CS 211: Basic Inheritance

Chris Kauffman

Week 5-1 and 6-1
Logistics

Goals Today

▶ Equality
▶ 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>
Inheritance

**Warning:** Inheritance is a tricky subject because...

- It’s not too bad to understand basic mechanics
- Creates behavior only observable at runtime
- **Spreads out code** to do one task into multiple places
- Advantages are not apparent until you have a large system
- Teaching examples do not reflect what inheritance is good for

**Our Approach**

- Spend today and part of Thursday on mechanics of inheritance
- These will involve little examples with mostly bad practice associated with it
- Then discuss good/bad applications of inheritance and why extends may in fact be evil
Basic Inheritance Mechanics: Animals.java

Primary reason for an inheritance hierarchy is to create a container for several kinds of things that can behave differently.

class Animal{ }
class Human extends Animal { }
class Mouse extends Animal { }

main(){
    Animal animals[] = new Animal[]{
        new Animal(),
        new Human(),
        new Mouse()
    };

    ... 
}

- Each animal implements its own proclaim() method
- Each behaves differently on
P3: Cipherous Symmetry

- Implement 2 simple ciphers to encrypt/decrypt + Morse Code
- Exist in an object hierarchy to
- Implement an alphabet class to handle character wrapping

**Cipher (abstract)**
- MorseCipher
- SymmetricCipher (abstract)
  - CaesarCipher
  - VigenereCipher

**Alphabet**

**Required Techniques**
- Inheritance
- Overriding methods
- Single Dynamic dispatch
- Throwing Exceptions
- Making Exception Classes
- Annotation @Override
- abstract classes
Extend an existing class like Exception or RuntimeException

```java
public class CKException extends Exception {
    public String toString() {
        return "Bogus!";
    }
}
```
Annotations

Java Annotations

- @Information for the compiler
- Like comments but the compiler may not completely ignore
- Metadata that summarizes the intent of code

Examples

- @Test This code tests other code (compiler may just ignore)
- @Deprecated This code is old, unsupported, may disappear
- @Override Error if not overriding parent method
Note on @Override

Annotating methods with @Override which are intended to override a parent method notifies the compiler to check for danger.

A Subtle Bug

```java
@Override
public boolean equals(Coord other){
    if(other==null || !(other instanceof Coord)){
        return false;
    }
    Coord c = (Coord) other;
    return this.row==c.row &&
            this.col==c.col;
}
```

Compiler Output

```
  > javac Coord.java
  Coord.java:17: error: method does not override or implement a method from a supertype
     @Override
^ 1 error
```
Fields are Inherited: 2D vs 3D Coord

```java
public class Coord {
    public final int row;
    public final int col;
    public Coord(int ir, int ic) {
        this.row = ir;
        this.col = ic;
    }
    public String toString() {
        return String.format("(%d,%d)", row, col);
    }
    public boolean equals(Coord c) {
        return this.row == c.row &&
                this.col == c.col;
    }
}

public class Coord3D extends Coord {
    // Fields row and col are inherited
    public final int height;
    public Coord3D(int ir, int ic, int h) {
        super(ir, ic); // Required
        this.height = h;
    }
    public String toString() {
        return String.format("(%d,%d,%d)", row, col, height);
    }
    public boolean equals(Coord3D other) {
        return this.row == other.row &&
                this.col == other.col &&
                this.height == other.height;
    }
}
```
Child Classes Must Call Parent Constructor

- Animal did not specify a constructor
- Java always provides a default 0-argument constructor if no constructors are specified
  ```java
  Animal a = new Animal();
  ```
- The constructor for Human initializes its parent class automagically as follows
  ```java
  public Human(){ // Created automatically
    super(); // Call done automatically
  }
  ```
- Coord has a two-argument constructor
  ```java
  Coord c = new Coord(1,2);
  ```
- That means it is now illegal to say
  ```java
  Coord c = new Coord();
  ```
  unless a zero-arg constructor is explicitly defined
- Coord3D must call a valid parent constructor
- Coord3D must therefore call constructor
  ```java
  super(ir,ic);
  ```
That’s super!

Keyword *this* gives access to *present* class’s fields and methods

```java
this(arg1, arg2, arg3); // call another constructor
this.someField = stuff; // access a field
this.doSomething(x, y); // call a method
```

Keyword *super* gives access to *parent* class’s fields and methods

```java
super(arg1, arg2, arg3); // call parent constructor
super.someField = stuff; // access parent field
super.doSomething(x, y); // call parent method
```
Equality Gets Trickier

What is printed on the right based on equals() definitions?

```java
public class Coord {
    public boolean equals(Coord c){
        return this.row==c.row &&
            this.col==c.col;
    }
}

public class Coord3D extends Coord{
    public boolean equals(Coord3D other){
        return this.row == other.row &&
            this.col == other.col &&
            this.height == other.height;
    }
}
```

```java
Coord a = new Coord(1,2);
Coord b = new Coord(10,12);
Coord3D c = new Coord3D(1,2,3);
Coord3D d = new Coord3D(10,12,14);

System.out.println( a.equals(b) );
System.out.println( a.equals(c) );
System.out.println( a.equals(d) );
System.out.println();
System.out.println( b.equals(c) );
System.out.println( b.equals(d) );
System.out.println();
System.out.println( c.equals(d) );
System.out.println();
String s = "(1,2)"
System.out.println( s.equals(a) );
System.out.println( a.equals(s) );
```
Everyone has `equals()` and `toString()`.

```java
package java.lang;
public class Object{ ... }
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

```java
public String toString()

Returns a string representation of the object.

public boolean equals(Object obj)

Indicates whether some other object is "equal to" this one.

```java
int a[] = {1,2,3}, b[] = {1,2,3};
System.out.println( a.equals(b) ); // ??
```
Checking type at run time: instanceof

X instanceof Y

- A keyword/syntax construct
- true if X has Y as an ancestor - X is a Y
  - Mascot is a Duck, Duck is a Animal, Animal is a Object
- false otherwise
Casting: Trust me, javac

```java
Object o = new Coord(1,2);
System.out.println(o.row);  // Compile error
Coord c = (Coord) o;        // Trust me, it’s a Coord
System.out.println(c.row);  // Voila!
```

- What can go wrong with casting: `(Coord) o`
- Try it interactively:

```java
Object o = new String("hi");
Coord c = (Coord) o;
```

- What about the following...

```java
Object x = new Coord(1,2);
Object y = new Coord(1,2);
System.out.println( x.equals(y) );
```
The most common case of casting

Compare current object like Coord to arbitrary other Objects

Coord Class Methods

// Are coordinates equal
public boolean equals(Coord c){
    return
    this.row==c.row &&
    this.col==c.col;
}

// Compare arbitrary object
public boolean equals(Object o){
    if(o instanceof Coord){
        Coord c = (Coord) o;
        return this.equals(c);
    }
    else{ return false; }
}

Equals works great now

Object x = new Coord(1,2);
Object y = new Coord(1,2);
System.out.println( x.equals(y) );

But what about...

Object w = new Coord(1,2);
Object z = new Coord3D(1,2,3);
System.out.println( w.equals(z) );
System.out.println( z.equals(w) );

(Hint: damn...)
Extending Classes You Can’t See Inside

When writing programs

- Create whole new class hierarchy: Rare
- Extend someone else’s class: Frequent

PrintWriter and Extensions

- Lab will have you extending the java.io.PrintWriter class
- Can’t see the source code (without searching for it)
- How do you extend it?
**PrintWriter**

A class that allows printing to the screen or to a file

```java
PrintWriter out = new PrintWriter(new File("myfile.txt"));
// PrintWriter out = new PrintWriter("myfile.txt");
// PrintWriter out = new PrintWriter(System.out);
out.println("Sweet foutput");
out.printf("An int: %d\nA double %.1f\nA string: %s\n", 1, 2.5, "Three");
out.close(); // May close System.out (bad)
```

Have a look at the [PrintWriter Java Doc](#).
Exercise: ScreamWriter

► It's bad form to SCREAM TEXT CONSTANTLY
► But some folks do it anyway
► Extend PrintWriter to ScreamWriter which screams output
► toggleVolume() turns screaming off/on
► ScreamWriters equal if screaming on/off matches

Welcome to DrJava.
> ScreamWriter out = new ScreamWriter(System.out);
> out.println("Hello there.");
HELLO THERE.
> out.toggleVolume()
> out.println("That’s better");
That’s better
> Object out2 = new ScreamWriter("somefile.txt");
> out2.equals(out)
false
> out.toggleVolume()
> out2.equals(out)
true
public class ScreamWriter extends PrintWriter

Write two constructors that allow ScreamWriters to be created. Will need to call parent class constructor with super(..)

public ScreamWriter(OutputStream o) throws Exception
public ScreamWriter(String filename) throws Exception

- Establish a field to control volume (SCREAM vs Normal)
- Create/Override the following methods
- Use parent version of println()

public void toggleVolume() // Turns screaming on/off
public void println(String s) // Print, maybe all caps
public boolean equals(Object o) // True for another SCREAMING writer

Grind on this one a few minutes. Answer in today’s code pack.
Dynamic Dispatch

Suppose we have an animal

Animal a = ...;

Methods: Single Dispatch

a.doSomething()

Call the method doSomething() with the most specific kind of thing a is as this

- Always done of method invocation
- There is runtime performance penalty

No Dispatch on Arguments

someFunction(a);

Call method someFunction() with a treated as a plain Animal as the argument

- Type of a determined at compile time and appropriate method is chosen
- No runtime performance penalty

SingleDispatch.java demonstrates this difference
Multiple Dispatch

Incredibly useful in some programming problems as it simplifies code but not present in java: see the code in DoubleDispatch.java

```java
public static void meets(Animal x, Animal y){
    System.out.println("Nothing special");
}
public static void meets(Snake x, Mouse y){
    System.out.println("Snake eats mouse");
}
public static void main(String args[]){
    Animal x = new Snake();
    Animal y = new Mouse();
    meet(x,y); // What do I print?
}
```