Arrays and Legacy Collections

Arrays Class

- Arrays class contains various methods for manipulating arrays (such as sorting and searching).
- This class also contains a static factory that allows arrays to be viewed as lists.
- The methods in this class all throw a NullPointerException, if the specified array reference is null.

Arrays Class

```
// Demonstrate Arrays
import java.util.*;
class ArraysDemo {
public static void main(String args[]) {
// Allocate and initialize array.
int array[] = new int[10];
for(int i = 0; i < 10; i++)
array[i] = -3 * i;
// Display, sort, and display the array.
System.out.print("Original contents:");
display(array);
Arrays.sort(array);
System.out.print("Sorted: ");
display(array);
// Fill and display the array.
Arrays.fill(array, 2, 6, -1);
System.out.print("After fill(): ");
display(array);
```

```
// Sort and display the array.
Arrays.sort(array);
System.out.print("After sorting again: ");
display(array);
// Binary search for -9.
System.out.print("The value -9 is at location ");
int index =
Arrays.binarySearch(array, -9);
System.out.println(index);
}
static void display(int array[]) {
for(int i: array)
System.out.print(i + " ");
System.out.println();
}
}
```

The Legacy Classes and Interfaces

- Early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects.
- When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces. Thus, they are now technically part of the Collections Framework.
- However, where a modern collection duplicates the functionality of a legacy class, you will usually want to use the newer collection class. In general, the legacy classes are supported because there is still code that uses them.
- The legacy classes defined by **java.util** are shown below:

Dictionary	Hashtable	Properties	Stack	Vector

- There is one legacy interface called **Enumeration.**

The Enumeration Interface

- The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects.
- This legacy interface has been superseded by **Iterator**. Although not deprecated, **Enumeration** is considered obsolete for new code.
- However, it is used by several methods defined by the legacy classes (such as **Vector** and **Properties**) and is used by several other API classes.

It has the following declaration:

```
interface Enumeration<E>
where E specifies the type of element being enumerated.
```

Enumeration specifies the following two methods:

```
boolean hasMoreElements( )
E nextElement( )
```

- It throws NoSuchElementException when the enumeration is complete.

Vector

```
Exp: // Demonstrate various Vector operations.
import java.util.*;
class VectorDemo {
public static void main(String args[]) {
// initial size is 3, increment is 2
Vector<Integer> v = new Vector<Integer>(3, 2);
System.out.println("Initial size: " + v.size());
System.out.println("Initial capacity: " +
v.capacity());
v.addElement(1); v.addElement(2);
v.addElement(3); v.addElement(4);
System.out.println("Capacity after four
additions: " + v.capacity());
v.addElement(5);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(6); v.addElement(7);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(9); v.addElement(10);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(11); v.addElement(12);
System.out.println("First element: " +
v.firstElement());
System.out.println("Last element: " +
v.lastElement());
if(v.contains(3))
System.out.println("Vector contains 3.");
// Enumerate the elements in the vector.
Enumeration<Integer> vEnum =
v.elements();
System.out.println("\nElements in
vector:"); while(vEnum.hasMoreElements())
System.out.print(vEnum.nextElement() + "
");
System.out.println();
}
}
```

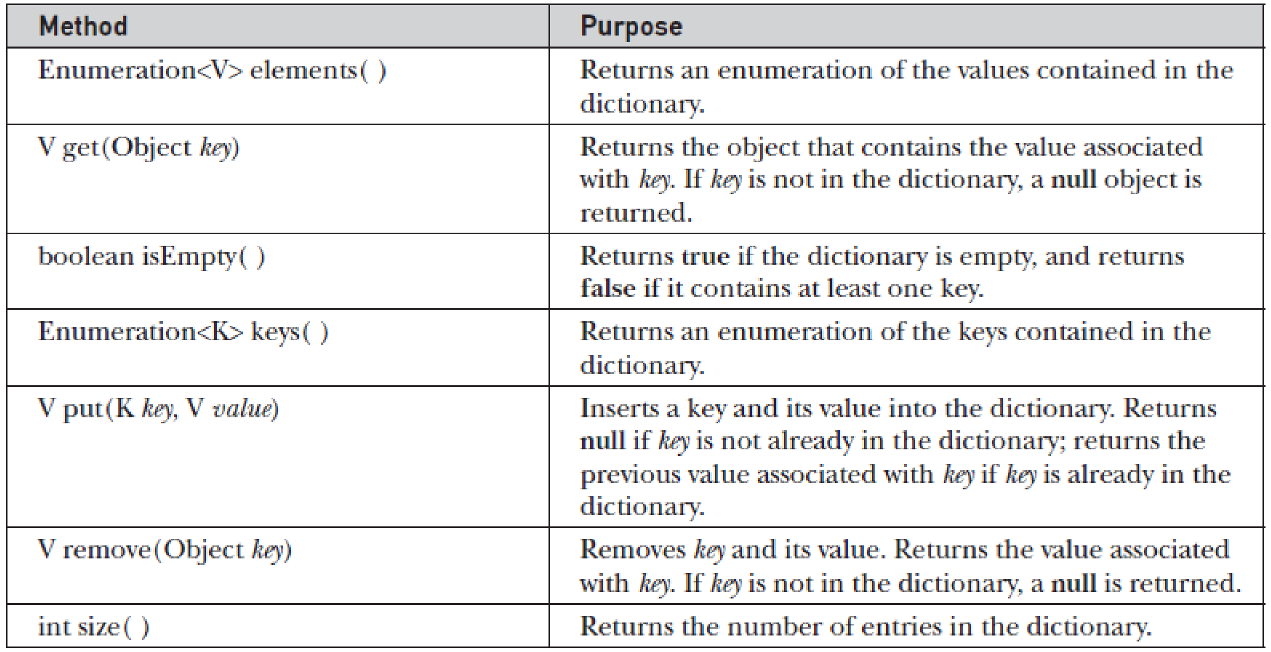
Stack

```
// Demonstrate the Stack class.
import java.util.*;
class StackDemo {
static void showpush(Stack<Integer> st, int
a) {
st.push(a);
System.out.println("push(" + a + ")");
System.out.println("stack: " + st);
}
static void showpop(Stack<Integer> st) {
System.out.print("pop -> ");
Integer a = st.pop();
System.out.println(a);
System.out.println("stack: " + st);
}
```

```
public static void main(String args[]) {
Stack<Integer> st = new Stack<Integer>();
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");
}
}
}
```

Dictionary

- Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
- Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.
- **Note:** Although not currently deprecated, Dictionary is classified as obsolete, because it is fully superseded by Map. However, Dictionary is still in use.



Method	Purpose
Enumeration<V> elements( )	Returns an enumeration of the values contained in the dictionary.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the dictionary, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the dictionary is empty, and returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the dictionary.
V put(K *key*, V *value*)	Inserts a key and its value into the dictionary. Returns **null** if *key* is not already in the dictionary; returns the previous value associated with *key* if *key* is already in the dictionary.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the dictionary, a **null** is returned.
int size( )	Returns the number of entries in the dictionary.

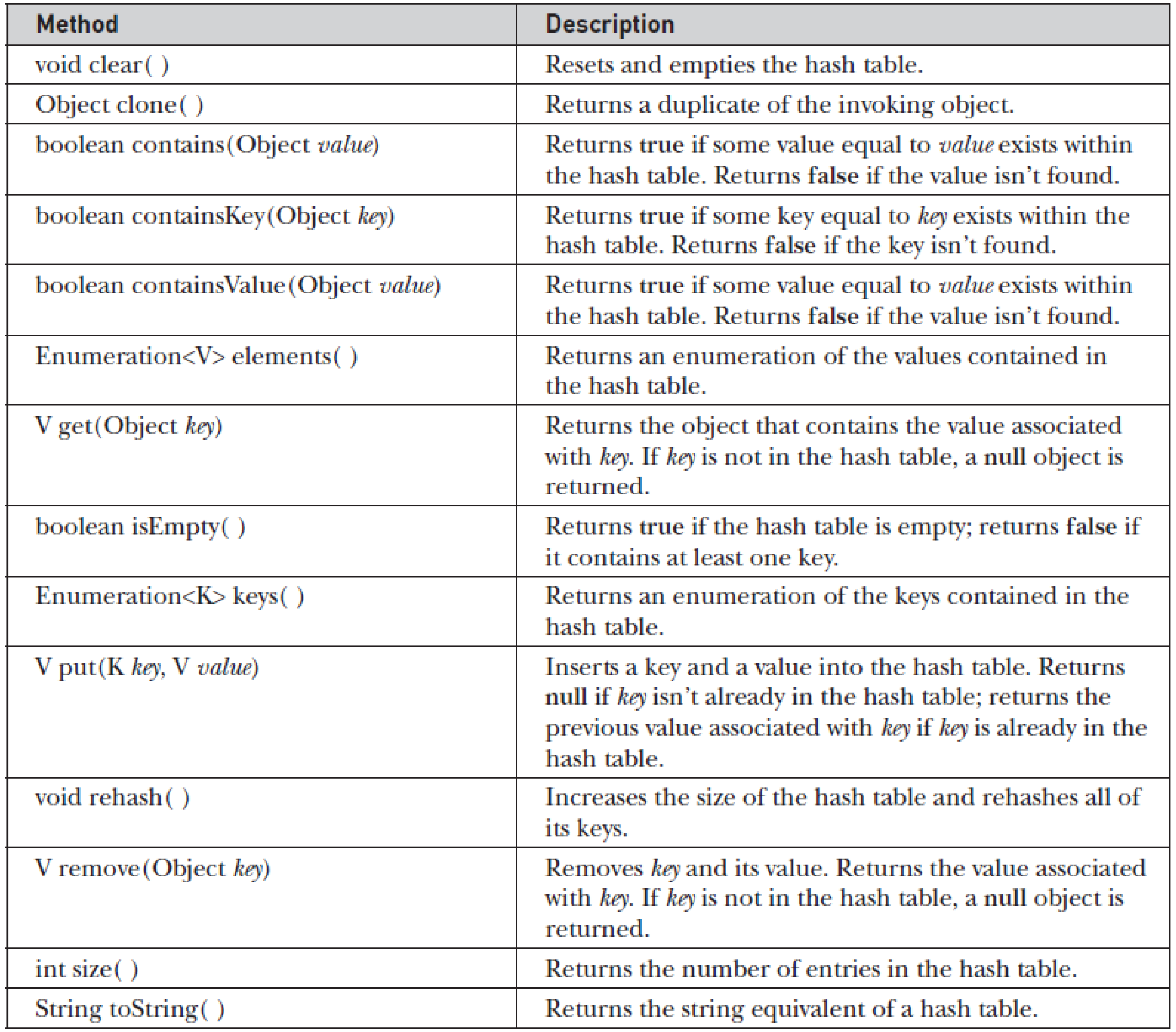
Hashtable

- Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
- However, with the advent of collections, Hashtable was reengineered to also implement the Map interface.
- Thus, Hashtable is integrated into the Collections Framework. It is similar to HashMap, but is synchronized.
- Like HashMap, Hashtable stores key/value pairs in a hash table. However, neither keys nor values can be null.
- When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key.
- The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

class Hashtable<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

Hashtable



Method	Description
void clear( )	Resets and empties the hash table.
Object clone( )	Returns a duplicate of the invoking object.
boolean contains(Object *value*)	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
boolean containsKey(Object *key*)	Returns **true** if some key equal to *key* exists within the hash table. Returns **false** if the key isn't found.
boolean containsValue(Object *value*)	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
Enumeration<V> elements( )	Returns an enumeration of the values contained in the hash table.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the hash table is empty; returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the hash table.
V put(K *key*, V *value*)	Inserts a key and a value into the hash table. Returns **null** if *key* isn't already in the hash table; returns the previous value associated with *key* if *key* is already in the hash table.
void rehash( )	Increases the size of the hash table and rehashes all of its keys.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
int size( )	Returns the number of entries in the hash table.
String toString( )	Returns the string equivalent of a hash table.

Hashtable

```
// Demonstrate a Hashtable.
import java.util.*;
class HTDemo {
public static void main(String
args[]) {
Hashtable<String, Double> balance =
new Hashtable<String, Double>();
Enumeration<String> names;
String str; double bal;
balance.put("John Doe", 3434.34);
balance.put("Tom Smith", 123.22);
balance.put("Jane Baker", 1378.00);
balance.put("Tod Hall", 99.22);
balance.put("Ralph Smith", -19.08);
// Show all balances in hashtable.
names = balance.keys();
while(names.hasMoreElements()) {
str = names.nextElement();
System.out.println(str + ": " +
balance.get(str)); }
System.out.println();
// Deposit 1,000 into John Doe's
account.
bal = balance.get("John Doe");
balance.put("John Doe", bal+1000);
System.out.println("John Doe's new
balance: " +
balance.get("John Doe"));
}
}
```

References

- http://java.sun.com/docs/books/tutorial/collections/interfaces/collection.html.