Chapter outline

- Lists
  - Collections
  - LinkedList vs. ArrayList
  - Iterators

- Sets
  - TreeSet vs. HashSet
  - Set operations

- Maps
  - Map operations
  - Map views
  - TreeMap vs. HashMap

Collections

- collection: an object that stores data inside it
  - the objects stored are called elements
  - some collections maintain an ordering, some don't
  - some collections allow duplicates, some don't
  - an array is like a very crude "collection"

- typical operations:
  - add element, remove element, clear all elements, contains or find element, get size
  - most collections are built with particular kinds of data, or particular operations on that data, in mind

- examples of collections:
  - list, bag, stack, queue, set, map, graph
**Java’s Collection interface**

- **The interface** `Collection<E>` in `java.util` represents many kinds of collections.
- Every collection has the following methods:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(Value)</code></td>
<td>adds the given value to the collection</td>
</tr>
<tr>
<td><code>addAll(collection)</code></td>
<td>adds all elements from given collection to this one</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all elements</td>
</tr>
<tr>
<td><code>contains(value)</code></td>
<td>returns <code>true</code> if the element is in the collection</td>
</tr>
<tr>
<td><code>containsAll(collection)</code></td>
<td>true if this collection contains all elements from the other</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>true if the collection does not contain any elements</td>
</tr>
<tr>
<td><code>removeAll(collection)</code></td>
<td>removes all values contained in the given collection from this collection</td>
</tr>
<tr>
<td><code>retainAll(collection)</code></td>
<td>removes all values not contained in the given collection from this collection</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in the list</td>
</tr>
<tr>
<td><code>toArray()</code></td>
<td>returns an array containing the elements of this collection</td>
</tr>
</tbody>
</table>

**Collection interface, cont’d.**

- `public boolean isEmpty()`
  Returns true if this list contains no elements.

- `public Iterator<E> iterator()`
  Returns a special object for examining the elements of the list in order (seen later).

- `public boolean remove(Object o)`
  Removes the first occurrence in this list of the specified element.

- `public int size()`
  Returns the number of elements in this list.

- `public Object[] toArray()`
  Returns an array containing all of the elements from this list.

**An example collection: List**

- **list**: an ordered sequence of elements, each accessible by a 0-based index
  - one of the most basic collections
List features

- Maintains elements in the order they were added (new elements are added to the end by default)
- Duplicates are allowed
- Operations:
  - add element to end of list
  - insert element at given index
  - clear all elements
  - search for element
  - get element at given index
  - remove element at given index
  - get size
    - some of these operations are inefficient (seen later)

- The list manages its own size; the user of the list does not need to worry about overfilling it.

Java's List interface

- Java also has an interface List<E> to represent a list of objects.
  - It adds the following methods to those in Collection<E>:
    - public void add(int index, E element)
      Inserts the specified element at the specified position in this list.
    - public E get(int index)
      Returns the element at the specified position in this list.
    - public int indexOf(Object o)
      Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain it.

List interface, cont'd.

- public int lastIndexOf(Object o)
  Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain it.
- public E remove(int index)
  Removes the object at the specified position in this list.
- public Object set(int index, E element)
  Replaces the element at the specified position in this list with the specified element.

- Notice that the methods added to Collection<E> by List<E> all deal with indices
  - a list has indices while a general collection may not

Array list limitations

- An add or remove operation on an ArrayList that is not at the end of the list will require elements to be shifted.
  - This can be slow for a large list.
  - What is the worst possible case?
The underlying issue

- the elements of an ArrayList are too tightly attached; can't easily rearrange them
- can we break the element storage apart into a more dynamic and flexible structure?

![Arraylist diagram]

Linked list

- linked list: a list implemented using a linked sequence of values
  - each value is stored in a small object called a node, which also contains references to its neighbor nodes
  - the list keeps a reference to the first and/or last node
  - in Java, represented by the class LinkedList

![LinkedList diagram]

LinkedList usage example

- A LinkedList can be used much like an ArrayList:

```java
LinkedList<String> words = new LinkedList<String>();
words.add("hello");
words.add("goodbye");
words.add("this");
words.add("that");
```

Adding elements to the list

1. Make a new node to hold the new element.
   ![New node diagram]
2. Connect the new node to the other nodes in the list.
   ![Node connection diagram]
3. Change the front of the list to point to the new node.
   ![Front change diagram]
Linked list performance

- To add, remove, get a value at a given index:
  - The list must advance through the list to the node just before the one with the proper index.
  - Example: To add a new value to the list, the list creates a new node, walks along its existing node links to the proper index, and attaches it to the nodes that should precede and follow it.
  - This is very fast when adding to the front or back of the list (because the list contains references to these places), but slow elsewhere.

![](image1.png)

A particularly slow idiom

```java
List<String> list = new LinkedList<String>();
// ... (put a lot of data into the list)

// print every element of linked list
for (int i = 0; i < list.size(); i++) {
    Object element = list.get(i);
    System.out.println(i + ": " + element);
}
```

- This code executes a slow operation (`get`) every pass through a loop that runs many times
  - this code will take prohibitively long to run for large data sizes

The problem of position

- The code on the previous slide is wasteful because it throws away the position each time.
  - Every call to `get` has to re-traverse the list.

- It would be much better if we could somehow keep the list in place at each index as we looped through it.

- Java uses special objects to represent a position of a collection as it's being examined
  - These objects are called `iterators`.  

![](image2.png)
**Iterators in Java**

- **interface** `Iterator<E>`
  - `public boolean hasNext()`  
    Returns `true` if there are more elements to see
  - `public E next()`  
    Returns the next object in this collection, then advances the iterator; throws an exception if no more elements remain
  - `public void remove()`  
    Deletes the element that was last returned by `next` (not always supported)

---

**Iterators on linked lists**

- An iterator on a linked list maintains (at least) its current index and a reference to that node
- When `iterator()` is called on a linked list, the iterator initially refers to the first node (index 0)

---

**Linked list iterator iteration**

- When `next()` is called, the iterator:
  - grabs the current node's element value ("a")
  - follows the `next` reference on its node and increments its index
  - returns the element it grabbed ("a")
- `hasNext` is determined by whether the iterator has reached the back of the list

---

**Iterator's remove**

- The `remove` removes the last value that was returned by a call to `next`
  - in other words, it deletes the element just before the iterator's current node
Fixing the slow LL idiom

```java
// print every element of the list
for (int i = 0; i < list.size(); i++) {
    Object element = list.get(i);
    System.out.println(i + " : " + element);
}

Iterator<Integer> itr = list.iterator();
for (int i = 0; itr.hasNext(); i++) {
    Object element = itr.next();
    System.out.println(i + " : " + element);
}
```

Iterator usage idiom

- The standard idiom of using an iterator:
  ```java
  Iterator<E> itr = <collection>.iterator();
  while (itr.hasNext()) {
      <do something with itr.next() >;
  }
  ```

- The following code efficiently removes all Strings with an even number of characters from a linked list:
  ```java
  // removes all strings of even length from the list
  public static void removeEvenLength(LinkedList<String> list) {
      Iterator<String> i = list.iterator();
      while (i.hasNext()) {
          String element = i.next();
          if (element.length() % 2 == 0) {
              i.remove();
          }
      }
  }
  ```

Benefits of iterators

- speed up loops over linked lists' elements
- a unified way to examine all elements of a collection
  - every collection in Java has an iterator method
  - in fact, that's the only guaranteed way to examine the elements of any Collection
  - don't need to look up different collections' method names to see how to examine their elements
- don't have to use indexes as much on lists

Iterator is still not perfect

```java
// print odd-valued elements, with their indexes
Iterator<Integer> itr = list.iterator();
for (int i = 0; itr.hasNext(); i++) {
    int element = itr.next();
    if (element % 2 == 1) {
        System.out.println(i + " : " + element);
    }
}
```

- We still had to maintain the index variable i so that we could print the index of each element.
- We can't use the iterator to add or set elements.
  - The iterator is programmed to crash if the list is modified externally while the iterator is examining it.
More iterator problems

```java
// add a 0 after any odd element
Iterator<Integer> itr = list.iterator();
int i = 0;
while (itr.hasNext()) {
    int element = itr.next();
    if (element % 2 == 1) {
        list.add(i, new Integer(0));    // fails
    }
}
```

- the iterator speeds up `get` and `remove` loops only
- the iterator really should be able to help us speed up `add` elements or `set` elements' values!

Concurrent modification

```java
public void doubleList(LinkedList<Integer> list) {
    Iterator<Integer> i = list.iterator();
    while (i.hasNext()) {
        int next = i.next();
        list.add(next);      // ConcurrentModificationException
    }
}
```

While you are still iterating, you cannot call any methods on the list that modify the list's contents.
- The above code crashes with a `ConcurrentModificationException`
- It is okay to call a method on the `iterator itself` that modifies the list (`remove`)

List abstract data type (ADT)

- **abstract data type (ADT)**: a general specification for a type of data structure
  - specifies what data the data structure can hold
  - specifies what operations can be performed on that data
  - does NOT specify exactly how the data structure holds the data internally, nor how it implements each operation
- Example ADT: **List**
  - list ADT specifies that a list collection will store elements in order with integer indices (allowing duplicates and null values)
  - list ADT specifies that a list collection supports `add`, `remove`, `get(index)`, `set(index)`, `size`, `isEmpty`, ...
  - **ArrayList** and **LinkedList** both implement the data/operations specified by the list ADT
- **ADTs in Java are specified by interfaces**
  - **ArrayList** and **LinkedList** both implement the **List** interface

ADT usage example

```java
// returns the longest string in the given list
// pre: list.size() > 0
public static String longest(List<String> list) {
    Iterator<String> i = list.iterator();
    String result = i.next();
    while (i.hasNext()) {
        String next = i.next();
        if (next.length() > result.length()) {
            result = next;
        }
    }
    return result;
}
```
Collections class

- The following static methods in the Collections class operate on either type of list.
  - Example:
    ```java
    Collections.replaceAll(list, "hello", "goodbye");
    ```

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(list, value)</td>
<td>searches a sorted list for a value and returns its index</td>
</tr>
<tr>
<td>copy(dest, source)</td>
<td>copies all elements from one list to another</td>
</tr>
<tr>
<td>fill(list, value)</td>
<td>replaces all values in the list with the given value</td>
</tr>
<tr>
<td>max(list)</td>
<td>returns largest value in the list</td>
</tr>
<tr>
<td>min(list)</td>
<td>returns smallest value in the list</td>
</tr>
<tr>
<td>replaceAll(list, oldValue, newValue)</td>
<td>replaces all occurrences of oldValue with newValue</td>
</tr>
<tr>
<td>reverse(list)</td>
<td>reverses the order of elements in the list</td>
</tr>
<tr>
<td>rotate(list, distance)</td>
<td>shifts every element's index by the given distance</td>
</tr>
<tr>
<td>sort(list)</td>
<td>places the list's elements into natural sorted order</td>
</tr>
<tr>
<td>swap(list, index1, index2)</td>
<td>switches element values at the given two indexes</td>
</tr>
</tbody>
</table>

Application: words in a book

- Write an application that reads in the text of a book (say, Moby Dick) and then lets the user type words, and tells whether those words are contained in Moby Dick or not.
  - How would we implement this with a List?
  - Would this be a good or bad implementation?

- Notice that the code to solve this problem doesn't use much of the list functionality (only add and search)
  - Does the ordering of the elements in the List affect the algorithm? Could we use this information to our advantage?

A new ADT: Set

- **set**: an unordered collection with no duplicates
  - The main purpose of a set is to search it, to test objects for membership in the set (contains).

  - Java has an interface named `Set<E>` to represent this kind of collection.
    - Set is an interface; you can't say `new Set()`

  - There are two `Set` implementations in Java:
    - TreeSet and HashSet
    - Java's set implementations have been optimized so that it is very fast to search for elements in them.
Java Set interface

- Interface `Set` has exactly the methods of the `Collection` interface.
- `TreeSet` and `HashSet` classes implement the `Set` interface.

Notice: These list methods are missing from `Set`:
- `get(index)`
- `add(index, value)`
- `remove(index)`

To access each element of a set, use its `iterator` method instead.

TreeSet vs. HashSet

The preceding code can use `TreeSet` instead:

```java
Set<String> stooges = new TreeSet<String>();
...
System.out.println(stooges);
```

Output:

```
[Curly, Larry, Moe, Shemp]
```

TreeSet vs. HashSet:
- A TreeSet stores its elements in the natural alphabetical order.
- TreeSet can only be used with elements with an ordering (it can't easily store `Point` objects, for example).
- TreeSet is slightly (often not noticeably) slower than HashSet.

Set operations

Sets support common operations to combine them with, or compare them against, other sets:
Typical set operations

- Sometimes it is useful to compare sets:
  - **subset**: $S_1$ is a subset of $S_2$ if $S_2$ contains every element from $S_1$.
  - `containsAll` tests for a subset relationship.

- It can be useful to combine sets in the following ways:
  - **union**: $S_1 \cup S_2$ contains all elements that are in $S_1$ or $S_2$.
  - `addAll` performs set union.
  - **intersection**: $S_1 \cap S_2$ contains only the elements that are in both $S_1$ and $S_2$.
  - `retainAll` performs set intersection.
  - **difference**: $S_1 \setminus S_2$ contains the elements that are in $S_1$ that are not in $S_2$.
  - `removeAll` performs set difference.

A variation: book word count

- Previously, we discussed an application that reads in the text of a book (say, *Moby Dick*) and then lets the user type words, and tells whether those words are contained in *Moby Dick* or not.

- What if we wanted to change this program to not only tell us whether the word exists in the book, but also how many times it occurs?

Mapping between sets

- Sometimes we want to create a mapping between elements of one set and another set
  - Example: map words to their count in the book
    - "the" --&gt; 325
    - "whale" --&gt; 14
  - Example: map people to their phone numbers
    - "Marty Stepp" --&gt; "692-4540"
    - "Jenny" --&gt; "867-5309"

- How would we do this with a list (or list(s))?  
  - A list doesn't map people to phone numbers; it maps ints from 0..size - 1 to objects
  - Could we map some int to a person's name, and the same int to the person's phone number?
  - How would we find a phone number, given the person's name? Is this a good solution?
A new ADT: Map

- **map**: an unordered collection that associates a collection of element values with a set of keys so that elements can be found very quickly
  - Each key can appear at most once (no duplicate keys)
  - A key maps to at most one value
  - the main operations:
    - `put(key, value)`
      - "Map this key to that value."
    - `get(key)`
      - "What value, if any, does this key map to?"
  - Maps are represented in Java by the `Map<K, V>` interface.
    - Two implementations: `HashMap` and `TreeMap`

Map methods

Maps don't implement the `Collection` interface, but they do have the following public methods:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clear()</code></td>
<td>removes all keys and values from the map</td>
</tr>
<tr>
<td><code>containsKey(key)</code></td>
<td>returns true if the given key exists in the map</td>
</tr>
<tr>
<td><code>containsValue(value)</code></td>
<td>returns true if the given value exists in the map</td>
</tr>
<tr>
<td><code>get(key)</code></td>
<td>returns the value associated with the given key (null if not found)</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns true if the map has no keys or values</td>
</tr>
<tr>
<td><code>keySet()</code></td>
<td>returns a collection of all keys in the map</td>
</tr>
<tr>
<td><code>put(key, value)</code></td>
<td>associates the given key with the given value</td>
</tr>
<tr>
<td><code>putAll(map)</code></td>
<td>adds all key/value mappings from given map</td>
</tr>
<tr>
<td><code>remove(key)</code></td>
<td>removes the given key and its associated value</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of key/value pairs in the map</td>
</tr>
<tr>
<td><code>values()</code></td>
<td>returns a collection of all values in the map</td>
</tr>
</tbody>
</table>

Basic Map usage

Maps are declared with two type parameters, one for the keys and one for the values:

```java
Map<String, Double> salaryMap = new HashMap<String, Double>();
salaryMap.put("Stuart", 20000.00);
salaryMap.put("Marty", 15500.00);
salaryMap.put("Jenny", 86753.09);
System.out.println(salaryMap);
// search the map for a name
if (salaryMap.containsKey("Jenny")) {
    double salary = salaryMap.get("Jenny");
    System.out.println("Jenny's salary is $" + salary);
} else {
    System.out.println("I don't have a record for Jenny");
}
```

Output:

```
{Jenny=86753.09, Stuart=20000.0, Marty=15500.0}
Jenny's salary is $86753.09
```
TreeMap vs. HashMap

- Remember that Map is an interface.
  - You can't say `Map m = new Map();`

- Java has two classes that implement the Map interface:
  - TreeMap
    - elements are stored in their natural Comparable order
    - slightly slower
    - can only be used on elements with an ordering
  - HashMap
    - elements are stored in an unpredictable order
    - faster to add, search, remove

Collection views

- A map itself is not regarded as a collection.
  - Map does not implement Collection interface
    - although, in theory, it could be seen as a collection of pairs

- Instead collection views of a map may be obtained:
  - a Set of its keys
  - a Collection of its values (not a set... why?)

Iterators and Maps

- Map has no iterator method; you can't get an Iterator directly

- You must first call either:
  - `keySet()` returns a Set of all the keys in this Map
  - `values()` returns a Collection of all the values in this Map

- Then call `iterator()` on the key set or value collection.

- Examples:
  - `Iterator<String> keyItr = grades.keySet().iterator();`
  - `Iterator<String> elementItr = grades.values().iterator();`

- You can also use the enhanced for loop over these collections:

```java
for (int ssn : ssnMap.values()) {
    System.out.println("Social security #: " + ssn);
}
```

Map example

```java
import java.util.*;
public class Birthday {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();
        m.put("Newton", 1642);
        m.put("Darwin", 1809);
        System.out.println(m);
        Set<String> keys = m.keySet();
        Iterator<String> itr = keys.iterator();
        while (itr.hasNext()) {
            String key = itr.next();
            System.out.println(key + " => " + m.get(key));
        }
    }
}
```

Output:

```
{Darwin=1809, Newton=1642}
Darwin => 1809
Newton => 1642
```
Map practice problems

- Write code to invert a Map; that is, to make the values the keys and make the keys the values.

```java
Map<String, String> byName = new HashMap<String, String>();
byName.put("Darwin", "748-2797");
byName.put("Newton", "748-9901");

Map<String, String> byPhone = new HashMap<String, String>();
// ... your code here!
System.out.println(byPhone);
```

Output:

```
{748-2797=Darwin, 748-9901=Newton}
```

- Write a program to count words in a text file, using a hash map to store the number of occurrences of each word.