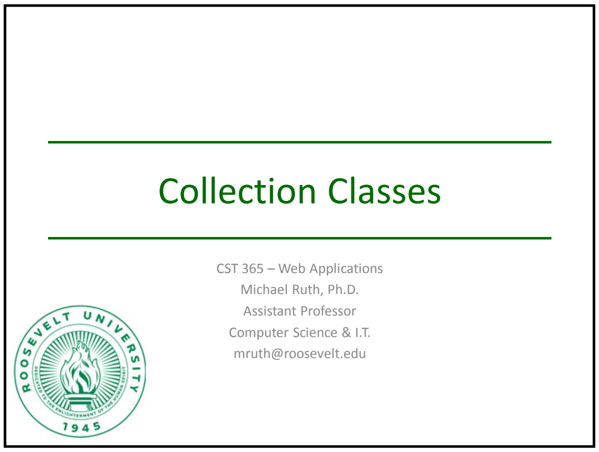
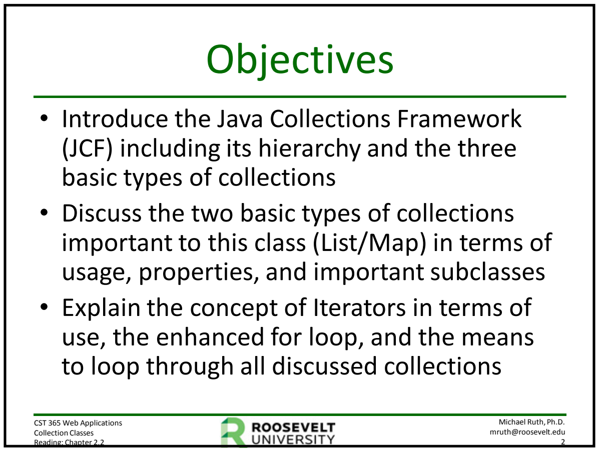


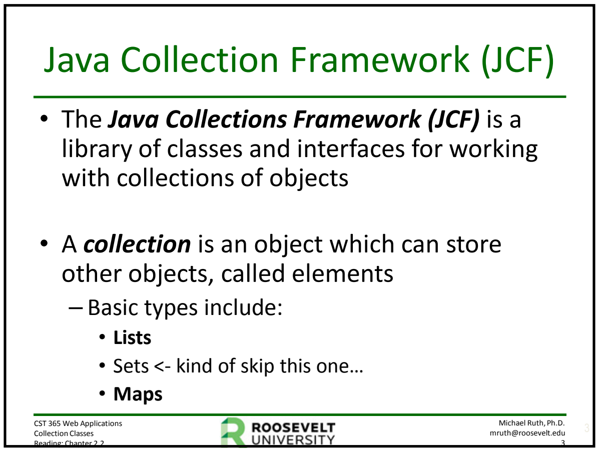
Collection Classes

CST 365 – Web Applications
Michael Ruth, Ph.D.
Assistant Professor
Computer Science & I.T.
mruth@roosevelt.edu



Objectives

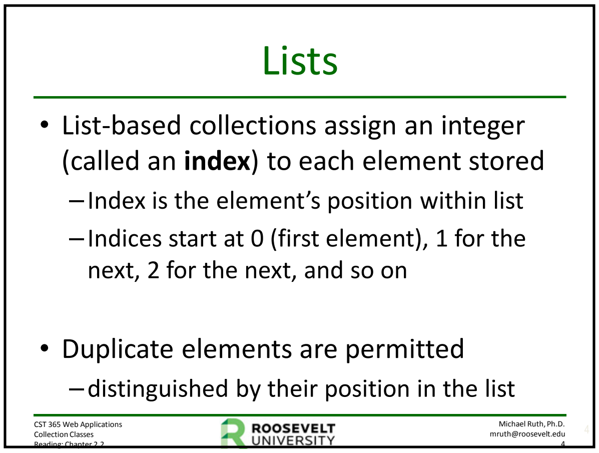
- Introduce the Java Collections Framework (JCF) including its hierarchy and the three basic types of collections
- Discuss the two basic types of collections important to this class (List/Map) in terms of usage, properties, and important subclasses
- Explain the concept of Iterators in terms of use, the enhanced for loop, and the means to loop through all discussed collections



Java Collection Framework (JCF)

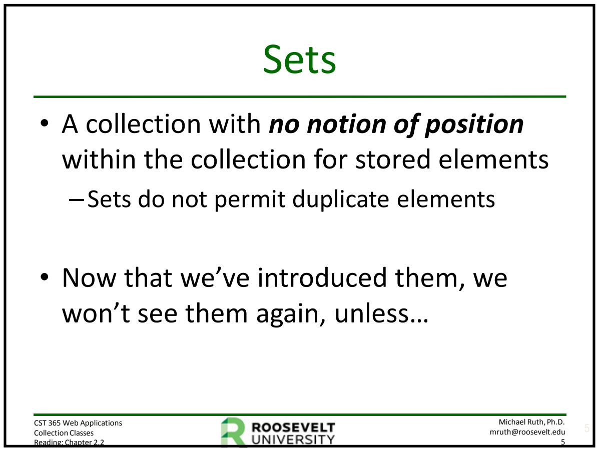
- The ***Java Collections Framework (JCF)*** is a library of classes and interfaces for working with collections of objects

- A ***collection*** is an object which can store other objects, called elements
  - Basic types include:
    - **Lists**
    - Sets <- kind of skip this one…
    - **Maps**



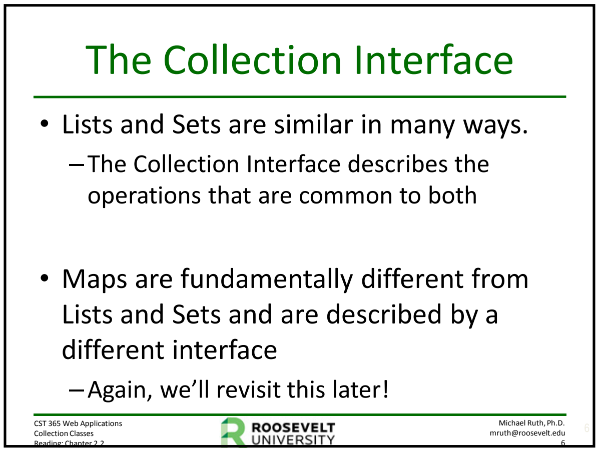
Lists

- List-based collections assign an integer (called an **index**) to each element stored
  - Index is the element’s position within list
  - Indices start at 0 (first element), 1 for the next, 2 for the next, and so on
- Duplicate elements are permitted
  - distinguished by their position in the list



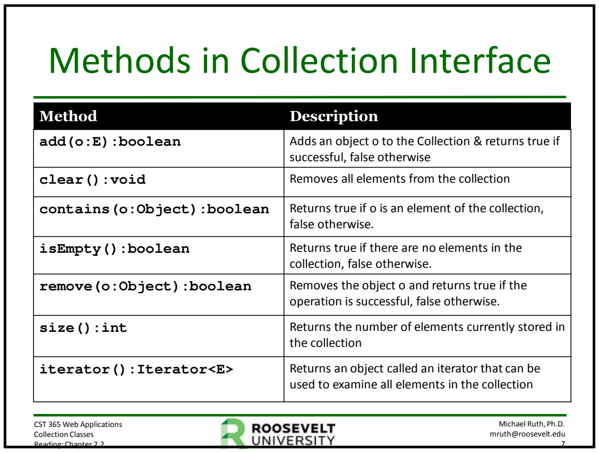
Sets

- A collection with ***no notion of position*** within the collection for stored elements
  - Sets do not permit duplicate elements
- Now that we’ve introduced them, we won’t see them again, unless…



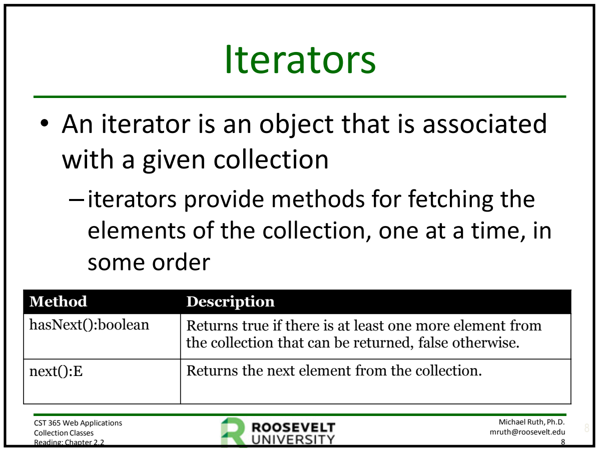
The Collection Interface

- Lists and Sets are similar in many ways.
  - The Collection Interface describes the operations that are common to both
- Maps are fundamentally different from Lists and Sets and are described by a different interface
  - Again, we’ll revisit this later!



Methods in Collection Interface

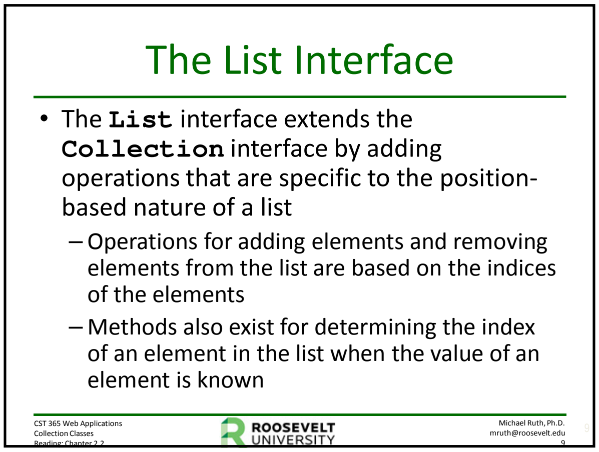
Method	Description
`add(o:E):boolean`	Adds an object o to the Collection & returns true if successful, false otherwise
`clear():void`	Removes all elements from the collection
`contains(o:Object):boolean`	Returns true if o is an element of the collection, false otherwise.
`isEmpty():boolean`	Returns true if there are no elements in the collection, false otherwise.
`remove(o:Object):boolean`	Removes the object o and returns true if the operation is successful, false otherwise.
`size():int`	Returns the number of elements currently stored in the collection
`iterator():Iterator<E>`	Returns an object called an iterator that can be used to examine all elements in the collection



Iterators

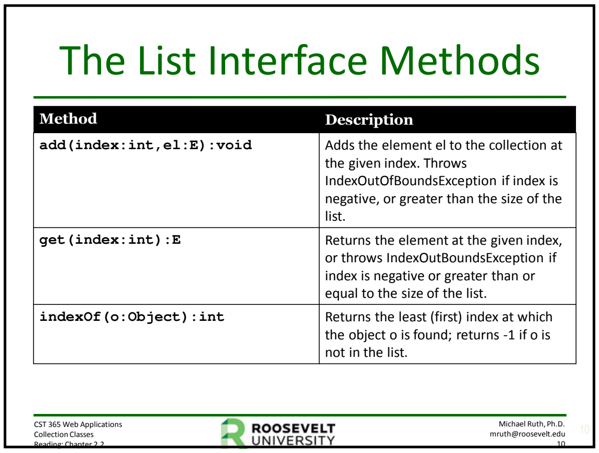
- An iterator is an object that is associated with a given collection
  - iterators provide methods for fetching the elements of the collection, one at a time, in some order

Method	Description
hasNext():boolean	Returns true if there is at least one more element from the collection that can be returned, false otherwise.
next():E	Returns the next element from the collection.



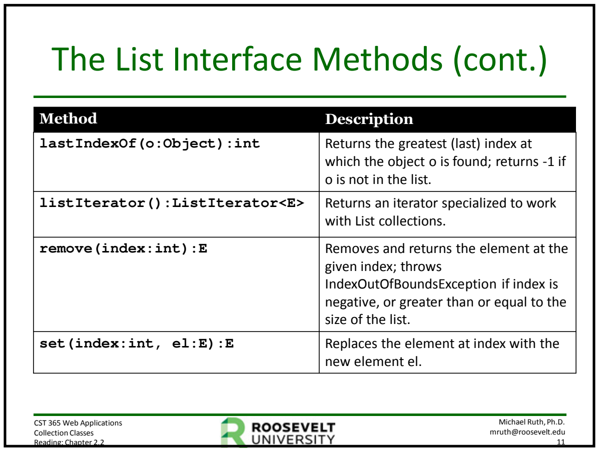
The List Interface

- The **`List`** interface extends the **`Collection`** interface by adding operations that are specific to the position-based nature of a list
  - Operations for adding elements and removing elements from the list are based on the indices of the elements
  - Methods also exist for determining the index of an element in the list when the value of an element is known



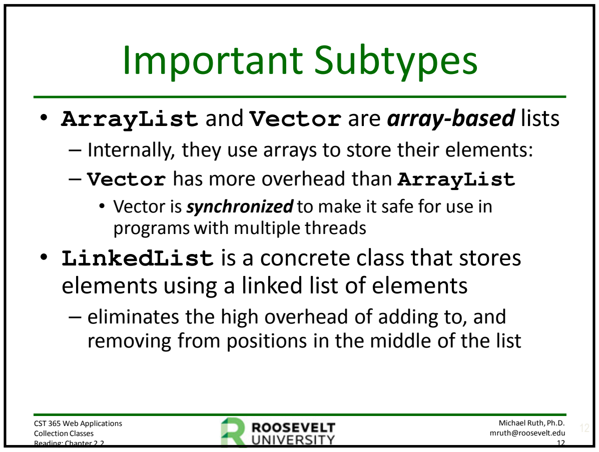
The List Interface Methods

Method	Description
`add(index:int,el:E):void`	Adds the element el to the collection at the given index. Throws IndexOutOfBoundsException if index is negative, or greater than the size of the list.
`get(index:int):E`	Returns the element at the given index, or throws IndexOutBoundsException if index is negative or greater than or equal to the size of the list.
`indexOf(o:Object):int`	Returns the least (first) index at which the object o is found; returns -1 if o is not in the list.



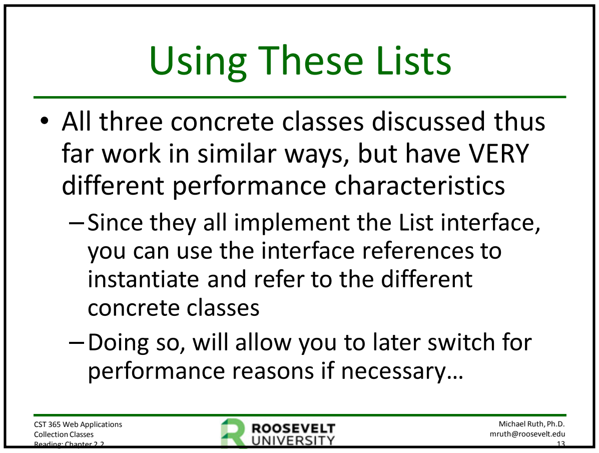
The List Interface Methods (cont.)

Method	Description
`lastIndexOf(o:Object):int`	Returns the greatest (last) index at which the object o is found; returns -1 if o is not in the list.
`listIterator():ListIterator<E>`	Returns an iterator specialized to work with List collections.
`remove(index:int):E`	Removes and returns the element at the given index; throws IndexOutOfBoundsException if index is negative, or greater than or equal to the size of the list.
`set(index:int, el:E):E`	Replaces the element at index with the new element el.



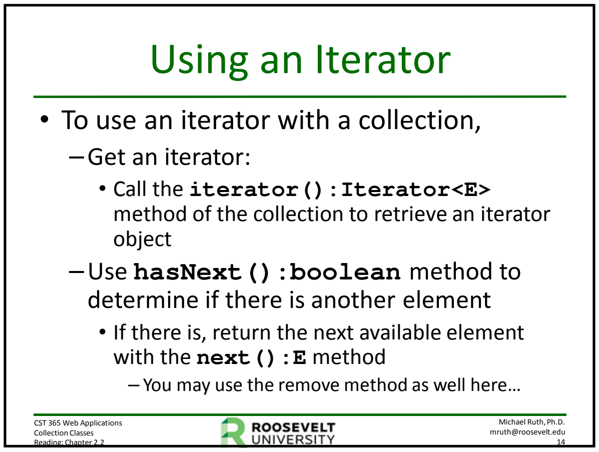
Important Subtypes

- **`ArrayList`** and **`Vector`** are ***array-based*** lists
  - Internally, they use arrays to store their elements:
  - **`Vector`** has more overhead than **`ArrayList`**
    - Vector is ***synchronized*** to make it safe for use in programs with multiple threads
- **`LinkedList`** is a concrete class that stores elements using a linked list of elements
  - eliminates the high overhead of adding to, and removing from positions in the middle of the list



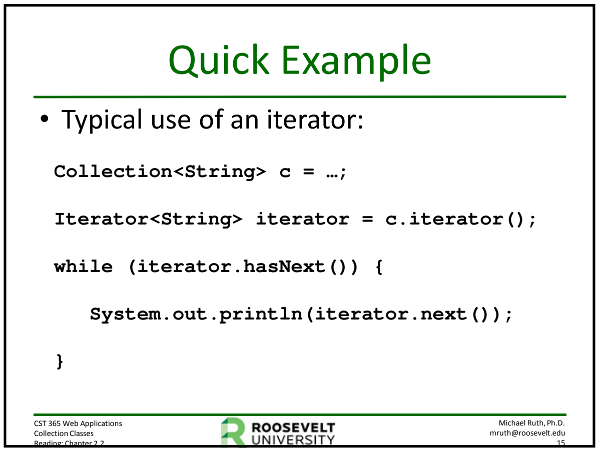
Using These Lists

- All three concrete classes discussed thus far work in similar ways, but have VERY different performance characteristics
  - Since they all implement the List interface, you can use the interface references to instantiate and refer to the different concrete classes
  - Doing so, will allow you to later switch for performance reasons if necessary…



Using an Iterator

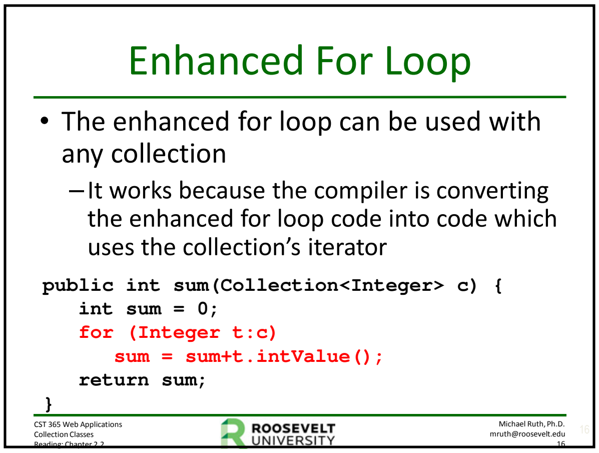
- To use an iterator with a collection,
  - Get an iterator:
    - Call the **`iterator():Iterator<E>`** method of the collection to retrieve an iterator object
  - Use **`hasNext():boolean`** method to determine if there is another element
    - If there is, return the next available element with the **`next():E`** method
      - You may use the remove method as well here…



Quick Example

- Typical use of an iterator:

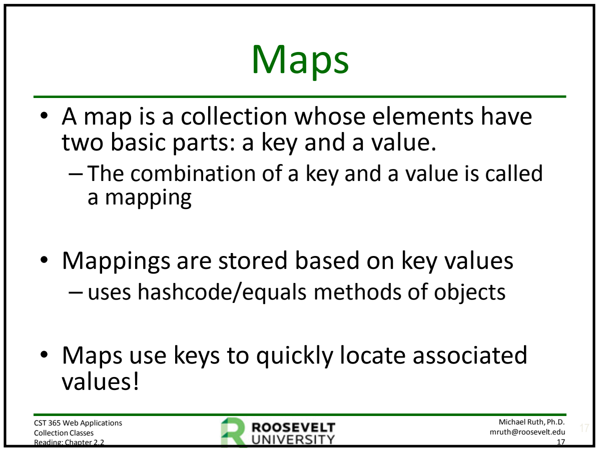
```
Collection<String> c = …;

Iterator<String> iterator = c.iterator();

while (iterator.hasNext()) {

   System.out.println(iterator.next());

}
```



Enhanced For Loop

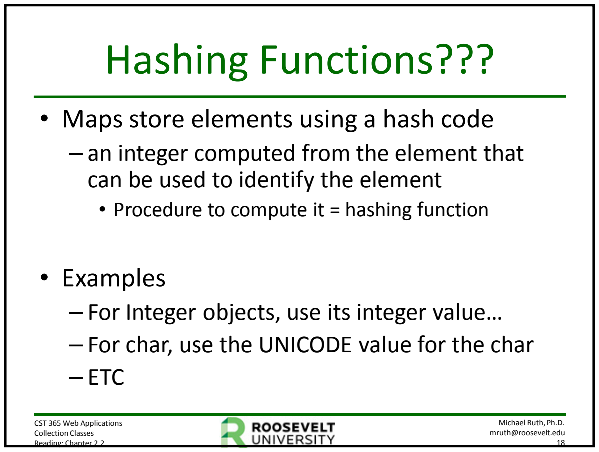
- The enhanced for loop can be used with any collection
  - It works because the compiler is converting the enhanced for loop code into code which uses the collection’s iterator

```
public int sum(Collection<Integer> c) {
   int sum = 0;
   for (Integer t:c)
      sum = sum+t.intValue();
   return sum;
}
```



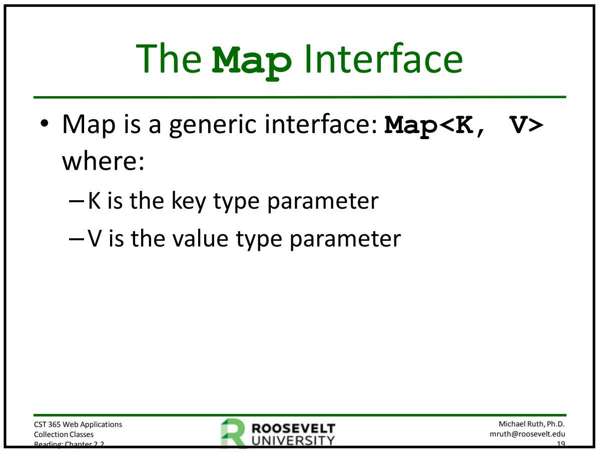
Maps

- A map is a collection whose elements have two basic parts: a key and a value.
  - The combination of a key and a value is called a mapping
- Mappings are stored based on key values
  - uses hashcode/equals methods of objects
- Maps use keys to quickly locate associated values!



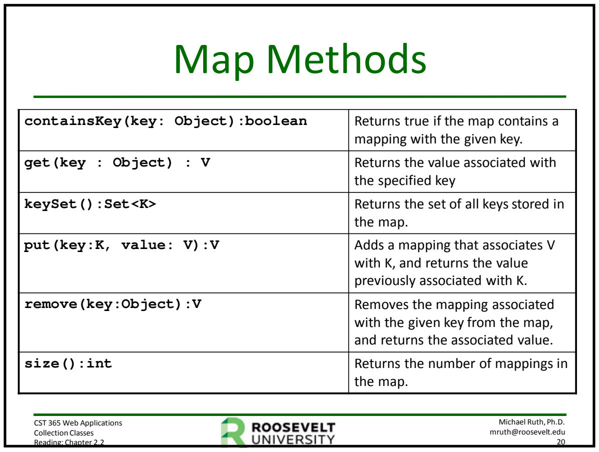
Hashing Functions???

- Maps store elements using a hash code
  - an integer computed from the element that can be used to identify the element
    - Procedure to compute it = hashing function
- Examples
  - For Integer objects, use its integer value…
  - For char, use the UNICODE value for the char
  - ETC



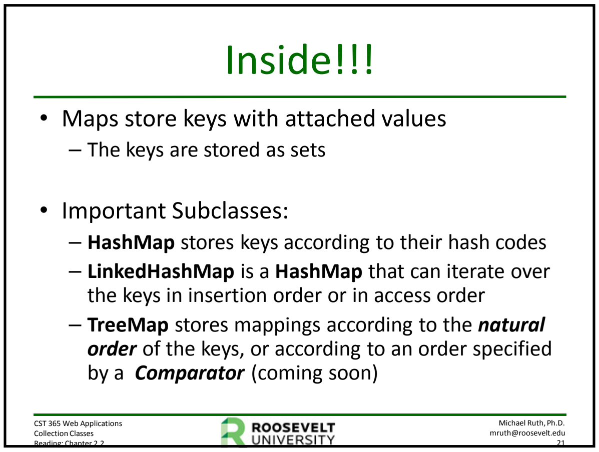
The **`Map`** Interface

- Map is a generic interface: **`Map<K, V>`** where:
  - K is the key type parameter
  - V is the value type parameter



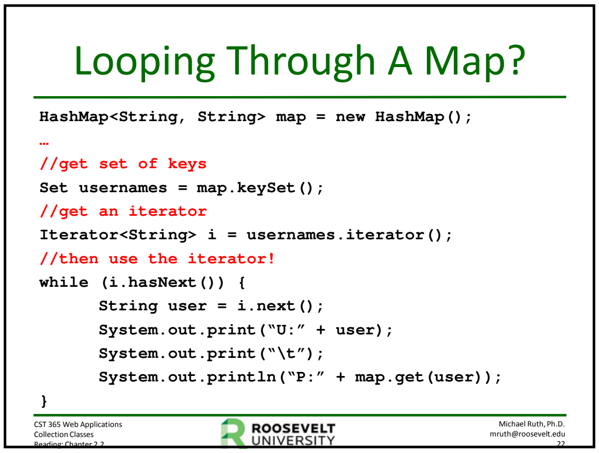
Map Methods

`containsKey(key: Object):boolean`	Returns true if the map contains a mapping with the given key.
`get(key : Object) : V`	Returns the value associated with the specified key
`keySet():Set<K>`	Returns the set of all keys stored in the map.
`put(key:K, value: V):V`	Adds a mapping that associates V with K, and returns the value previously associated with K.
`remove(key:Object):V`	Removes the mapping associated with the given key from the map, and returns the associated value.
`size():int`	Returns the number of mappings in the map.



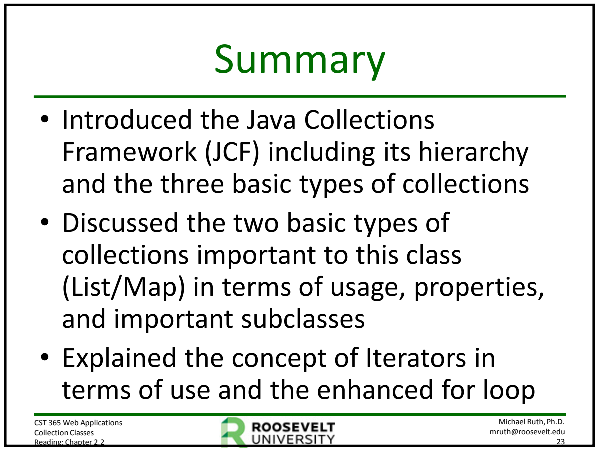
Inside!!!

- Maps store keys with attached values
  - The keys are stored as sets
- Important Subclasses:
  - **HashMap** stores keys according to their hash codes
  - **LinkedHashMap** is a **HashMap** that can iterate over the keys in insertion order or in access order
  - **TreeMap** stores mappings according to the ***natural order*** of the keys, or according to an order specified by a ***Comparator*** (coming soon)



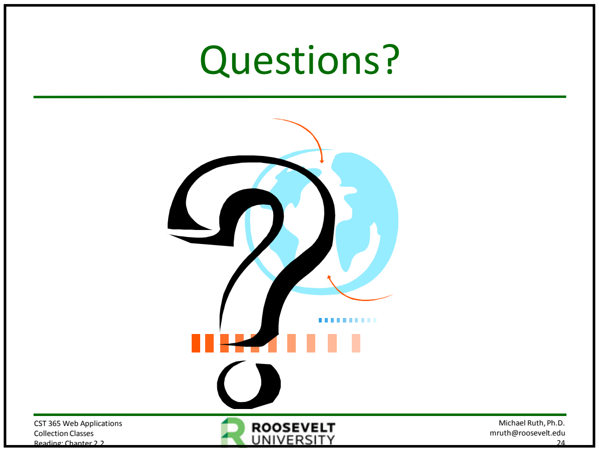
Looping Through A Map?

```
HashMap<String, String> map = new HashMap();
…
//get set of keys
Set usernames = map.keySet();
//get an iterator
Iterator<String> i = usernames.iterator();
//then use the iterator!
while (i.hasNext()) {
      String user = i.next();
      System.out.print(“U:” + user);
      System.out.print(“\t”);
      System.out.println(“P:” + map.get(user));
}
```



Summary

- Introduced the Java Collections Framework (JCF) including its hierarchy and the three basic types of collections
- Discussed the two basic types of collections important to this class (List/Map) in terms of usage, properties, and important subclasses
- Explained the concept of Iterators in terms of use and the enhanced for loop



Questions?