CS11 – Java

Winter 2010-2011

Lecture 8

Java Collections

- Very powerful set of classes for managing collections of objects
  - Introduced in Java 1.2
- Provides:
  - Interfaces specifying different kinds of collections
  - Implementations with different characteristics
  - Iterators for traversing a collection's contents
  - Some common algorithms for collections
- Very useful, but nowhere near the power and flexibility of C++ STL

Why Provide Collection Classes?

- Reduces programming effort
  - Most programs need collections of some sort
  - Makes language more appealing for development
- Standardized interfaces and features
  - Reduces learning requirements
  - Facilitates interoperability between separate APIs
- Facilitates fast and correct programs
  - Java API provides high-performance, efficient, correct implementations for programmers to use

Collection Interfaces

- Generic collection interfaces defined in **`java.util`**
  - Defines basic functionality for each kind of collection
- **`Collection`** – generic "bag of objects"
- **`List`** – linear sequence of items, accessed by index
- **`Queue`** – linear sequence of items "for processing"
  - Can add an item to the queue
  - Can "get the next item" from the queue
  - What is "next" depends on queue implementation
- **`Set`** – a collection with no duplicate elements
- **`Map`** – associates values with unique keys

More Collection Interfaces

- A few more collection interfaces:
  - **`SortedSet`** (extends **`Set`**)
  - **`SortedMap`** (extends **`Map`**)
  - These guarantee iteration over elements in a particular order
- Requires elements to be comparable
  - Must be able to say an element is “less than” or “greater than” another element
  - Provide a total ordering of elements used with the collection

Common Collection Operations

- Collections typically provide these operations:
  - **`add(Object o)`** – add an object to the collection
  - **`remove(Object o)`** – remove the object
  - **`clear()`** – remove all objects from collection
  - **`size()`** – returns a count of objects in collection
  - **`isEmpty()`** – returns true if collection is empty
  - **`iterator()`** – traverse contents of collection
- Some operations are optional
  - Throws **`UnsupportedOperationException`** if not supported by a specific implementation
- Some operations are slower/faster

Collection Implementations

- Multiple implementations of each interface
  - All provide same basic functionality
  - Different storage requirements
  - Different performance characteristics
  - Sometimes other enhancements too
    - e.g. additional operations not part of the interface
- Java API Documentation gives the details!
  - See interface API Docs for list of implementers
  - Read API Docs of implementations for performance and storage details

`List` Implementations

- **`LinkedList`** – doubly-linked list
  - Each node has reference to previous and next nodes
  - O(N)-time access of i^{th} element
  - Constant-time append/prepend/insert
  - Nodes use extra space (previous/next references, etc.)
  - Best for when list grows/shrinks frequently over time
  - Has extra functions for get/remove first/last elements
- **`ArrayList`** – stores elements in an array
  - Constant-time access of i^{th} element
  - Append is usually constant-time
  - O(N)-time prepend/insert
  - Best for when list doesn't change much over time
  - Has extra functions for turning into a simple array

`Set` Implementations

- **`HashSet`**
  - Elements are grouped into “buckets” based on a hash code
  - Constant-time add/remove operations
  - Constant-time “contains” test
  - Elements are stored in no particular order
  - Elements must provide a hash function
- **`TreeSet`**
  - Elements are kept in sorted order
    - Stored internally in a balanced tree
  - O(log(N))-time add/remove operations
  - O(log(N))-time “contains” test
  - Elements must be comparable

`Map` Implementations

- Very similar to `Set` implementations
  - These are *associative containers*
  - Keys are used to access values stored in maps
  - Each key appears only once
    - (No multiset/multimap support in Java collections)
- `HashMap`
  - Keys are hashed
  - Fast lookups, but random ordering
- `TreeMap`
  - Keys are sorted
  - Slower lookups, but kept in sorted order

Collections and Java 1.5 Generics

- Up to Java 1.4, collections only stored **`Objects`**

```
LinkedList points = new LinkedList();
points.add(new Point(3, 5));
Point p = (Point) points.get(0);
```

  - Casting everything gets annoying
  - Could add non-**`Point`** objects to **`points`** collection too!
- Java 1.5 introduces generics

```
LinkedList<Point> points = new LinkedList<Point>();
points.add(new Point(3, 5));
Point p = points.get(0);
```

  - No more need for casting
  - Can only add **`Point`** objects to **`points`** too
  - Syntactic sugar, but quite useful!

Using Collections

- Lists and sets are easy:

```
HashSet<String> wordList = new HashSet<String>();
LinkedList<Point> waypoints = new LinkedList<Point>();
```

  - Element type must appear in both variable declaration and in **`new`**-expression

- Maps are more verbose:

```
TreeMap<String, WordDefinition> dictionary =
  new TreeMap<String, WordDefinition>();
```

  - First type is key type, second is the value type

- See Java API Docs for available operations

Iteration Over Collections

- Often want to iterate over values in collection
- **`ArrayList`** collections are easy:

```
ArrayList<String> quotes;
...
for (int i = 0; i < quotes.size(); i++)
  System.out.println(quotes.get(i));
```

  - Impossible/undesirable for other collections!
- Iterators are used to traverse contents
- **`Iterator`** is another simple interface:
  - **`hasNext()`** – Returns **`true`** if can call **`next()`**
  - **`next()`** – Returns next element in the collection
- **`ListIterator`** extends **`Iterator`**
  - Provides many additional features over **`Iterator`**

Using Iterators

- Collections provide an **`iterator()`** method
  - Returns an iterator for traversing the collection
- Example:

```
HashSet<Player> players;
...
Iterator<Player> iter = players.iterator();
while (iter.hasNext()) {
  Player p = iter.next();
  ... // Do something with p
}
```

  - Iterators also use generics
  - Can use iterator to delete current element, etc.

Java 1.5 Enhanced For-Loop Syntax

- Setting up and using an iterator is annoying
- Java 1.5 introduces syntactic sugar for this:

```
for (Player p : players) {
  ... // Do something with p
}
```

  - Can't access the actual iterator used in the loop
  - Best for simple scans over a collection's contents
- Can also use enhanced for-loop syntax with arrays:

```
float sum(float[] values) {
  float result = 0.0f;
  for (float val : values)
    result += val;
  return result;
}
```

Collection Algorithms

- **`java.util.Collections`** class provides *some* common algorithms
  - …not to be confused with the **`Collection`** interface
  - Algorithms are provided as static functions
  - Implementations are fast, efficient, and generic
- Example: sorting

```
LinkedList<Product> groceries;
...
Collections.sort(groceries);
```

  - Collection is sorted in-place: **`groceries`** is changed
- Read Java API Docs for more details
  - Also see **`Arrays`** class for array algorithms

Collection Elements

- Collection elements may require certain capabilities
- **`List`** elements don't need anything special
  - …unless **`contains()`**, **`remove()`**, etc. are used!
  - Then, elements should provide a correct **`equals()`** implementation
- Requirements for **`equals()`**:
  - **`a.equals(a)`** returns true
  - **`a.equals(b)`** same as **`b.equals(a)`**
  - If **`a.equals(b)`** is true and **`b.equals(c)`** is true, then **`a.equals(c)`** is also true
  - **`a.equals(null)`** returns false

Set Elements, Map Keys

- Sets and maps require special features
  - Sets require these operations on set-elements
  - Maps require these operations on the keys
- **`equals()`** must definitely work correctly
- **`TreeSet`**, **`TreeMap`** require sorting capability
  - Element or key class must implement **`java.lang.Comparable`** interface
  - Or, an appropriate implementation of **`java.util.Comparator`** must be provided
- **`HashSet`**, **`HashMap`** require hashing capability
  - Element or key class must provide a good implementation of **`Object.hashCode()`**

`Object.hashCode()`

- **`java.lang.Object`** has a **`hashCode()`** method

  ```
  public int hashCode()
  ```

  - Compute a hash code based on object's values
  - **`hashCode()`** is used by **`HashSet`**, **`HashMap`**, etc.
- Rule 1:
  - If **`a.equals(b)`** then their hash codes must be the same!
  - OK for two non-equal objects to have the same hash code
    - "Same hash-codes" just means "they *might be* equal"
- Rule 2:
  - If you override **`equals()`** on a class then you should also override **`hashCode()`**!
  - (See Rule 1)

Implementing `hashCode()`

- Is this a correct implementation?

```
public int hashCode() {
  return 42;
}
```

  - It satisfies the rules, so *technically* yes…
  - In practice, will cause programs to be very inefficient
- Hash func should generate a wide range of values
  - Specifically, should produce a uniform distribution of values
  - Facilitates most efficient operation of hash tables
  - Requirement is that equal objects must produce identical hash values…
  - Also good if unequal objects produce different hash values

Implementing `hashCode()` (2)

- If a field is included in `equals()` comparison, should also include it in the hash code
- Combine individual values into a hash code:

```
int hashCode() {
  int result = 17;     // Some prime value

  // Use another prime value to combine
  result = 37 * result + field1.hashCode();
  result = 37 * result + field2.hashCode();
  ...
  return result;
}
```

More Hash-Code Hints

- A few more basic hints:
  - If field is a boolean, use 0 or 1 for hash code
  - If field is an integer type, cast value to **`int`**
  - If field is a non-array object type:
    - Call the object's hashCode() function, or use 0 for **`null`**
  - If field is an array:
    - Include every array-element into final hash value!
  - See Effective Java, Item 8 for more guidelines!
- If computing the hash is expensive, cache it.
  - Must recompute hash value if object changes!

Changing Elements and Keys

- Java sets/maps assume that their elements/keys don't change
  - e.g. a key's hash code shouldn't change while it's in the collection
- Don't change a map-key after adding it to a map
  - Remove the key/value mapping, change the key, then re-add the key/value mapping
- Don't change a set element after adding it to a set
  - Remove the element from the set, change the element, then re-add the element to the set

Comparing and Ordering Objects

- Objects implement **`java.lang.Comparable<T>`** interface to allow them to be ordered

  ```
  public int compareTo(T obj)
  ```

- Returns a value that imposes an order:
  - result < 0 means **`this`** is less than **`obj`**
  - result == 0 means **`this`** is "same as" **`obj`**
  - result > 0 means **`this`** is greater than **`obj`**
- This defines the *natural ordering* of a class
  - i.e. the "usual" or "most reasonable" sort-order
- Natural ordering should be *consistent with* **`equals()`**
  - **`a.compareTo(b)`** returns 0 only when **`a.equals(b)`** is true
- Implement this interface correctly for using **`TreeSet`** / **`TreeMap`**

Alternate Orderings

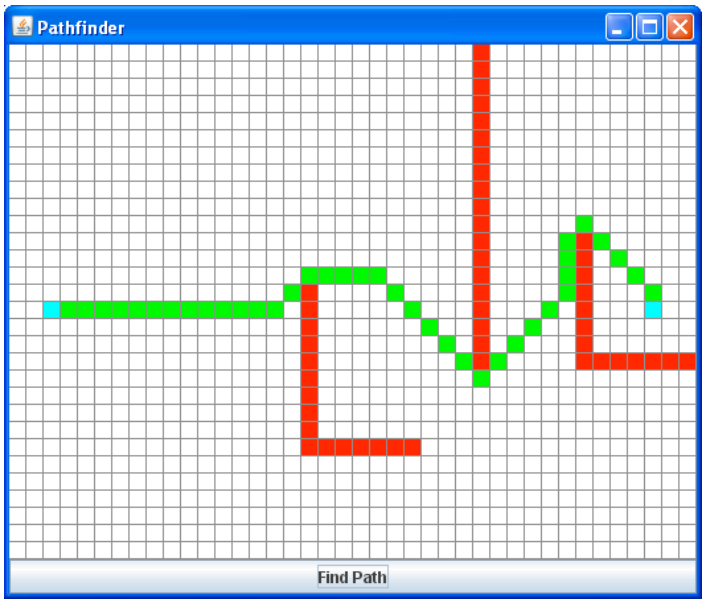
- Can provide extra comparison functions
  - Provide a separate object that implements **`java.util.Comparator<T>`** interface
  - Simple interface:

```
int compare(T o1, T o2)
```

- Sorted collections, sort algorithms can also take a comparator object
  - Allows sorting by all kinds of things!
- Comparator impls are typically nested classes
  - e.g. **`Player`** class could provide a **`ScoreComparator`** nested class

Lab 8 – A* Path-Finding Algorithm

- A* path-finding algorithm is used extensively for navigating maps with obstacles
  - Finds an optimal path from start to finish, if a path exists
- Example:



A* Implementation

- A* algorithm requires two collections
  - A collection of “open waypoints” to be considered
  - Another collection of “closed waypoints” that have already been examined
- Your tasks:
  - Provide **`equals()`** and **`hashCode()`** impls. for **`Location`** class
  - Complete the **`AStarState`** class, which manages open and closed waypoints for A* algorithm
  - Play with the fun A* user interface ☺