Inheritance and Polymorphism

Concepts

Game plan

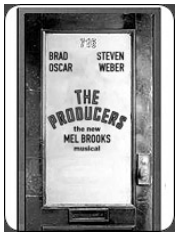
- Inheritance, Subclassing, Polymorphism
  - Today: Basic concepts
  - Tomorrow: How it's done in Java

Before we begin

- Questions from last time?
- Project to be released this week

Something to think about for next class!

- Any theatre fans in the house?



When we last left our wannabe producer

- Developed a payroll application that will calculate total weekly salary paid to performers
- New requirements:
  - Add musicians to the mix
  - Musicians get reimbursed for instrument rental as well as getting a base pay per performance
  - The pay rate for musicians is different than that for actors.
- Wish to achieve with maximum reuse of code

Potential Problems

- Musicians
  - Have data items that other performers do not
  - Determine their pay differently than other performers.
- Yet…
  - The producer still needs to maintain pay info about musician AND other performers

Consider this code in payroll

```
/**
 *  Caculates the weekly pay for all of the performers
 *
 * @returns the total weekly pay
 */
public double calculateTotalPay()
{
   double sum = 0.0;
   for (int i=0; i < nPerf; i++)
        sum += performers[i].calculatePay();
   return sum;
}
```

Adding musicians does not change this algorithm!

What would be nice

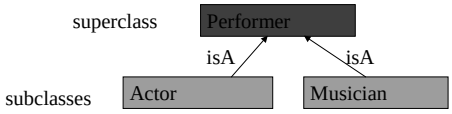
- Keep the same code
- Let each object in the array compute it's pay based on the kind of object it is.
- We can achieve this using inheritance / subclassing.

Subclassing

- Defining a class as a specialization or extension of another class.
- The more general class is called the superclass.
- The more specific class is called the subclass.
- Implies an IS-A relationship.

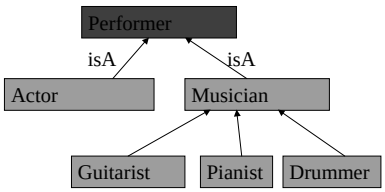
Subclassing in our example

- Define 2 classes "Actor" and "Musician"
- Both Actors and Musicians are specializations of Performer



Class Heirarchies

- Class heirarchies can be as deep as needed:



Subclassing and Inheritance

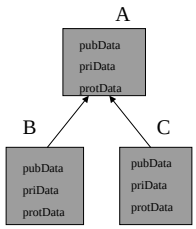
- When you define a class as a subclass:
  - The subclass **inherits** all of the data members and methods of the superclass.
  - In addition, a subclass can have data/methods that are it's own.
  - Inheritance is transitive:
    - I.e. If B is a subclass of A and C is a subclass of B, then C inherits the data/methods from both B and A.

Data/method access

- Protected data/methods
  - Recall that data/methods can be classified as:
    - Public – Objects of all classes can access the data / method
    - Private – Data / methods accessible only by the objects that belong to the same class.
  - New access control category
    - Protected – data/methods accessible by objects of the same class and all subclasses.

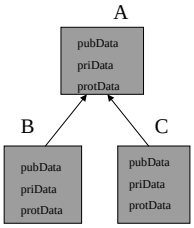
Data / Method access

- An object of class B or C can access:
  - A.pubData
  - A.protData
- It cannot access
  - A.priData



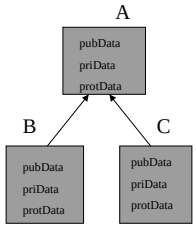
Data / Method access

- An object of class A can access:
  - B.pubData
- It cannot access
  - B.priData
  - B.protData



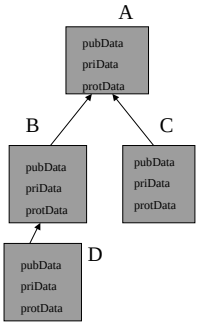
Data / Method access

- An object of class C can access:
  - B.pubData
- It cannot access
  - B.priData
  - B.protData



Data / Method access

- An object of class D can access:
  - B.pubData
  - B.protData
  - A.pubData
  - A.protData
- It cannot access
  - B.priData
  - A.priData



A break for questions…

Method Overriding

- A subclass can define it's own version of any non-final method of it's superclass.
  - This is called method overriding
  - In our example,
    - Both Actor and Musician can override the definition of calculatePay()

Polymorphism

- A variable of a superclass can reference an object of any one of it's subclasses.
- The variable remembers what subclass of object is referenced so that the correct methods of the subclass are called.

Polymorphism in Action

- Example

```
Performer A = new Actor("foo");
Performer M = new Musician ("bar");
Performer P = new Performer ("fred");

// calls Actor's calculatePay
float Apay = A.calculatePay();

// calls Musician's calculatePay
float Mpay = M.calculatePay();

// calls Performer's calculatePay
Float Ppay = P.calculatePay();
```

Polymorphism

```
/**
 *  Caculates the weekly pay for all of the performers
 *
 * @returns the total weekly pay
 */
public double calculateTotalPay()
{
   double sum = 0.0;
   for (int i=0; i < nPerf; i++)
        sum += performers[i].calculatePay();
   return sum;
}
```

Polymorphism is performed at RUNTIME

Polymorphism

- Must use methods defined in superclass

```
Performer A = new Actor("foo");
Performer M = new Musician ("bar");
Performer P = new Performer ("fred");

// Let's say that Musician has an
// addInstrument() method which
// Performer does not

M.addInstrument();   // would cause a compile
                     // error
```

Polymorphism

- Must use methods defined in superclass

```
Performer A = new Actor("foo");
Musician M = new Musician ("bar");
Performer P = new Performer ("fred");

// Let's say that Musician has an
// addInstrument() method which
// Performer does not

M.addInstrument();   // This would be okay
```

Abstract method

- Suppose Performer had no implementation of calculatePay()
  - Then, it is required that all subclasses define a caculatePay() method
  - Performer::calculatePay() is considered an abstract method
    - I.e. the method is not defined by the Performer class and must be implemented by all subclasses of Performer

Abstract method

- Private superclass methods cannot be declared as abstract.
  - Anyone know why?
- Static superclass methods cannot be declared as abstract.

Abstract class

- Any class with abstract methods are considered abstract classes.
  - Generalized placeholder/definition for more specific objects
  - Specialized subclasses fill in the details for the general abstract class methods.
  - Cannot instantiate an object of an abstract class directly! Must always instantiate an object of a subclass
    - `Performer P = new Performer ("foo")` would not be allowed if Performer was abstract.

Abstract class

- Using abstract classes
  - A superclass should be declared abstract if we know that all types of that superclass can/will be defined by some subclass.
  - If it's acceptable for a given superclass not to be further specialized, then the superclass should not be declared as abstract.

Forms of inheritance

- Specialization – The subclass is a specialized form of the superclass
- Specification – The subclass defines new behavior to existing functionality in the superclass
- Extension – The subclass defines new functionality to the parent class
- Limitation – The subclass restricts the use of functionality of the superclass.
- Generalization – The subclass overrides methods in the superclass

To summarize

- Subclasses
- Inheritance
- Protected data/methods
- Method override
- Polymorphism
- Abstract classes
- Forms of inheritance

Next time…

- How we do all this in Java
- Questions?