Daily Joke

• Q: What’s the object-oriented way to become wealthy?
• A: Inheritance
Inheritance

• Important questions:
  – What is inheritance?
  – How to use inheritance?

• The biggest difficulty:
  – Inheritance is specifically used for “better design”
  – Design is harder than implementation, so you haven’t done much design
Inheritance

• A way to organize classes
• Derived classes share the characteristics of base classes
• Usually referred as subclass and superclass
  – We don’t use child class and parent class because it’s inaccurate
Example: Bike

```java
public class Bicycle {
    // the Bicycle class has three fields
    public int cadence, gear, speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear; cadence = startCadence; speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }
    public void speedUp(int increment) {
        speed += increment;
    }
}
```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear); // introduce later
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}

Example: MountainBike
Syntax Rules

- public class **Derived_CLASS_Name** extends **Base_CLASS_Name**
  - public class **MountainBike** extends **Bicycle**
- After the inheritance, the subclass inherits all the **public** variables and methods of the superclass
  - Also, the subclass can add new variables and methods
    - Bicycle class has **cadence, gear, speed**, constructor and four setters
    - MountainBike class has **cadence, gear, speed**, seatHeight, constructor, **four setters** and a new setter setHeight()
First Summary

• Subclasses inherit all public variables and methods from superclass
  – They can use these variables and methods as their own
    • MountainBike mb = new MountainBike(110, 50, 30, 4);
    • mb.setGear(5);
  – You don’t have to copy and paste the duplicate methods. It **seems** a good way to reuse your old code
More Inheritance: Override

• Moreover, you can write a method (and variables) in the subclass to **hide** the method **with the same name** in the superclass
  – In this example, the MountainBike has a powerful break so it immediately reduce the speed to 0

```java
public class MountainBike extends Bicycle {
   // the MountainBike subclass overrides one method
   public void applyBrake(int decrement) {
       speed = 0;
   }
}
```

– Now if we call `mb.applyBrake(3)`, the speed will be 0
  • It won’t be the old speed minus 3, as the superclass defines
Wait a Minute......

- What’s the point of overriding a method
  - If we want to reuse a method by inheritance, why do we rewrite the method?
- If we think more – why do we reuse our code by inheritance?
  - We can simply use the old class in the new class
  - Remember that we only inherit the public variables and methods – there is no difference between using the superclass
public class MountainBike2 {
    public int seatHeight;
    // the Bicycle class is used -- instead of inherited
    public Bicycle mb;

    public MountainBike2(int startHeight, int startCadence, int startSpeed, int startGear) {
        mb = new Bicycle(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    public void setGear(int newValue) {
        mb.setGear(newValue);
    }

    public void applyBrake(int decrement) {
        mb.speed = 0;
    }
}
Inheritance is NOT for Reusability

• Though inheritance can be good for reusability, it is not intended for reusability
  – That means, if you want to reuse your code, you shall not think about inheritance first!

• Inheritance is for flexibility
  – It is used when different objects need different methods
  – We call this property “polymorphism”
Polymorphism

• It means “many forms”
• Same instruction to mean different things in different contexts.
  – Example: “Go play your favorite sport.”
    • I’d go play soccer
    • Others of you would play basketball or football instead.
• In programming, this means that the same method name can cause different actions depending on what object it is applied to
Why is Polymorphism Required?

• Let’s consider if we want to design a set of classes that represents animals
  – Every animal can play its own sound
  – If we have to write a method for each animal, the class design will be a disaster
public class Animal {
    private String animalName;
    private String species;
    private void playDuckSound() {
        // play "QUACK"
    }
    private void playDogSound() {
        // play "WOOF"
    }
    private void playCatSound() {
        // play "MEW"
    }

    public void speak() {
        if (species.equals("Duck")) {
            this.playDuckSound();
        } else if (species.equals("Dog")) {
            this.playDogSound();
        } else if (species.equals("Cat")) {
            this.playCatSound();
        }
    }
}
If We Want to Add Cow to the Class

• We must add a method called playCowSound()
  – Let it play “moo”

• Then we must change the speak() method by adding a new case in the multibranch statement
  – If there is more than one method that depends on the species, we need more
    • eat(), hunt(), sleep()
  – Again, modifying this class is a disaster
Loops, Arrays and Polymorphism

- Loops are used to repeatedly access similar statements
- Arrays are used to repeatedly access similar variables
- Polymorphism are used to access similar methods

- Their syntax rules are very different, but you shall see a similar purpose
Polymorphism and Overriding

• Key point:
  – You can create a subclass object for a superclass type variable
  – When you invoke the methods from the superclass variable, the overridden method is called

```java
// Animal.java
public class Animal {
    private String animalName;
    public void speak() {
        // default method -- can be empty
    }
}

// In another file Cat.java
public class Cat extends Animal {
    public void speak() {
        // play "MEW"
    }
    public static void main(String[] args) {
        Animal c = new Cat();
        c.speak(); // will play "MEW"
    }
}
```
Polymorphism and Overriding

```java
public class Animal {
    private String animalName;
    public void speak() {
        // default method -- can be empty
    }

    public static void main(String[] args) {
        Animal a[] = new Animal[3];
        a[0] = new Cat();
        a[1] = new Dog();
        a[2] = new Duck();
        for (int i = 0; i < 3; i++) {
            a[i].speak();
        }
    }
}

public class Cat extends Animal {
    public void speak() {
        System.out.println("MEW");
    }
}

public class Dog extends Animal {
    public void speak() {
        System.out.println("WOOF");
    }
}

public class Duck extends Animal {
    public void speak() {
        System.out.println("QUACK");
    }
}
```

Output: MEW, WOOF, QUACK
Polymorphism and Dynamic Binding

What if we want to add a new animal: cow?

- Just write a new class `Cow`
  - Nothing in `Animal` shall be changed
  - If you have another method in `Animal` that calls `speak()`, it won’t be affected

- The method invocation is not bound to the method definition until the program executes
- Java dynamically decide what method to call at `run-time`

```
public class Cow extends Animal {
    public void speak() {
        System.out.println("MOO");
    }
}
```

```
public class Animal {
    public static void groupSpeak(Animal[] group) {
        for (int i = 0; i < group.length; i++)
            group[i].speak();
    }
}
```
Second Summary: Polymorphism

• In programming, this means that the same method name can cause different actions depending on what object it is applied to
  – You can create a subclass object for a superclass type variable
  – When you invoke the methods from the superclass variable, the overridden method is called
The *is-a* Relationship

- This inheritance relationship is known as an *is-a* relationship
  - A Bear *is a* Mammal
  - A Mammal *is an* Animal
- Is a Mammal a Bear?
  - Not necessarily!
The *is-a* Relationship

```java
public class Animal {
    public void eat() {
        System.out.println("Get anything to eat");
    }
}

public class Mammal extends Animal {
}

public class Bear extends Mammal {
    public void eat() {
        System.out.println("Find a fish to eat");
    }
    public void hibernate() {
        System.out.println("Zzzzzz");
    }
}

public static void main(String[] args) {
    Animal a = new Mammal();
    // YES! A Mammal is an Animal
    Animal b = new Bear();
    // YES! A Bear is an Animal
    Mammal c = new Bear();
    // YES! A Bear is a Mammal
    // Bear d = new Mammal(); NO! A Mammal may not be a Bear!
    a.eat(); // OK. Mammal doesn't override eat(). Eat anything.
    b.eat(); // OK. Bear overrides eat(). Eat fish.
    // c.hibernate(); WRONG! Mammal doesn't have this method!
}
```
• Who is a whom?

![More Complicated Hierarchy Diagram]

- Person
  - Student
    - Undergrad
    - Grad
    - Masters
    - Doctoral
    - Nondegree
  - Employee
    - Faculty
    - Staff
Third Summary

• A subclass object can be assigned to a superclass type variable
  – After the assignment, it loses its newly added methods
  – However, it can still perform its own action from overridden methods
• Therefore, a superclass object acts as a superclass all the time, though it can be actually a subclass object
Liskov Substitution Principle

• Derived types must be **completely substitutable** for their base types
  – Inheritance in fact means “detailed substitute”
    • A bear can do anything that a mammal can do
  – Therefore we don’t name them as parent/child class
    • Children is not substitutes of their parent

• In a design, you must understand if a class **is** another class, or **uses** another class
  – Never inherit another class just because you want to use it!
is-a vs. use-a

- Sometimes it is easy to determine
  - A sedan is a car; a sedan uses an engine
- Sometimes it is hard
  - Is Square a Rectangle?
    - In program design, a square is not a rectangle!
    - Because a square can not substitute a rectangle!
      - In a rectangle, changing length won’t change its width
      - In a square, it will – it’s not acting like a rectangle!
  - Square can be implemented by using a rectangle
    - Still, not straightforward
      - Basically they are different
Square vs. Rectangle

```java
public class Rectangle {
    protected int m_width;
    protected int m_height;

    public void setWidth(int width) {
        m_width = width;
    }

    public void setHeight(int height) {
        m_height = height;
    }

    public int getWidth() {
        return m_width;
    }

    public int getHeight() {
        return m_height;
    }

    public int getArea() {
        return m_width * m_height;
    }
}

public class Square extends Rectangle {
    public void setWidth(int width) {
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height) {
        m_width = height;
        m_height = height;
    }

    public static void main(String args[]) {
        Rectangle r = new Square();
        r.setWidth(5);
        r.setHeight(10);
        // user knows that r it's a rectangle.
        // It assumes that he's able to set the
        // width and height as for the base
        // class
        System.out.println(r.getArea());
        // now he's surprised to see that the
        // area is 100 instead of 50.
    }
}
```
public, protected and private

• **private** instance variables and **private** methods in the base class are NOT inherited by derived classes
  – **private** instance variables and **private** methods are inaccessible in all other classes – including its subclasses

• **protected** instance variables and **protected** methods in the base class are inherited by derived classes
  – **protected** instance variables and **protected** methods are inaccessible in other classes except its subclasses
**public, protected and private**

- **private** instance variables and **private** methods exist in subclasses – they are just **invisible**
  - You can call them from public methods in superclasses

```java
public class Person {
    private int ID;
    protected int age;
    public int getID() {
        return ID;
    }
}

public class Student extends Person {
    public void printInfo() {
        System.out.println(age);
        // OK. Age is accessible by Student
        System.out.println(ID);
        // WRONG! ID is invisible to Student;
        System.out.println(this.getID());
        // It is OK. getID() is public
    }
}
```
Using the Keyword super

- If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword **super**

```java
public class Animal {
    public void eat() {
        System.out.println("Get anything to eat");
    }
}

public class Bear extends Animal {
    public void eat() {
        super.eat();
        System.out.println("Finding a fish to eat is better");
    }
}
```
Using the Keyword *super*

- *super* can also be used to invoke superclass's constructor. It must be the first line in the subclass constructor.

```
public class MountainBike extends Bicycle {
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
}
```

- The default constructor super() will be automatically called. If the super class does not have a no-argument constructor, you **must** invoke the superclass constructor with a matching parameter list.
Overriding and Overloading

• If a derived class defines a method of the same name, same number and types of parameter as a base class method (in short, the same signature), this is **overriding**

• You can still have another method of the *same name* in the same class, as long as its number or types of parameters are different: **overloading**
Overriding and Overloading

```java
public class BaseClass {
    public void m(int a) {
        System.out.println("Method with one int in BaseClass");
    }

    public void m(int a, int b) {
        System.out.println("Method with two int in BaseClass");
    }
}

public class DeriveClass extends BaseClass {
    public void m(int a) {
        System.out.println("Method with one int in DeriveClass");
    }

    public static void main(String[] args) {
        BaseClass c = new DeriveClass();
        c.m(0);
    }
}
```

c is a DeriveClass object. The method `m(int)` is defined (overridden) in c

Will print: Method with one int in DeriveClass
Overriding and Overloading

```java
public class BaseClass {
    public void m(int a) {
        System.out.println("Method with one int in BaseClass");
    }

    public void m(int a, int b) {
        System.out.println("Method with two int in BaseClass");
    }
}

public class DeriveClass extends BaseClass {
    public void m(int a) {
        System.out.println("Method with one int in DeriveClass");
    }

    public static void main(String[] args) {
        BaseClass c = new DeriveClass();
        c.m(0,0);
    }
}
```

**Will print: Method with two int in BaseClass**

c is a DeriveClass object. However, the method \(m(int, int)\) is not defined (overridden) in c. Therefore, it will call the inherited and overloaded method in BaseClass.
Overriding and Overloading

```java
public class BaseClass {
    public void m(int a) {
        System.out.println("Method with one int in BaseClass");
    }
}
```

```java
public class DeriveClass extends BaseClass {
    public void m(int a) {
        System.out.println("Method with one int in DeriveClass");
    }

    public void m(int a, int b) {
        System.out.println("Method with two int in DeriveClass");
    }

    public static void main(String[] args) {
        BaseClass c = new DeriveClass();
        c.m(0,0);
        // You can declare c as DeriveClass c = new DeriveClass()
    }
}
```

Will cause a syntax error

```java
// c is in BaseClass type. There is no m(int, int) method defined in BaseClass type.
```
Keyword `instanceof`

- `instanceof` is very similar to a comparison operator. It returns a boolean value indicating if an object is in a given class type.
- The syntax rule: `Variable_of_Object instanceof Class_Name`
  - If the value of the expression is true, it means the variable is (or can be treated as) in the class type.
Keyword `instanceof`

- `Person a = new Grad();`
- `Grad b = new Doctoral();`
- `Employee c = new Faculty();`
- `a instanceof Grad` is **true**
- `a instanceof Doctoral` is **false**
- `b instanceof Doctoral` is **true**
- `c instanceof Person` is **true**
- `c instanceof Employee` is **true**
Type Casting

Similar to primitive types, you can cast a variable to a different type

- Syntax rule: \((\text{Class} \_\text{Name}) \ \text{variable\_of\_object})\;
  - \text{double} \ d = 13.5;
  - \text{int} \ a = (\text{int}) \ d;
  - \text{Person} \ p = \text{new Student}();
  - \text{Student} \ s = (\text{Student}) \ p;

- A run-time error happens if you can’t cast the object
  - \text{Person} \ p = \text{new Student}();
  - \text{Student} \ s = (\text{Student}) \ p;
  - \text{Doctoral} \ d = (\text{Doctoral}) \ p; // WRONG! \ p \ is \ not \ in \ Doctoral \ type!
Type Casting

• You can cast the object only if the object is an instance of the class type
  – Therefore, you can always use
    
    \[
    \text{if (objectVariable instanceof ClassName)}
    \]
    
    
    ClassName newVar = (ClassName) objectVariable;
  – The casting can be to a higher level (to superclass) or to a lower level (to subclass). Usually we only use the explicit casting if to a lower level
    • Student s = new Doctoral();
    • Person p = s;
    • Doctoral d = (Doctoral) s;
The Class Object

- Every class in Java inherits a base class "Object"
  - You don’t have to write "extends" explicitly
  - Every class in Java is an object

- Class Object has several methods that can be overridden
  - The most important one is public boolean equals (Object obj)
  - This method compares if two Object variables are the same
  - We’ve used the overridden one in String class
equals() Method

- Read Chapter 8.2 for more details

```java
public class Student {
    private String name;
    private int studentNumber;

    public boolean sameName(Student otherStudent) {
        return this.name.equals(otherStudent.name);
    }

    public boolean equals(Object otherObject) {
        boolean isEqual = false;
        if (otherObject instanceof Student) {
            Student otherStudent = (Student) otherObject;
            isEqual = this.sameName(otherStudent)
                    && (this.studentNumber == otherStudent.studentNumber);
        }
        return isEqual;
    }
}
```
Two More Keywords: abstract & final

• If a method is **abstract**, subclasses **must** override it.
• If a method is **final**, subclasses **can not** override it.
• There are more details:
  – A class with at least one abstract method is called an abstract class. You can not create objects in this class. It can only be used as a base class.
  – A class can be declared as final. Then you can not inherit this class.
  – A variable can also be declared as final. You know that it also means the variable is not changeable.
Two More Keywords: abstract & final

```java
public abstract class AbstractClass {
    public abstract void m();
    // An abstract method cannot have method body
    // Also, you must declare the class as abstract
}

class ClassWithFinal {
    public final void m() {
        System.out.println("Can't override!");
    }
    public void n() {
        System.out.println("Can override");
    }
}

class FinalClass {
    public void m() {
        System.out.println("Can't inheritance!");
    }
    // A final class cannot be inherited
}

class Class1 extends AbstractClass {
    public void m() {
        System.out.println("Must override!");
    }
}

class Class2 extends ClassWithFinal {
    // Can not override method m();
    public void n() {
        System.out.println("Override n()!");
    }
}
```
Take-Home Message

• A subclass object can be assigned to a superclass type variable
• When you invoke a method, what is called depends on what **object** it is invoked from
• Polymorphism means you can write methods for superclass only, and the behavior depends on detailed subclass implementation
Announcement

• On next lecture, we will start reviewing the important contents, with sample questions from example exams
  – Be sure to attend!

• Read Lab 6 and Lab 7, and review sheets
  – No submission required for new labs. Solutions are given

• Send me an email if you want to attend our tic-tac-toe AI tournament
  – You need (eventually) a version with CPU moves first