**Java Data Structures**

* Enumeration
* BitSet
* Vector
* Stack
* Dictionary
* Hashtable
* Properties

**Vector**

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 we don't know the size of the array in advance or we just need one that can change sizes over the lifetime of a program.

**Constructors :-**

* **Vector()**

This creates a vector of initial size = 10 and capacityIncrement = 0 which means the vector will grow by double its size every time when the capacity is exceeded

* **Vector(int size)**

This creates a vector of initial size = size and capacityIncrement = 0 which means the vector will grow by double its size every time when the capacity is exceeded

* **Vector(int size, int increment)**

This creates a vector of initial size = size and capacityIncrement = increment

* **Vector(Collection c)**

This creates a vector from the Collection provided to it in an order in which the elements are returned by collection’s iterator.

**Enumeration**

The Enumeration interface defines the methods by which we 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This legacy interface has been superceded by Iterator.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.

Methods declared in Enumeration class are :-

* **hasMoreElements( ) : boolean**
* **nextElement( ) : Object**

e = v.elements();

int i = 0;

while(e.hasMoreElements()) {

if(i == 2)

v.add(8);

System.out.print(e.nextElement() + " ");

i++;

}

If the same thing is done with Iterator, the Java Runtime will give **ConcurrentModificationException.**

The iterators returned by this class's iterator and listIterator methods are fail-fast: if the vector is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. The Enumerations returned by the elements method are not fail-fast.

**BitSet**

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.

**BitSet( )**

**BitSet(int size)**

This constructor allows we to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero

BitSet implements the Cloneable interface

**Dictionary**

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.

Given a key and value, we can store the value in a Dictionary object. Once the value is stored, we can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

**Object creation :-**

Dictionary<String, Integer> dict = new Hashtable<String, Integer>();

**Methods:-**

isEmpty() : boolean

elements() : Enumeration – returns Enumeration for all the values of the Dictionary

get(String Key) : Object

keys() : Enumeration – returns Enumeration for all the Keys of the Dictionary

put(Object key, Object value) : Object

remove(Object key) : Object

size() : int

**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.

**Constructors :-**

* **Properties( ) -** This constructor creates a Properties object that has no default values
* **Properties(Properties propDefault) -** creates an object that uses propDefault for its default values. In both cases, the property list is empty

**Methods :-**

* getProperty(String key) : String – return null if key not found
* getProperty(String key, String defaultProperty) : String – return default if key not found
* list(PrintStream streamOut) : void – sends the list of properties to PrintStream object
* list(PrintWriter streamOut) : void
* load(InputStream streamIn) throws IOException – load the properties from InputStream object, # is considered as comment and will not be loaded into the property
* propertyNames( ) : Enumeration
* StringPropertyNames() : Set
* setProperty(String key, String value) : Object
* store(OutputStream streamOut, String description) : void
* loadFromXML(InputStream in) : void
* storeToXML(OutputStream os, String comment)throws IOException
* storeToXML(OutputStream os, String comment, String encoding)throws IOException

**Hashtable**

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. Java 2 re-engineered 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.

**Constructors :-**

* Hashtable( )
* Hashtable(int size)
* Hashtable(int size, float fillRatio) - This creates a hash table 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 hash table can be before it is resized upward.
* Hashtable(Map<? extends K, ? extends V> t)

**Stack**

The Stack class implements a last-in-first-out (LIFO) stack of elements. 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.

**Constructors**

Stack() creates an empty stack

**Methods**

* empty() : Boolean
* peek() : Object – returns last element of the stack without removing it from the stack
* pop() : Object – returns last element of stack and removes it from the stack
* push(E item) : E – adds element to the stack
* search(Object obj) : int – returns 1-based position of element in the stack from left to right, -1 if element is not found