java.util.Hashtable

- This class implements a hash table, which maps keys to values
- A Hashtable is *generic*: its components are **`Objects`**
  - Any non-null object can be used as a key or as a value
  - To successfully store and retrieve objects from a hash table, the objects used as *keys* must implement the **`hashCode`** method and the **`equals`** method.
- An instance of Hashtable has two parameters that affect its efficiency: its *capacity* and its *load factor*
  - The load factor should be between 0.0 and 1.0
  - When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is *increased automatically*
  - Larger load factors use memory more efficiently, at the expense of larger expected time per lookup

Some of the Hashtable methods
(there are lots more!)

- `public Hashtable()`
  - Constructs a new, empty hash table with a default capacity (seems to be 11!) and load factor (0.75)
- `public Hashtable(int initialCapacity)`
  - Constructs a new, empty hash table with the specified initial capacity and a default load factor (0.75)

-
  ```
  public Hashtable(int initialCapacity,
                   float loadFactor)
  ```
  - Constructs a new, empty hash table with the specified initial capacity and the specified load factor
  - Example:

```
private Hashtable h =
               new Hashtable (1000,0.5f);
```

- **`public int size()`**
  - Returns the number of keys in this hash table
  - Example:
    ```
    int n = h.size();
    ```
- **`public Object put(Object key, Object value)`**
  - Inserts the specified key with the specified value into this hash table. Neither the key nor the value may be null
  - Returns the previous value of the specified key in this hash table, or **`null`** if it did not have one
  - Example:
    ```
    String n = ". . .";
    StudentData s = new StudentData(. . . );
    h.put(n,s);     // Can ignore the result
    ```
- **`public boolean containsKey(Object key)`**
  - Tests if the specified object is a key in this hash table
  - Example:
    ```
    if (h.containsKey(n)) . . .
    ```

- **`public Object get(Object key)`**
  - Returns the value to which the specified key is mapped in this hash table
  - Note that the result is an **`Object`** - so *casting* will usually be required
  - Example:

```
StudentData s = (StudentData)h.get(n);
```

- **`public Object remove(Object key)`**
  - Removes the key (and its corresponding value) from this hash table. This method does nothing if the key is not in the hash table
  - Returns the removed *value*, or **`null`** if none
  - Example:

```
h.remove(n);  // Can ignore the result
```

java.util.Vector

- The Vector class implements a growable array of objects
  - A Vector is *generic*: its components are `Objects`
  - Like an array, it contains components that can be accessed using an integer index
  - However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created
- Each vector tries to optimize storage management by maintaining a *capacity* and a *capacity increment*
  - The capacity is always at least as large as the vector size
  - It is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of capacity increment

Some of the Vector methods
(there are lots more!)

- `public Vector()`
  - Constructs an empty vector with capacity 10, and increment zero
- `public Vector(int initialCapacity)`
  - Constructs an empty vector with the specified initial capacity , and increment zero

-
  ```
  public Vector(int initialCapacity,
                int capacityIncrement)
  ```
  - Constructs an empty vector with the specified initial capacity and capacity increment
  - Example:

    ```
    private Vector v = new Vector(1000,500);
    ```

- `public int size()`
  - Returns the number of components in this vector
  - Example:

    ```
    int n = v.size();
    ```

- `public void addElement(Object obj)`
  - Adds the specified component to the end of this vector, increasing its size by one
  - Example:

    ```
    Student s = new Student(. . . );
    v.addElement(s);
    ```

  - The capacity of this vector is increased if its size becomes greater than its current capacity

- **`public int indexOf(Object elem)`**
  - Searches for the first occurrence of the given argument, testing for equality using the object's **`equals`** method
  - Returns the index of the first occurrence of the argument in this vector; returns -1 if the object is not found
  - Example:

```
int n = v.indexOf(s);
```

- **`public Object elementAt(int index)`**
  - Returns the component at the specified index
  - Note that the result is an **`Object`** - so *casting* will usually be required
  - Example:

    ```
    Student s = (Student)v.elementAt(n);
    ```
  - *Analogous to the following for arrays:*

    ```
    s = v[n];
    ```
- **`public void setElementAt(Object obj, int index)`**
  - Sets the component at the specified index of this vector to be the specified object
  - Example:

    ```
    v.setElementAt(s,n);
    ```
  - *Analogous to the following for arrays:*

    ```
    v[n] = s;
    ```