

Chapter 2

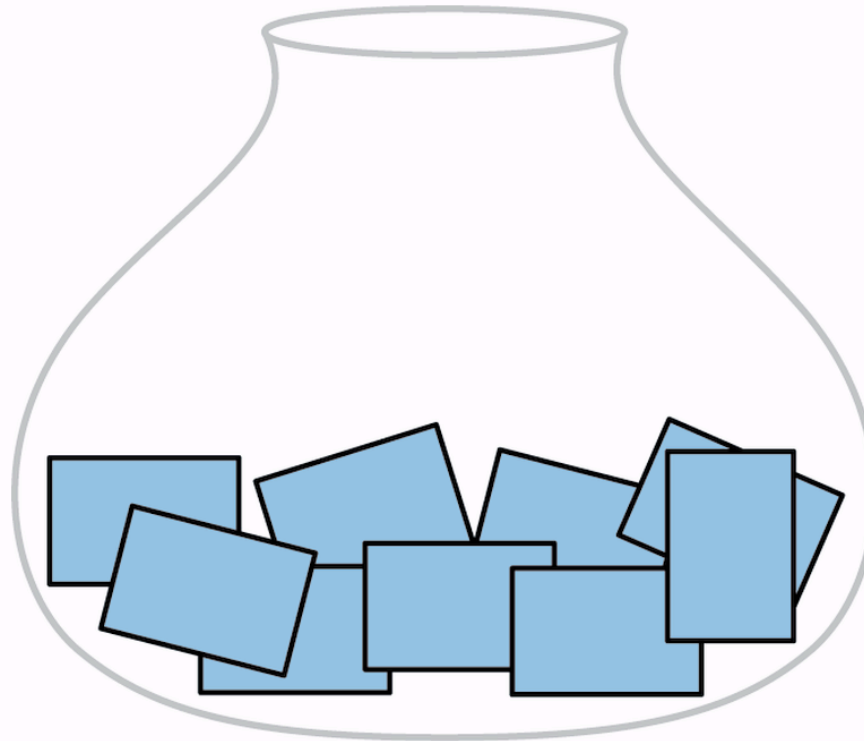
Collections

A Bag Collection

- Let's look at an example of a collection
- A bag collection groups elements without regard to their relationship to each other
- It's as if you threw them all in a bag
- You can reach into a bag 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 2.3

The conceptual view of a bag 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 2.4

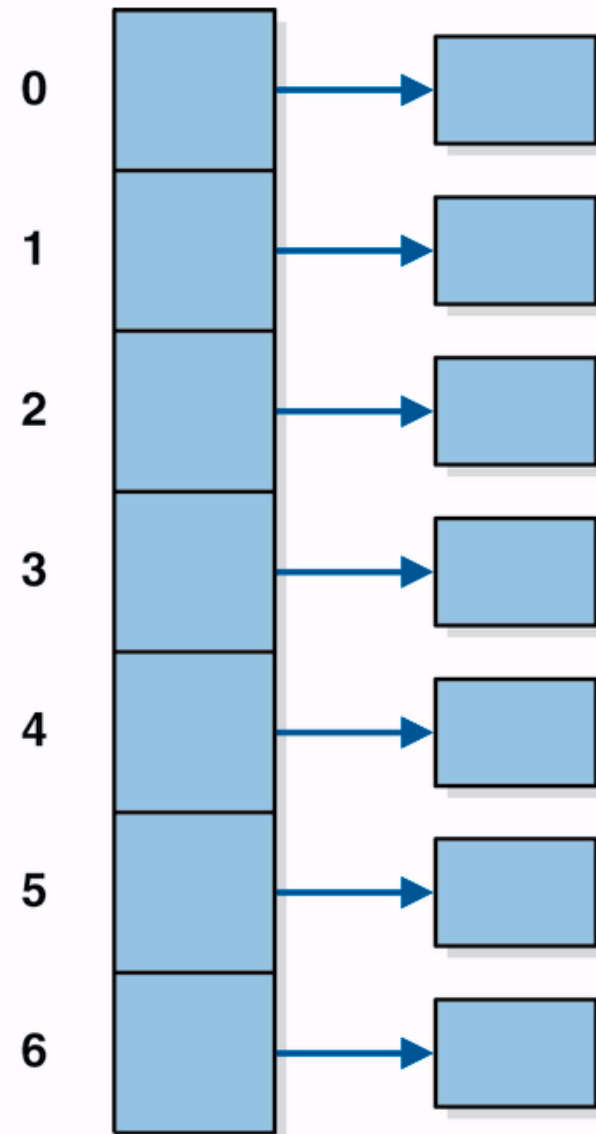
The operations on a bag collection

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

Bag Exceptions

- Collections must always manage problem situations carefully
- For example: attempting to remove an element from an empty bag
- 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

figure 2.8 An array of object references

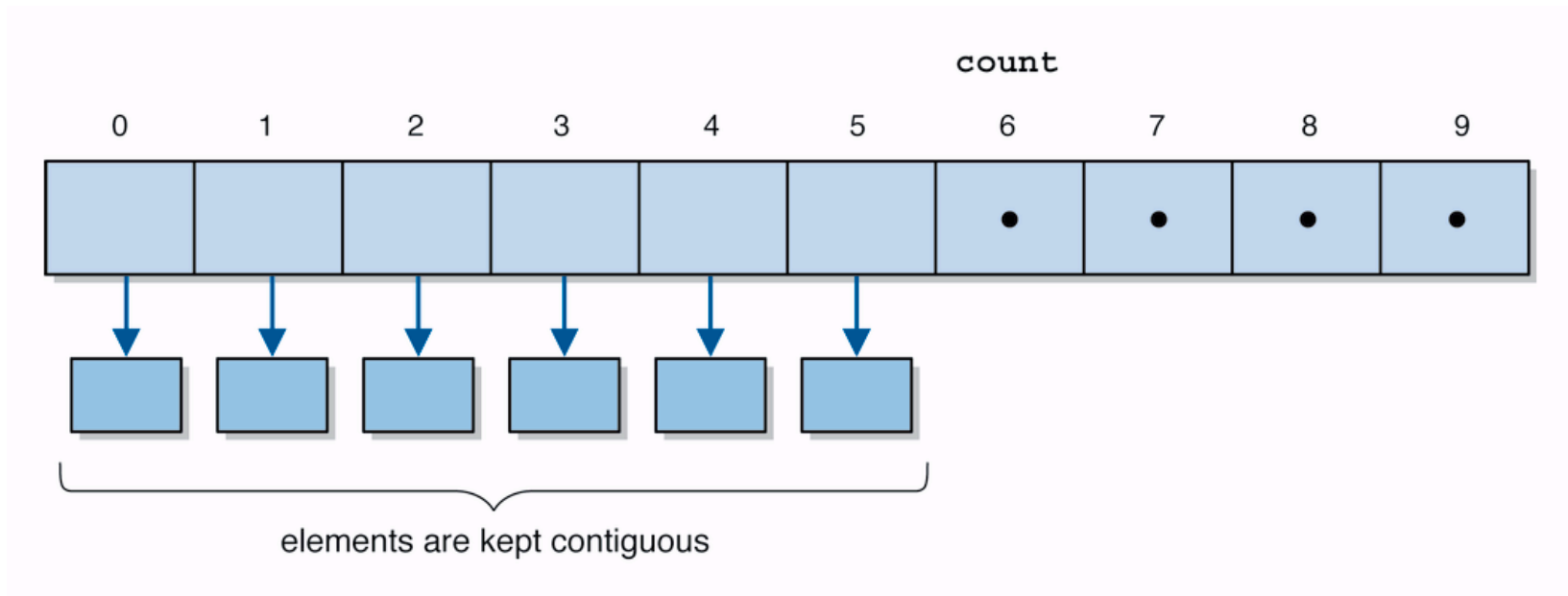


The ArrayBag Class

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

figure 2.9

An array implementation of a bag



ArrayBag Constructors

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

size and isEmpty Operations

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

The contains Operation

```
//-----  
// Returns true if this bag contains the specified target  
// element.  
//-----  
public boolean contains (Object 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);  
}
```

The toString Operation

```
//-----  
// Returns a string representation of this bag.  
//-----  
public String toString()  
{  
    String result = "";  
  
    for (int index=0; index < count; index++)  
        result = result + contents[index].toString() + "\n";  
  
    return result;  
}
```

The add Operation

```
//-----  
// Adds the specified element to the bag, expanding the capacity  
// of the bag array if necessary.  
//-----  
public void add (Object element)  
{  
    if (size() == contents.length)  
        expandCapacity();  
  
    contents[count] = element;  
    count++;  
}
```

Managing Capacity

- An array has a particular number of cells when it is created – its capacity
- So the array's capacity is also the bag's capacity
- What do we do when the bag is 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 `expandCapacity` Method

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

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 bag, 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

Array Iterator Operations

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

package jss2;

import java.util.*;

public class ArrayIterator implements Iterator
{
    private int count;    // the number of elements in the
    collection
    private int current; // the current position in the iteration
    private Object[] items;
}
```

Array Iterator Operations

```
//-----  
// Sets up this iterator using the specified items.  
//-----  
public ArrayIterator (Object[] 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);  
}
```

Array Iterator Operations

```
//-----  
// Returns the next element in the iteration. If there are no  
// more elements in this iteration, a NoSuchElementException is  
// thrown.  
//-----  
public Object 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();  
}
```

Copyright © 2004 Pearson Addison-Wesley. All rights reserved.

The addAll Operation

```
//-----  
// Adds the contents of the parameter to this bag.  
//-----  
public void addAll (ArrayBag bag)  
{  
    Iterator scan = bag.iterator();  
  
    while (scan.hasNext())  
        add (scan.next());  
}
```

The removeRandom Operation

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

The remove Operation

```
//-----  
// Removes one occurrence of the specified element from the bag  
// and returns it. Throws an EmptyBagException if the bag is  
// empty and a NoSuchElementException if the target is not in  
// the bag.  
//-----  
public Object remove (Object target) throws EmptyBagException,  
                                                             NoSuchElementException  
{  
    int search = NOT_FOUND;  
  
    if (isEmpty())  
        throw new EmptyBagException();  
  
    for (int index=0; index < count && search == NOT_FOUND; index++)  
        if (contents[index].equals(target)  
            search = index;
```

remove continued

```
    if (search == NOT_FOUND)
        throw new NoSuchElementException();

    Object result = contents[search];

    contents[search] = contents[count-1];
    contents[count-1] = null;
    count--;

    return result;
}
```

The union Operation

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

The equals Operation

```
//-----  
// Returns true if this bag contains exactly the same elements  
// as the parameter.  
//-----  
public boolean equals (BagADT bag)  
{  
    boolean result = false;  
    ArrayBag temp1 = new ArrayBag();  
    ArrayBag temp2 = new ArrayBag();  
    Object obj;  
  
    if (size() == bag.size())  
    {  
        temp1.addAll(this);  
        temp2.addAll(bag);  
  
        Iterator scan = bag.iterator();
```

equals continued

```
while (scan.hasNext())
{
    obj = scan.next();
    if (temp1.contains(obj))
    {
        temp1.remove(obj);
        temp2.remove(obj);
    }
}

result = (temp1.isEmpty() && temp2.isEmpty());
}

return result;
}
```

The `iterator` Operation

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

See [ArrayIterator.java](#) (page 58)

Analysis of ArrayBag

- If the array is not full, adding an element to the bag 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 bag is $O(n)$
- The union of two bags is $O(n+m)$, where m is the size of the second bag