Objectives

- Collections
- Traversing Collections

1

Review

1. (from Monday) How do we indicate that a class is part of a particular package?
2. What are the 3 components of the Java Collection Framework?
3. What data types can collections hold?
4. How can we convert a primitive type into its respective Wrapper Class type?
5. What are 3 of the main interfaces for collections that Java provides?
   a) What are the properties of those collections?
   b) What are operations you can perform on those collections?
6. What is the syntax to say what type the collection holds?
7. Why can Eclipse provide the functionality it does?
8. Why did I wait until now to show you Eclipse?

2

Eclipse

- Very helpful – *after* you know what you're doing
  - You know
    - Code is compiled before executed
    - Structure of classes
    - How to fix errors
- Eclipse can handle the "routine" for you
  - That wasn't "routine" for you a few weeks ago
- Gives suggestions for fixes
  - You need to think through what the appropriate fix is
  - Don't say "Eclipse made me do <something>"
- Eclipse is a beast (memory hog)
  - If you have less than ~8GB of memory, Eclipse will be slow

Eclipse Hints

- After you have written a method, type

  /**

before the method, and then hit enter and the Javadocs comment template will be automatically generated for you

- Use command-spacebar for possible completions

Eclipse Discussion

- Helpful hints
  - Control-spacebar
  - Format the file
  - Auto-templates for Javadoc comments

5

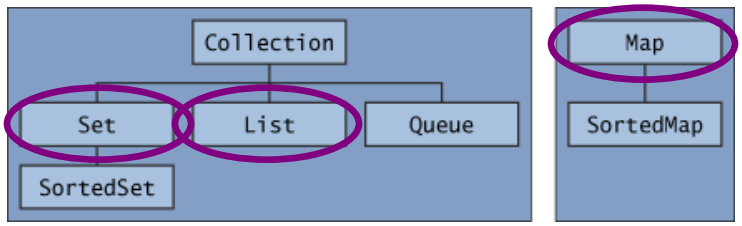
Review: Collections Framework

- **Interfaces**
  - Abstract data types that represent collections
  - Collections can be manipulated *independently* of implementation
- **Implementations**
  - Concrete implementations of collection interfaces
  - Reusable data structures
- **Algorithms**
  - Methods that perform useful computations on collections, e.g., searching and sorting
  - Reusable functionality
  - ***Polymorphic***: same method can be used on many different implementations of collection interface

6

Review: Core Collection Interfaces

- Encapsulate different types of collections



Similarly, abstract class inheriting from an abstract class:

```
public abstract class AbstractList<E>
extends AbstractCollection<E> implements
List<E>
```


Comparing: Before & After Generics

- Before Generics

```
List myList = new LinkedList();
myList.add(new Card(4, "clubs"));
…
Card x = (Card) myList.get(0);
```

- After Generics

```
List<Card> myList = new LinkedList<>();
myList.add(new Card(4, "clubs"));
…
Card x = myList.get(0);
```

✓ Improved readability and robustness

LISTS

9

List

- An *ordered* collection of elements
- Can contain duplicate elements
- Has control over where objects are stored in the list

10

List Interface

- boolean add(<E> o)
  - Boolean so that List can refuse some elements
    - e.g., refuse adding null elements
- <E> get(int index)
  - Returns element at the position index
  - Different from Python: no shorthand
    - Can't write ~~list[pos]~~
- int size()
  - Returns the number of elements in the list
- And more!
  - contains, remove, toArray, …

Common List Implementations

- ArrayList
  - Resizable array
  - Used most frequently
  - Fast
- LinkedList
  - Use if adding elements to ends of list
  - Use if often delete from middle of list
  - Implements Deque and other methods so that it can be used as a stack or queue

How would you find the other implementations of List?

Implementation vs. Interface

Implementation choice only affects performance

- Preferred Style:
  1. Choose an implementation
  2. Assign collection to variable of corresponding **interface** type

```
Interface variable = new Implementation();
```

- Methods should accept interfaces—not implementations

Why is this the preferred style?

Implementation vs. Interface

Implementation choice only affects performance

- Preferred Style:
  1. Choose an implementation
  2. Assign collection to variable of corresponding **interface** type
- Why?
  - Program does not depend on a given implementation's methods
    - Access only using interface's methods
  - Programmer can change implementations
    - Performance concerns or behavioral details

Discussion of Deck Class

`cards.Deck.java`

15

SETS

16

Set Interface

- No duplicate elements
  - Needs to determine if two elements are "logically" the same (`equals` method)
- Models mathematical set abstraction

Set Interface

- `boolean add(<E> o)`
  - Add to set, only if not already present
- `int size()`
  - Returns the number of elements in the list
- And more! (`contains, remove, toArray, …`)
  - Note: no `get` method -- `get` #3 from the set?

Some `Set` Implementations

- `HashSet` 
  - Implements set using *hash table*
    - `add`, `remove`, and `contains` each execute in O(1) time
  - Used more frequently
  - Faster than `TreeSet`
  - No ordering
- `TreeSet`
  - Implements set using a *tree*
    - `add`, `remove`, and `contains` each execute in O(log n) time
  - Sorts

FindDuplicates Problem

- From the array of command-line arguments, identify the duplicates

```
public static void main(String args[]) {




}
```

FindDuplicates: One solution

```
public static void main(String args[]) {
    Set<String> s = new HashSet<>();
    for (String a : args) {
      if (!s.add(a)) {
           System.out.println(
                "Duplicate detected: " + a);
      }
    }
    System.out.println(s.size() +
       " distinct words detected: " + s);
}
```

How much does code changes if s is a TreeSet?

MAPS

Maps

- Maps keys (of type <K>) to values (of type <V>)
- No duplicate keys
  - Each key maps to at most one value

Declaring Maps

- Declare types for both keys and values
- `class HashMap<K,V>`

```
Map<String, Integer> map = new HashMap<>();
```

Keys are Strings
Values are Integers

```
Map<String, List<String>> map
                          = new HashMap<>();
```

Keys are Strings
Values are Lists of Strings

Map Interface

- `<V> put(<K> key, <V> value)`
  - Returns old value that key mapped to
- `<V> get(Object key)`
  - Returns value at that key (or null if no mapping)
- `Set<K> keySet()`
  - Returns the set of keys

And more …

A few Map Implementations

- `HashMap`
  - Fast
- `TreeMap`
  - Sorting
  - Key-ordered iteration
- `LinkedHashMap`
  - Fast
  - Insertion-order iteration

ALGORITHMS

27

Collections Framework's Algorithms

- *Polymorphic algorithms*
- Reusable functionality
- Implemented in the `Collections` class
  - Similar to `Arrays` class, which operates on arrays
  - Static methods, 1st argument is the Collection

28

Overview of Available Algorithms

- Sorting – optional `Comparator`
- Shuffling
- Searching – binarySearch

* Only `Lists`

- Routine data manipulation: reverse*, copy*, fill*, swap*, addAll
- Composition – frequency, disjoint
- Finding min, max

TRAVERSING COLLECTIONS

Traversing Collections: For-each Loop

- For-each loop:

Or whatever data type is appropriate

```
for (Object o : collection)
    System.out.println(o);
```

- Valid for all `Collections`
  - `Maps` (and its implementations) are not `Collections`
    - But, `Map`'s `keySet()` is a `Set` and `values()` is a `Collection`

Traversing Lists: Iterator

- Always between two elements


```
Iterator<Integer> i = list.iterator();
while( i.hasNext()) {
     int value = i.next();
     …
}
```

Iterator API

- <E> next()
  - Get the next element
- boolean hasNext()
  - Are there more elements?
- void remove()
  - Remove the previous element
  - **Only safe way** to remove elements during iteration
    - Not known what will happen if remove elements in for-each loop

Polymorphic Filter Algorithm

```
static void filter(Collection c) {
    Iterator i = c.iterator();
    while( i.hasNext() ) {
        // if the next element does not
        // adhere to the condition, remove it
        if ( ! condition(i.next()) ) {
            i.remove();
        }
    }
}
```

Polymorphic: works regardless of Collection implementation

Traversing Lists: `ListIterator`

- Methods to traverse list backwards too
  - `hasPrevious()`
  - `previous()`
- To get a `ListIterator`:
  - `listIterator(int position)`
    - Pass in `size()` as position to get at end of list

Key difference

Sept 18, 2020 Spren `ListIteratorExamples.java`

35

How Not to Iterate

- Don't use `get` to access `List`
  - If implementation is a `LinkedList`, performance is reeeeeally slow

```
for (int i = 0; i < list.size(); i++) {
   count += list.get(i); // do something
}
```

What to do instead?

Sept 18, 2020 Sprenkle - CSCI209 36

36

Enumeration

- Legacy class
- Similar to `Iterator`
- Example methods:
  - `boolean hasMoreElements()`
  - `Object nextElement()`
- Longer method names
- Doesn't have remove operation

Synchronized Collection Classes

- For multiple threads sharing same collection
- Slow down typical programs
  - Avoid for now
- e.g., `Vector, Hashtable`
- See `java.util.concurrent`

Another example: `StringBuffer` is synchronized, whereas `StringBuilder` is not

Benefits of Collections Framework

- ?

39

Benefits of Collections Framework

- **Provides common, well-known interface**
  - Allows interoperability among unrelated APIs
  - Reduces effort to learn and to use new APIs for different implementations
- **Reduces programming effort:** provides useful, reusable data structures and algorithms
- **Increases program speed and quality:** provides high-performance, high-quality implementations of data structures and algorithms; interchangeable implementations → tuning
- **Reduces effort to design new APIs:** use standard collection interface for your collection
- **Fosters software reuse:** New data structures/algorithms that conform to the standard collection interfaces are reusable

40

Looking Ahead

- Assignment 7 – due Wednesday
- Exam 1 – next Friday
  - Online, timed exam
    - No class next Friday
    - Start time, due time TBD
  - Open book/notes/slides – but **do not** rely on that
    - NOT open internet
  - Prep document online
  - 3 sections:
    - Very Short Answer, Short Answer, Coding