EmptySetException  
▶ {  
▶     if (isEmpty())  
▶         throw new EmptySetException();  
  
▶     int choice = rand.nextInt(count);  
  
▶     T result = contents[choice];  
  
▶     contents[choice] = contents[count-1]; // fill the gap  
▶     contents[count-1] = null;  
▶     count--;  
  
▶     return result;  
▶ }
```

ArraySet - remove

```
▶ //-----  
▶ // Removes the specified element from the set and returns it.  
▶ // Throws an EmptySetException if the set is empty and a  
▶ // NoSuchElementException if the target is not in the set.  
▶ //-----  
▶ public T remove (T target) throws EmptySetException,  
▶                               NoSuchElementException  
▶ {  
▶     int search = NOT_FOUND;  
  
▶     if (isEmpty())  
▶         throw new EmptySetException();  
  
▶     for (int index=0; index < count && search == NOT_FOUND; index++)  
▶         if (contents[index].equals(target))  
▶             search = index;  
  
▶     if (search == NOT_FOUND)  
▶         throw new NoSuchElementException();  
  
▶     T result = contents[search];  
  
▶     contents[search] = contents[count-1];  
▶     contents[count-1] = null;  
▶     count--;  
  
▶     return result;  
▶ }
```

ArraySet - union

```
▶ //-----  
▶ // Returns a new set that is the union of this set and the  
▶ // parameter.  
▶ //-----  
▶ public SetADT<T> union (SetADT<T> set)  
▶ {  
▶     ArraySet<T> both = new ArraySet<T>();  
  
▶     for (int index = 0; index < count; index++)  
▶         both.add (contents[index]);  
  
▶     Iterator<T> scan = set.iterator();  
▶     while (scan.hasNext())  
▶         both.add (scan.next());  
  
▶     return both;  
▶ }
```

ArraySet - contains

```
▶ //-----  
▶ // Returns true if this set contains the  
▶ // specified target element.  
▶ //-----  
▶ public boolean contains (T target)  
▶ {  
▶     int search = NOT_FOUND;  
  
▶     for (int index=0; index < count && search ==  
▶         NOT_FOUND; index++)  
▶         if (contents[index].equals(target))  
▶             search = index;  
  
▶     return (search != NOT_FOUND);  
▶ }
```

ArraySet - equals

```
▶ //-----  
▶ // Returns true if this set contains  
▶ // exactly the same elements  
▶ // as the parameter.  
▶ //-----  
▶ public boolean equals (SetADT<T> set)  
▶ {  
▶     boolean result = false;  
▶     ArraySet<T> temp1 = new ArraySet<T>() ;  
▶     ArraySet<T> temp2 = new ArraySet<T>() ;  
▶     T obj;
```

ArraySet – equals (continued)

```
▶ if (size() == set.size())
▶     {
▶         temp1.addAll(this);
▶         temp2.addAll(set);
▶
▶         Iterator<T> scan = set.iterator();
▶
▶         while (scan.hasNext())
▶         {
▶             obj = scan.next();
▶             if (temp1.contains(obj))
▶             {
▶                 temp1.remove(obj);
▶                 temp2.remove(obj);
▶             }
▶         }
▶
▶         result = (temp1.isEmpty() && temp2.isEmpty());
▶     }
▶
▶     return result;
▶ }
```

ArraySet - iterator

```
▶ //-----  
▶ // Returns an iterator for the elements  
▶ // currently in this set.  
▶ //-----  
▶ public Iterator<T> iterator()  
▶ {  
▶     return new ArrayIterator<T> (contents, count);  
▶ }
```

Listing 3.4

```
▶ /**
 *
 * ArrayIterator.java          Authors: Lewis/Chase
 *
 * Represents an iterator over the elements of an array.
 */
```

```
▶ package setStorage;
```

```
▶ import java.util.*;
```

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

```
▶ {
```

```
▶     private int count;    //the number of elements in the collection
```

```
▶     private int current; //the current position in the iteration
```

```
▶     private T[] items;
```

```
▶     //-----
```

```
▶     // Sets up this iterator using the specified items.
```

```
▶     //-----
```

Listing 3.4 (continued)

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

▶ //-----
▶ // Returns true if this iterator has at least one more element
▶ // to deliver in the iteration.
▶ //-----
▶ public boolean hasNext()
▶ {
▶     return (current < count);
▶ }

▶ //-----
▶ // Returns the next element in the iteration. If there are no
▶ // more elements in this iteration, a NoSuchElementException is
▶ // thrown.
▶ //-----
```

Listing 3.4 (continued)

```
▶ public T next()  
▶ {  
▶     if (! hasNext())  
▶         throw new NoSuchElementException();  
  
▶     current++;  
  
▶     return items[current - 1];  
▶  
▶ }  
  
▶ //-----  
▶ // The remove operation is not supported in this  
collection.  
▶ //-----  
▶ public void remove() throws  
UnsupportedOperationException  
▶ {  
▶     throw new UnsupportedOperationException();  
▶ }  
▶ }
```

Analysis of ArraySet

- ▶ If the array is not null, adding an element to the set is $O(1)$
- ▶ Expanding the capacity is $O(n)$
- ▶ Removing a particular element, because it must be found, is $O(n)$
- ▶ Removing a random element is $O(1)$
- ▶ Adding all elements of another set is $O(n)$
- ▶ The union of two sets is $O(n+m)$, where m is the size of the second set