7 More Collections: Sets and HashMaps

What's in This Set of Notes ?

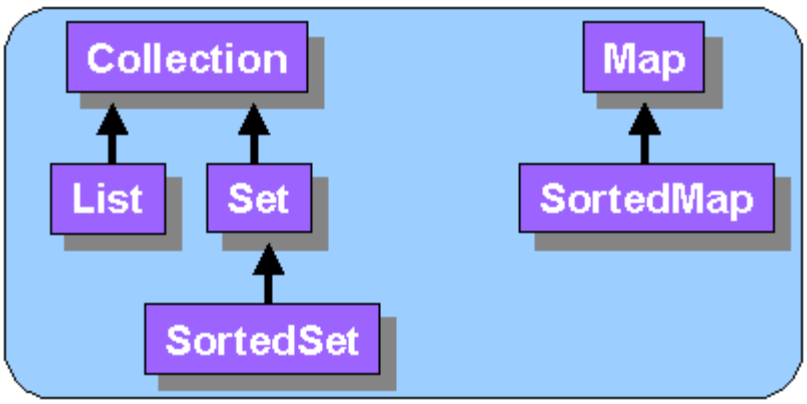
In computer science, we are often concerned about issues pertaining to efficiency. That is, we always want to write code that is fast and efficient. In terms of collections, we want to make sure that when we put stuff in there, we can find it quickly. Think about how you unpack your groceries. Don't you place things in your cupboards in an organized way ? It helps you find what you are looking for quickly. Also, how will you find the items at the grocery store if they are not organized by sections ? **HashMaps** are collections in JAVA which store objects like **ArrayLists**. However, the items are stored in a **HashMap** in a way that the items can be retrieved quickly, when compared to an **ArrayList**. We will discuss the **Set, TreeSet, HashSet, Map, HashMap/Hashtable, TreeMap** collections in JAVA and also give a larger example using **HashMaps** to store data in a MovieStore.


Here are the individual topics found in this set of notes (click on one to go there):

- 7.1 Collections Re-Visited
- 7.2 The Set Classes
- 7.3 The Map Interface and Its Classes
- 7.4 HashMaps
- 7.5 The MovieStore Example

7.1 Collections Re-Visited

In COMP1405/1005, we examined some **Collection** classes and worked quite a bit with **ArrayLists.** We also briefly looked at the **List** interface. Recall that there were other **Collection** classes as well that implemented the **Set** and **Map** interfaces:



Recall that an interface merely specifies a list of method signatures ... not actual code. Recall as well that the **Collection** interface defined many messages. Below is a list of a few of these grouped as querying methods and modifying methods.

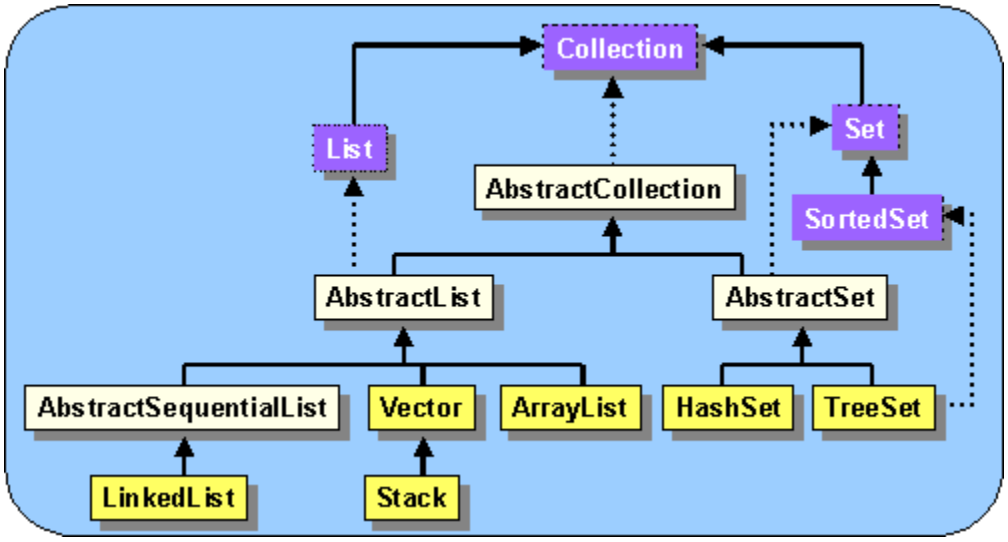
Querying methods (returns some kind of value):

- **size()** - returns the number of elements in the collection
- **isEmpty()** - returns whether or not there are any elements in the collection
- **contains(Object obj)** - returns whether or not the given object is in the collection (uses **.equals()** method for comparison)
- **containsAll(Collection c)** - same as above but looks for ALL elements specified in the given collection parameter.

Modifying methods (changes the collection in some way):

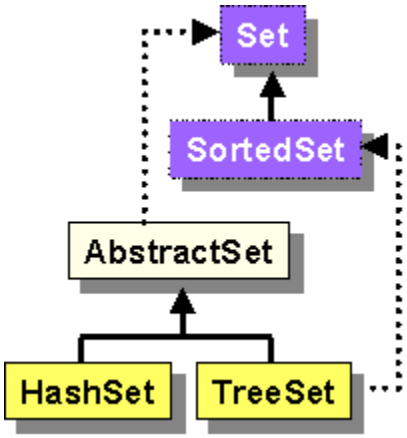
- **add(Object obj)** - adds the given object as an element to the collection (location is not specified)
- **addAll(Collection c)** - same as above but adds ALL elements specified in the given collection parameter.
- **remove(Object obj)** - removes the given object from the collection (uses **.equals()** method for comparison)
- **removeAll(Collection c)** - same as above but removes ALL elements specified in the given collection parameter.
- **retainAll(Collection c)** - same as above but removes all elements EXCEPT THOSE specified in the given collection parameter.
- **clear()** - empties out the collection by removing all elements.

Do you remember the hierarchy of classes implementing the **Collection** interface ?



7.2 The Set Classes

Let us continue where we left off from COMP1405/1005 and discuss the **Set** interface as well as the subclasses that implement it: **HashSet** and **TreeSet.**



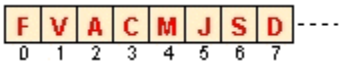
The **Set** classes are much like **Lists**, except that they do not allow duplicates.

- there cannot be two elements e1 and e2 such that e1.equals(e2)
- any modifications to the elements that affect the equality results in unspecified behaviour
- attempts to add duplicates are not satisfied

There are 2 main **Set** implementations:

HashSet

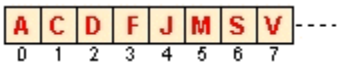
- elements are **not kept in order**
- **fast adding/removing** operations
- **searching is slow** for a particular element



- elements MUST implement **`.equals()`** and **`.hashCode()`**

TreeSet

- elements are kept in **sorted order**, but not indexable
- order is user-defined (default is ascending)
- **adding is slow** since it takes longer to find the proper location
- **searching is fast** now since items are in order



Hashing is a high speed scheme for converting sparse (i.e., quite spread apart) keys into unique array subscripts. Hashing essentially generates for each item, a number (called the ***hashcode***) which is used as the ***key*** (or index). The generated number has no significant meaning other than to allow efficient storage and retrieval of data.

It is used as a way of getting quickly to the "vicinity" of the object you are looking for. For example, this is exactly what post offices do when they sort mail. They use the postal code to determine "roughly" where in the city your mail should go. People living in the same area have the same postal code, so it is easier for the post office to locate, gather and deliver your mail.

So **HashSets** (as well as **Hashtables** and **HashMaps** which we will see later) store their objects according to a ***hash function*** which:

- returns an integer representation of the object (i.e., its *hashcode*).
- the integer should reflect the object's characteristics.
- equal objects should return the same hashcode.

Let us now look at an example that helps us understand the differences between these two sets.

Consider this code that adds some **BankAccounts** to an **ArrayList**:

```
String[]  names = {"Al", "Zack", "Sally", "Al", "Mel", "Zack",
"Zack", "Sally"};
ArrayList<String> aCollection = new ArrayList<String>();

// Fill up the collection
for(int i=0; i<names.length; i++)
    aCollection.add(names[i]);
```

```
        // Now print out the elements
        for(String s: aCollection)
            System.out.println(s);
```

Here is the output, as expected:

```
Al
Zack
Sally
Al
Mel
Zack
Zack
Sally
```

Consider what happens when we replace the line:

```
ArrayList<String> aCollection = new ArrayList<String>();
```

with:

```
HashSet<String> aCollection = new HashSet<String>();
```

When we run the code, we obtain the following output:

```
Mel
Al
Sally
Zack
```

What happened to the duplicates ? They were removed. We only see the unique names coming out on the console window. But hold on! This is not the order that we added the names into the collection!! Well ... recall that the elements are not kept in any kind of order, and in fact, JAVA has its own pre-determined order based on hashcode values of the elements in which we added.

Now what about a **TreeSet** ? Shouldn't that keep them in order ?

Consider what happens when we replace:

```
HashSet<String> aCollection = new HashSet<String>();
```

with:

```
TreeSet<String> aCollection = new TreeSet<String>();
```

Now we obtain this output:

```
Al
Mel
```

```
Sally
Zack
```

Notice that the items are now sorted alphabetically. If we wanted to sort in a different order, we would need to make our own objects, instead of using Strings so that we could implement the **Comparable** interface.

What if we want to store **Customer** objects in the sets instead of simply strings:

```
public class Customer {
    private String name;

    public Customer(String n) { name = n; }
    public String getName() { return name; }
    public String toString() { return "Customer: " + name; }
}
```

Here is some code that makes a **HashSet** of **Customer** objects and prints them:

```
String[]  names = {"Al", "Zack", "Sally", "Al", "Mel", "Zack",
"Zack", "Sally"};
HashSet<Customer> aCollection = new HashSet<Customer>();
for(int i=0; i<names.length; i++) {
    Customer c = new Customer(names[i]);
    aCollection.add(c);
}
for(Customer c: aCollection)
    System.out.println(c);
```

Here is the output:

```
Customer: Sally
Customer: Zack
Customer: Zack
Customer: Mel
Customer: Al
Customer: Sally
Customer: Al
Customer: Zack
```

Hey! There are duplicates! What's up with that ?

A unique **Customer** object is actually created for each customer and so they are all unique by default regardless of their name. How can we fix it so that only **Customers** with unique names can be added ? Recall that in a set, there cannot be two elements e1 and e2 such that **e1.equals(e2)**. But we don't have an **equals()** method in our **Customer** class! So in fact, we inherit a default one that checks identity, not equality. So ... we have to create our own **equals()** method for the **Customer** class:

```
public boolean equals(Object obj) {
    if (!(obj instanceof Customer))
        return false;
    return getName().equals(((Customer)obj).getName());
}
```

In addition, since objects are "hashed" in order to find their position in the ***Hash***Set, we must also implement a **hashCode()** method. The **hashCode()** method should return an integer that attempts to represent the object uniquely. Usually, it simply returns the combined hashcodes of its instance variables:

```
public int hashCode() {
    return name.hashCode();
}
```

Now when we run the code, we see that the duplicates are gone:

```
Customer: Mel
Customer: Al
Customer: Sally
Customer: Zack
```

We could change **HashSet** to **TreeSet** to get the items in sorted order. However, we would then need to make sure that our **Customer** objects are **Comparable**. We could add the following to our **Customer** class:

```
public class Customer implements Comparable {
    ...

    public int compareTo(Object obj) {
        if (obj instanceof Customer)
          return getName().compareTo(((Customer)obj).getName());
        else
            throw new ClassCastException();
    }
}
```

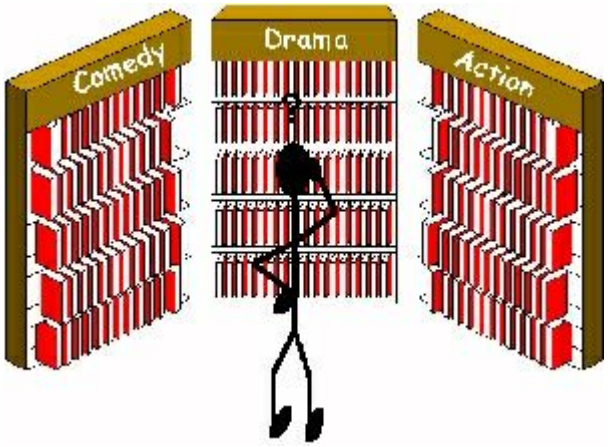
Then when we run the code, we see that the duplicates are gone and that the items are in sorted order:

```
Customer: Al
Customer: Mel
Customer: Sally
Customer: Zac
```

So remember ... we can use a **HashSet** or **TreeSet** to eliminate duplicates from a collection.

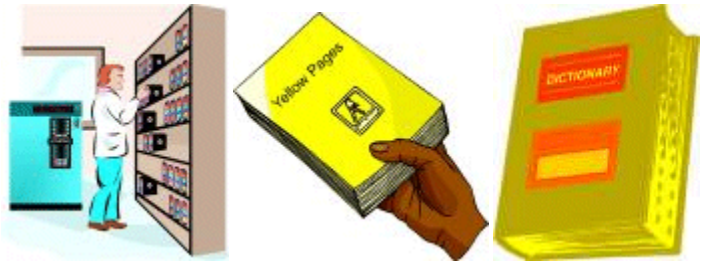
7.3 The Map Interface and Its Classes

It is often necessary to store objects in collections in such a way that they can be retrieved quickly later on. That is, there is a need for a collection which allows quick searching. For example, consider a video store. Isn't it a nice idea to have the movies arranged by category so that you don't waste time looking over movies that are not of an interesting nature (such as musicals or perhaps drama) ?



Like the **Collection** interface, the **Map** interface stores objects as well. So what is different ? Well, a **Map** stores things in a particular way such that the objects can be easily located later on. A ***Map*** really means a "Mapping" of one object to another. Maps are used when it is necessary to access the elements quickly by particular ***keys***. Examples are:

- storing many items by category (e.g., a video store is organized by sections (comedy/action/drama)).
- phone books where a value (e.g., number) is associated with a particular key (e.g., name).
- dictionaries where a value (e.g., definition) is associated with a particular key (e.g., word).



All **Maps** store a group of object pairs called ***entries***.
Each map ***entry*** has:

- ***key*** **-** identifies values uniquely (maps cannot have duplicate keys)
- ***value*** **-** accessed by their keys (each key maps to at most one value)

So, the ***key*** MUST be used to obtain a particular value from the **Map**.

The **Map** interface defines many messages:

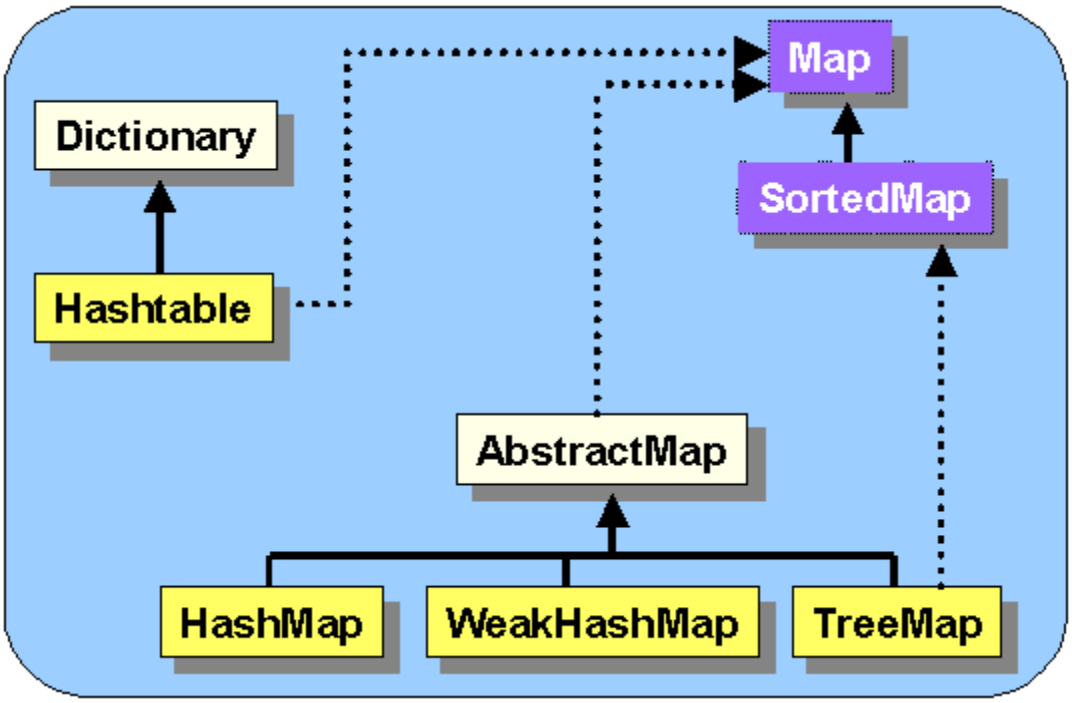
Querying methods (returns some kind of value):

- **size()** - returns the number of elements in the map.
- **isEmpty()** - returns whether or not there are any elements in the map.
- **containsKey(Object key)** - returns whether or not the given object is a key of the map (uses **.equals()** method for comparison).
- **containsValue(Object val)** - returns whether or not the given object is a value in the map (uses **.equals()** method for comparison).
- **getKey(Object key)** - returns the **Object** in the map that is associated with the given key.
- **values(~~Object key~~)** - returns a **Collection** containing all the values in the map.
- **keySet(~~Object key~~)** - returns a **Set** of all the keys in the map.
- **entrySet(~~Object key~~)** - returns a **Set** of all the key/value pairs in the map (i.e., **Map.Entry** objects represent key/value pairs).

Modifying methods (changes the collection in some way):

- **put(Object key, Object val)** - adds a new entry in the map with the given key and value. The key MUST be unique, otherwise this method replaces the value that was previously there by the given value.
- **putAll(Map m)** - adds all entries that are in the given map to the receiver map.
- **remove(Object key)** - removes the given key/value entry from the map based on the key only.
- **clear()** - empties out the map by removing all elements.

Here is the hierarchy showing most of the **Map** classes:



Notice that there are 4 main **Map** implementations:

Hashtable

- elements **not kept in order**
- **fast adding/removing**
- **fast searching** for a particular element
- elements MUST implement `.equals()` and `.hashCode()`
- cannot have **null** key
- synchronized

HashMap

- works similar to **Hashtables**
- can have one **null** key
- unsynchronized

TreeMap

- works similar to **HashMap**
- elements are **kept in sorted order**, but not indexable
- **slow adding** since it takes longer to find the proper location
- **efficient access and modification** methods

WeakHashMap

- works similar to **HashMap**

- **keys removed more efficiently** by the garbage collector.

7.4 HashMaps

HashMaps are not fixed-size, so they can grow as needed (just like **ArrayLists**). Items are added to the **HashMap** by specifying the key AND the value. The key is used as a unique identifier that distinguishes its associated value from any other value. Both the keys and values can be arbitrary objects (but no **null** key for **Hashtables**). We will look at **HashMaps** here, but the **Hashtable** class works the same way, but is synchronized (slower than HashMap).

To create a general **HashMap** that can store arbitrary keys and values we can use this constructor:

```
HashMap table = new HashMap();
```

However, typically we will use a **HashMap** that has all the same type of keys and the same type of values. In this case, we can specify the keys and values when declaring our variables. For example, if we want a HashMap where the keys are peoples names and the values for each person is a collection of **Items**, we could declare our **HashMap** something like this:

```
HashMap<String,ArrayList<Item>> table = new
HashMap<String,ArrayList<Item>>();
```

Of course, this will prevent us from making non-String keys and non-ArrayList values ... but it will simplify the usage of the class by eliminating the need for typecasting in many places where we access/modify the **HashMap**.

Remember, if you want to use your own created objects within **Hashtables, HashMaps, HashSets** or **TreeSets**, you MUST implement a **hashCode()** method. All objects inherit a default **hashCode()** from **Object**, but it may not be efficient. The **hashcode()** method should look similar to an **equals()** method in what it checks.
Here are some of the standard **HashMap** methods along with a simple example of how they work:

Method	Description and Example
`Object put(Object key, Object value)`	Add the given value to the **HashMap** with the given key. If there is no value there for that key then **null** is returned, otherwise the original value in the **HashMap** is returned. `HashMap<String,String> aPhoneBook = new HashMap<String,String>();`

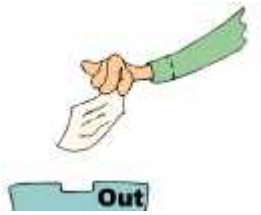
```
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.put("Mark", "555-2238");
aPhoneBook.put("Norm", "555-3789");
System.out.println(aPhoneBook); // displays
{Norm=555-3789, Mark=555-2238, Arthur=555-8813}
```

Object get(Object key)

Return the value associated with the given key. If there is no value, **null** is returned.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.put("Mark", "555-2238");
aPhoneBook.put("Norm", "555-3789");
aPhoneBook.get("Mark");         // returns "555-2238"
aPhoneBook.get("Betty");        // returns null
aPhoneBook.get("555-3789");     // returns null
(must specify key, not value)
```

Object remove(Object key)



Remove the key/value pair from the **HashMap** and return the value that was removed. If it is not there, return **null**.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.put("Mark", "555-2238");
aPhoneBook.put("Norm", "555-3789");
aPhoneBook.remove("555-8813");  // No error but
returns null
aPhoneBook.remove("Arthur");    // returns "555-
8813"
System.out.println(aPhoneBook);
// displays {Norm=555-3789, Mark=555-2238}
```

boolean isEmpty()

Return whether or not there are any values in the **HashMap**.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.isEmpty(); // returns true
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.isEmpty(); // returns false
```

boolean containsKey(Object key)

Return whether or not the given key is in the HashMap.

```
HashMap<String,String>  aPhoneBook = new
HashMap<String,String>();
aPhoneBook.put("Arthur", "555-8813");
aPhoneBook.put("Mark", "555-2238");
aPhoneBook.put("Norm", "555-3789");
aPhoneBook.containsKey("555-8813"); // returns false
aPhoneBook.containsKey("Mark");     // returns true
aPhoneBook.remove("Mark");
```

	`aPhoneBook.containsKey("Mark");  // returns false`

boolean containsValue(Object value)
 
	Return whether or not the given value is in the **HashMap**.
	`HashMap<String,String>  aPhoneBook = new HashMap<String,String>();`
	`aPhoneBook.put("Arthur", "555-8813");`
	`aPhoneBook.put("Mark", "555-2238");`
	`aPhoneBook.put("Norm", "555-3789");`
	`aPhoneBook.containsValue("Mark");  // returns false`
	`aPhoneBook.containsValue("555-3789");  // returns true`
	`aPhoneBook.remove("Norm");`
	`aPhoneBook.containsValue("555-3789");  // returns false`

void clear()
 
	Empty the **HashMap**.
	`HashMap<String,String>  aPhoneBook = new HashMap<String,String>();`
	`aPhoneBook.put("Arthur", "555-8813");`
	`aPhoneBook.put("Mark", "555-2238");`
	`System.out.println(aPhoneBook); // displays {Mark=555-2238, Arthur=555-8813}`
	`aPhoneBook.clear();`
	`System.out.println(aPhoneBook); // displays {}`

Collection **values()**
 
	Return a **Collection** of the values in the **HashMap**.
	`HashMap<String,String>  aPhoneBook = new HashMap<String,String>();`
	`aPhoneBook.put("Arthur", "555-8813");`
	`aPhoneBook.put("Mark", "555-2238");`
	`aPhoneBook.put("Norm", "555-3789");`
	`aPhoneBook.values(); // returns Collection [555-3789, 555-2238, 555-8813]`

Set **keySet()**
 
	Return a **Set** of the keys in the **HashMap**.
	`HashMap<String,String>  aPhoneBook = new HashMap<String,String>();`
	`aPhoneBook.put("Arthur", "555-8813");`
	`aPhoneBook.put("Mark", "555-2238");`
	`aPhoneBook.put("Norm", "555-3789");`
	`aPhoneBook.keySet(); // returns Set [Norm, Mark, Arthur]`

Set **entrySet()** 	Return a **Set** of the key/value pairs (i.e., as **Map.Entry** objects) in the **HashMap**.
	
	`HashMap<String,String>  aPhoneBook = new HashMap<String,String>();`
	`aPhoneBook.put("Arthur", "555-8813");`
	`aPhoneBook.put("Mark", "555-2238");`
	`aPhoneBook.put("Norm", "555-3789");`
	`aPhoneBook.entrySet(); // returns Set`
	`[Norm=555-3789, Mark=555-2238, Arthur=555-8813]`

What is the difference between a **HashMap** and a **TreeMap** ? As with **HashSets** and **TreeSets**, **TreeMaps** maintain the keys in sorted order, whereas **HashMaps** do not maintain the keys in sorted order.

Consider this code:

```
String[]  names = {"Al", "Zack", "Sally", "Al", "Mel", "Zack",
"Zack", "Sally"};
HashMap<String,Customer> aMap = new HashMap<String,Customer>();

// Fill up the collection
for (int i=0; i<names.length; i++)
    aMap.put(names[i], new Customer(names[i]));

System.out.println("Here are the keys:");
for (String key: aMap.keySet())
     System.out.println("    " + key);

System.out.println("Here are the values:");
for (Customer val: aMap.values())
     System.out.println("    " + val);

System.out.println("Here are the key/value pairs:");
for (Map.Entry pair: aMap.entrySet())
     System.out.println("    " + pair);
```

Here we see that a **HashMap** is formed with keys being the names of the **Customer** and values being the **Customers** themselves. This is the output:

```
Here are the keys:
    Mel
    Al
    Sally
    Zack
Here are the values:
    Customer: Mel
    Customer: Al
    Customer: Sally
    Customer: Zack
```

```
Here are the key/value pairs:
    Mel=Customer: Mel
    Al=Customer: Al
    Sally=Customer: Sally
    Zack=Customer: Zack
```

Notice that there are only 4 keys, even though we added many items ... this is because one key overwrites another when we call the **put()** method more than once with the same key.

If we replace the **HashMap** with a **TreeMap** in the above code, the code still works, we just get the items in sorted order according to the keys:

```
Here are the keys:
    Al
    Mel
    Sally
    Zack
Here are the values:
    Customer: Al
    Customer: Mel
    Customer: Sally
    Customer: Zack
Here are the key/value pairs:
    Al=Customer: Al
    Mel=Customer: Mel
    Sally=Customer: Sally
    Zack=Customer: Zack
```

Using **WeakHashMap**, you won't notice a difference. We will not talk about this class in this course.

7.5 The MovieStore Example

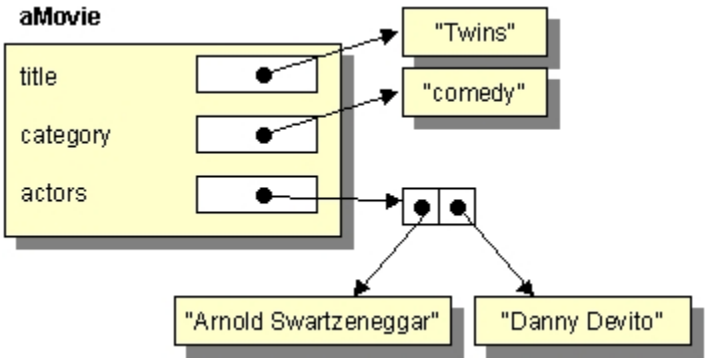
Consider an application which represents a movie store that maintains movies to be rented out. Assume that we have a collection of movies. When renting, we would like to be able to find movies quickly. For example, we may want to:

- ask for a movie by title and have it found right away
- search for movies in a certain category (e.g., new release, comedy, action)
- find movies containing a specific actor/actress (e.g., Jackie Chan, Peter Sellers, Jude Law etc...)

Obviously, we could simply store all moves in one big **ArrayList**. But how long would we waste finding our movies ? Imagine a video store in which the movies are not sorted in any particular order ... just randomly placed on the shelves !! We would have to go through them one by one !!!

We will use **HashMaps** to store our movies efficiently so that we can quickly get access to the movies that we want.

Let us start out with the representation of a **Movie** object. Each movie will maintain a **title**, list of **actors** and a **type** (category). Obviously, in a real system, we would need to keep much more information such as ID, rental history, new releases vs. oldies, etc... Here is the diagram representing the **Movie** object:



Let us now define this **Movie** class.

```
import java.util.*;
public class Movie {
    private String             title, type;
    private ArrayList<String>  actors;

    public String getTitle() { return title; }
    public String getType() { return type; }
    public ArrayList<String> getActors() { return actors; }
    public void setTitle(String aTitle) { title = aTitle; }
    public void setType(String aType) { type = aType; }

    public Movie() { this("???", "???"); }

    public Movie(String aTitle, String aType) {
        title = aTitle;
        type = aType;
        actors = new ArrayList<String>();
    }

    public String toString() { return("Movie: " + "\"" + title + "\"");
}
```

```
        public void addActor(String anActor) { actors.add(anActor); }
    }
```

Notice that there is no "set" method for the actors. We do not need it (just like in the autoshow example). Now lets look at the **`addActor()`** method. It merely adds the given actor (just a name) to the actors arrayList. We can make some example methods to represent some movies. Add the following methods to the **Movie** class:

```
    public static Movie example1() {
        Movie aMovie = new Movie("The Matrix","SciFic");
        aMovie.addActor("Keanu Reeves");
        aMovie.addActor("Laurence Fishburne");
        aMovie.addActor("Carrie-Anne Moss");
        return aMovie;
    }

    public static Movie example2() {
        Movie aMovie = new Movie("Blazing Saddles","Comedy");
        aMovie.addActor("Cleavon Little");
        aMovie.addActor("Gene Wilder");
        return aMovie;
    }

    public static Movie example3() {
        Movie aMovie = new Movie("The Matrix Reloaded","SciFic");
        aMovie.addActor("Keanu Reeves");
        aMovie.addActor("Laurence Fishburne");
        aMovie.addActor("Carrie-Anne Moss");
        return aMovie;
    }

    public static Movie example4() {
        Movie aMovie = new Movie("The Adventure of Sherlock Holmes' Smarter
Brother","Comedy");
        aMovie.addActor("Gene Wilder");
        aMovie.addActor("Madeline Kahn");
        aMovie.addActor("Marty Feldman");
        aMovie.addActor("Dom DeLuise");
        return aMovie;
    }

    public static Movie example5() {
        Movie aMovie = new Movie("The Matrix Revolutions","SciFic"));
        aMovie.addActor("Keanu Reeves");
        aMovie.addActor("Laurence Fishburne");
        aMovie.addActor("Carrie-Anne Moss");
        return aMovie;
    }

    public static Movie example6() {
        Movie aMovie = new Movie("Meet the Fockers","Comedy");
        aMovie.addActor("Robert De Niro");
        aMovie.addActor("Ben Stiller");
        aMovie.addActor("Dustin Hoffman");
        return aMovie;
    }
```

```
    public static Movie example7() {
        Movie aMovie = new Movie("Runaway Jury","Drama");
        aMovie.addActor("John Cusack");
        aMovie.addActor("Gene Hackman");
        aMovie.addActor("Dustin Hoffman");
        return aMovie;
    }

    public static Movie example8() {
        Movie aMovie = new Movie("Meet the Parents","Comedy");
        aMovie.addActor("Robert De Niro");
        aMovie.addActor("Ben Stiller");
        aMovie.addActor("Teri Polo");
        aMovie.addActor("Blythe Danner");
        return aMovie;
    }

    public static Movie example9() {
        Movie aMovie = new Movie("The Aviator","Drama");
        aMovie.addActor("Leonardo DiCaprio");
        aMovie.addActor("Cate Blanchett");
        return aMovie;
    }

    public static Movie example10() {
        Movie aMovie = new Movie("Envy","Comedy");
        aMovie.addActor("Ben Stiller");
        aMovie.addActor("Jack Black");
        aMovie.addActor("Rachel Weisz");
        aMovie.addActor("Amy Poehler");
        aMovie.addActor("Christopher");
        return aMovie;
    }
```

Of course, we should test our class:

```
    public class MovieTester {
        public static void main(String args[]) {
            Movie     aMovie, anotherMovie;

            aMovie = Movie.example1();
            anotherMovie = Movie.example2();
            System.out.println(aMovie);
            System.out.println("is a " + aMovie.getType() +
                       " with actors " + aMovie.getActors());
            System.out.println(anotherMovie);
            System.out.println("is a " + anotherMovie.getType() +
                       " with actors " + anotherMovie.getActors());
        }
    }
```

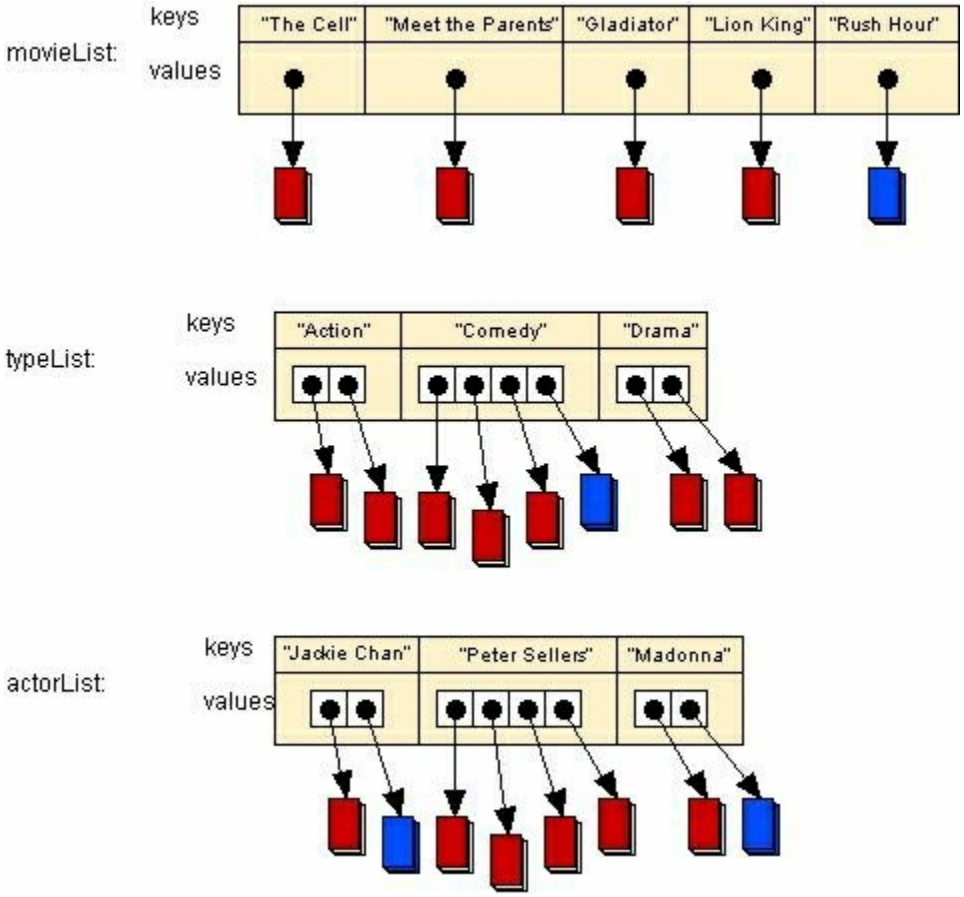
Here is the output:

```
Movie: "The Matrix"
is a SciFic with actors [Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss]
```

```
Movie: "Blazing Saddles"
is a Comedy with actors [Cleavon Little, Gene Wilder]
```

Now we need to consider the making a **MovieStore** object. Recall, that we want to store movies efficiently using **HashMaps**.

For the **MovieStore**, we will maintain three **HashMaps**. One will be the **movieList** where the keys are titles and the values are the movie objects with that title. The second will be the **actorList** which will keep actor/actress names as keys and the values will be ArrayLists of all movies that the actor/actress stars in. The last one will be the **typeList** in which the keys will be the "types" (or categories) of movies and the values will be ArrayLists of all movies belonging to that type.



Notice that one of the movies is "blue" in the picture. Why ?
This represents the same exact movie. So in fact, the reference to this movie is stored in 4 different places.

Isn't this wasteful ? Keep in mind that we are not duplicating all the movie's data ... we are only duplicating the pointer to the movie. So in fact, each time we duplicate a movie in our **HashMaps**, we are simply duplicating its reference (or pointer) which takes 4 bytes.

So, yes, we are taking slightly more space, but at the benefit of allowing quick access to the data. You will learn more about efficiency when you do your second-year course on data structures.

The basic **MovieStore** definition is as follows:

```
import java.util.*
public class MovieStore {
    //These are the instance variables
    private HashMap<String,Movie>              movieList;
    private HashMap<String,ArrayList<Movie>>   actorList;
    private HashMap<String,ArrayList<Movie>>   typeList;

    //These are the get methods, not set methods are needed
    public HashMap<String,Movie> getMovieList() { return movieList; }
    public HashMap<String,ArrayList<Movie>> getActorList() { return
actorList; }
    public HashMap<String,ArrayList<Movie>> getTypeList() { return
typeList; }

    //This is the constructor
    public MovieStore() {
        movieList = new HashMap<String,Movie>();
        actorList = new HashMap<String,ArrayList<Movie>>();
        typeList  = new HashMap<String,ArrayList<Movie>>();
    }

    //This method returns a String representation of the Movie
    public String toString() {
        return ("MovieStore with " + movieList.size() + " movies.");
    }
}
```

Why do not we need "set" methods for the **HashMaps** ? You should be able to reason on that ;).

Now, how do we add a movie to the store ? Well ... how do the instance variables change ?

- the movie must be added to the movieList
- the movie must be added to the typeList. What if it is the first/last movie from this category ?
- the movie must be added to the actorList. What if it is the first/last movie for this actor ?

Here is the code:

```
//This method adds a movie to the movieStore.
public void addMovie(Movie aMovie) {
    //Add to the movieList
    movieList.put(aMovie.getTitle(), aMovie);

    //If there is no category yet matching this movie's type, make a
new category
```

```
        if (!typeList.containsKey(aMovie.getType()))
            typeList.put(aMovie.getType(), new ArrayList<Movie>());

        //Add the movie to the proper category.
        typeList.get(aMovie.getType()).add(aMovie);

        //Now add all of the actors
        for (String anActor: aMovie.getActors()) {
            //If there is no actor yet matching this actor, make a new
    actor key
           if (!actorList.containsKey(anActor))
                actorList.put(anActor, new ArrayList<Movie>());

            //Add the movie for this actor
            actorList.get(anActor).add(aMovie);
        }
    }
```

In fact, removing a movie is just as easy:

```
    //This private method removes a movie from the movie store
    private void removeMovie(Movie aMovie) {
        //Remove from the movieList
        movieList.remove(aMovie.getTitle());

        //Remove from the type list vector.  If last one, remove the type.
        typeList.get(aMovie.getType()).remove(aMovie);
        if (typeList.get(aMovie.getType()).isEmpty())
            typeList.remove(aMovie.getType());

        //Now Remove from the actors list.  If actor has no more, remove
    him.
        for(String anActor: aMovie.getActors()) {
            actorList.get(anActor).remove(aMovie);
            if (actorList.get(anActor).isEmpty())
                actorList.remove(anActor);
        }
    }
```

However, what if we do not have a hold of the **Movie** object that we want to delete ? Perhaps we just know the title of the movie that needs to be removed. We can write a method which asks to remove a movie with a certain title. All it needs to do is grab a hold of the movie and then call the remove method that we just wrote.

```
    //This method removes a movie (given its title) from the movie store
    public void removeMovieWithTitle(String aTitle) {
        if (movieList.get(aTitle) == null)
            System.out.println("No movie with that title");
        else
            removeMovie(movieList.get(aTitle));
    }
```

Well, perhaps the final thing we need to do is list the movies (or print them out). How do we do this ? What if we want them in some kind of order ? Perhaps any order, by actor/actress, or by type. Here's how to display them in the order that they were added to the MovieStore:

```
    //This method lists all movie titles that are in the store
    public void listMovies() {
        for (String s: movieList.keySet())
            System.out.println(s);
    }
```

As you can see, with the automatic type-casting due to the generics of JAVA 1.5, everything is easy. What about listing movies that star a certain actor/actress ? Well it just requires an additional search. Can you guess what HashMap is needed ?

```
    //This method lists all movies that star the given actor
    public void listMoviesWithActor(String anActor) {
        for (Movie m: actorList.get(anActor))
             System.out.println(m);
    }
```

Lastly, let us list all of the movies that belong to a certain category (type). For example, someone may wish to have a list of all comedy movies in the store. It is actually very similar to the actor version.

```
    //This method lists all movies that have the given type
    public void listMoviesOfType(String aType) {
        for (Movie m: typeList.get(aType))
            System.out.println(m);
    }
```

Ok, now we better test everything:

```
public class MovieStoreTester {
            public static void main(String args[]) {
                MovieStore  aStore = new MovieStore();
                aStore.addMovie(Movie.example1());
                aStore.addMovie(Movie.example2());
                aStore.addMovie(Movie.example3());
                aStore.addMovie(Movie.example4());
                aStore.addMovie(Movie.example5());
                aStore.addMovie(Movie.example6());
                aStore.addMovie(Movie.example7());
                aStore.addMovie(Movie.example8());
                aStore.addMovie(Movie.example9());
                aStore.addMovie(Movie.example10());

                System.out.println("Here are the movies in: " + aStore);
                aStore.listMovies();
                System.out.println();

                //Try some removing now
                System.out.println("Removing The Matrix");
                aStore.removeMovieWithTitle("The Matrix");
                System.out.println("Trying to remove Mark's Movie");
                aStore.removeMovieWithTitle("Mark's Movie");

                //Do some listing of movies
                System.out.println("\nHere are the Comedy movies in: " +
            aStore);
                aStore.listMoviesOfType("Comedy");
                System.out.println("\nHere are the Science Fiction movies in:
            " + aStore);
                aStore.listMoviesOfType("SciFic");
                System.out.println("\nHere are the movies with Ben
```

```
            Stiller:");
                aStore.listMoviesWithActor("Ben Stiller");
                System.out.println("\nHere are the movies with Keanu
            Reeves:");
                aStore.listMoviesWithActor("Keanu Reeves");
            }

    }
```

Here is the output:

```
Here are the movies in: MovieStore with 10 movies.
Envy
Blazing Saddles
The Matrix
The Matrix Reloaded
Meet the Fockers
Meet the Parents
Runaway Jury
The Matrix Revolutions
The Adventure of Sherlock Holmes' Smarter Brother
The Aviator

Removing The Matrix
Trying to remove Mark's Movie
No movie with that title

Here are the Comedy movies in: MovieStore with 9 movies.
Movie: "Blazing Saddles"
Movie: "The Adventure of Sherlock Holmes' Smarter Brother"
Movie: "Meet the Fockers"
Movie: "Meet the Parents"
Movie: "Envy"

Here are the Science Fiction movies in: MovieStore with 9 movies.
Movie: "The Matrix Reloaded"
Movie: "The Matrix Revolutions"

Here are the movies with Ben Stiller:
Movie: "Meet the Fockers"
Movie: "Meet the Parents"
Movie: "Envy"

Here are the movies with Keanu Reeves:
Movie: "The Matrix Reloaded"
Movie: "The Matrix Revolutions"
```