Advanced Java

Collections

What is a Collection?

An object that groups multiple elements into a single unit

- Stores, retrieves & transmits data from one method to another
- Typically represent data items that form a natural group, a card hand, a mail folder, a telephone directory

The Java Collections Framework

- Provides a basic set of collections
  - Core data structures which are frequently required for coding

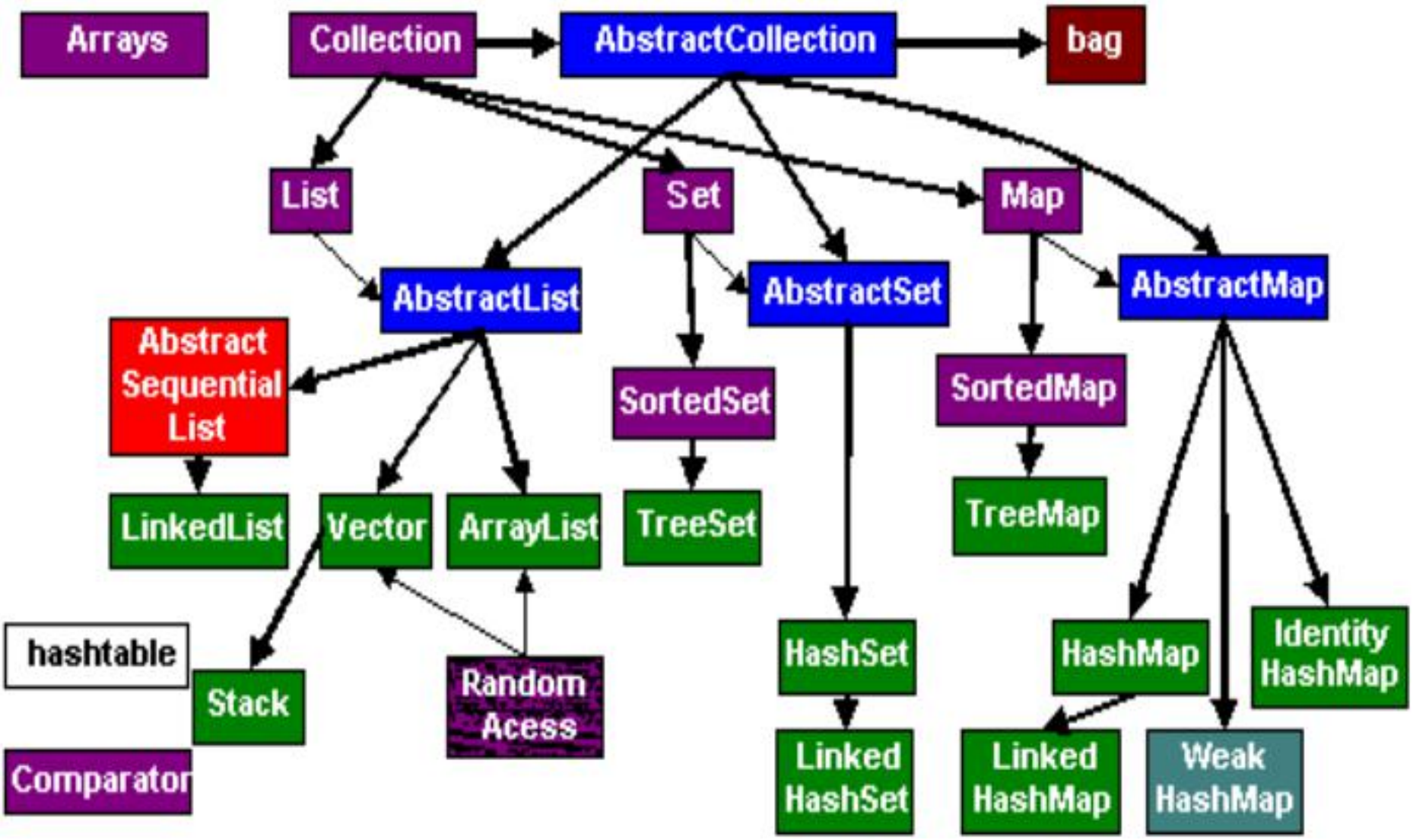
- The Java Collections Framework provides:

Interfaces	Abstract data types representing collections
Implementations	Concrete implementations of the Collection interfaces
Algorithms	Methods that perform useful computations, like searching & sorting on objects which implement Collection interfaces

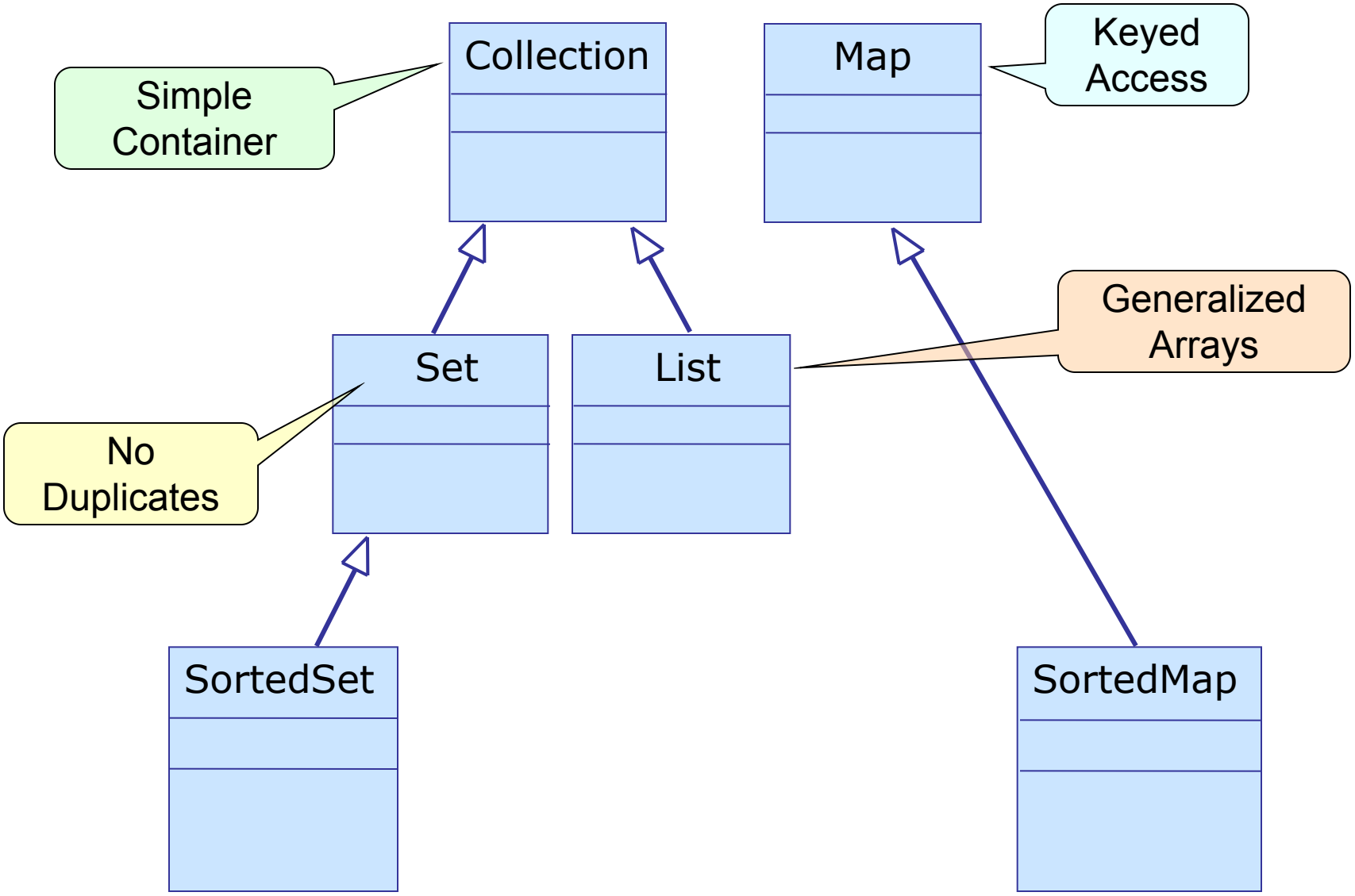
The Java Collections Framework (Contd…)

- Core Interfaces:
  - Collection
  - Set
  - List
  - Map
  - SortedSet
  - SortedMap
- Utility Interfaces:
  - Comparator
  - Iterator
  - Enumerator
- Utility Classes:
  - Collections
  - Arrays

The Java Collections Framework (Contd…)



The Main Interfaces



Collection & Map Interfaces

- The Collection interface is a group of objects, with duplicates allowed
  - *Set* extends *Collection* but forbids duplicates
  - *List* extends *Collection* and allows duplicates and positional indexing

- *Map* extends neither *Set* nor *Collection* and forbids duplicates

Implementation Classes

Interface	Implementation				Historical
	Hash table	Resizable array	Tree (sorted)	Linked list	
Set	HashSet		TreeSet		
List		ArrayList		LinkedList	Vector Stack
Map	HashMap		TreeMap		HashTable Properties

List Implementations

- *ArrayList*
  - A resizable array implementation
  - Unsynchronized
  - Constructed as follows:

Constructor	Description
ArrayList()	Constructs an empty list with an initial capacity of ten
ArrayList(Collection c)	Constructs a list containing the elements of a specified collection, in the order they are returned by the collection's iterator
ArrayList(int initialCapacity)	Constructs an empty list with the specified initial capacity

ArrayListDemo.java

List Implementations (Contd…)

- Vector
  - A resizable array like *ArrayList*
  - Synchronized
  - Constructed as follows:

Constructor	Description
Vector()	Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero
Vector(Collection c)	Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator
Vector(int initialCapacity)	Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero

VectorTest.java

List Implementations

Vector

```
import java.util.Vector;

public class MyVector {

  public static void main(String args[ ]) {

        Vector vecky = new Vector( );

        vecky.add(new Integer(1));
        vecky.add(new Integer(2));
        vecky.add(new Integer(3));

        for(int x=0; x<3; x++) {
           System.out.println(vecky.get(x));
        }
  }
}
```

Back!

List Implementations

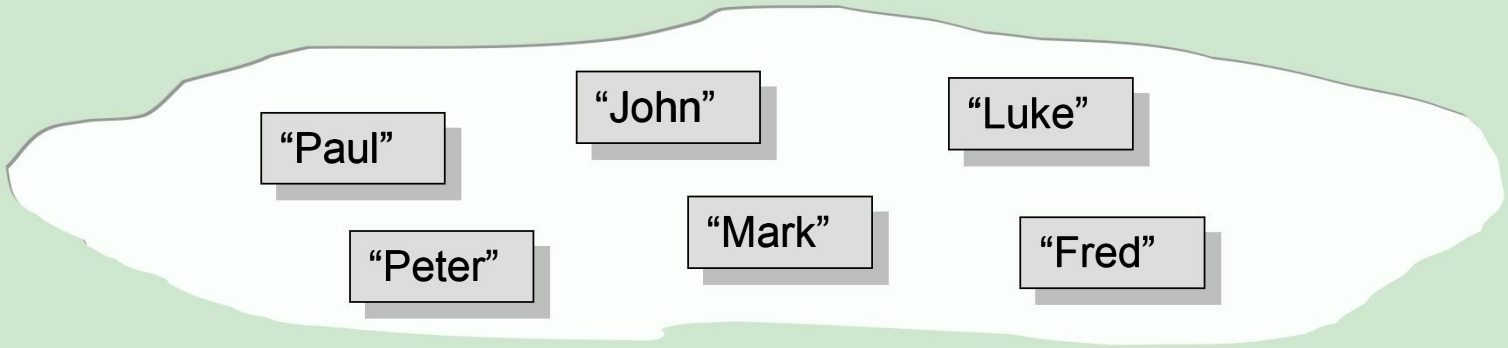
LinkedList

```
1.0
2.0
3.0
```

```
import java.util.LinkedList;

public class MyLinkedList {

  public static void main(String args[ ]) {

        LinkedList link = new LinkedList( );

        link.add(new Double(2.0));
        link.addLast(new Double(3.0));
        link.addFirst(new Double(1.0));

        Object array[ ] = link.toArray( );

        for(int x=0; x<3; x++) {
           System.out.println(array[x]);
        }
  }
}
```

Back!

Set



A Set cares about uniqueness, it doesn’t allow duplicates.

HashSet	LinkedHashSet	TreeSet

Set Implementations

HashSet

```
d
a
c
```

```
import java.util.*;

public class MyHashSet {

  public static void main(String args[ ]) {

        HashSet hash = new HashSet( );

        hash.add("a");
        hash.add("b");
        hash.add("c");
        hash.add("d");

        Iterator iterator = hash.iterator( );

        while(iterator.hasNext( )) {
           System.out.println(iterator.next( ));
        }
  }
}
```

Back!

Set Implementations

LinkedHashSet

```
import java.util.LinkedHashSet;

public class MyLinkedHashSet {

  public static void main(String args[ ]) {

        LinkedHashSet lhs = new LinkedHashSet();

        lhs.add(new String("One"));
        lhs.add(new String("Two"));
        lhs.add(new String("Three"));

        Object array[] = lhs.toArray( );

        for(int x=0; x<3; x++) {
           System.out.println(array[x]);
        }
  }
}
```

```
One
Two
Three
```

Back!

Set Implementations

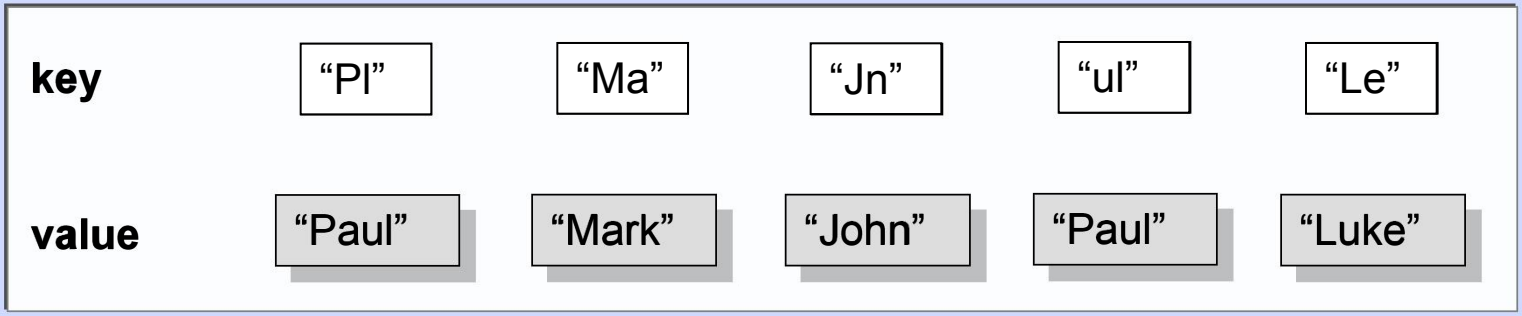
TreeSet

```
import java.util.TreeSet;
import java.util.Iterator;

public class MyTreeSet {

  public static void main(String args[ ]) {

        TreeSet tree = new TreeSet();

        tree.add("Jody");
        tree.add("Remiel");
        tree.add("Reggie");
        tree.add("Philippe");

        Iterator iterator = tree.iterator( );

        while(iterator.hasNext( )) {

  System.out.println(iterator.next( ).toString( ));
        }
  }
```

```
Jody
Philippe
Reggie
Remiel
```

Back!

Map



A Map cares about unique identifiers.

- HashMap
- Hashtable
- LinkedHashMap
- TreeMap

Map Implementations

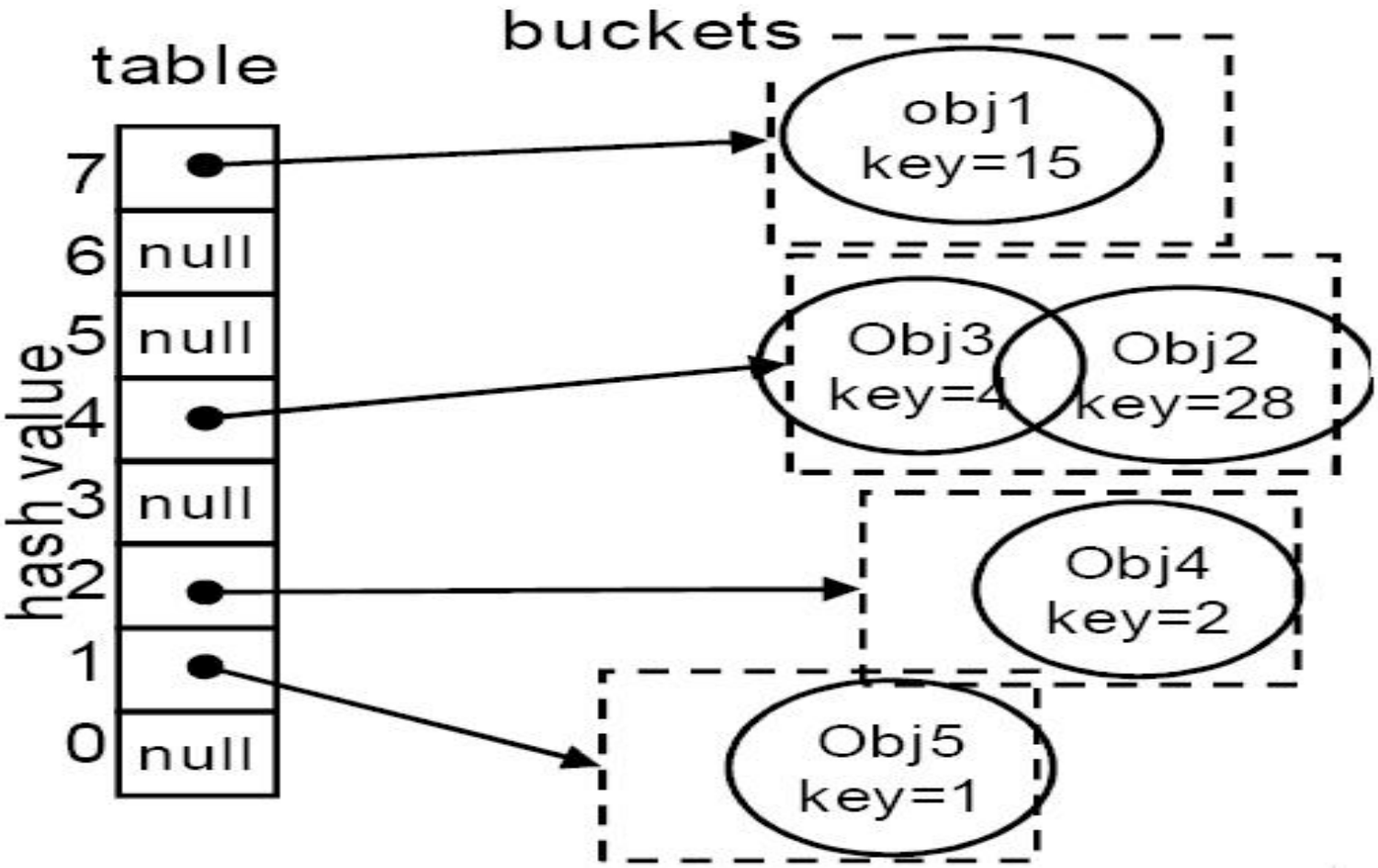
- Hash Table
  - Implementation of a Map interface
  - Synchronized
  - Any non-null object can be used as a key or as a value
  - To store & retrieve objects from a Hashtable, objects are used as keys
  - These must implement the *hashCode()* method and the *equals()* method

Constructor	Description
Hashtable()	Constructs a new, empty hash table with a default initial capacity (11) and load factor 0.75
Hashtable(int initialCapacity)	Constructs a new, empty hash table with the specified initial capacity & default load factor 0.75
Hashtable(int initialCapacity,float loadFactor)	Constructs a new, empty hash table with the specified initial capacity and load factor

HashtableDemo.java (Hashtab

Map Implementations (Contd…)

- Hash Table



Map Implementations (Contd…)

- HashMap
  - A Hash Table implementation of Map
  - Unsynchronized
  - Like Hashtable, but supports null keys & values

Constructor	Description
HashMap()	Constructs an empty HashMap with the default initial capacity 16 and the default load factor 0.75
HashMap(int initialCapacity)	Constructs an empty HashMap with the specified initial capacity and the default load factor 0.75
HashMap(Map m)	Constructs a new HashMap with the same mappings as the specified Map

HashMapDemo.java

Map Implementations

HashMap

```
import java.util.HashMap;

public class MyHashMap {

  public static void main(String args[ ]) {

        HashMap map = new HashMap( );
        map.put("name", "Jody");
        map.put("id", new Integer(446));
        map.put("address", "Manila");

        System.out.println("Name: " + map.get("name"));
        System.out.println("ID: " + map.get("id"));
        System.out.println("Address: " +
  map.get("address"));
  }
}
```

```
Name: Jody
ID: 446
Address: Manila
```

Back!

Map Implementations

Hashtable

```
Table of Contents:
{address=Manila, name=Jody,
  id=1001}
```

```
import java.util.Hashtable;

public class MyHashtable {

  public static void main(String args[ ]) {

              Hashtable table = new
  Hashtable( );
       table.put("name", "Jody");
       table.put("id", new Integer(1001));
       table.put("address", new
  String("Manila"));

       System.out.println("Table of
  Contents:" + table);
  }
}
```

Back!

Map Implementations

LinkedHashMap

```
Jody
446
Manila
Savings
```

```
import java.util.*;

public class MyLinkedHashMap {

  public static void main(String args[ ]) {
        int iNum = 0;
        LinkedHashMap myMap = new LinkedHashMap( );

        myMap.put("name", "Jody");
        myMap.put("id", new Integer(446));
        myMap.put("address", "Manila");
        myMap.put("type", "Savings");

        Collection values = myMap.values( );
        Iterator iterator = values.iterator( );

        while(iterator.hasNext()) {
           System.out.println(iterator.next( ));
        }
  }
}
```

Back!

Map Implementations

TreeMap

```
import java.util.*;

public class MyTreeMap {

  public static void main(String args[]) {

        TreeMap treeMap = new TreeMap( );

        treeMap.put("name", "Jody");
        treeMap.put("id", new Integer(446));
        treeMap.put("address", "Manila");

        Collection values = treeMap.values()
        Iterator iterator = values.iterator( );
        System.out.println("Printing the
  VALUES....");

        while (iterator.hasNext()) {
           System.out.println(iterator.next( ));
        }
  }
```

```
Printing the VALUES....
Manila
446
Jody
```

Back!

21.5 Properties Class

- Properties
  - *Persistent* Hashtable
    - Can be written to output stream
    - Can be read from input stream
  - Provides methods setProperty and getProperty
    - Store/obtain key-value pairs of Strings
    - The **load(Reader) / store(Writer, String)** methods load and store properties from and to a character based stream.
    - The **loadFromXML(InputStream)** and **storeToXML(OutputStream, String, String)** methods load and store properties in a simple XML format.
    - An XML properties document has the following DOCTYPE declaration:
    - <!DOCTYPE properties SYSTEM http://java.sun.com/dtd/properties.dtd>

21.5 Example

```
FileInputStream fis = new FileInputStream("hello.properties");
Properties p = new Properties();
p.load(fis);

Set<Entry<Object, Object>> set = p.entrySet();

for (Entry<Object, Object> entry : set) {
System.out.println(entry.getKey() + "=======" + entry.getValue());
}

System.out.println("a is " + p.getProperty("a"));

p.setProperty("e", "elephant");
p.setProperty("f", "from");
p.setProperty("g", "golu");

p.store(new FileOutputStream("hello.properties", true), null);
```

21.5 Properties Class

emp.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
   "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>This is xml program</comment>
<entry key="java">A programming language</entry>
<entry key="tomcat">java web server</entry>
</properties>
```

```
Properties p = new Properties();
p.loadFromXML(new FileInputStream("emp.xml"));

p.list(System.out);
```

Using Enumeration, Iterator

- Enumeration
  - This interface defines methods by which we can enumerate, i.e. obtain, one at a time, the elements in a collection of objects

VectorDemo.java

- Iterator
  - An iterator over a collection
  - Iterator has superseded Enumeration in the Java collections framework

VectorItr.java

- Iterators differ from Enumerations in two ways:
  - Iterators allow the caller to remove elements from the underlying collection during iteration with well-defined semantics
  - Method names are improved

Stacks, Queues, and Deques

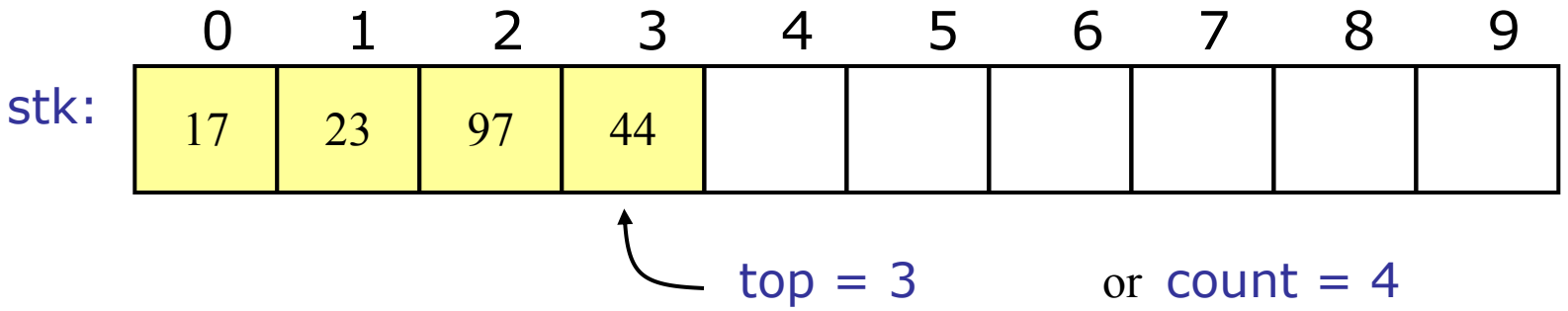
Stacks, Queues, and Deques

- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
- A queue is a first in, first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
- A deque is a double-ended queue—items can be inserted and removed at either end

Array implementation of stacks

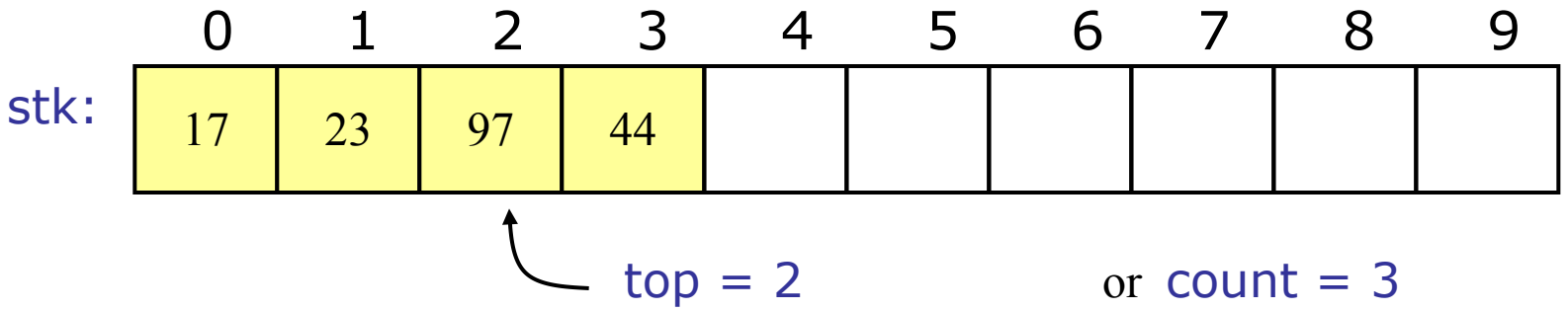
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack

Pushing and popping



- If the bottom of the stack is at location 0, then an empty stack is represented by top = -1 or count = 0
- To add (push) an element, either:
  - Increment top and store the element in stk[top], or
  - Store the element in stk[count] and increment count
- To remove (pop) an element, either:
  - Get the element from stk[top] and decrement top, or
  - Decrement count and get the element in stk[count]

After popping



- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the "deleted" array element to null
  - Why? To allow it to be garbage collected!

- Stack is a subclass of Vector that implements a standard last-in, first-out stack.

- Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

- Stack( )

Methods with Description

1 boolean empty()

Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.

2 Object peek( )

Returns the element on the top of the stack, but does not remove it.

3 Object pop( )

Returns the element on the top of the stack, removing it in the process.

4 Object push(Object element)

Pushes element onto the stack. element is also returned.

5 int search(Object element)

Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned.

```
public class StackDemo {

  static void showpush(Stack st, int a) {
    st.push(new Integer(a));
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }
  static void showpop(Stack st) {
    System.out.print("pop -> ");
    Integer a = (Integer) st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }
  public static void main(String args[]) {
    Stack st = new Stack();
    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("empty stack");
    }
  }
}
```

Queue

- Queue — a collection used to hold multiple elements prior to processing.

- Queue provides additional insertion, extraction, and inspection operations.
- Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner.
- The Queue interface follows.

```
public interface Queue<E> extends Collection<E> {
E element();
boolean offer(E e);
E peek();
E poll();
E remove();
}
```

- Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner.

- Priority queues, which order elements according to their values .
- Whatever ordering is used, the head of the queue is the element that would be removed by a call to remove or poll.
- In a FIFO queue, all new elements are inserted at the tail of the queue.
- Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

countdown time

```
public class Countdown {
    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();

        for (int i = time; i >= 0; i--)
            queue.add(i);

        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

The queue is preloaded with all the integer values in descending order.

Then, the values are removed from the queue and printed at one-second intervals.

Class PriorityQueue<E>

- java.lang.Object
  - java.util.AbstractCollection<E>
    - java.util.AbstractQueue<E>
      - java.util.PriorityQueue<E>

- An unbounded priority queue based on a priority heap.
- The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
- A priority queue does not permit null elements.
- A priority queue relying on natural ordering.

Example

```
public class PriorityQueueDemo {
   public static void main(String args[]) {
      // create priority queue
      PriorityQueue < Integer >  prq = new PriorityQueue < Integer > ();

      // insert values in the queue
      for ( int i = 3; i  <  10; i++ ){
         prq.add (new Integer (i)) ;
      }

      System.out.println ( "Initial priority queue values are: "+ prq);

      // get the head from the queue
      Integer head = prq.poll();

      System.out.println ( "Head of the queue is: "+ head);

      System.out.println ( "Priority queue values after poll: "+ prq);
   }
}
```

Arrays and Collections

Overview

- Arrays
  - Working with arrays
  - Java API support for arrays
- Collection classes
  - Working with Collections

Java Arrays – The Basics

- Declaring an array

```
int[] myArray;
int[] myArray = new int[5];
String[] stringArray = new String[10];
String[] strings = new String[] {“one”, “two”};
```

- Checking an arrays length

```
int arrayLength = myArray.length;
```

- Looping over an array

```
for(int I=0; I<myArray.length; i++)
{
   String s = myArray[i];
}
```

Java Arrays – Bounds Checking

- Bounds checking
  - Java does this automatically.
  - Impossible to go beyond the end of an array (unlike C/C++)
  - Automatically generates an `ArrayIndexOutOfBoundsException`

Java Arrays – Copying

- Don’t copy arrays “by hand” by looping over the array
- The System class has an `arrayCopy` method to do this efficiently

```
int array1[] = new int[10];
int array2[] = new int[10];
//assume we add items to array1

//copy array1 into array2
System.arrayCopy(array1, 0, array2, 0, 10);
//copy last 5 elements in array1 into first 5 of
  array2
System.arrayCopy(array1, 5, array2, 0, 5);
```

Java Arrays – Sorting

- Again no need to do this "by hand".
- The **java.util.Arrays** class has methods to sort different kinds of arrays

```
int myArray[] = new int[] {5, 4, 3, 2, 1};
java.util.Arrays.sort(myArray);
//myArray now holds 1, 2, 3, 4, 5
```

- Sorting arrays of *objects* is involves some extra work, as we'll see later…

Java Arrays – BinarySearch()

- Again no need to do this "by hand".
- The **java.util.Arrays** class has methods to search

```
int myArray[] = new int[] {5, 4, 3, 2, 1};
java.util.Arrays.binarySearch(myArray,pos);
```

- BinarySearch() Searches the specified array for the specified value using the binary search algorithm.
- The array must be sorted prior to making this call.
- For unsorted arrays, the results are undefined.

- Returns:
- index of the search key, if it is contained in the array; otherwise, (-(insertion point) - 1)

Java Arrays

- Advantages
  - Very efficient, quick to access and add to
  - Type-safe, can only add items that match the declared type of the array
- Disadvantages
  - Fixed size, some overhead in copying/resizing
  - Can’t tell how many items in the array, just how large it was declared to be
  - Limited functionality, need more general functionality

Collections – Other Functions

- The **`java.util.Collections`** class has many useful methods for working with collections
  - min, max, sort, reverse, search, shuffle

- Virtually all require your objects to implement an extra interface, called `Comparable`.

Give this a Try…

1. Which class extends Collection to handle list of objects in synchronized manner?

1. Set allows duplicate elements. State True / False

Generics

- It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array or an array of any type that supports ordering.
- Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- Using Java Generic concept we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

- We can write a single generic method declaration that can be called with arguments of different types.
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately

- Following are the rules to define Generic Methods:
  - All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

  - Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

Generic Methods

- Following are the rules to define Generic Methods:
  - The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
  - A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types not primitive types (like int, double and char).

Generics

```
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E [ ] inputArray )   {
      // Display array elements
         for ( E element : inputArray ) {
            System.out.printf( "%s ", element );
         }
}
public static void main( String args[ ] )    {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array
        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array
        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // pass a Character array    }
}
```

Generic Classes

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section

- As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas.

- These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Generic Classes

```
public class Box<T> {
  private T t;
  public void add(T t) {
    this.t = t;
  }
  public T get() {
    return t;
  }

  public static void main(String[] args) {
     Box<Integer> integerBox = new Box<Integer>();
     Box<String> stringBox = new Box<String>();

     integerBox.add(new Integer(10));
     stringBox.add(new String("Hello World"));

     System.out.printf("Integer Value :%d\n\n", integerBox.get());
     System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

Summary

In this session, we have covered:

- Introduction to Collection Framework
- Types of Collections
- Using collections (*ArrayList, Vector, Hashtable, HashMap*)