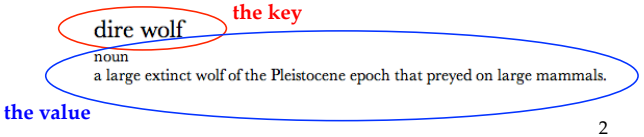
CSE 1720

Lecture 23

The Collections Framework, Revisited

10.1.1 The Map Interface

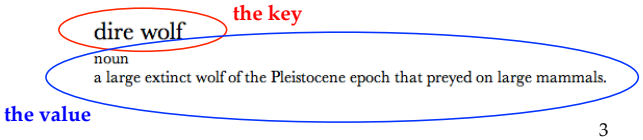
- a *collection* of **pairs**
- a pair has two elements: the **key** and the **value**
- an example of a map
  - the dictionary is a list of pairs
  - the keys are sorted in lexicographic order



10.1.1 The Map Interface

- Maps are parameterized
  - When declaring a map, we need to specify:
    - the type of the keys
    - the type of the values

```
Map<String, String> theDictionary;
Map<String, Integer> theTally;
```



Review of Generics

- All classes in the collections framework support generics.
- By specifying the type (between < and >) the client ensures:
  - No rogue element can be inserted
  - No casting is needed upon retrieval

```
List<Stock> bag = new ArrayList<Stock>();
// bag.add("Hello"); will not compile!
bag.add(new Stock(".ab"));
Stock s = bag.get(0); // no cast!
```

Sets

- The interface is `Set`
- Implementing classes are `HashSet` and `TreeSet`

```
Set<String> s = new HashSet<String>();
Set<String> s = new TreeSet<String>();
```

Declaration Instantiation

Sets

- Best Practises:
  - *Declaration* as high up the hierarchy as possible
  - *Instantiation* lower in the hierarchy
- The following is ok; it will compile
  - but it is not using best practises

```
HashSet<String> s = new HashSet<String>();
TreeSet<String> s = new TreeSet<String>();
```


Lists

- The interface is `List`
- Implementing classes are `ArrayList`, `LinkedList`, and `Vector`

```
List<String> s = new ArrayList<String>();
List<String> s = new LinkedList<String>();
List<String> s = new Vector<String>();
```

Declaration Instantiation

Use LinkedList only if your app tends to add or remove elements at index 0

Maps

- The interface is `Map`
- The basic operations
  - `put` a pair into the `Map`
  - `remove` a pair
  - given a key, `get` its corresponding value
  - iterate *over the keys*
  - iterate *over the values*

Maps

- `put` a pair into the `Map`
  - what if the key is already present?
- `remove` a pair
  - what if the key is not present?
- given a key, `get` its corresponding value
  - what if the key is not present?

`HashMap` and `TreeMap`

- Main differences:
  - iterating over keys
    - `HashMap`: keys **will not be** sorted
    - `TreeMap`: keys **will be** sorted
  - putting involves a test to determine whether the key is already present
    - `HashMap`: test on the basis of
      - `hashCode()` first, if not equals, then `.equals(Object)`
    - `TreeMap`: test on the basis of `compareTo()`

10.2.1 API Highlights

- Use put to add an element to a map

```
Map<Integer, String> map;
map = new HashMap<Integer, String>();
map.put(55, "Clock Rate");
```


API Highlights

- To delete a map element given its key:

```
String gone = map.remove(55);
```

Note that remove in maps returns the value of the element that was removed or null if the specified key was not found.

The Iterator in Maps

The Map interface has no iterator() method but we can obtain a set of the map's keys:

```
public Set<K> keySet()
```

And by iterating over the obtained set, we can, in effect, iterate over the map's elements:

```
Iterator<Integer> it = map.keySet().iterator();
for (; it.hasNext();)
{
   int key = it.next();
   String value = map.get(key);
   output.println(key + " --> " + value);
}
```


10.2.4 Summary of Features

LIST	SET	MAP
Adding Elements		
boolean add(E e) void add(int index, E e)	boolean add(E e)	V put(K key, V value)
Removing Elements		
boolean remove(E e) E remove(int index)	boolean remove(E e)	V remove(K key)
Accessing an Element		
E get(int index)	*none*	V get(K key)
Searching the Elements		
boolean contains(E o)	boolean contains(E o)	boolean containskey(K key)
Traversing the Elements		
Iterator iterator() *invoke on it:* E next() boolean hasNext()	Iterator iterator() *invoke on it:* E next() boolean hasNext()	Iterator keySet().iterator() *invoke on it:* E next() boolean hasNext()
Other methods (available in all three interfaces)		
equals, size, toString		
Algorithms for lists only (static *methods in the* Collections *class)*		
binarySearch, copy, fill, reverse, shuffle, sort		

List-Only Utilities in Collections

- **Sort / binarySearch**
  We covered these
- **shuffle**
  Rearranges the elements randomly
- **reverse**
  Rearranges the elements in reverse order
- **copy**
  Returns a deep copy of the collection
- **fill**
  Populates all the elements with a given value