Objectives

- Overloading
- Inheritance
  - Final methods, fields
- Abstract Classes
- Interfaces

Review

- How do we make a class inherit from a parent class?
- How does a class refer to its parent class?
- What does a class inherit from its parent class?
  - What is not inherited?
- What are the access modifiers, ordered from least restrictive to most restrictive?
- How can we verify that an object variable is a certain type?
- How can we specify that an object variable has a different type (a derived type)?
- How does Java decide which method to call on an object?
  - Example: chicken[1].feed();

Inheritance Rules: Access Modifiers

Access modifiers in child classes

- Can make access to child class less restrictive but not more restrictive
- Why?

  - What would happen if a method in the parent class is public but the child class’s method is private?

Summary of Inheritance

- Remove repetitive code by modeling the “is-a” hierarchy
  - Move “common denominator” code up the inheritance chain
- Don’t use inheritance unless all inherited methods make sense
- Use polymorphism
Overload: Static Dispatch

```java
public class OverloadPlay {
    public static void print(Parent parent) {
        System.out.println("Static Parent");
    }
    public static void print(Child child) {
        System.out.println("Static Child");
    }
    public void instancePrint(Parent parent) {
        System.out.println("Parent");
        parent.method();
    }
    public void instancePrint(Child child) {
        System.out.println("Child");
        child.method();
    }
    public static void main(String[] args) {
        Child child = new Child();
        Parent p = child;
        print(p);
        new OverloadPlay().instancePrint(p);
    }
}
```

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, simply add the keyword final in front of the class definition:

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

- Example of final class: System

Why final methods and classes?

- **Efficiency**
  - Compiler can replace a final method call with an inline method
  - Does not have to worry about another form of this method that belongs to a derived class
  - JVM does not need to determine which method to call dynamically
- **Safety**
  - No alternate form of the method; straightforward which version of the method you called
Abstract Classes

- Some methods defined, others not defined
  - Partial implementation

- Classes in which not all methods are implemented are abstract classes
  - public abstract class ZooAnimal

- Blank methods are labeled as abstract
  - public abstract void exercise(Environment env);

Abstract Classes

- An abstract class cannot be instantiated
  - i.e., can’t create an object of that class
  - But can have a constructor!

- Child class of an abstract class can only be instantiated if it overrides and implements every abstract method of parent class
  - If child class does not override all abstract methods, it is also abstract

Examples of abstract classes

- Example 1:
  - java.net.Socket
  - java.net.SSLSocket (abstract)

- Example 2:
  - java.util.Calendar (abstract)
  - java.util.GregorianCalendar

Summary: Defining Abstract Classes

- Define a class as abstract when have partial implementation
**Better Organization of Game Classes**

- GamePiece should be abstract

  *Why?*

---

**Interfaces**

- Like abstract classes with *all* abstract methods
  - A set of requirements for classes to conform to
- Pure specification, no implementation
- Classes can implement one or more interfaces

---

**Example of an Interface**

- We can call Arrays.sort(array)
  - Arrays.sort() sorts arrays of any object class that implements the Comparable interface
- Classes that implement Comparable must provide a way to decide if one object is less than, greater than, or equal to another object

  *Sound similar to anything in Python?*

---

**java.lang.Comparable**

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

- Any object that is Comparable must have a method named compareTo()
- Returns:
  - `< 0` for less than
  - `0` for equals
  - `> 0` for greater than
- Similar to Python’s `__cmp__` method

---

**Comparable Interface API/Javadoc**

- Specifies what the compareTo() method should do:
  - Return a –1 if the first object is less than the second object (passed as a parameter)
  - Return a 1 if the second object (passed as a parameter) is less than the first object
  - Return a 0 if the two objects are equal
- Says which Java library classes implement Comparable
Implementing an Interface

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

```java
public class Chicken implements Comparable {
    // How to determine Chicken order?
}
```

Comparable Chickens

One way: order by height

```java
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?

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

Testing for Interfaces

- 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

```java
if (obj instanceof Comparable) {
    Comparable comp = (Comparable) obj;
    boolean res = comp.compareTo(obj2);
} 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...

```java
Object obj;
if (obj instanceof Comparable) {
    Comparable comp = (Comparable) obj;
    boolean res = comp.compareTo(obj2);
}
```

Interface Definitions

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

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

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

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

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

Constants in an Interface

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

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

- 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

  ```java
  public final class String implements Serializable, Comparable, CharSequence {
  ...
  }
  ```

- But 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

- Can implement multiple interfaces
Using an Interface or Abstract Class

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Abstract Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Any class can use</td>
<td>✓ Contain partial implementation</td>
</tr>
<tr>
<td>✓ Can implement multiple interfaces</td>
<td>✓ Can’t extend/subclass multiple classes</td>
</tr>
<tr>
<td>✓ No implementation</td>
<td>✓ Add non-abstract methods without breaking subclasses</td>
</tr>
</tbody>
</table>

One Option: Use Both!

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

Abstract Classes and Interfaces

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

TODO

- Assignment 5: due Wednesday