Objectives

- Collections Framework
  - Maps
  - Algorithms
  - Traversing
- Enumerated Types

Review

- What is the Java Collection Framework made up of?
- What interfaces/data structures did we discuss?
- Why do we use Interface objects instead of Implementations in our programs?
- How do we declare/initialize a new Collection object?

Review: Collections Framework

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

MAPS

Map Interface

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

Map Implementations

- `HashMap`
  - Fast
- `TreeMap`
  - Sorting
  - Key-ordered iteration
- `LinkedHashMap`
  - Fast
  - Insertion-order iteration
  - Remove stale mappings → custom caching

Declaring Maps

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

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

Keys are Strings
Values are Lists of Strings

Rethinking PetSurvey.java

- How did we keep track of a pet's votes in PetSurvey.java?
- Any limitations? Inefficiencies?
  - Could we do better? Be more efficient?

Implement: castVote, getAnimals

PetSurvey3.java

ALGORITHMS

Collections Framework's Algorithms

- *Polymorphic algorithms*
- Reusable functionality
- Implemented in the `Collections` class
  - Static methods, 1st argument is the collection

Overview of Available Algorithms

- Sorting – optional `Comparator`
- Shuffling
- Routine data manipulation: reverse, copy, fill, swap, addAll
- Searching – binarySearch
- Composition – frequency, disjoint
- Finding min, max

Update `Deck` class

TRAVERSING COLLECTIONS

Traversing Collections

- For-each loop:

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

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

Traversing Collections: `Iterator`

- Java Interface
- To get an `Iterator` from a `Collection` object:

```
Iterator<E> iterator()
```

  - Returns an `Iterator` over the elements in this collection
  - Example:

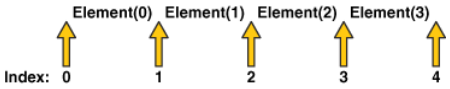
```
Iterator<String> iter = keys.iterator();
```


`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

Iterator: Like a Cursor

- Always between two elements

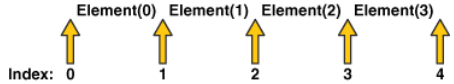


Polymorphic Filter Algorithm

```
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 (!cond(i.next())) {
         i.remove();
      }
   }
}
```


Traversing Lists: `ListIterator`

- Methods to traverse list backwards too
  - `hasPrevious()`
  - `previous()`
- To get a `ListIterator`:
  - `listIterator(int position)`
    - Pass in `size()` as index to get at end of list
- Used for insertion/modification/deletion in linked lists in the middle



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`

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

ENUMERATED TYPES

Enumerated Types

Type whose legal values consist of a fixed set of constants

- Also called *enums*
  - More powerful than enums in C
  - Added Java 1.5

Old Way: *Int Enum Pattern*

```
public static final int APPLE_FUJI             = 0;
public static final int APPLE_PIPPIN           = 1;
public static final int APPLE_GRANNY_SMITH     = 2;

public static final int ORANGE_NAVEL           = 0;
public static final int ORANGE_TEMPLE          = 1;
public static final int ORANGE_BLOOD           = 2;
```

- Drawbacks
  - No type safety (ORANGE vs APPLE?)
  - Compile-time constants
    - Change associated int, other code needs to be recompiled
  - Weak debug information
  - Can't iterate over them reliably; size of group?
- Similar: *String enum pattern*

Enum

```
public enum Apple  {FUJI, PIPPIN, GRANNY_SMITH};
public enum Orange {NAVEL, TEMPLE, BLOOD};
```

Use:

```
Apple lunch = Apple.FUJI;
```

Each is a public static final instance

- Full-fledged class
  - Can add arbitrary methods and fields
  - Implementations of `Object` methods, `Comparable` interface, …
  - Effectively final

Enumerated Types

- Are like **inner** classes in Java
  - *Entirely nested within another class*
- Implicitly inherits from `java.lang.Enum`
  - `boolean equals(Object other)`
  - `int compareTo(E o)`
  - `String name()`
    - Returns the name of this enum constant, exactly as declared in its enum declaration
  - `int ordinal()`
    - Returns the ordinal of this enumeration constant, i.e., its position in its enum declaration, where the initial constant is assigned an ordinal of zero

More Sophisticated Enum: `Planet`

`PlanetTest.java`

Enums

- Has static `values()` method
  - Returns *array* of values in order declared
  - E.g., `FIJI, PIPPIN, GRANNY_SMITH`
- Can be used in `switch` statements

```
switch(lunch) {
     case Apple.FUJI:
          price = 1.43;
     …
}
```


Designing the Playing Card Class

- State?
  - How to represent?
- API?

Implement:

```
boolean sameSuit(Card c)
int getRummyValue()
```

For Next Week

- Assignment 8: Due Wednesday
  - Practice with Collections, User interface