CHP-27
Java Data Structures

The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes:

- Enumeration
- BitSet
- Vector
- Stack
- Dictionary
- Hashtable
- Properties

All these classes are now legacy and Java-2 has introduced a new framework called Collections Framework, which is discussed in next tutorial:

The Enumeration:

The Enumeration interface isn't itself a data structure, but it is very important within the context of other data structures. The Enumeration interface defines a means to retrieve successive elements from a data structure.
For example, Enumeration defines a method called nextElement that is used to get the next element in a data structure that contains multiple elements.
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This legacy interface has been superceded by Iterator.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, is used by several other API classes, and is currently in widespread use in application code.
The methods declared by Enumeration are summarized in the following table:

CHAPTER

SN Methods with Description

1 **boolean hasMoreElements( )**
When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.

2 **Object nextElement( )**
This returns the next object in the enumeration as a generic Object reference.

Example:

Following is the example showing usage of Enumeration.

```
import java.util.Vector;
import java.util.Enumeration;
public class EnumerationTester{
```

```
public static void main(String args[]){
Enumeration days;
Vector dayNames =newVector();
dayNames.add("Sunday");
dayNames.add("Monday");
dayNames.add("Tuesday");
dayNames.add("Wednesday");
dayNames.add("Thursday");
dayNames.add("Friday");
dayNames.add("Saturday");
days = dayNames.elements();
while(days.hasMoreElements()){
System.out.println(days.nextElement());
}
}
}
```

This would produce the following result:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

The BitSet

The BitSet class implements a group of bits or flags that can be set and cleared individually.
This class is very useful in cases, where you need to keep up with a set of Boolean values; you just assign a bit to each value and set or clear it as appropriate.
A BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits.
This is a legacy class but it has been completely re-engineered in Java 2, version 1.4.
The BitSet defines two constructors. The first version creates a default object:


```
BitSet()
```

The second version allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero.

```
BitSet(int size)
```

BitSet implements the Cloneable interface and defines the methods listed in table below:

SN Methods with Description

1
void and(BitSet bitSet)
ANDs the contents of the invoking BitSet object with those specified by bitSet. The result is placed into the invoking object.

2 **void andNot(BitSet bitSet)**
For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.

3 **int cardinality( )**
Returns the number of set bits in the invoking object.

4 **void clear( )**
Zeros all bits.

5 **void clear(int index)**
Zeros the bit specified by index.

6 **void clear(int startIndex, int endIndex)**
Zeros the bits from startIndex to endIndex.1.

7 **Object clone( )**
Duplicates the invoking BitSet object.

8
boolean equals(Object bitSet)
Returns true if the invoking bit set is equivalent to the one passed in bitSet. Otherwise, the method returns false.

9 **void flip(int index)**
Reverses the bit specified by index. (

10 **void flip(int startIndex, int endIndex)**
Reverses the bits from startIndex to endIndex.1.
11 **boolean get(int index)**
Returns the current state of the bit at the specified index.
12
BitSet get(int startIndex, int endIndex)
Returns a BitSet that consists of the bits from startIndex to endIndex.1. The invoking object is not changed.
13 **int hashCode( )**
Returns the hash code for the invoking object.
14 **boolean intersects(BitSet bitSet)**
Returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
15 **boolean isEmpty( )**
Returns true if all bits in the invoking object are zero.
16 **int length( )**
Returns the number of bits required to hold the contents of the invoking BitSet. This value is

determined by the location of the last 1 bit.
17
int nextClearBit(int startIndex)
Returns the index of the next cleared bit, (that is, the next zero bit), starting from the index specified by startIndex
18
int nextSetBit(int startIndex)
Returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, .1 is returned.
19
void or(BitSet bitSet)
ORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.
20 **void set(int index)**
Sets the bit specified by index.
21 **void set(int index, boolean v)**
Sets the bit specified by index to the value passed in v. true sets the bit, false clears the bit.
22 **void set(int startIndex, int endIndex)**
Sets the bits from startIndex to endIndex.1.
23
void set(int startIndex, int endIndex, boolean v)
Sets the bits from startIndex to endIndex.1, to the value passed in v. true sets the bits, false clears the bits.
24 **int size( )**
Returns the number of bits in the invoking BitSet object.
25 **String toString( )**
Returns the string equivalent of the invoking BitSet object.
26
void xor(BitSet bitSet)
XORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object

Example:

The following program illustrates several of the methods supported by this data structure:

```
import java.util.BitSet;
public class BitSetDemo{
public static void main(String args[]){
BitSet bits1 =new BitSet(16);
BitSet bits2 =new BitSet(16);
// set some bits
for(int i=0; i<16; i++){
if((i%2)==0) bits1.set(i);
if((i%5)!=0) bits2.set(i);
}
System.out.println("Initial pattern in bits1: ");
```

```
System.out.println(bits1);
System.out.println("\nInitial pattern in bits2: ");
System.out.println(bits2);
// AND bits
bits2.and(bits1);
```



```
System.out.println("\nbits2 AND bits1: ");
System.out.println(bits2);
// OR bits
bits2.or(bits1);
System.out.println("\nbits2 OR bits1: ");
System.out.println(bits2);
// XOR bits
bits2.xor(bits1);
System.out.println("\nbits2 XOR bits1: ");
System.out.println(bits2);
}
}
```

This would produce the following result:

```
Initial pattern in bits1:
{0,2,4,6,8,10,12,14}
Initial pattern in bits2:
{1,2,3,4,6,7,8,9,11,12,13,14}
bits2 AND bits1:
{2,4,6,8,12,14}
bits2 OR bits1:
{0,2,4,6,8,10,12,14}
bits2 XOR bits1:
{}
```

The Vector

The Vector class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements.

Like an array, elements of a Vector object can be accessed via an index into the vector.

The nice thing about using the Vector class is that you don't have to worry about setting it to a specific size upon creation; it shrinks and grows automatically when necessary.

Vector implements a dynamic array. It is similar to ArrayList, but with two differences:

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

The Vector class supports four constructors. The first form creates a default vector, which has an initial size of 10:

```
Vector()
```

The second form creates a vector whose initial capacity is specified by size:


```
Vector(int size)
```

The third form creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward:

```
Vector(int size,int incr)
```

The fourth form creates a vector that contains the elements of collection c:

```
Vector(Collection c)
```

Apart from the methods inherited from its parent classes, Vector defines the following methods:

SN Methods with Description

1 **void add(int index, Object element)**

Inserts the specified element at the specified position in this Vector.

2 **boolean add(Object o)**

Appends the specified element to the end of this Vector.

3

boolean addAll(Collection c)

Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.

4 **boolean addAll(int index, Collection c)**
Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5 **void addElement(Object obj)**
Adds the specified component to the end of this vector, increasing its size by one.
6 **int capacity()**
Returns the current capacity of this vector.
7 **void clear()**
Removes all of the elements from this Vector.
8 **Object clone()**
Returns a clone of this vector.
9 **boolean contains(Object elem)**
Tests if the specified object is a component in this vector.
10 **boolean containsAll(Collection c)**
Returns true if this Vector contains all of the elements in the specified Collection.
11 **void copyInto(Object[] anArray)**
Copies the components of this vector into the specified array.
12 **Object elementAt(int index)**
Returns the component at the specified index.
13 **Enumeration elements()**
Returns an enumeration of the components of this vector.
14
void ensureCapacity(int minCapacity)
Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
15 **boolean equals(Object o)**
Compares the specified Object with this Vector for equality.

16 **Object firstElement()**
Returns the first component (the item at index 0) of this vector.
17 **Object get(int index)**
Returns the element at the specified position in this Vector.
18 **int hashCode()**
Returns the hash code value for this Vector.
19
int indexOf(Object elem)
Searches for the first occurence of the given argument, testing for equality using the equals method.
20
int indexOf(Object elem, int index)
Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
21 **void insertElementAt(Object obj, int index)**
Inserts the specified object as a component in this vector at the specified index.
22 **boolean isEmpty()**
Tests if this vector has no components.
23 **Object lastElement()**
Returns the last component of the vector.
24 **int lastIndexOf(Object elem)**
Returns the index of the last occurrence of the specified object in this vector.
25
int lastIndexOf(Object elem, int index)
Searches backwards for the specified object, starting from the specified index, and returns an index to it.
26 **Object remove(int index)**
Removes the element at the specified position in this Vector.
27
boolean remove(Object o)
Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
28 **boolean removeAll(Collection c)**
Removes from this Vector all of its elements that are contained in the specified Collection.
29 **void removeAllElements()**

Removes all components from this vector and sets its size to zero.
30 **boolean removeElement(Object obj)**
Removes the first (lowest-indexed) occurrence of the argument from this vector.
31 **void removeElementAt(int index)**
removeElementAt(int index)
32
protected void removeRange(int fromIndex, int toIndex)
Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
33 **boolean retainAll(Collection c)**
Retains only the elements in this Vector that are contained in the specified Collection.
34 **Object set(int index, Object element)**
Replaces the element at the specified position in this Vector with the specified element.

35 **void setElementAt(Object obj, int index)**
Sets the component at the specified index of this vector to be the specified object.
36 **void setSize(int newSize)**
Sets the size of this vector.
37 **int size()**
Returns the number of components in this vector.
38 **List subList(int fromIndex, int toIndex)**
Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
39 **Object[] toArray()**
Returns an array containing all of the elements in this Vector in the correct order.
40
Object[] toArray(Object[] a)
Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
41
String toString()
Returns a string representation of this Vector, containing the String representation of each element.
42 **void trimToSize()**
Trims the capacity of this vector to be the vector's current size.

Example:

The following program illustrates several of the methods supported by this collection:

```
import java.util.*;
public class VectorDemo{
public static void main(String args[]){
// initial size is 3, increment is 2
Vector v =new Vector(3,2);
System.out.println("Initial size: "+ v.size());
System.out.println("Initial capacity: "+v.capacity());
v.addElement(newInteger(1));
v.addElement(new Integer(2));
v.addElement(new Integer(3));
v.addElement(new Integer(4));
System.out.println("Capacity after four additions: "+v.capacity());
v
.addElement(new Double(5.45));
System.out.println("Current capacity: "+v.capacity());
v.addElement(new Double(6.08));
v.addElement(new Integer(7));
System.out.println("Current capacity: "+v.capacity());
v.addElement(new Float(9.4));
v.addElement(new Integer(10));
System.out.println("Current capacity: "+v.capacity());
v.addElement(new Integer(11));
v.addElement(new Integer(12));
System.out.println("First element: "+(Integer)v.firstElement());
System.out.println("Last element: "+(Integer)v.lastElement());
```

```
if(v.contains(new Integer(3)))
System.out.println("Vector contains 3.");
// enumerate the elements in the vector.
Enumeration vEnum = v.elements();
```

```
System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
System.out.print(vEnum.nextElement()+" ");
System.out.println();
}
}
```
This would produce the following result:
```
Initial size:0
Initial capacity:3
Capacity after four additions:5
Current capacity:5
Current capacity:7
Current capacity:9
First element:1
Last element:12
Vector contains 3.
Elements in vector:
12345.456.0879.4101112
```

The Stack

The Stack class implements a last-in-first-out (LIFO) stack of elements.
You can think of a stack literally as a vertical stack of objects; when you add a new element, it gets stacked on top of the others.
When you pull an element off the stack, it comes off the top. In other words, the last element you added to the stack is the first one to come back off.
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()
```
Apart from the methods inherited from its parent class Vector, Stack defines the following methods:

SN 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.

Example:

The following program illustrates several of the methods supported by this collection:
```
import java.util.*;
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");
}
}
}
```

This would produce the following result:

```
stack:[]
push(42)
stack:[42]
push(66)
stack:[42,66]
push(99)
stack:[42,66,99]
pop ->99
stack:[42,66]
pop ->66
stack:[42]
pop ->42
stack:[]
pop -> empty stack
```


The Dictionary

The Dictionary class is an abstract class that defines a data structure for mapping keys to values.
This is useful in cases where you want to be able to access data via a particular key rather than an integer index.
Since the Dictionary class is abstract, it provides only the framework for a key-mapped data structure rather than a specific implementation.
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.
The abstract methods defined by Dictionary are listed below:

SN Methods with Description

1 **Enumeration elements( )**
Returns an enumeration of the values contained in the dictionary.

2
Object 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.

3 **boolean isEmpty( )**
Returns true if the dictionary is empty, and returns false if it contains at least one key.

4 **Enumeration keys( )**
Returns an enumeration of the keys contained in the dictionary.

5
Object put(Object key, Object 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.

6

Object 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.
7 **int size( )**
Returns the number of entries in the dictionary.
The Dictionary class is obsolete. You should implement the **Map interface** to obtain key/value storage functionality.

Map Interface

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
- Several methods throw a NoSuchElementException when no items exist in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.

- A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
- An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

SN Methods with Description
1 **void clear( )**
Removes all key/value pairs from the invoking map.
2 **boolean containsKey(Object k)**
Returns true if the invoking map contains k as a key. Otherwise, returns false.
3 **boolean containsValue(Object v)**
Returns true if the map contains v as a value. Otherwise, returns false.
4
Set entrySet( )
Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.
5 **boolean equals(Object obj)**
Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
6 **Object get(Object k)**
Returns the value associated with the key k.
7 **int hashCode( )**
Returns the hash code for the invoking map.
8 **boolean isEmpty( )**
Returns true if the invoking map is empty. Otherwise, returns false.
9
Set keySet( )
Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
10
Object put(Object k, Object v)
Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11 **void putAll(Map m)**
Puts all the entries from m into this map.
12 **Object remove(Object k)**
Removes the entry whose key equals k.
13 **int size( )**
Returns the number of key/value pairs in the map.
14
Collection values( )
Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Example:

Map has its implementation in various classes like HashMap, Following is the example to explain map functionality:

```
import java.util.*;
public class CollectionsDemo{
```

```
public static void main(String[] args){
Map m1 =new HashMap();
m1.put("Zara","8");
m1.put("Mahnaz","31");
m1.put("Ayan","12");
m1.put("Daisy","14");
System.out.println();
System.out.println(" Map Elements");
System.out.print("\t"+ m1);
}
}
```

This would produce the following result:

```
MapElements
{Mahnaz=31,Ayan=12,Daisy=14,Zara=8}
```

The Hashtable

The Hashtable class provides a means of organizing data based on some user-defined key structure.
For example, in an address list hash table you could store and sort data based on a key such as ZIP code rather than on a person's name.
The specific meaning of keys in regard to hashtables is totally dependent on the usage of the hashtable and the data it contains.
Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
However, Java 2 reengineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.
Like HashMap, Hashtable stores key/value pairs in a hashtable. 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.
The Hashtable defines four constructors. The first version is the default constructor:

```
Hashtable()
```

The second version creates a hashtable that has an initial size specified by size:

```
Hashtable(int size)
```

The third version creates a hashtable that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hashtable can be before it is resized upward.

```
Hashtable(int size,float fillRatio)
```

The fourth version creates a hashtable that is initialized with the elements in m.
The capacity of the hashtable is set to twice the number of elements in m. The default load factor of 0.75 is used.

```
Hashtable(Map m)
```

Apart from the methods defined by Map interface, Hashtable defines the following methods:

SN Methods with Description

1 **void clear( )**
Resets and empties the hash table.

2 **Object clone( )**
Returns a duplicate of the invoking object.

3
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.

4
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.

5
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.

6 **Enumeration elements( )**
Returns an enumeration of the values contained in the hash table.

7
Object 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.

8 **boolean isEmpty( )**
Returns true if the hash table is empty; returns false if it contains at least one key.
9 **Enumeration keys( )**
Returns an enumeration of the keys contained in the hash table.
10
Object put(Object key, Object 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.
11 **void rehash( )**
Increases the size of the hash table and rehashes all of its keys.
12
Object 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.
13 **int size( )**
Returns the number of entries in the hash table.
14 **String toString( )**
Returns the string equivalent of a hash table.

Example:

The following program illustrates several of the methods supported by this data structure:

```
import java.util.*;
public class HashTableDemo{
public static void main(String args[]){
// Create a hash map
Hashtable balance =new Hashtable();
```



```
Enumeration names;
String str;
double bal;
balance.put("Zara",new Double(3434.34));
balance.put("Mahnaz",new Double(123.22));
balance.put("Ayan",new Double(1378.00));
balance.put("Daisy",new Double(99.22));
balance.put("Qadir",new Double(-19.08));
// Show all balances in hash table.
names = balance.keys();
while(names.hasMoreElements()){
str =(String) names.nextElement();
System.out.println(str +": "+balance.get(str));
}
System.out.println();
// Deposit 1,000 into Zara's account
bal =((Double)balance.get("Zara")).doubleValue();
balance.put("Zara",new Double(bal+1000));
System.out.println("Zara's new balance: "+balance.get("Zara"));
}
}
```

This would produce the following result:

```
Qadir:-19.08
Zara:3434.34
Mahnaz:123.22
Daisy:99.22
Ayan:1378.0
Zara's new balance: 4434.34
```

The Properties

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.
Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value

is also a String.
The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.
Properties define the following instance variable. This variable holds a default property list associated with a Properties object.

```
Properties defaults;
```

The Properties define two constructors. The first version creates a Properties object that has no default values:

```
Properties()
```

The second creates an object that uses propDefault for its default values. In both cases, the property list is empty:


```
Properties(Properties propDefault)
```

Apart from the methods defined by Hashtable, Properties define the following methods:

SN Methods with Description

1
String getProperty(String key)
Returns the value associated with key. A null object is returned if key is neither in the list nor in the default property list.

2
String getProperty(String key, String defaultProperty)
Returns the value associated with key. defaultProperty is returned if key is neither in the list nor in the default property list.

3 **void list(PrintStream streamOut)**
Sends the property list to the output stream linked to streamOut.

4 **void list(PrintWriter streamOut)**
Sends the property list to the output stream linked to streamOut.

5 **void load(InputStream streamIn) throws IOException**
Inputs a property list from the input stream linked to streamIn.

6
Enumeration propertyNames( )
Returns an enumeration of the keys. This includes those keys found in the default property list, too.

7
Object setProperty(String key, String value)
Associates value with key. Returns the previous value associated with key, or returns null if no such association exists.

8
void store(OutputStream streamOut, String description)
After writing the string specified by description, the property list is written to the output stream linked to streamOut.

Example:

The following program illustrates several of the methods supported by this data structure:

```
import java.util.*;
public class PropDemo{
public static void main(String args[]){
Properties capitals =new Properties();
Set states;
String str;
capitals.put("Illinois","Springfield");
capitals.put("Missouri","Jefferson City");
capitals.put("Washington","Olympia");
capitals.put("California","Sacramento");
capitals.put("Indiana","Indianapolis");
// Show all states and capitals in hashtable.
states = capitals.keySet();// get set-view of keys
Iterator itr = states.iterator();
while(itr.hasNext()){
str =(String) itr.next();
System.out.println("The capital of "+str +" is "+capitals.getProperty(str)+".");
}
```



```
System.out.println();
```

```
// look for state not in list -- specify default
str = capitals.getProperty("Florida","Not Found");
System.out.println("The capital of Florida is "+ str +".");
}
}
```

This would produce the following result:

```
The capital of Missouri is JeffersonCity.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.
The capital of Florida is NotFound.
```

CHP-28
Java Collections

Priorto Java 2, Java provided ad hoc classes such as **Dictionary, Vector, Stack**, and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used Vector was different from the way that you used Properties.

The collections framework was designed to meet several goals.

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- Extending and/or adapting a collection had to be easy.

Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList, HashSet,** and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations, i.e., Classes:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

The Collection Interfaces:

The collections framework defines several interfaces. This section provides an overview of each interface:

SN Interfaces with Description

CHAPTER

1 The Collection Interface

This enables you to work with groups of objects; it is at the top of the collections hierarchy.

2 The List Interface

This extends **Collection** and an instance of List stores an ordered collection of elements.

3 The Set

This extends Collection to handle sets, which must contain unique elements
4 The SortedSet
This extends Set to handle sorted sets
5 The Map
This maps unique keys to values.
6 The Map.Entry
This describes an element (a key/value pair) in a map. This is an inner class of Map.
7 The SortedMap
This extends Map so that the keys are maintained in ascending order.
8
The Enumeration
This is legacy interface and 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 superceded by Iterator.

The Collection Classes:

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.
The standard collection classes are summarized in the following table:

SN Classes with Description
1 **AbstractCollection**
Implements most of the Collection interface.
2 **AbstractList**
Extends AbstractCollection and implements most of the List interface.
3
AbstractSequentialList
Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
4 LinkedList
Implements a linked list by extending AbstractSequentialList.
5 ArrayList
Implements a dynamic array by extending AbstractList.
6 **AbstractSet**
Extends AbstractCollection and implements most of the Set interface.
7 HashSet
Extends AbstractSet for use with a hash table.
8 LinkedHashSet
Extends HashSet to allow insertion-order iterations.

9 TreeSet
Implements a set stored in a tree. Extends AbstractSet.
10 **AbstractMap**
Implements most of the Map interface.
11 HashMap
Extends AbstractMap to use a hash table.
12 TreeMap
Extends AbstractMap to use a tree.
13 WeakHashMap
Extends AbstractMap to use a hash table with weak keys.
14 LinkedHashMap
Extends HashMap to allow insertion-order iterations.
15 IdentityHashMap
Extends AbstractMap and uses reference equality when comparing documents.
The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.
The following legacy classes defined by java.util have been discussed in previous tutorial:

SN Classes with Description
1 Vector
This implements a dynamic array. It is similar to ArrayList, but with some differences.
2 Stack

Stack is a subclass of Vector that implements a standard last-in, first-out stack.
3
Dictionary
Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
4 Hashtable
Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
5
Properties
Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
6
BitSet
A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed.

The Collection Algorithms:

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.
Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.
Collections define three static variables: EMPTY_SET, EMPTY_LIST, and EMPTY_MAP. All are immutable.

SN Algorithms with Description
1 The Collection Algorithms
Here is a list of all the algorithm implementation.

How to use an Iterator?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.
The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.
Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.

SN Iterator Methods with Description
1 Using Java Iterator
Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces.

Using Java Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.
The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.
Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.
In general, to use an iterator to cycle through the contents of a collection, follow these steps:

- Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
- Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
- Within the loop, obtain each element by calling next( ).

For collections that implement List, you can also obtain an iterator by calling ListIterator.

The Methods Declared by Iterator:

SN Methods with Description
1 **boolean hasNext( )**
Returns true if there are more elements. Otherwise, returns false.
2 **Object next( )**

Returns the next element. Throws NoSuchElementException if there is not a next element.
3
void remove( )
Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

The Methods Declared by ListIterator:

SN Methods with Description
1 **void add(Object obj)**
Inserts obj into the list in front of the element that will be returned by the next call to next( ).
2 **boolean hasNext( )**
Returns true if there is a next element. Otherwise, returns false.
3 **boolean hasPrevious( )**
Returns true if there is a previous element. Otherwise, returns false.
4 **Object next( )**
Returns the next element. A NoSuchElementException is thrown if there is not a next element.
5 **int nextIndex( )**
Returns the index of the next element. If there is not a next element, returns the size of the list.
6 **Object previous( )**
Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
7 **int previousIndex( )**
Returns the index of the previous element. If there is not a previous element, returns -1.
8
void remove( )
Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
9 **void set(Object obj)**
Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Example:

Here is an example demonstrating both Iterator and ListIterator. It uses an ArrayList object, but the general principles apply to any type of collection.
Of course, ListIterator is available only to those collections that implement the List interface.

```
import java.util.*;
public class IteratorDemo {
public static void main(String args[]) {
// Create an array list
ArrayList al = new ArrayList();
// add elements to the array list
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
```



```
al.add("F");
// Use iterator to display contents of al
System.out.print("Original contents of al: ");
Iterator itr = al.iterator();
while(itr.hasNext()) {
Object element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Modify objects being iterated
ListIterator litr = al.listIterator();
while(litr.hasNext()) {
Object element = litr.next();
litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
```

```
itr = al.iterator();
while(itr.hasNext()) {
Object element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Now, display the list backwards
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
Object element = litr.previous();
System.out.print(element + " ");
}
System.out.println();
}
}
```

This would produce the following result:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

How to use a Comparator?

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Also, this interface can be used to sort any instances of any class(even classes we cannot modify).

SN Iterator Methods with Description

1 Using Java Comparator

Here is a list of all the methods with examples provided by Comparator Interface.

Using Java Comparator

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

TUTORIALS POINT Simply Easy Learning

The Comparator interface defines two methods: compare( ) and equals( ). The compare( ) method, shown here, compares two elements for order:

The compare Method:

```
int compare(Object obj1, Object obj2)
```

obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned.

By overriding compare( ), you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The equals Method:

The equals( ) method, shown here, tests whether an object equals the invoking comparator:

```
boolean equals(Object obj)
```

obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

Overriding equals( ) is unnecessary, and most simple comparators will not do so.

Example:

```
class Dog implements Comparator<Dog>, Comparable<Dog>{
private String name;
private int age;
Dog(){
}
Dog(String n, int a){
name = n;
age = a;
}
public String getDogName(){
return name;
```

```
}
public int getDogAge(){
return age;
}
// Overriding the compareTo method
public int compareTo(Dog d){
return (this.name).compareTo(d.name);
}
// Overriding the compare method to sort the age
public int compare(Dog d, Dog d1){
return d.age - d1.age;
}
}
```



```
public class Example{
public static void main(String args[]){
// Takes a list o Dog objects
List<Dog> list = new ArrayList<Dog>();
list.add(new Dog("Shaggy",3));
list.add(new Dog("Lacy",2));
list.add(new Dog("Roger",10));
list.add(new Dog("Tommy",4));
list.add(new Dog("Tammy",1));
Collections.sort(list);// Sorts the array list
for(Dog a: list)//printing the sorted list of names
System.out.print(a.getDogName() + ", ");
// Sorts the array list using comparator
Collections.sort(list, new Dog());
System.out.println(" ");
for(Dog a: list)//printing the sorted list of ages
System.out.print(a.getDogName() +" : "+
a.getDogAge() + ", ");
}
}
```

This would produce the following result:

```
Lacy, Roger, Shaggy, Tammy, Tommy,
Tammy : 1, Lacy : 2, Shaggy : 3, Tommy : 4, Roger : 10,
```

Note: Sorting of the Arrays class is as the same as the Collections.

Summary:

The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.

A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.

The classes and interfaces of the collections framework are in package java.util.