Sets and Maps

Ye Yang
Stevens Institute of Technology

Sets

- A set is defined as a collection of items that
  - Are unordered
  - Are unique (no duplicates)
- What kind of operations do we think of?
  - element membership (add, remove, isMember, etc.)
  - manipulation (union, intersection,…)
- Java has:
  - An interface: Set
  - Several implementation classes: TreeSet, HashSet

Java’s Set Interface

- The Set interface looks just like the Collection interface, but with more specific behaviors
  - boolean add(elem) – returns false if already there
  - boolean remove(elem) – returns false if not there
- What’s sweet here:
  - Try to add something that’s already there?
  - Remove something that’s not there?
  - No problem! It basically ignores that attempt!
- This allows some powerful methods to be very short

Examples using Set

- How many unique items in any Collection object?

```
int numUniq = new HashSet(aCollection).size();
```

  – Creates a new set “on the fly” using the constructor that adds all items from an existing collection
- Remove duplicates from any Collection object?

```
Set noDups = new HashSet( aCollection );
aCollection.clear();
aCollection.addAll( noDups );
```

  – Note we don’t have to iterate
  – We rely on useful constructors and bulk operations
- Procedural Abstraction opportunity:
  – Put these lines of code into a method with a good names

Classes Implementing Set Interface

- public class TreeSet
  - Implements Set interface using TreeMap
  - Sorted set will be ascending order according to the natural order
- public class HashSet
  - Implements Set interface using HashTable
  - Offers constant time performance

Using Sets in Java: TreeSet class

- TreeSet is a concrete class that implements Set
- How's it work inside? How is it implemented?
  - Through a balanced binary search tree
- By default:
  - All items must be Comparable
  - All items must correctly implement compareTo()
  - Iterating over a TreeSet uses compareTo()'s ordering. So prints in "correct" order
- Performance:
  - O(logN): contains, add, remove, findMin, findMax, findKth

Using Sets in Java: HashSet class

- HashSet is a concrete class that implements Set
  - Cant be used to enumerate items in sorted order
  - Cant be used to obtian the smallest or largest item
  - Items don’t have to be comparable in any way
- How’s it work inside? How is it implemented?
  - Through hashing

Hash Values for Java Classes

- Hashing to get a hash-value is very useful, so…
- Java's superclass Object defines a method hashCode() for every class
  - If you want to use HashSet or any other class that uses hashing, override this method
  - **Important:** when you override equals(), you should override hashCode() too!
    - Otherwise things break when you don't expect it.
    - Make sure that things that are equals() as you define that will also hash to the same value
  - Overriding hashCode() is often very very easy

Overriding hashCode() in Java

- Goal: make sure that two items that are equals() return the same value when hashCode() is called

- Focus on what fields determine identity of objects
  - Example #1: Class Person
    - Say two objects are equals() if their SSN fields are equal
  - Example #2: Class Song
    - Say two objects are equals() if their title and artist fields are both equal

- Example solutions:
  - If your object's "key" is a String:
    - Most Java library classes already implement hashCode().
    - If just one String value: Your hashCode() method should just return the result of calling hashCode() on that String object
  - If multiple fields, here's a simple solution:
    - Get the hash-value for each individual field and just add them
    - (Example #2) for class Song where title and artist must be equals(), the method just does:
      - **return artist.hashCode() + title.hashCode();**

Summary on Set, HashSet, TreeSet

- HashSet is an implementation of Set you can use
  - Don’t forget to implement hashCode() for items you want to store!
  - You **MUST** over-ride hashCode() and equals() together! If you do one, always do the other!
- Items in Sets are not ordered
  - But you can construct a List from a Set and sort it
- add() handles duplicates, and remove() ignores missing items
- TreeSet is ordered

Maps

- Maps
  - Like sets, but have <key, value> instead of <value>
  - Keys must be unique, but several keys can map to the same value.
    - Dictionary example
    - Other examples
      - Student's email to Facebook Profile
      - Host Names to IP addresses
- How to declare: Map<KeyType, ValueType>…
  Examples:

```
Map<String, Integer> // maps string to a number
Map<Student, String> // maps Student to a String
Map<String, List<Course>> // e.g. a schedule
```

Important Map Methods

- Keys and values
  - put (key, value), get(key), remove(key)
  - containsKey(key), containsValue(value)
- Can think of a Map as a set of keys and a set of values
  - keySet() // returns a Set of keys in the map
  - values() // returns a Collection of values
  - (Note: Map itself does not extend Collection, but it uses Sets and is part of the Java Collection Framework)

Concrete Classes

- HashMap
  - Most commonly used. Allows nulls as values.
  - Need a hashCode() method
- TreeMap
  - Uses a search tree data structure
  - Values stored in a sorted order
  - Keys must have a compareTo() method
    - Or can create with a Comparator object
- HashTable
  - Old class -- deprecated! Don’t use!

Why Are Maps Useful and Important?

- "Lookup values for a key" -- an important task
  - We often want to do this!
  - Need flexibility for what values are stored for a given key
  - Need to use objects other than Strings or ints as key values
- Efficiency matters for large sets of data
  - HashMap has O(1) lookup cost
  - TreeMap is O(lg n) which is still very good

Summary

- Set and Map are important ADTs
  - Java supports these directly
- Set interface
- Concrete classes: HashSet, TreeSet
  - Note:
    - How add() and remove() fail "nicely" (I.e. return false)
    - How used to achieve solve more complex tasks with Collections
  - HashSet relies on proper implementation of hashCode()
    - hashCode() and equals() must be "consistent"
  - TreeSet relies on compareTo()

Summary (2)

- Map Interface
  - Concrete classes: HashMap and TreeMap
  - Declared using both key-type and value-type, e.g.
    Map<Student, List<Course>>
  - Important because “lookup” is a frequent problem
  - Lookup for maps is very efficient for large amounts of data
  - Code often needs to take different actions for:
    - The first time a key/value pair is inserted into a map
    - We want to update information for an existing key

Note: Map not IS-A Collection

- It's wrong to say a Map is a Collection
  - It does not implement the Collection interface
    - add(E) doesn't make sense for a map
    - We need to add a (key, value) pair
- But Map and its concrete classes are part of the JCF
  - They use Sets as part of their definition and API

BACK UP SLIDES

Big Picture on Java Collections Unit

- Java Collections Framework
  - A large collections of interfaces and classes
  - Includes things we can use “as is” (concrete classes) and building blocks for new things (abstract classes)
  - Classes that exist to contain static methods, e.g. Collections class (see also Arrays class)
- Power of inheritance, interfaces, polymorphism
  - See how “type” Collection and List are used in parameter lists and in Collections class’ methods (e.g. sort)
- Be aware of and start to use Java 5.0 generics with your collections

Big Picture on Java Collections Unit (2)

- Iterators: both concepts and the practice
  - Example of procedural abstraction (use, methods) and data abstraction (what's really in a iterator object?)
  - Using these in Java for Collections, Lists (methods, cursor, etc.)
- Procedural abstraction: the concepts, examples
- Function-objects and procedural abstraction
  - Encapsulating "functionality" or "control" in an object
  - Comparator objects (contrast to implementing Comparable interface)
  - Java trick: anonymous classes

Big Picture on Java Collections Unit (3)

- Java skills: Concrete collection classes and methods for you to use
  - ArrayList and ListIterator
  - HashSet
    - Must over-ride hashCode()
- Collection<u>s</u> static methods
  - See JavaDoc
    - E.g. sort, reverse, binarySearch, fill, shuffle, min, max,…
  - Implement Comparable for your collection items
  - Define Comparator classes for more powerful, flexible use