Objectives

- Collections Framework
- Wrapper classes
- Autoboxing, autounboxing

Review

- What are jar files? How are they used?
- What is the classpath?
- What is the syntax for Generics? How are they used?
- What is the keyword for specifying that your class adheres to an interface?
- Compare and contrast abstract classes and interfaces
  - When should a class be abstract?
  - When should you create/use an interface?
- True or False (with explanation):
  - If you extend an abstract class, you must override all abstract methods.
  - You can instantiate an abstract class.
  - An abstract class can have a constructor.
  - You can have an object variable of an abstract class.
  - You can have an object variable of an interface.
- What is the Collection framework made up of?
- 112 review: what are *lists*, *sets*, and *dictionaries*?

Review: Interfaces vs Abstract Classes

Interfaces

- Only specification (no implementation)
- Any class can implement
  - Because classes can implement multiple interfaces
- Implementing methods multiple times
- Adding a method to interface will break classes that implement that interface

Abstract Classes

- Contain partial implementation
- Child classes can't extend/subclass multiple classes
- Add non-abstract methods without breaking subclasses

Case Study

USING AN IMPLEMENTATION VS. INTERFACE

Implementation vs. Interface: Case Study

- The Java Collections Framework has both interfaces and implementations
- When should you use an implementation vs an interface?

Implementation vs. Interface: Case Study

Implementation choice only affects performance

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

```
Interface variable = new Implementation();
Example: List<Card> hand = new ArrayList<>();
```

Why is this the preferred style?

- Rather than

```
Implementation variable = new Implementation();
Example: ArrayList<Card> hand = new ArrayList<>();
```

Implementation vs. Interface

Implementation choice only affects performance

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

```
Interface variable = new Implementation();
Example: List<Card> hand = new ArrayList<>();
```

- 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

Design Principle: Program to an Interface

(Not to an implementation)

- Implementation choice only affects performance
- In practice:
  - Use interface variables (as seen with Collections)
  - Methods should accept interfaces—not implementations

```
public void method( Interface var ) {…}
Example: public void method( List var ) {…}
```

- Makes code more resilient to change
  - Can change implementation and not affect implementation of the rest of code because … you programmed to the interface

WRAPPER CLASSES

Types Allowed with Generics

- Can only contain `Objects`, not primitive types

- Autoboxing and Autounboxing to the rescue!

Wrapper Classes

- Sometimes need an instance of an Object
  - Ex: to store in Lists and other Collections
- Each primitive type has a **Wrapper class**
  - Examples: `Integer`, `Double`, `Long`, `Character`, …
- Include functionality of parsing their respective data types

```
int x = 10;
Integer y = Integer.valueOf(x);
Integer z = Integer.valueOf("10");
```

Wrapper Classes

- ***Autoboxing*** – automatically create a wrapper object

```
Integer y = 11; // implicitly 11 converted to Integer,
                // e.g., Integer.valueOf(11)
```

- ***Autounboxing*** – automatically extract a primitive type

```
Integer x = Integer.valueOf(11);
int y = x.intValue();
int z = x; // implicitly, x is x.intValue();
```

Converts right side to whatever is needed on the left

Effective Java: Unnecessary Autoboxing

```
Long sum = 0L;
for (long i=0; i < Integer.MAX_VALUE; i++) {
        sum += i;
}
System.out.println(sum);
```

- What is the inefficiency from object creation?
- How can you fix the inefficiency?

Effective Java: Unnecessary Autoboxing

```
Long sum = 0L;
for (long i=0; i < Integer.MAX_VALUE; i++) {
        sum += i;
}
System.out.println(sum);
```

Constructs 2^{31} `Long` instances

How can you fix the inefficiency?

`Integer.MAX_VALUE` is an `int`

`Autobox.java`

`AutoboxFixed.java`

Effective Java: Unnecessary Autoboxing

```
Long sum = 0L;
for (long i=0; i < Integer.MAX_VALUE; i++) {
        sum += i;
}
System.out.println(sum);
```

Constructs 2^{31} Long instances

Lessons:
- Prefer primitives to boxed primitives
- Watch for unintentional autoboxing

Autobox.java
AutoboxFixed.java

BACK TO COLLECTIONS FRAMEWORK

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`

ALGORITHMS

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`

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

Discussion of Deck Class

Deck.java

SETS

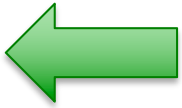
`Set` Interface

- No duplicate elements
  - Needs to determine if two elements are "logically" the same (i.e., uses `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 – can't get #3 from the set because sets aren't ordered.

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

MAPS

Maps

- Python called these *dictionaries*
- 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

MapExample.java

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();
        …
}
```

Helpful to use if removing elements from list during iteration

Benefits of Collections Framework

- ?

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

FindDuplicates Problem

1. Design algorithm
2. Implement

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

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




}
```

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 – can't get #3 from the set because sets aren't ordered.

FindDuplicates

```
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 change if `s` is a `TreeSet`?

Looking Ahead

- Assignment 3 Due Before Class Wednesday