Objectives

- Wrap up Inheritance
- Interfaces
- Collections
- Generics

1

Iteration over Code: Assignment 5

- Demonstrates typical design/implementation process
  - Start with original code design
    - Inheritance from GamePiece class
  - Realize it could be designed better
    - Make GamePiece class abstract
    - Use an array of GamePiece objects
    - Easier to add new functionality to Game
- Major part of problem-solving is figuring out how to break problem into smaller pieces
- Reminders
  - Heed my warnings
  - Start simple, small

2

Review

- How does Java decide which method to execute for an object?
  - Example: `chicken[1].feed();`
- Compare and contrast abstract classes and interfaces
  - When should a class be abstract?
  - When should you create/use an interface?

Mostly 112 review

- True or False:
  - If you extend an abstract class, you have to override all abstract methods.
  - You can instantiate an abstract class
  - You can have an object variable of an abstract class
  - You can have an object variable of an interface
- 112 review: what are *lists*, *sets*, and *dictionaries*?

Interfaces vs Abstract Classes

Interfaces

- No implementation
- ✓ *Any* class can use
  - ✓ Can implement multiple interfaces
- – Implementing methods multiple times
- – Adding a method to interface will break classes that implement

Abstract Classes

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

PREVENTING INHERITANCE

Preventing Inheritance

- Sometimes, you do not want a class to derive from one of your classes
- A class that cannot be extended is known as a `final` class
- To make a class final, add the keyword `final` in the class definition:

```
public final class Rooster extends Chicken {
        . . .
}
```

- Example of `final` class: `System`

Final methods

- Can make a method `final`
  - Any class derived from this class cannot override the `final` methods

```
class Chicken {
        . . .
        public final String getName() { . . . }
        . . .
}
```

- By default, **all** methods in a `final` class are `final` methods.

Why would we want to use `final`?
What are possible benefits to us, the compiler, …?

7

INTERFACES

8

Example Interface: java.lang.Comparable

```
public interface Comparable {
        int compareTo(Object other);
}
```

- Any object that implements Comparable must have a method named compareTo()
- Returns:
  - Return a negative integer if this object is less than the object passed as a parameter
  - Return a positive integer if this object is greater than the object passed as a parameter
  - Return a 0 if the two objects are equal

Implementing an Interface

- In the class definition, specify that the class will implement the specific interface

```
public class Chicken implements Comparable
```

- Provide a definition for all methods specified in interface

How to determine Chicken order?

Comparable Chickens

One way: order by height

```
public class Chicken implements Comparable {
  . . .
  public int compareTo(Object otherObject) {
        Chicken other = (Chicken) otherObject;
        if (height < other.getHeight() )
              return -1;
        if (height > other.getHeight())
              return 1;
        return 0;
        // simpler: return height - other.getHeight();
  }
}
```

What if `otherObject` is not a `Chicken`?

Testing for Interfaces

- Can also use the `instanceof` operator to see if an object implements an interface
  - e.g., to determine if an object can be compared to another object using the `Comparable` interface

```
if (obj instanceof Comparable) {
    // runs if obj is an object variable of a class
    // that implements the Comparable interface
}
else {
    // runs if it does not implement the interface
}
```

Interface Object Variables

- Can use an object variable to refer to an object of any class that implements an interface
- Using this object variable, can only access the interface's methods
- For example…

```
public void aMethod(Object obj) {
    …
    if (obj instanceof Comparable) {
         Comparable comp = (Comparable) obj;
         boolean res = comp.compareTo(obj2);
    }
}
```


Interface Definitions

```
public interface Comparable {
    int compareTo(Object other);
}
```

- Interface methods are `public` by default
  - Do not *need* to specify methods as `public`

Interface Definitions and Inheritance

- Can extend interfaces
  - Allows a chain of interfaces that go from general to more specific
- For example, define an interface for an object that is capable of moving:

```
public interface Movable {
      void move(double x, double y);
}
```


Interface Definitions and Inheritance

- A powered vehicle is also `Movable`
  - Must also have a `milesPerGallon()` method, which will return its gas mileage

```
public interface Powered extends Movable {
      double milesPerGallon();
}
```

Interface Definitions and Inheritance

- Powered interface extends Movable interface
- An object that implements Powered interface must satisfy all requirements of that interface as well as the parent interface.
  - A Powered object must have a `milesPerGallon()` and `move(double x, double y)` method

Constants in an Interface

- If a variable is specified in an interface, it is automatically a constant:
  - `public static final variable`

```
public interface Powered extends Movable {
        double SPEED_LIMIT = 95;
        double milesPerGallon();
}
```

- Example: An object that implements Powered interface has a constant SPEED_LIMIT defined

Multiple Interfaces

- A class can implement *multiple* interfaces
  - Must fulfill the requirements of each interface

```
public final class String implements
      Serializable, Comparable, CharSequence { …
```

- Recall: NOT possible with inheritance
  - A class can only extend (or inherit from) **one** class

Benefits of Interfaces

- Abstraction
  - Separate the interface from the implementation
- Allow easier type substitution
  - We'll see this with Collections
- Classes can implement multiple interfaces

Interface Summary

- Contain only object (*not class*) methods
- All methods are `public`
  - Implied if not explicit
- Fields are constants that are `static` and `final`
- A class can implement multiple interfaces
  - Separated by commas in definition

Interfaces vs Abstract Classes

Interfaces

- No implementation
- ✓ *Any* class can use
  - ✓ Can implement multiple interfaces
- – Implementing methods multiple times
- – Adding a method to interface will break classes that implement

Abstract Classes

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

One Option: Use Both!

- Define interface, e.g., `MyInterface`
- Define abstract class, e.g., `AbstractMyInterface`
  - Implements interface
  - Provides implementation for some methods

Summary: Abstract Classes and Interfaces

- Important structures
  - Make code easier to change
- Will return to/apply these ideas throughout the course
- Concepts are used in many languages besides Java
  - Java provides tools to enforce these concepts

COLLECTIONS

25

Collections

- Sometimes called *containers*
- Group multiple elements into a single unit
- Store, retrieve, manipulate, and communicate aggregate data
- Represent data items that form a natural group
  - Poker hand (a collection of cards)
  - Mail folder (a collection of messages)
  - Telephone directory (a mapping of names to phone numbers)

26

Java Collections Framework

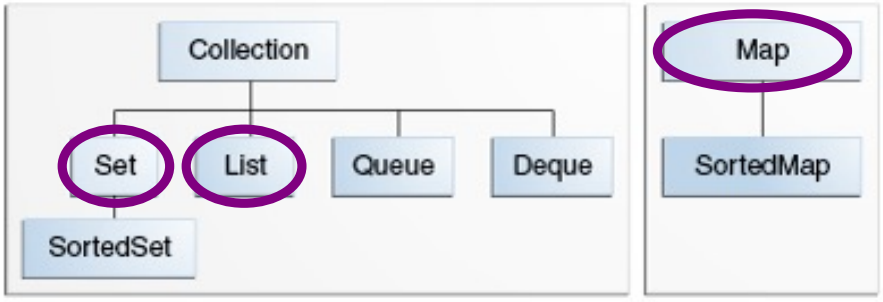
- *Unified architecture* for representing and manipulating collections
- More than arrays
  - More flexible, functionality, dynamic sizing
- In `java.util` package

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

Core Collection Interfaces

- Encapsulate different types of collections



GENERICS

Generic Collection Interfaces

- Declaration of the `Collection` interface:

```
public interface Collection<E> …
```

Type parameter

  - `<E>` means interface is generic for **e**lement class
- When declare a `Collection`, **specify type** of object it contains
  - Allows compiler to verify that object's *type* is correct
    - Reduces errors at runtime
- Example, a hand of cards:

Always declare type contained in collections

```
List<Card> hand = new ArrayList<Card>();
```

Added in Java 7:

```
List<Card> hand = new ArrayList<>();
```


Comparing: Before & After Generics

- Before Generics

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

- List of `Objects`
- Need to cast to the desired child class

Comparing: Before & After Generics

- Before Generics

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

- List of Objects
- Need to cast to the desired child class

- After Generics

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

- If you try to add not-a-Card, compiler gives an error

✓ Improved readability and robustness

Comparable Interface

- Also uses Generics

```
public interface Comparable<T>
```

The type it compares

```
int compareTo(T o)
```

Generics Use Comparison for Comparable

Without Generics

```
public int compareTo(Object otherObject) {
        if( ! (otherObject instanceof Chicken) ) {
            return 1;
        }
        Chicken other = (Chicken) otherObject;
        if (height < other.getHeight() )
            return -1;
        if (height > other.getHeight())
            return 1;
        return 0;
}
```

With Generics

```
public int compareTo(Chicken other) {
        if (height < other.getHeight() )
            return -1;
        if (height > other.getHeight())
            return 1;
        return 0;
}
```


Types Allowed with Generics

- Can only contain `Objects`, not primitive types
- Autoboxing and Autounboxing to the rescue!

WRAPPER CLASSES

Wrapper Classes

- Sometimes need an instance of an Object
  - 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 type

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

Wrapper Classes

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

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

- Can you find 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?

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

LISTS

`List` Interface

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

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

When should you use one vs the other?

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

- Methods should accept interfaces—not implementations

Why is this the preferred style?

```
public void method( Interface var ) {…}
```

47

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

Looking Ahead

- Assignment 5 – due Tuesday at 11:59 p.m.
- Exam 1 on Friday!
  - Bring your questions on Wednesday