Inheritance and Polymorphism

Recitation 04/10/2009

CS 180
Department of Computer Science,
Purdue University

Reminders

- Project 7 is due next Wednesday.

Introduction

- Recall:
  - *Inheritance* allows data and methods to be inherited by a subclass.
  - *Polymorphism* simplifies code by automatically using the appropriate method for a given object/type.

Inheritance

- A *subclass* is a *specialization* of the class it inherits from.
- The common behavior is *implemented once* in the superclass and automatically inherited by the subclasses.

Consider the example of a roster of students with different grading for graduates and undergrads.

Overriding

- A derived class may **override** an inherited method
- Simply define a method with the same method header.
  - An overridden method cannot change the return type!
  - Can be prevented with **`final`**
- Note Difference: A subclass may **overload** *any* method by using the same name, but *different* signature.

The Java Inheritance Hierarchy

- ***private*: data members and methods are accessible only to instances of the class.**
- ***protected*: visible to instances of the class and individual descendant instances.**
- ***public*: accessible to all**

Inheritance and Constructors

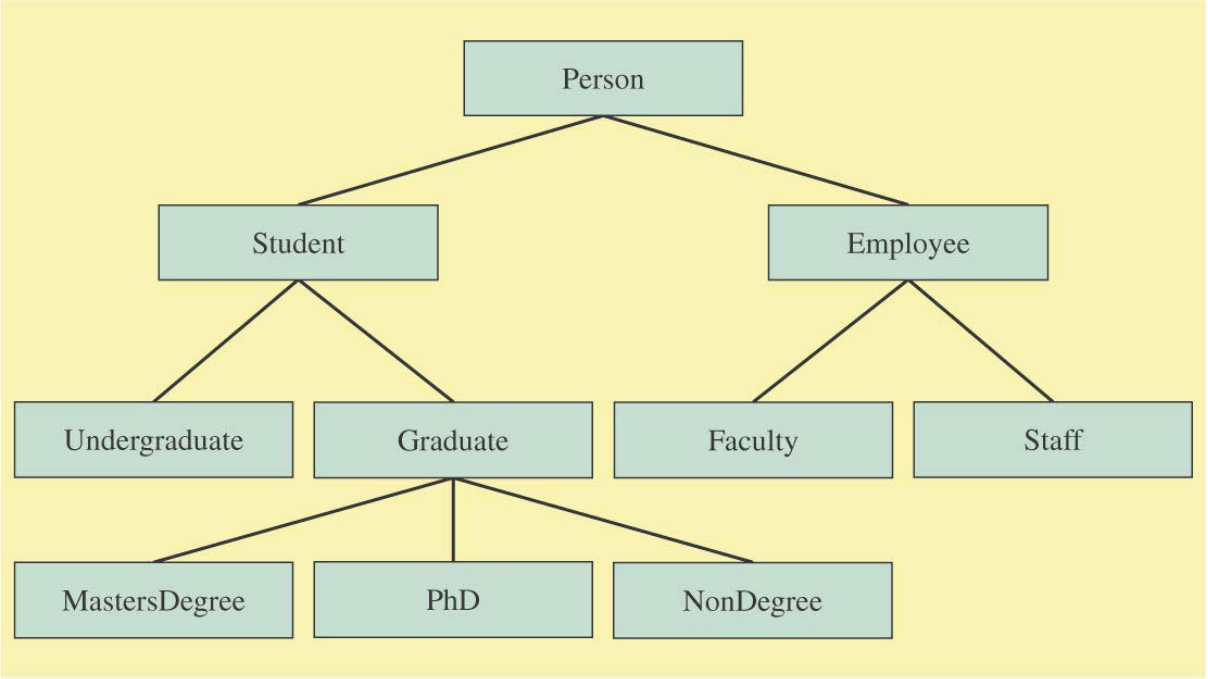
- Constructors are not inherited.
- Default constructors are made iff no others found
- A call to the superclass constructor, **`super()`**, must be the first line of a constructor
  - It is automatically added if not present
- You may optionally call some other constructor of the base class,
  - e.g.: `super( "some string" );`
- **`super` also has other meanings**

Super keyword

- It can also be used to call a method of the parent class:
  - e.g.: super.methodA();
- This can be useful to:
  - Call an overridden method
  - Access data members of the parent

Example 1

- Consider a college record-keeping system with records about students, faculty and staff.
- All these specific groups are sub-classes of the class *Person.*



Example 1, cont.

```
class Person
{
   private String name;
   public Person()
   {
      name = “no name”;
   }
   public Person(String _name)
   {
      name = _name;
   }
   public void setName(String) { … }
   public String getName() { … }

   public void output()
   {
      System.out.println(name);
   }

}
```

Example 1, cont.

```
public class Student extends Person{
      private int studentNumber;
      public Student(String _name, int _num){
             super(_name);
             studentNumber = _num;
      }
      // override the method in Person class
       public void output()
       {
             System.out.println(name);
             System.out.println(studentNumber);
       }
       //more methods not in Person class
       …
}
```

- `class Student` is a *subclass* of `class Person` and `class Person` is called the *superclass*.

Polymorphism

- Polymorphism allows a single variable to refer to objects from different subclasses in the same inheritance hierarchy
- For example, if Cat and Dog are subclasses of Pet, then the following statements are valid:

```
Pet myPet;

myPet = new Dog();
. . .
myPet = new Cat();
```

Dynamic Binding

- At compile time, the version of a polymorphic method to be executed is unknown.
  - Determined at run-time by the class of the object
- This is called dynamic (late) binding

Object Type

- Consider the inheritance hierarchy:

  Object ← A ← B

- An instance of B is also an instance of A and Object.
  - Instances of class B can be used where objects of class A can be used.
  - The relationship is one way (thus the arrows)

- A reference of type A can hold an object of type B. It can only be treated like an instance of A unless cast.

The instanceof Operator

- The instanceof operator can help us discover the class of an object at runtime.
- The following code counts the number of undergraduate students.

```
int undergradCount = 0;
for (int i = 0; i < numberOfStudents; i++) {
if ( roster[i] instanceof UndergraduateStudent ) {
        undergradCount++;
}
}
```

Definition: Abstract Class

- An *abstract class* is a class
  - defined with the modifier **abstract** OR
  - that contains an abstract method OR
  - that does not provide an implementation of an inherited abstract method

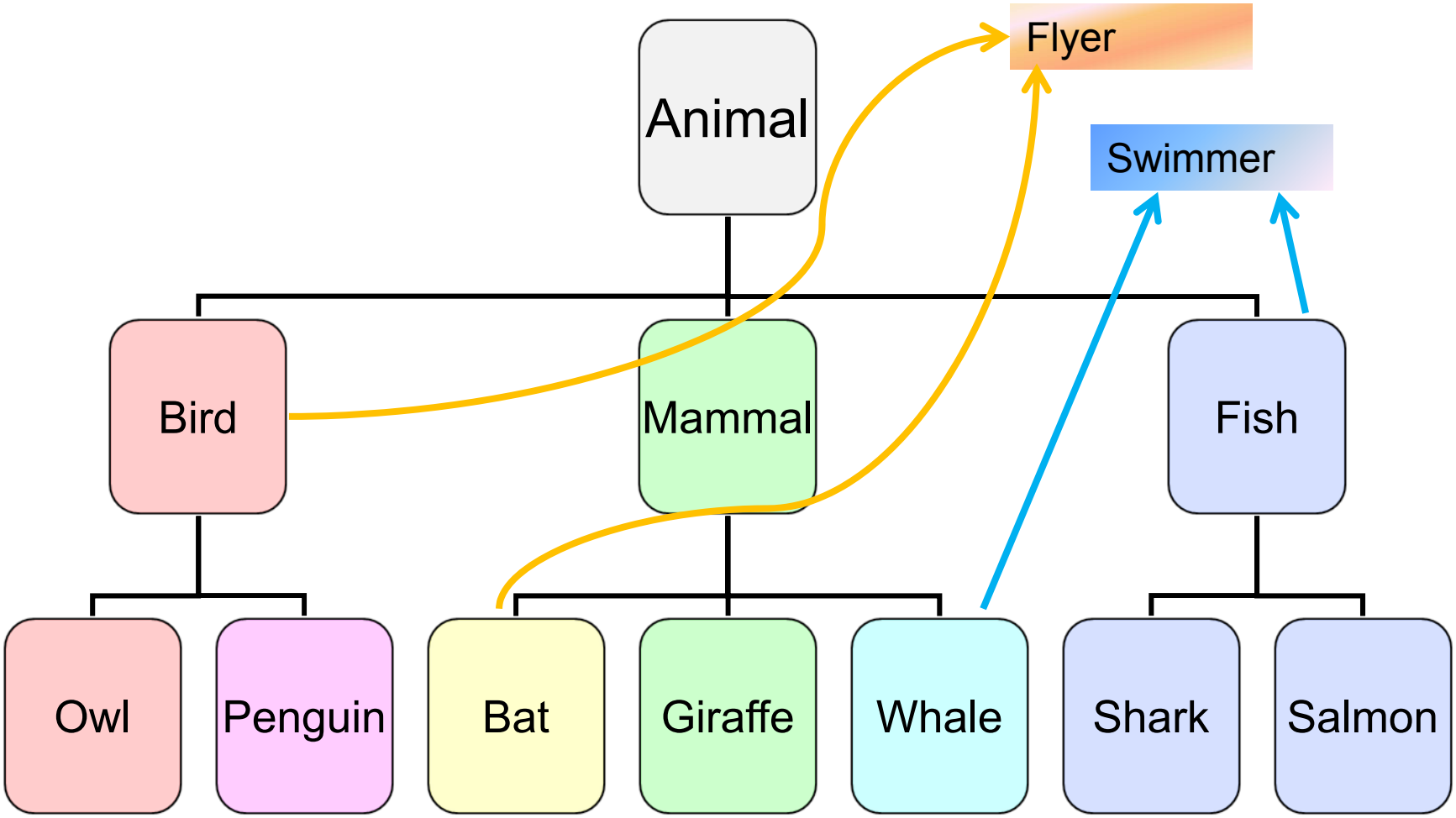
- An *abstract method* is a method with the keyword **`abstract`**, and it ends with a semicolon instead of a method body.
  - Private methods and static methods may not be declared **abstract**.

- No instances can be created from an abstract class.

Inheritance versus Interface

- Interfaces are like a contract to share or guarantee behavior
- Inheritance is used to share common code when one class is a specialized form of another.

Example 2


Flyer
Animal
Swimmer
Bird
Mammal
Fish
Owl
Penguin
Bat
Giraffe
Whale
Shark
Salmon

Example 2

Our animals:

public class Owl {…}

public class Penguin {…}

public class Bat {…}

public class Giraffe {…}

public class Whale {…}

public class Shark {…}

public class Salmon {…}

Example 2

Create appropriate superclasses:

```
public class Animal {…}
public class Bird {…}
public class Mammal {…}
public class Fish {…}
public class Owl {…}
public class Penguin {…}
public class Bat {…}
public class Giraffe {…}
public class Whale {…}
public class Shark {…}
public class Salmon {…}
```

Example 2

Connect the hierarchy:

```
public class Animal {…}
public class Bird extends Animal {…}
public class Mammal extends Animal {…}
public class Fish extends Animal {…}
public class Owl extends Bird {…}
public class Penguin extends Bird {…}
public class Bat extends Mammal {…}
public class Giraffe extends Mammal {…}
public class Whale extends Mammal {…}
public class Shark extends Fish {…}
public class Salmon extends Fish {…}
```

Example 2

Add in appropriate interfaces at the highest levels:

```
public class Animal {…}
public class Bird extends Animal {…}
public class Mammal extends Animal {…}
public class Fish extends Animal implements Swimmer {…}
public class Owl extends Bird implements Flyer {…}
public class Penguin extends Bird {…}
public class Bat extends Mammal implements Flyer {…}
public class Giraffe extends Mammal {…}
public class Whale extends Mammal implements Swimmer {…}
public class Shark extends Fish {…}
public class Salmon extends Fish {…}
```

Example 2

Make appropriate classes abstract:

```
abstract public class Animal {…}
abstract public class Bird extends Animal {…}
abstract public class Mammal extends Animal {…}
abstract public class Fish extends Animal implements Swimmer {…}
public class Owl extends Bird implements Flyer {…}
public class Penguin extends Bird {…}
public class Bat extends Mammal implements Flyer {…}
public class Giraffe extends Mammal {…}
public class Whale extends Mammal implements Swimmer {…}
public class Shark extends Fish {…}
public class Salmon extends Fish {…}
```

Example 2 - Quiz

- abstract public class Animal {…}
  abstract public class Bird extends Animal {…}
  abstract public class Mammal extends Animal {…}
  abstract public class Fish extends Animal implements Swimmer {…}
  public class Owl extends Bird implements Flyer {…}
  public class Penguin extends Bird {…}
  public class Bat extends Mammal implements Flyer {…}
  public class Giraffe extends Mammal {…}
  public class Whale extends Mammal implements Swimmer {…}
  public class Shark extends Fish {…}
  public class Salmon extends Fish {…}

Which are valid instantiations?

```
Animal a = new Animal();
Animal b = new Fish();
Animal c = new Flyer();
Mammal d = new Bat();
Fish e = new Swimmer();
Swimmer f = new Shark();
Flyer g = new Owl();
Swimmer h = new Whale();
Swimmer i = new Fish();
```

Timer and TimerTask

- Timer
  - scheduleAtFixedRate(TimerTask task, long delay, long period)
    - task: task to be scheduled.
    - delay: delay in **milliseconds** before task is to be executed for the first time
    - perid: time in **milliseconds** between successive task executions

Timer and TimerTask

- TimerTask
  - Inherit this class
  - Override the run() method
    - Put everything you want to be executed into this method.

Timer and TimerTask Example

```
public class TimerTest {
        private Timer timer;
        private TimerTask task;

        public TimerTest(TimerTask task) {
                this.timer = new Timer();
                this.task = task;
        }
        public void start(int delay, int internal) {
                timer.scheduleAtFixedRate(task, delay * 1000,
        internal * 1000);
        }
        public static void main(String[] args) {
                TimerTask task1 = new MyTask(“ Job 1");
                TimerTask task2 = new MyTask("Job 2");
                TimerTest tt1 = new TimerTest(task1);
                tt1.start(1,3);
                TimerTest tt2 = new TimerTest(task2);
                tt2.start(1,1);
        }
}
```

delay * 1000 (ms) = delay (sec)

Polymorphism

Inherit the TimerTask

```
class MyTask extends TimerTask {
        private String jobName;

        //override
        public void run() {
                System.out.println(jobName);
        }

        public MyTask(String jobName) {
        }               this.jobName = jobName;
}
```

Override the abstract method in the class TimerTask.

PURDUE UNIVERSITY