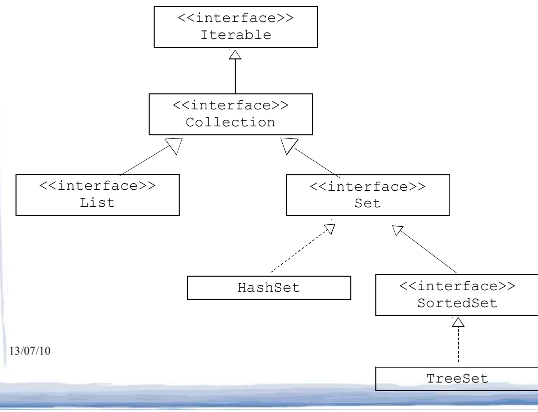
Review

- Sets
  - Collections without duplicates
- HashSet
  - Overriding hashCode() and equals()
- TreeSet
  - Keeps things sorted
  - Implementing Comparable or Comparator

The `Set` interface



Methods of the `Set` interface

- Note that the `Set` interface extends the `Collection` interface. An implementation of `Set` therefore supports the methods defined in the `Collection` interface:
  - `add(o)` – add a specified element to the set (if not already a member)
  - `remove(o)` – remove the specified element from the set
  - `contains(o)` – is the specified element in the set?
  - etc.
- Note that the add method:

```
public boolean add( E item );
```

  adds the item only if it isn't already in the set. The method returns true if the item is added and false if it's already in the set.
- Similarly the `addAll` method does not add duplicates.

Methods of the `Set` interface

- We can use these methods to define known set operations:
  - `c1.containsAll(c2)` - true if `c2` is a subset of `c1`
  - `c1.addAll(c2)` - `c1` becomes union of c1 and c2
  - `c1.retainAll(c2)` - `c1` becomes intersection of c1 and c2
  - `c1.removeAll(c2)` - `c1` becomes set difference of c1 and c2

Using Sets

- Since Set is specified as an interface, to use it we have to pick a particular implementation (e.g., `HashSet`, or `TreeSet`)
- Example:

```
public class PlayList
{
private Set<Song> songs;

public PlayList()
{
       songs= new HashSet< Song >();
}
```


The `HashSet` implementation

- The `HashSet` implementation provides an efficient implementation of the `Set` interface that allows us to add or remove an item or check if the set contains an item in O(1) time provided certain conditions are met (more later).

- That is, if
  `Set<…>  s = new HashSet<…>();`
  - `s.add(o)` is O(1)
  - `s.remove(o)` is O(1)
  - `s.contains(o)` is O(1)

The `HashSet` implementation

- As mentioned, certain conditions must be met if we are to add, remove or determine if the set contains an item in O(1) time.

- To understand these conditions, we must have a basic understanding of how the hash set works.

- A hash set uses a ***hash table*** as the underlying structure in which data is stored.

- A hash table is an array of linked lists…

The `HashSet` implementation

- We add elements to the table using a hash code, an integer that represents the object

- A hash set maintains a list of groups.
- All members of the group at position i have a hashCode of i.
- We'll talk more in a moment about where these hashCodes come from

The `HashSet` implementation

- In a HashSet the operations are performed as following:

`add(o)`
  - compute the hashcode of o, say i
  - add o in the ith group

`remove(o)`
  - compute the hashcode of o, say i
  - search the ith group and remove o

`contains(o)`
  - compute the hashcode of o, say i
  - search the ith group to find o

- If each group is small (and of constant size) each of the above operations is O(1).

The `HashSet` Implementation

- What makes these operations so efficient?
  - Take add() for example
- Rather than iterating over a collection and checking at each step whether the object already exists, we just compute the hashCode and check that index in the array
- We then check whether the object exists in that bucket
- If we have a good hashCode and hash table, there will be few collisions, meaning few items to search through in the bucket
- If we can get close to 1 item per bucket, these operations will be 0(1) – constant time

Where do hash codes come from?

- Each Java class inherits a hashCode() method from the Java class Object
  - when invoked, hashCode() returns an integer that represents the object
  - a class' hashCode() is usually defined in terms of the hash codes of its attributes
  - if two objects are equal according to equals(), they must have the same hash code
  - objects with the same hash code are not necessarily equal
- It would be nice to rely upon the Java Object's class definition of hashCode() but you can't if you override equals() because two instances of an object that are equal according to equals() may not return the same hashCode() unless you ensure they do!
- The rule is then:
  "If you override equals() you should always override hashCode()"
  - See page 36 of http://java.sun.com/developer/Books/effectivejava/Chapter3.pdf for a complete description

Default hashCode() and equals()

- If you rely on the default inherited equals() and hashCode(), you are okay in the sense that they both rely on the memory location of the object and are therefore consistent with one another
- But then you are left with a very restrictive definition of equals() which might not be what you want

How do you write a good hashCode()

- Writing a fantastic hashCode() method for a class is hard
  - The kind of thing people write PhD theses about
- Writing a decent hashCode() method for a class is straightforward
  - Page 38 of http://java.sun.com/developer/Books/effectivejava/Chapter3.pdf provides a recipe.
    - Start with a non-zero value (preferably a prime number, like 11, 17, etc.) in the result value
    - Pick another prime number, say 37, as a multiplier
    - For each attribute that is taken into account in the equals() method
      - if attribute is of a primitive type (i.e. an integer, float, etc.) , **result = 37 * result + attribute's value casted to an integer**
      - if attribute is an object, **result = 37 * result + attribute.hashCode()**
      - and so on…

The `SortedSet` Interface

- Allows the user to retrieve objects from the set in sorted order
- To sort a collection, the objects within the collection must be comparable:
  - the corresponding class must implement either the `Comparable` interface or the `Comparator` interface.

The `Comparable` Interface

- The `Comparable` interface is declared as follows:

```
public interface Comparable<T> {
   int compareTo(T other)
}
```

- the integer returned by `a.compareTo(b)` must adhere to the following convention:
  - negative if a < b
  - zero if `a.equals(b)`
  - positive if a > b
- `compareTo` defines the ***natural ordering*** for the class

compareTo()

- If `a` already exists in a set, and you try to add `b`, and a.compareTo(b) == 0, `b` will not be added
- These are considered duplicates

Implementing compareTo

- Rules to follow when you implement this method in a class C:
  - C must implement Comparable<C>
  - must be asymmetric
    - a.compareTo(b) and b.compareTo(a) must both equal 0 or have opposite signs
  - must be transitive
    - if a.compareTo(b) < 0 and b.compareTo(c) < 0 then a.compareTo(c) < 0
  - must be consistent with equals()
    - a.equals(b) is true iff a.compareTo(b) is zero and b.compareTo(a) is zero

The Comparator Interface

- Some classes may not have a single natural ordering
  - employees may be ordered by name or by salary or…
- A comparator is an object that defines (encapsulates) one ordering for a class
- A comparator has to implement:

```
public interface Comparator<T> {
    int compare(T object1, T object2);
}
```


The Comparator Interface

- The return value for this method
  - is defined in the same way as for the compareTo method of the Comparable interface:

compare(a,b) is like *a.compareTo(b)*

- We may define many comparators for a class if we need to order objects of that type in different ways.

Example

- Create a Comparator that compares Accounts by id numbers.

```
public class AccountIdComparator
             implements Comparator<Account>
{
   public int compare( Account ac1, Account ac2 )
   {
      return (ac1.getId() - ac2.getId() );
   }
}
```

The `SortedSet` Interface

```
public interface SortedSet<E> extends Set<E>
{
  // Views on the sorted set
  SortedSet<E>  subSet(E from, E to);
  SortedSet<E>  headSet(E toElement);
  SortedSet<E>  tailSet(E fromElement);

  // Endpoints
  E first();
  E last();

  // Comparator access
  Comparator<? super E>  comparator();
}
```


The `SortedSet` Interface

- Like `Set` but keeps elements in ascending order according to
  - the *natural order* defined by the `compareTo` method of `Comparable`, or
  - the `compare` method of a `Comparator`
- Iterator will traverse elements in the defined order
- Array produced by `toArray` methods is sorted
- Additional operations:
  - `first()` and `last()` return min and max elements in set
  - `comparator()` returns the `Comparator` used to sort the set, or `null` if the *natural order* is used

The `TreeSet` Class

- The `TreeSet` class implements the `SortedSet` interface. It has the following constructors (among others):

```
public TreeSet()
// orders the elements according to their
// natural order


public TreeSet( Comparator< ? super E > c )
// orders the elements according to the
// comparator c
```


The `TreeSet` Class

- Note the use of the bounded wildcard:
  - Comparator< ? super E > c
- This indicates that the `Comparator` must compare types that are supertypes of `E` (including `E` itself).
- For example, if `SavingsAccount` is a subclass of `Account` and `BalanceComparator` implements the `Comparator<Account>` interface, then we can create the following `TreeSet` of `SavingsAccount` objects:

```
TreeSet<SavingsAccount> accts
      = new TreeSet<SavingsAcount>(
                  new BalanceComparator() );
```

TreeSet - Time Complexity

- The add, remove and contains methods all have a guaranteed O(log N) time complexity.
- So these operations on a `TreeSet` are less efficient than for a `HashSet` (assuming a good `hashCode()` implementation) but we have to remember that the `TreeSet` maintains the data in sorted order.

TreeSet vs. HashSet

- If you don't care about sorting but just want efficient add(), remove() and contains() operations, the question of which Set to use depends on how confident you are in your hash code method
- If you have a good hash code, there will be few collisions, which means few objects in each bucket, which means less to search through
- Otherwise, you might want to use a TreeSet

Using TreeSet

- Now we can do this:

```
Set<Golfer> gSet = new TreeSet<Golfer>();
gSet.add(bob);
gSet.add(jane);
gSet.add(jim);
Iterator<Golfer> itr = gSet.iterator();
while (itr.hasNext())
{
System.out.println(itr.next().getName());
}
```

Golfer implements Comparable

Using TreeSet

- Or we can supply a Comparator

```
Set<Golfer> gSet = new TreeSet<Golfer>(new HandicapComparator());
gSet.add(bob);
gSet.add(jane);
gSet.add(jim);
Iterator<Golfer> itr = gSet.iterator();
while (itr.hasNext())
{
System.out.println(itr.next().getName());
}
```

Using TreeSet

- A different Comparator if we choose...

```
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(bob);
gSet.add(jane);
gSet.add(jim);
Iterator<Golfer> itr = gSet.iterator();
while (itr.hasNext())
{
System.out.println(itr.next().getName());
}
```


Using TreeSet

- Now Java will use either the compareTo() method if we implement Comparable, or the compare() method if we use Comparators, and will keep our items nicely sorted
- Whenever we add something, Java will determine where it belongs by calling those methods
- Note: if we don't supply a Comparator and our class doesn't implement Comparable, we will get an error. We need one or the other.

compareTo() and equals()

- We stated earlier that one rule for implementing compareTo() - and compare() - is that it must be consistent with equals()
- a.equals(b) and (a.compareTo(b) == 0) should have the same boolean value
- If these are not consistent, the Set contract can be violated and you can end up with strange behaviour
- *In practice*, however, you will often see compareTo() that is not consistent with equals()

compareTo() and equals()

- If you define compareTo() or compare() in a way that is not consistent with equals(), you should note this in the comments for that method
- You should also be aware of the behaviour that can result
- For example, when trying to add b to a set containing a, if (!a.equals(b)) and (a.compareTo(b) == 0), b will not be added even though they are not equal according to equals()

Inconsistency I – Golfer example

- Say we define the Golfer equals() method as such:

```
public boolean equals(Object other)
   {
      if (other == null) return false;
      if (other.getClass() != getClass()) return false;
      Golfer og = (Golfer) other;
      return (this.name.equals(og.getName()) && this.handicap == og.getHand());
   }
```


Inconsistency I – Golfer example

- Then we use a Comparator based on the best score attribute

```
public class BestScoreComparator implements Comparator<Golfer>
{

public int compare(Golfer g1, Golfer g2)
{
return g1.getBest() - g2.getBest();
}
}
```


Inconsistency I – Golfer Example

- So Golfers are considered equal if they have the same name and same handicap
- And the Comparator sorts Golfers based on their best scores, with the compare() method returning 0 when Golfers have the same score
- As far as the sorted set is concerned, this 0 value means that the Golfers are equal, and so this is what will happen...


```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer betty = new Golfer(8, 76, "betty");
gSet.add(betty);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(betty.equals(jane));
```

Create a couple Golfers

```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer betty = new Golfer(8, 76, "betty");
gSet.add(betty);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(betty.equals(jane));
```

Create a new TreeSet with a BestScoreComparator


```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer betty = new Golfer(8, 76, "betty");
gSet.add(betty);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(betty.equals(jane));
```

Create a new Golfer Betty who happens to have the same best score as Jane


```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer betty = new Golfer(8, 76, "betty");
gSet.add(betty);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(betty.equals(jane));
```

Now what output do we get if we iterate through the set and print out each Golfer's name and best score?


```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer betty = new Golfer(8, 76, "betty");
gSet.add(betty);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(betty.equals(jane));
```

```
>
jane 76
jim 105
```

Betty never got added to the set, because the BestScoreComparator considered her and Jane to be equal (returned 0)

```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer betty = new Golfer(8, 76, "betty");
gSet.add(betty);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(betty.equals(jane));
```

>
jane 76
jim 105
false

Even though jane and betty are not considered equal according to the equals() method

Inconsistency II – Golfer Example

- On the other hand, if a.equals(b) and (a.compareTo(b)) != 0), b *will* be added to the set even though a and b are equal
- For example, using the same Golfer example, with the same equals() method and same BestScoreComparator, this could happen...


```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer jane2 = new Golfer(5, 88, "jane");
gSet.add(jane2);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(betty.equals(jane));
```

We create a new Golfer with the same name and handicap as jane, thus making them equal according to the equals() method


```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer jane2 = new Golfer(5, 88, "jane");
gSet.add(jane2);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(betty.equals(jane));
```

However, they have different best scores, so compare() will return a non-zero value.
The golfer jane2 gets added to the set even though jane and jane2 are equal

```
Golfer jane = new Golfer(5, 76, "jane");
Golfer jim = new Golfer(15, 105, "jim");
Set<Golfer> gSet = new TreeSet<Golfer>(new
BestScoreComparator());
gSet.add(jane);
gSet.add(jim);

Golfer jane2 = new Golfer(5, 88, "jane");
gSet.add(jane2);

for (Golfer g: gSet)
{
System.out.print(g.getName());
System.out.println(" "+g.getBest());
}
System.out.println(jane2.equals(jane));
```

```
>
jane 76
jane 88
jim 105
true
```


compareTo() and equals()

- By making compareTo() and equals() inconsistent, we now have unexpected behaviour for sets
- So it's recommended to make them consistent
- Otherwise, just be aware of this behaviour

Maps, Stacks, Queues and Generic Algorithms

You will be expected to:
- program to the generic `Map` and `SortedMap` interfaces by reading and using the API
- compare and contrast `HashMap` and `TreeMap` classes (benefits of using each, basic run time analysis)
- program to the generic `Queue` interface
- program to the API of the generic `Stack` class
- program to the API of the generic `Deque` class
- identify (in words or through code) appropriate types for collections of data needed in a given software system
- write code (solve problems) that uses the generic algorithms provided in the `Collections` class

Reading:
- 2nd Ed: 20.4, 21.2, 21.7
- 3rd Ed: 15.4, 16.2, 16.7

Additional references:
Online Java Tutorial at
http://java.sun.com/docs/books/tutorial/collections/

The `Map` Interface

- A ***map*** structure is also known as a ***table*** or ***dictionary*** or ***association list***.
- A map is a collection of pairs (key, value).
  - The keys are unique within the map
  - The map associates exactly one value with each key
- Examples:
  - map of student ids to student records
  - map of words to frequency of occurrence in a document

The `Map` Interface

- The `Map` interface is *not* related to the other interfaces in the Java Collections Framework:
  - `Map` does not extend `Iterable` (so you can't iterate over a map or use the for-each loop)
  - `Map` does not extend `Collection`
  - a `Collection` operates on items
  - a `Map` manipulates (key, value) pairs

The `Map` Interface (cont')

```
public interface Map<K,V>
{
   // Basic Operations
   int size();
   boolean isEmpty();
   boolean containsKey(Object key);
   boolean containsValue(Object value);

   V  get(Object key);
   V  remove(Object key);
   V  put(K key, V value);
   // Bulk Operations
   void clear();
   void putAll(Map<? extends K, ? extends V> m);
```


The `Map` Interface (cont')

```
   // Collection Views
   Set<K> keySet();
   Collection<V> values();
   Set<Map.Entry<K,V>> entrySet();

   // Interface for entrySet elements; defined inside Map
   interface Entry<K,V>
   {
      K  getKey();
      V  getValue();
      V  setValue(V value);
   }
}
```


`Map` Methods

- Basic methods:
  - `put` adds a (key, value) entry; returns previous value for that key or `null`
  - `containsKey`, `containsValue` check if a key or value is in the map
  - `get` returns the value of a given key; returns `null` if key is not in the map
    - problems if map allows `null` values; must use `containsKey`
  - `remove` removes the entry for that key; returns the value removed or `null` if the map doesn't contain the given key

`Map` Methods

- Collection Views: restructure the map (or parts of it) as a `Collection` so we can use iterators:
  - `keySet` returns a `Set` containing all keys in the map
  - `values` returns a `Collection` with all map values (may have duplicates)
  - `entrySet` returns a `Set` of entries; each entry represents a (key, value) pair and it is defined by the interface `Entry` defined inside `Map`

`Map` Implementations

- The class `HashMap` provides an implementation of the `Map` interface.
- The underlying implementation is similar to a `HashSet` (it uses a hash table)
- When a (key, value) pair is added to a `HashMap`, the hash code is generated using only the key (not the value)
- Assuming a good `hashCode()` method for the `Key`, the `put`, `get` and `remove` methods run in O(1) time.

`Map` Examples

- A generic method that prints out all the key-value pairs of a `Map`:

```
public static <K,V> void printMap( Map<K,V> theMap )
{
   for(Map.Entry<K,V> e : theMap.entrySet()) {
      System.out.println(e.getKey() + ": "
                              + e.getValue());
}
```


`Map` example, cont'd

```
public static void main(String[] args)
    {
     Map<String, String> ourJobs = new
HashMap<String, String>();
     ourJobs.put("Grover", "researcher");
     ourJobs.put("Geneva", "librarian");
     ourJobs.put("Gina", "architect");
     printMap(ourJobs);
    }
```

`Map` example, cont'd

```
public static void main(String[] args)
    {
     Map<String, String> ourJobs = new
HashMap<String, String>();
     ourJobs.put("Grover", "researcher");
     ourJobs.put("Geneva", "librarian");
     ourJobs.put("Gina", "architect");
     printMap(ourJobs);
    }
```

Gina: architect
Geneva: librarian
Grover: researcher

`Map` Examples

- We could have used an Iterator instead (same output)

```
public static <K, V> void printMapIterator(Map<K, V> theMap)
{
Iterator<Map.Entry<K, V>> i = theMap.entrySet().iterator();
while (i.hasNext()) {
  Map.Entry<K, V> e = i.next();
  System.out.println(e.getKey() + ": " + e.getValue());
}
}
```


What gets printed here?

```
Map<String, String> ourJobs = new
HashMap<String, String>();
ourJobs.put("Grover", "researcher");
ourJobs.put("Geneva", "librarian");
ourJobs.put("Gina", "architect");
ourJobs.put("Gina", "dog trainer");
ourJobs.remove("Grover");
printMapIterator(ourJobs);
```


What gets printed here?

```
Map<String, String> ourJobs = new
HashMap<String, String>();
ourJobs.put("Grover", "researcher");
ourJobs.put("Geneva", "librarian");
ourJobs.put("Gina", "architect");
ourJobs.put("Gina", "dog trainer");
ourJobs.remove("Grover");
printMapIterator(ourJobs);
```

>
Gina: dog trainer
Geneva: librarian

What gets printed here?

```
Map<String, String> ourJobs = new
HashMap<String, String>();
ourJobs.put("Grover", "researcher");
ourJobs.put("Geneva", "librarian");
ourJobs.put("Gina", "architect");
ourJobs.put("Gina", "dog trainer");
ourJobs.remove("Grover");
printMapIterator(ourJobs);
```

Gina gets overwritten here. You can't have duplicate keys.

>
Gina: dog trainer
Geneva: librarian

What gets printed here?

```
Map<String, String> ourJobs = new
HashMap<String, String>();
ourJobs.put("Grover", "researcher");
ourJobs.put("Geneva", "librarian");
ourJobs.put("Gina", "architect");
ourJobs.put("Gina", "dog trainer");
ourJobs.remove("Grover");
printMapIterator(ourJobs);
```

Grover gets removed – both the key and its value.

>
Gina: dog trainer
Geneva: librarian

What gets printed here?

- Notice that `put` has a return type of the previous value for that key, or null

```
Map<String, String> ourJobs = new HashMap<String,
String>();
ourJobs.put("Grover", "researcher");
ourJobs.put("Geneva", "librarian");
ourJobs.put("Gina", "architect");
String ginaOld = ourJobs.put("Gina", "dog trainer");
System.out.println("Gina used to be an "+ginaOld);
System.out.println("Now she is a "+ourJobs.get("Gina"));
```

Get the previous value

What gets printed here?

- Notice that `put` has a return type of the previous value for that key, or null

```
Map<String, String> ourJobs = new HashMap<String,
String>();
ourJobs.put("Grover", "researcher");
ourJobs.put("Geneva", "librarian");
ourJobs.put("Gina", "architect");
String ginaOld = ourJobs.put("Gina", "dog trainer");
System.out.println("Gina used to be an "+ginaOld);
System.out.println("Now she is a "+ourJobs.get("Gina"));
```

Get current value for key “Gina”

What gets printed here?

- Notice that `put` has a return type of the previous value for that key, or null

```
Map<String, String> ourJobs = new HashMap<String,
String>();
ourJobs.put("Grover", "researcher");
ourJobs.put("Geneva", "librarian");
ourJobs.put("Gina", "architect");
String ginaOld = ourJobs.put("Gina", "dog trainer");
System.out.println("Gina used to be an "+ginaOld);
System.out.println("Now she is a "+ourJobs.get("Gina"));
```

Gina used to be an architect
Now she is a dog trainer

In-Class Exercise I

- A function that returns a map of words to the number of times each word occurs in a text document (doc):

```
public static Map<String, Integer>
                 frequencies(Collection<String>  doc)
{
    Map<String,Integer> map = new HashMap<String,Integer>();
    for (String word : doc) {
       if ( ________________________________ )
          ____________________________________
       else
          ____________________________________
    }
    return map;
}
```


Tea break!

Sorted Map

- The `SortedMap` interface extends the `Map` interface.
- A `SortedMap` maintains the entries in the map sorted by their key (not their value)
- The class `TreeMap` implements the `SortedMap` interface
- `TreeMap` uses a similar underlying implementation to `TreeSet`. The `get`, `put` and `remove` operations run in O(log N) time.

Sorted Map

- The `TreeMap<K,V>` has the following constructors (note the similarity with `TreeSet`):

```
• public TreeMap()
  // orders the entries using the natural
  // ordering for the Key

• public TreeMap( Comparator< ? super K > c )
  // orders the entries using the given
  // comparator for the Key
```


SortedMap Interface

```
public interface SortedMap<K, V>  extends Map<K, V>
{
   // Range-view
   SortedMap<K, V>  subMap(K fromKey, K almostToKey);
   SortedMap<K, V>  headMap(K almostToKey);
   SortedMap<K, V>  tailMap(K fromKey);
   // Endpoints
   K  firstKey();
   K  lastKey();
   // Comparator access
   Comparator<? super K>  comparator();
}
```


SortedMap Example

```
Map<Golfer, String> golfTeams = new TreeMap<Golfer,
String>(new HandicapComparator());
```

Let's create a map associating Golfers with team names.

We'll supply the HandicapComparator that we defined before.

SortedMap Example

```
Map<Golfer, String> golfTeams = new TreeMap<Golfer,
String>(new HandicapComparator());
Golfer betty = new Golfer(12, 76, "betty");
Golfer jane = new Golfer(10, 88, "jane");
Golfer jim = new Golfer(15, 99, "jim");
```

We create a few Golfer instances.

SortedMap Example

```
Map<Golfer, String> golfTeams = new TreeMap<Golfer,
String>(new HandicapComparator());
Golfer betty = new Golfer(12, 76, "betty");
Golfer jane = new Golfer(10, 88, "jane");
Golfer jim = new Golfer(15, 99, "jim");
golfTeams.put(betty, "fairweather fairways");
golfTeams.put(jane, "the water hazards");
golfTeams.put(jim, "the water hazards");
```

Then we associate our Golfers with team names. Notice again that it's possible to have duplicates values, just not duplicate keys.

SortedMap Example

- Our map should now be sorted according to key
- We can get the entrySet and iterate through to see

```
for (Map.Entry<Golfer, String> e: golfTeams.entrySet())
      {
         System.out.println(e.getKey().getName()+"
"+e.getValue());
      }
```


SortedMap Example

- It seems like there's a lot happening in a few lines, so let's unpack it

```
for (Map.Entry<Golfer, String> e: golfTeams.entrySet())
      {
         System.out.println(e.getKey().getName()+"
"+e.getValue());
      }
```


SortedMap Example

EntrySet() returns a Set of type Entry.

Entry is defined within Map

```
for (Map.Entry<Golfer, String> e: golfTeams.entrySet())
      {
         System.out.println(e.getKey().getName()+"
"+e.getValue());
      }
```

SortedMap Example

The Entry has two type parameters, Golfer and String, representing the keys and the values.

```
for (Map.Entry<Golfer, String> e: golfTeams.entrySet())
      {
         System.out.println(e.getKey().getName()+"
"+e.getValue());
      }
```


SortedMap Example

```
for (Map.Entry<Golfer, String> e: golfTeams.entrySet())
      {
         System.out.println(e.getKey().getName()+"
"+e.getValue());
      }
```

An Entry object has a getKey() method and a getValue() method. In this case getKey() returns a Golfer object and getValue() returns a String.

SortedMap Example

```
for (Map.Entry<Golfer, String> e: golfTeams.entrySet())
      {
         System.out.println(e.getKey().getName()+"
"+e.getValue());
      }
```

We can then call the getName() method of the Golfer object.

SortedMap Example

```
for (Map.Entry<Golfer, String> e: golfTeams.entrySet())
      {
         System.out.println(e.getKey().getName()+"
"+e.getValue());
      }
```

So we get the name associated with each key (Golfer) and the value for each key (String), with the result sorted using the supplied comparator.

```
>
jane the water hazards
betty fairweather fairways
jim the water hazards
```

In-Class Exercise II

- Replace the for loop in that example with an iterator

```
for (Map.Entry<Golfer, String> e: golfTeams.entrySet())
      {
         System.out.println(e.getKey().getName()+"
"+e.getValue());
      }
```


Some other structures

Queues

- A ***queue*** is a collection of items organized in a structure with the ***First-In-First-Out (FIFO)*** property.
  - new items join the queue at the end,
  - the first item to enter the queue is the first to exit
  - like a line-up at a cashier
- Typical queue operations
  - add an item to a queue (***enqueue***)
  - remove and return the first item of a queue (***dequeue***)
  - return a queue's first item without removing it (***front***)
  - check if a queue is empty (***empty***)

Applications of Queues.

- Queues are frequently used in operating systems and networking software modules
  - processor queue
  - network router queues of outgoing packets, etc.
- Queues are very useful structures for computer based simulation
  - event queues, etc.

Interface Queue

```
public interface Queue<E> extends Collection<E>
{
   // enqueue
   public boolean offer(E element);

   // front
   public E peek();       // returns null if queue is empty
   public E element();    // throws a NoSuchElementException
                          // if queue is empty
   // dequeue
   public E poll();       // returns null if queue is empty
   public E remove();     // throws a NoSuchElementException
                             // if queue is empty

}
```


Stacks

- A ***stack*** is a collection of items organized in a structure with the ***Last-In-First-Out (LIFO)*** property:
  - new items are stacked on top of older items
  - the item on top of the stack is the first to be removed from it
  - like a stack of plates in a cafeteria
- Stack operations
  - **push** an item onto a stack
  - **pop** an item off the stack and return it
  - **peek** at the item on top of the stack (without popping it)
  - check if the stack is **empty**

`Stack` class (partial)

```
class Stack <E>  extends Vector<E>
{
   // Return true if the stack is empty.
   boolean empty();

   // Returns the item on top of the stack.
   E peek();

   // Pops item on top of stack and returns it.
   E pop();

   // Pushes a new item onto the stack, and returns it.
   E push(E item);
}
```

NOTE: A more consistent Stack interface is provided by Deque interface and its implementations

Applications of Stacks.

- **A computer uses a run-time stack to keep track of the function calls**
  - when a function is called, a new stack frame (with space for the parameters and local variables) is created and pushed onto the stack
  - when the function returns, the system pops the frame from the stack

Example Using Stacks

- **Problem**: Given an algebraic expression (with single letter operands) with brackets, check if brackets are balanced
  - `[a*(b+c)-{(d/a)-(d-a)}]` is balanced
  - `[a*(b+c)-{(d/a}-(d-a))]` is not balanced
  - `a*(b+c)-{(d/a)-(d-a)}]` is not balanced
  - `[a*(b+c)-{(d/a)-(d-a)]` is not balanced

Example (cont'd)

- **Solution Strategy**
  - use a stack
  - scan expression string from left to right once
  - when we find a left bracket, we push it onto the stack
  - when we find a right bracket:
    - if the stack is empty, report "not balanced"
    - otherwise:
      - pop an item off the stack
      - if the two brackets don't match, report "not balanced"
  - when you reach the end of the expression
    - if the stack is not empty, report "not balanced"
    - otherwise report "balanced"

Example (cont'd)

```
public static boolean match(String expression)
{
  final String LEFTS = "([{<";
  final String RIGHTS = ")]}>";

  char nxtChar;  // next character in expression
  char topChar;   // character on top of stack
  Stack<Character> brackets = new Stack<Character>();

  for (int index = 0; index < expression.length(); index++)
  {
    nxtChar = expression.charAt(index);
    if (LEFTS.indexOf(nxtChar) != -1)
    {
      // ch is left bracket
      brackets.push(nxtChar); // autoboxing
    }
```


Example (cont'd)

```
    else if(RIGHTS.indexOf(nxtChar) != -1)
    {
      // ch is a right bracket
      if (brackets.empty())
      {
        return false; // stack empty, so no match.
      }

      topChar = brackets.pop(); // auto-unboxing
      if (LEFTS.indexOf(topChar) != RIGHTS.indexOf(nxtChar))
      {
        return false;  // mismatched pair
      }
    }
  }
  return brackets.empty();
}
```

`Deque<E>` Interface

- A deque is a double ended queue:
  - can insert and remove items from both ends of a deque
  - Methods throw exception or return a special value

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

- Deque implementations:
  - `ArrayDeque, Linked List`

Generic Algorithms

- The `Collections` class (in `java.util`) has many useful static methods that operate on collections including the following:

```
• public static void reverse( List<?> list )
  // reverses the items in the list

• public static void shuffle( List<?> list )
  // randomly permutes the items in the list
```


Generic Algorithms

- Two methods to sort a list:

```
• public static <T> void sort( List<T> list,
          Comparator<? super T> c )
  // sorts the list using the given comparator

• public static <T extends Comparable<? super T>>
      void sort( List<T> list )
  // sorts the list according to the natural
  // ordering of its elements
```

- And many more…

Learning Goals Review

You will be expected to:

- program to the generic `Map` and `SortedMap` interfaces by reading and using the API
- compare and contrast `HashMap` and `TreeMap` classes (benefits of using each, basic run time analysis)
- program to the generic `Queue` interface
- program to the API of the generic `Stack` class
- program to the API of the generic `Deque` class
- identify (in words or through code) appropriate types for collections of data needed in a given software system
- write code (solve problems) that uses the generic algorithms provided in the `Collections` class

Sneak Preview

- In an upcoming lecture we will talk in detail about implementing associations
- But this is useful for your current assignment, so here's a preview

Unidirectional one-to-one associations

- The simplest type of association to implement is a unidirectional one-to-one association between two classes
- With a unidirectional association, you can navigate from an object of one class to an object of the other class (as indicated by the direction of the arrow) but not vice-versa
- This kind of association is easily implemented using an attribute that holds a reference to an object of the other class

Unidirectional one-to-one cont'd

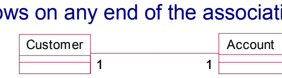
- Consider the following association:

- Implementation:

```
public class Watch
{
   private Display theDisplay;
   ...
}
```


Bidirectional one-to-one associations

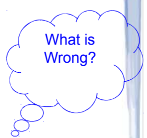
- The following UML diagram indicates a bidirectional association
  - no arrows on any end of the association



- We must be able to navigate from an account to the corresponding customer and vice versa
- Implementation:
  - each class needs an attribute that holds a reference to an object of the other class
  - each class must have setter methods that allow the reference to the object of the other class be established

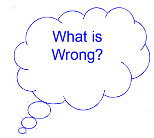
A Bad Implementation

```
public class Customer
{
  private Account theAccount;
  public void setAccount(Account account)
  {
    theAccount = account;
  }
  // etc.
}
public class Account
{
  private Customer theCustomer;
  public void setCustomer(Customer customer)
  {
    theCustomer = customer;
  }
  // etc.
}
```



How about this?

```
public class Customer
{
  private Account theAccount;
  public void setAccount(Account account)
  {
    theAccount = account;
    account.setCustomer(this)
  }
}
public class Account
{
  private Customer theCustomer;
  public void setCustomer(Customer customer)
  {
    theCustomer = customer;
    customer.setAccount(this)
  }
}
```



A Better Solution

```
public class Customer
{
  private Account theAccount;
  public void setAccount(Account account)
  {
    if (theAccount != account) {
      theAccount = account;
      account.setCustomer(this)
    }
  }
}
public class Account
{
  private Customer theCustomer;
  public void setCustomer(Customer customer)
  {
    if (theCustomer != customer) {
      theCustomer = customer;
      customer.setAccount(this)
     }
  }
}
```


One-to-many Associations

- One-to-many associations can also be bidirectional:

or unidirectional:

depending on the needs of the application. In either case the “many” part of the association is realized using a collection of references.

One-to-many Associations cont'd

```
public class Customer
{
    private Set<Video> rentedVideos;

    public void addVideo(Video video)
    {
      …
    }
   // etc.
}
```

- The particular type of collection that is used will depend on the needs of the application.
- If ordering matters, we may use an `Array` or `List`.
- If an element appears in the collection only once, we may choose to use a `Set`, etc.

One-to-many Associations cont'd

- Assuming a bidirectional association between customer and Video, the implementation of the Video class would look something like:

```
public class Video
{
    private Customer rentee;

    public void setRentee(Customer c)
    {
      …
    }
}
```


One-to-many associations cont'd

- Again, we have to be careful to ensure consistency. Would this be ok?

```
public class Customer {
    //…
    public void addVideo(Video video)
    {
       rentedVideos.add(video);
       video.setRentee(this);
    }
}

public class Video {
    //…
    public void setRentee(Customer c)
    {
       rentee = c;
      rentee.addVideo(this);
    }
}
```


One-to-many associations cont'd

- Here's a better implementation …

```
public class Customer {
    //…
    public void addVideo(Video video)
    {
       if (rentedVideos.add(video)){
          video.setRentee(this);
       }
    }
}

public class Video {
    //…
    public void setRentee(Customer c)
    {
        if (rentee != c)
        {
            rentee = c;
            rentee.addVideo(this);
        }
    }
}
```

Many-to Many Associations

- Consider the following many-to-many association between `SalesRep` and `Customer`:

- One way of implementing it is for both classes to maintain collections of references to instances of the other class.

- Again, operations need to be added that preserve consistency between the two collections of references.

Aggregations and Compositions

- The implementation of an aggregation does not differ from an association

- The implementation of a composition should ensure that when the whole is deleted, the parts are also deleted
  - In Java, we need to make sure that when the whole is deleted, there are no references to its parts, so the parts are garbage collected.