# **CS100M**

#### Introduction to Computer Programming

Spring 2004 Inheritance Summary

#### Announcements

- Game Open House:
  Wed, May 12, 3:30-6:30,Upson 315, 319
  www.cs.cornell.edu/projects/game
- A6, Lab
- course evaluations: part of your grade! http://www.engineering.cornell.edu/courseeval
- Final exam, review, etc

#### **Motivation/Overview**

- Classes as types
- Try to link related classes for code reuse
  - subtyping: create a subcategory of another type
     polymorphism: can use multiple types to manipulate objects
  - keyword: **extends**
- How to use code reuse? *inheritance* 
  - to say code is inherited means that it is copied to a subclass
  - for code reuse, you do not actually rewrite the code
- Rules for...
  - members (fields, methods for CS100) can be inherited
  - constructors: not inherited-they need their own rules

#### Aliases

- Reminders:
  - upcasting is legal (always works)
     Supertype var = new Subtype(...)
  - downcasting requires a cast (might not work)
  - Subtype var = (Subtype) new Supertype(...)
- A bit more about subtyping: aliases?
  - Coin c1 = new Dime();
  - Coin c2 = new Penny();
  - c2 = c1;

1

## **Inheritance of Fields**

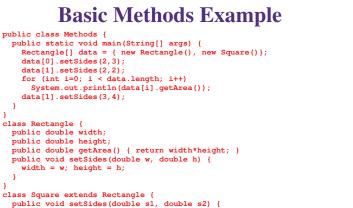
- inheritance of **public** fields:
  - automatically inherited
  - values set for current object, but code used is from superclass
  - avoid: shadowing fields (**public** fields with same names in super and sub classes)
- access of fields:
  - use type of reference (not actual object type) to access
  - rule really affects special cases of shadowing and private fields

#### **Basic Field Example**

```
public class Fields {
   public static void main(String[] args) {
      Student s = new Student();
      s.name = "Borgir";
      System.out.println(s);
      Person p = new Student();
      p.name = "Dimmu";
      System.out.println(p);
   }
}
class Person {
   public String name;
   }
class Student extends Person {
      public String toString() { return "Student: "+name; }
}
```

**Methods and Overriding** 

- inheritance of **public** methods:
  - automatically inherited unless overriden (see below)
- dynamic method binding: use actual object type to access
- overriding:
  - inherited method uses fields that have also been inherited
  - maybe the subclass should have a different behavior?
    - you can write the same method header in the subclass
    - the method body differs
    - the subclass method is said to *override* the superclass method



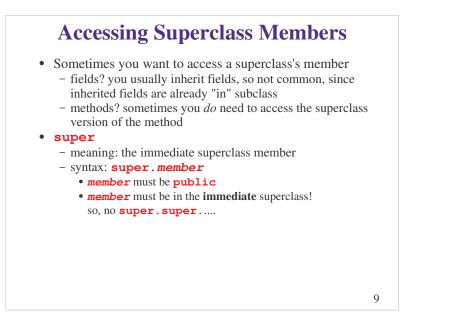
```
if (s1!=s2) {
   System.out.println("not a square!");
   System.exit(0);
   width = height = s1;
```

¥

7

5

8



```
class Square extends Rectangle {
    public void setSides(double s1, double s2) {
        if (s1!=s2) {
            System.out.println("not a square!");
            System.exit(0);
        }
        super.setSides(s1, s1);
    }
}
```

#### **Information Hiding and Inheritance**

- information hiding and abstraction
  - good style for OOP!
  - problem with **private**: involves blizzard of more rules
  - solution: allow subclasses to access superclass members but not let non-related classes have access: protected
- rules:
  - style: private members (fields and internally-used methods) now become protected
  - syntax: effectively **private** for the same package (defined group of classes) but not for outside class
- what's best? **private** if possible
  - see also package visibility (no modifier)

```
11
```

# **Information Hiding Example**

```
class Rectangle {
  protected double width;
  protected double height;
  public double getArea() { return width*height; }
  public void setSides(double w, double h) {
    width = w;
    height = h;
  }
}
class Square extends Rectangle {
    public void setSides(double s1, double s2) {
        if (s1!=s2) {
            System.out.println("not a square!");
            System.exit(0);
        }
        super.setSides(s1, s1);
    }
}
```

### **Constructor Chaining**

- Constructors aren't members, so...
  - they don't inherity
  - but they do call each other
  - concept: superclasses set general info for subclasses, and subclasses handle their own specific info
- Gist of chaining....

#### **More Constructor Chaining**

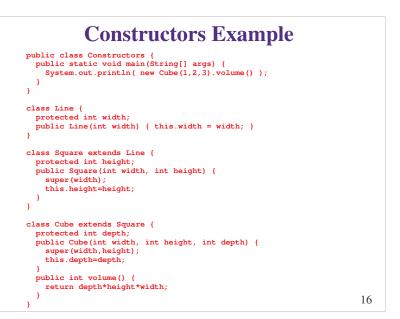
- Rules:
  - all classes must have a constructor
  - if you do not provide a constructor, Java provides the empty constructor as the default
  - 1<sup>st</sup> statement of constructor must be call to another constructor of same class (this(...)) or a call to the immmediate superclass constructor (super(...))
  - if you do not provide a super (...), Java will call super (), which means the superclass better have an empty constructor! (see 2<sup>nd</sup> rule)

14

#### **Constructor Rules (continued)**

- Order of construction
  - set all fields to default values of "zero" even if they have an assignment statement!
    - eg) the field assignment int x = 9; means x gets 0
  - invoke only the chain of this (...) and super (...) (constructor invocation)
    - at the "top" you reach Object's constructor and...
    - Set all the field assignments (if any) for the top class
    - Execute the rest of the top's constructor (constructor execution)
    - Go to next highest subclass in the chain and repeat
- Why bother?
  - actually, usually you don't need to
  - sometimes need to know when fields are set
  - affects shadowing, which you should avoid

15



# More Information Hiding (advanced!)

- package
  - group classes together
  - syntax: package name;
  - first statement of program
  - see Savitch 5.7
- private members "bind to their class "
  - no overriding! no external access, even by subclass and super
  - need to provide public members in subclass to access a private member
  - "bind to their class": called *static binding*: assocation created when compiling
  - dynamic binding: when associations occur at run time

17

#### **Information Hiding (continued)**

- static:
  - also set at compile time, no dynamic binding
  - consequence: **static** methods cannot be overridden
  - if you have two static methods with same header, they are completely different methods with no relation to eachother! (bad style)
  - someone ask me why the name **static** is now explained...
- final:
  - fields? cannot change after initialization and constuctor sets
  - methods? cannot override
  - classes? cannot make subclass

18

#### **Example** public class Shadowing { public static void main(String[] args) { A = new B();a.test4(); } class A { public int x; public A() { test1(); test2(); test3(); } private void test1() { System.out.println(x); } public void test2() { System.out.println(x); } public void test3() { System.out.println(x); } public static void test4() { System.out.println("Hi"); } } class B extends A { public boolean x = true;private void test1() { System.out.println(x); } public void test3() { System.out.println(x); } public static void test4() { System.out.println("Bye!"); } 19

### **Class Object**

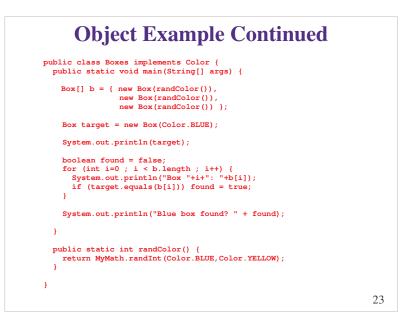
- **Object**: Superest superclass of them all!
  - source of **toString**, **equals**, and others
  - see API for full list
- Uses
  - generic code! data structure can hold pretty much anything
  - convenience methods (see above)

#### **Object Example**

```
// color constants for boxes
interface Color {
  public final int BLUE = 0;
  public final int RED = BLUE+1;
  public final int YELLOW = RED+1;
}
// handy dandy random int generator
class MyMath {
  public static int randInt(int low, int high) {
    return (int) (Math.random()*(high-low+1)) + (int)low;
  }
}
```

21

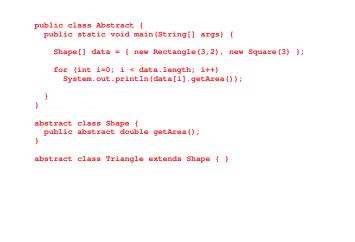
#### **Object Example Continued** class Box implements Color { private int color; public Box(int color) { this.color=color; 3 public int getColor() { return color; } public boolean equals(Object other) { return color==((Box)other).color; 3 public String toString() { switch (color) { case Color BLUE return "Blue"; case Color.RED: return "Red"; case Color.YELLOW: return "YELLOW"; default: return "UNKNOWN": - } 1 22



#### **Abstract Classes**

- Design issues:
  - completely specify full class hierarchy
  - specify only types (interfaces, which can include constants and method headers)
  - anything inbetween?
- abstract class
  - partially specified class
  - can contain at least one abstract method (no body)
  - cannot make objects from abstract class
- syntax for abstract class, abstract method: modifiers abstract class Name { ... } modifiers abstract RetType Name(...);

#### **Abstract Class Example**



#### **Abstract Class Example (continued)**

```
abstract class Quadrilateral extends Shape {
 protected double s1, s2, s3, s4;
 public Quadrilateral(double s1, double s2, double s3, double s4) {
    this.s1=s1; this.s2=s2; this.s3=s3; this.s4=s4;
  3
-F
class Rectangle extends Ouadrilateral {
 public Rectangle(double s1, double s2) {
    super(s1, s2, s1, s2);
  public double getArea() { return s1*s2; }
3
class Square extends Rectangle {
 public Square(double s) {
    super(s,s);
  }
ъ
                                                                          26
```

#### **Interface vs Abstract**

- interface resembles a completely abstract class
- abstract:
  - need to reuse code
  - abstract class resembles a repository
  - also helps define classification scheme from a very high to low level
- interface:
  - want to share method name, but perhaps little relation
  - building a hierarchy would take a lot of abstract classes
  - worried only about subtyping, not code reuse
- examples?

#### **Design Revisted**

- brainstorm
- research: nouns, verbs
  - nouns:
    - constant, whole noun? field, local, constant, static (sharing)
    - composite noun? class
    - class related to another class, code reuse? inheritance
    - class relation, no code reuse? interfaces
  - verbs:
    - known operation? operator
    - action you define and name? method

# **More Design**

- outline:
  - algorithm, steps to solve problem
    - pseudocode to keep general
    - stepwise refinement: write and test a little bit at a time
    - stubbing: define all class and method signatures (use interfaces to ensure consistency)

29

- top-down:
  - start at top of stubs
  - comment and write and test
- bottom-up:
  - start in utility methods, utility classes
  - test code with basic test cases and build up

# **More Design**

- polishing:
  - baby steps!!!
  - special trick...?
- testing:
  - test cases up front?
  - known, simple values by hand
  - exhaustive test cases?
  - special checks inside program?
- iteration?