CS 211: Basic Inheritance

Chris Kauffman

Week 5

Announcement: Engineers Week

National Engineers Week is February 21st-27th. Special activities begin with a Kick-off Social in the Atrium on Monday, February 22 at 3:00 p.m.. Enjoy cookies, cake, and light refreshments with friends and colleagues.

- Tue 2/21: 10:30am-1:00pm: Patriot Hackers in ENGR Atrium
- All Activities: https://volgenau.gmu.edu/sites/common/files/Engineers%20Week%202017_0.pdf

Announcement: Spring Career Fair

- Wednesday, February 22: Science, Technology, Engineering and Math focus
- 11 a.m. - 4 p.m.
- Fairfax Campus, Johnson Center, Dewberry
- Thursday, February 23: Business, Public Service and Non-Tech Focus

Logistics

Labs

05 Exercises: Inheritance with PrintWriter

P3 Up, Discuss

Spellcheckers via Inheritance

Reading: Inheritance

- Building Java Programs Ch 9
- Lab Manual Ch 7

Goals Today

- Overview P3
- Continue Discussion of Inheritance
- Equals Methods

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

Fields are Inherited: 2D vs 3D Coord

```
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;
  }
}
```

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

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

Compiler Output

```
> javac Coord.java

Coord.java:17: error:
method does not override or
implement a method from a
supertype
  @Override
  ^
1 error
```

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

  ```
  Animal a = new Animal()
  ```
- The constructor for `Human` initializes it's parent class automagically as follows

  ```
  public Human(){ // Created automatically
    super();      // Call done automatically
  }
  ```
- `Coord` has a two-argument constuctor

  ```
  Coord c = new Coord(1,2);
  ```
- That means it is now illegal to say

  ```
  Coord c = new Coord();
  ```

  unless a zero-arg constructor is explicitly defined
- `Coord3D` must call a valid parent constructor
- `Coord3D` must therefore call constructor

  ```
  super(ir,ic);
  ```

That's super!

Keyword `this` gives access to present class's fields and methods

```
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 method

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

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

```
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
```

ScreamWriter Strategy

```
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.

Recap: Inheritance

- Inherited attributes:
  - Class `Coord3D` extends class `Coord`: what attributes does `Coord3D` get through inheritance?
  - Class `ScreamWriter` extended `PrintWriter`: what attributes did it get?
  - What attributes does any child class get through inheritance?
- What must be done if a child class wants to behave differently than the parent class?
- What is the difference between the keywords `this` and `super`?
- How does a child class initialize its parent class?
- How does a child class invoke its parent class's version of a method?
- What methods does every class and why?

Inherited Fields: `protected` vs `private`

Modifier	Class	Package	Subclass	World	Note
public	Y	Y	Y	Y	Children see it
protected	Y	Y	Y	N	Children see it
no modifier	Y	Y	N	N	Children don't
private	Y	N	N	N	Children don't

```
class Parent{
  protected int prot;
  private int priv;
  public Parent(int i, int j){
    this.prot=i; this.priv=j;
  }
}
class Child extends Parent{
  public Child(){
    super(1,2);
  }
  public void show(){
    System.out.println(prot);
    System.out.println(priv);
  }
}
```

```
> javac ProtectedFields.java
ProtectedFields.java:17:
 error: priv has private access
        in Parent

    System.out.println(priv);
```

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. . .

Exercise: Finish ScreamWriter

Current Solution

```
import java.io.*;
public class ScreamWriter
extends PrintWriter
{
  public ScreamWriter(OutputStream o){
    super(o);
  }
  public ScreamWriter(File f)
  throws Exception
  {
    super(f);
  }
  public ScreamWriter(String filename)
  throws Exception
  {
    super(new File(filename));
  }
  public void println(String s){
    String output = s.toUpperCase();
    super.println(output);
    this.flush();
  }
}
```

Allow Volume Toggling

```
// Turns screaming on/off
public void toggleVolume()
```

Example use:

```
> ScreamWriter out =
    new ScreamWriter(System.out);
> out.println("Hello there.");
HELLO THERE.
> out.toggleVolume()
> out.println("That's better");
That's better
> out.toggleVolume()
> out.println("how about now?");
HOW ABOUT NOW?
```

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`

```
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?
}
```