CS 211: Final Classes, Abstract, Interfaces

Chris Kauffman

Week 6-2
Logistics

Goals Today

▶ equals(Object o) method
▶ Dynamic Dispatch
▶ Abstract Classes

P3: Cipherous Symmetry

▶ Due before spring break (next Friday)
▶ Questions?

Reading: Inheritance

▶ Building Java Programs Ch 9
▶ Lab Manual Ch 7

Exam 1 Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 2/23</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Wed 2/25</td>
<td>Dispatch/Abstract</td>
</tr>
<tr>
<td>Thu 2/26</td>
<td>Lab Quiz</td>
</tr>
<tr>
<td>Mon 2/29</td>
<td>Review</td>
</tr>
<tr>
<td>Wed 3/2</td>
<td>Exam 1</td>
</tr>
<tr>
<td>Thu 3/3</td>
<td>Lab Exercises</td>
</tr>
<tr>
<td>Fri 3/4</td>
<td>Project 3 Due</td>
</tr>
<tr>
<td>Mon 3/7</td>
<td>Spring Break</td>
</tr>
</tbody>
</table>
Access Modifiers Again

Access Levels for Fields/Methods by other stuff

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>no modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- private fields are not accessible by child classes
- protected fields are accessible by child classes
- Useful when children need to do stuff with a parent's field

Official docs on access modifiers

http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
Quick Note on Shadowing

class Parent {
    protected int field;
    Parent(int f){ field = f; } 
    public void reportField(){
        System.out.println(field);
    }
}

class ProperChild extends Parent{
    ProperChild(int f){ super(f); } 
    public void anotherReport(){
        System.out.println(field);
    }
}

class ShadyChild extends Parent{
    protected int field;
    public ShadyChild(int f){
        super(f); field = 2*f;
    }
    public void anotherReport(){
        System.out.println(field);
    }
}

What Gets Printed?

public class Shadowing{
    public static void main(String args[]){
        Parent p = new Parent(1);
        p.reportField();
        ProperChild pc = new ProperChild(2);
        pc.reportField();
        pc.anotherReport();
        ShadyChild sc = new ShadyChild(3);
        sc.reportField();
        sc.anotherReport();
    }
}

Don’t write code like this...
Preventing Inheritance

- Occasionally want to prevent inheritance of a class
- Keyword `final` useful to prevent change; Examples:

```java
class C {
    public final int x
    
    Assign variable/field x once

    public final class C {
        ...
    }

    Class C is defined once, no subclasses allowed, will never have children

    public class D {
        public final int doIt() {
            ...
        }
        public int fakeIt() {
            ...
        }
    }

    Class D can have children, children can override fakeIt() but cannot override doIt()
```
Why Make a Class/Method final?

- Somewhat beyond the scope of this course
- Canonical example: String is final to keep it immutable
- Prevents any crazy, change-able child strings from being used in place of immutable version
- Enables compiler/runtime optimizations and potentially some security
- final methods may enable somewhat better performance to avoid dynamic dispatch
Forcing Inheritance

- Sometimes want to set up a hierarchy but don’t have a good default behavior
- Force implementation of certain methods

```java
abstract public class Animal{
    abstract public void proclaim(String s);
}
```

- Every animal must be able to proclaim(..)
- No default way to proclaim: make it abstract
Why abstract class?

Interchangeable parts
Interchangeable components can be set up via 3 mechanisms
- Inheritance (abstract classes)
- Interfaces (soon)
- Generics (later in the course)

All rely on interchangeable part having similar/same methods

When to use abstract class
Following factors indicate abstract class is the correct mechanism
- Obvious hierarchy of objects
- No need to mix in methods: class Z does NOT need methods from both class X and Y
- Want to share implementation and fields between some classes
- No complete default implementation
P3: Example of shared implementation

- abstract class Cipher
  - Every child must have encrypt()/decrypt()
  - No default implementation for these
- abstract class SymmetricCipher
  - Implements encrypt(), decrypt(), rotate(), wrapInt()
  - Leaves encrypt1(), decrypt1() as abstract
- class CaesarCipher extends SymmetricCipher
  - Inherits methods encrypt(), decrypt(), rotate(), wrapInt()
  - Implements encrypt1(), decrypt1()
- class VigenereCipher extends SymmetricCipher
  - Inherits methods encrypt(), decrypt(), rotate(), wrapInt()
  - Implements encrypt1(), decrypt1()
- class MorseCipher extends Cipher
  - No encryption keys, just a lookup table for encrypt()/decrypt()
  
  
  
  
  ".-",   /* A */   "-...",   /* B */   ".-.",   /* C */
Swing: Java GUI Classes

These set up a deep hierarchy, many abstract classes

Will trace down to JButton
Top of the Hierarchy

public abstract class Component extends Object
implements ImageObserver, MenuContainer, Serializable

A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user.

public class Container
extends Component

A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components.

public abstract class JComponent extends Container
implements Serializable

The base class for all Swing components except top-level containers. To use a component that inherits from JComponent, you must place the component in a containment hierarchy whose root is a top-level Swing container.
public abstract class AbstractButton extends JComponent
implements ItemSelectable, SwingConstants

Defines common behaviors for buttons and menu items.

public class JButton extends AbstractButton
implements Accessible

An implementation of a "push" button.
Quick Input Calculation

Consider simple data file `scores.dat` of name / score

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adama</td>
<td>17.0</td>
</tr>
<tr>
<td>Baltar</td>
<td>18.0</td>
</tr>
<tr>
<td>Thrace</td>
<td>16.0</td>
</tr>
<tr>
<td>Tye</td>
<td>10.0</td>
</tr>
<tr>
<td>Rosslyn</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Write some quick code that computes

- Mean score: average of all scores
- Max score with name of max earner

Sample output

```
> cat scores.dat
Adama 17.0
Baltar 18.0
Thrace 16.0
Tye 10.0
Rosslyn 15.0
> javac SimpleScores.java
> java SimpleScores
Mean: 15.20   Max: 18.00 by Baltar
```
Generalize

More Columns

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
<th>HW1</th>
<th>HW2</th>
<th>Exam1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lee</td>
<td>Adama</td>
<td>17.0</td>
<td>12.0</td>
<td>34.0</td>
</tr>
<tr>
<td>Gaius</td>
<td>Baltar</td>
<td>18.0</td>
<td>13.0</td>
<td>38.0</td>
</tr>
<tr>
<td>Kara</td>
<td>Thrace</td>
<td>16.0</td>
<td>10.0</td>
<td>24.5</td>
</tr>
<tr>
<td>Saul</td>
<td>Tye</td>
<td>10.0</td>
<td>13.5</td>
<td>34.0</td>
</tr>
<tr>
<td>Laura</td>
<td>Rosslyn</td>
<td>15.0</td>
<td>12.0</td>
<td>36.0</td>
</tr>
</tbody>
</table>

More Statistics

Mean, max(name), min(name), total, standard deviation, mode, median...

- Which of these are easy/hard?
- What is the general pattern of a statistic?
- Can we generalize?
Dremel: A tool with Interchangeable Parts
Car: Many Interchangeable Parts

Source
Heels... Okay this is just ridiculous

Source
A Possible Pattern

A statistic

- Has an initial value (may be NaN)
- Can be *updated* with new input value
- Can report its current *value*
- Can be stringified

Is there a default implementation of these that fits for several statistics?

- Mean, max(name), min(name), total, standard deviation, mode, median...
Abstract Statistic

// What can statistics "do"?
public abstract class Statistic {
    public abstract double value(); // Current value
    public abstract void update(String s, double x); // Update
    public abstract String toString(); // Pretty print
}

Establish A hierarchy rooted at Statistic
Statistic mean = new Mean();
Statistic stdev = new StandardDev();

Benefit abstract methods don’t need to be written: no body present, just prototype

Cost Can’t actually create a plain Statistic
Statistic s = new Statistic();
// Error: Statistic is abstract;
// cannot be instantiated
Implementations

Want substitutable behaviors

▸ Substitute: hard-coded \(\rightarrow\) parameterized
▸ Components have internal state and behavior
▸ Statistic is a class, Mean, Max, Total are its children
▸ Children can do anything a parent can do

Exercise

▸ Adapt SimpleScores.java to use Statistic
▸ SimpleStats.java handles a single name/score data file
▸ Uses a Mean and a Max

Input Data

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adama</td>
<td>17.0</td>
</tr>
<tr>
<td>Baltar</td>
<td>18.0</td>
</tr>
<tr>
<td>Thrace</td>
<td>16.0</td>
</tr>
<tr>
<td>Tye</td>
<td>10.0</td>
</tr>
<tr>
<td>Rosslyn</td>
<td>15.0</td>
</tr>
</tbody>
</table>
If Statistic isn’t going to have any fields nor any concrete methods, why make it a class at all.

**Abstract Parent**

```java
public abstract class Statistic {
    public abstract double value();
    public abstract void update(String s, double x);
    public abstract String toString();
}
```

**Interface**

```java
public interface Statistic {
    public double value();
    public void update(String s, double x);
    public String toString();
}
```

All methods automatically abstract, can’t specify a body for any of them.
Abstract vs Interface

- **Ethereal**
  - Interface
    - No fields allowed
    - Methods can't be specified
  - Abstract Class
    - Some fields/methods specified
    - Some methods abstract
  - Regular Class
    - All fields/methods specified

- **Concrete**
  - Enumeration
    - All instances specified

interface ≈ abstract class with all abstract methods

**Similar Except**

- Classes can only descend from one parent, abstract or not
- Classes can implement many interfaces
- Methods from both places referred to as "inherited from..."

**Example:** JButton (javadoc)
extends how many parents? vs implements how many interfaces?
Interface

Like a "capabilities badge" that classes can wear

Interfaces

I'm interface Savable. You must implement void save(String fn) to wear my badge.

```java
public interface Savable{
    void save(String fname);
}
```

I'm interface Describable. You must implement both versions of the describe() method to wear my badge.

```java
public interface Describable{
    void describe(AudioStream o);
    void describe(PrintStream o);
}
```

Implementing Classes

I'm class C and I can be saved because I implements Savable

```java
public class C implements Savable {
    public void myMeth(){...}
    public void save(String fn){...}
}
```

I'm class D, my parent is X and I'm both Savable and Describable

```java
public class D extends X implements Savable, Describable {
    public void save(String fn){...}
    public void describe(AudioStream o){...}
    public void describe(PrintStream o){...}
    public String dooDad(){...}
```
Interface Particulars

Methods are automatically public abstract

```java
public interface Savable{
    void save(String fname);
}
```

IDENTICAL TO

```java
public interface Savable{
    public abstract
    void save(String fname);
}
```

Can form a Hierarchy (infrequent)

```java
public interface Savable{
    void save(String fname);
}
```

```java
public interface ReadWritable extends Savable {
    void load(String fname);
}
```

```java
public class C implements ReadWritable {
    public void save(String f){...}
    public void load(String f){...}
}
```
Interfaces Set Up Interchangeable Parts

- Any class that implements `Savable` can be in an array of `Savable` objects
- Dynamic dispatch to the specific object’s version of `save(f)`

```java
public class X {
    public static void saveAll(Savable[] arr,
        String[] fnames)
    {
        for(int i=0; i<arr.length; i++){
            Savable s = arr[i];
            String f = fnames[i];
            s.save(f);
        }
    }
    public static void main(String args[]){
        Savable[] sa = {new C(),
            new D(),
            new Y()};
        X.saveAll(sa);
    }
}
```

```java
public class D extends X
implements Savable, Describable {
    ..
}
```

```java
public class C
implements Savable {
    ..
}
```

```java
public interface Savable{
    void save(String fname);
}
```
Interface Examples

Code pack has versions of `Statistic` as both
- Abstract class hierarchy
- Interface implemented by classes
Both look similar
Implements vs Extends

Recall Statistic: formerly abstract class, now interface

Interface Definition

```java
public interface Statistic {
    public double value();
    public void update(String s, double x);
    public String toString();
}
```

Implementing Class

```java
public class Total implements Statistic {
    protected double sum;
    public Total(){
        this.sum=0.0;
    }
    public void update(String s, double d){
        this.sum += d;
    }
    public double value(){
        return sum;
    }
    public String toString(){
        return String.format("total %5.2f", this.sum);
    }
}
```
Why have both?

During the memorable Q&A session, someone asked [James Gosling]: "If you could do Java over again, what would you change?" "I’d leave out classes," he replied. After the laughter died down, he explained that the real problem wasn’t classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible.

– Allen Holub, Why extends is evil (2003)
Exercise

Recall Op From Exam 1

- **Group 1** Rewrite using
  enum Op
  with single static
  transform() method in Op

- **Group 2** Rewrite using
  interface Op
  with many implementing
classes to get dynamic
dispatch

```java
class Op{}
class Double extends Op{}
class Incr extends Op{}
public class OpDemo{
  public static int
  transform(Op o, int x) { return x; }
  public static int
  transform(Double o, int x) { return 2 * x; }
  public static int
  transform(Incr o, int x) { return x + 1; }

  public static void main(String args[]){
    Op [] ops = {new Double(),
                 new Incr()};
    int arg = 10;
    for(Op op : ops){
      int ans = transform(op, arg);
      System.out.println( ans );
    }
  }
}
```