Inheritance and Polymorphism

Notes

- Lab 4:
  - Career fair overlaps sessions 2 (012) and 3 (013), so they will be short handed
  - Anyone may attend any lab session this week
  - Universal lab deadline: 7pm on Saturday (this week only)
- Project 1 is due in a little more than a week

Relationships Between Classes

So far: we have looked at class aggregation

- Class A ***has-a*** instance of class B
- This allows A to make use of what has already been done in class B

Sharing Data Between Classes

Aggregation (*Has-A*) is one way to share data between classes

- Can only use public parts of the class
- Is this a limitation or an advantage?

Sharing Data Between Classes

Another way to share data is inheritance

Sharing Data Between Classes

Another way to share data is inheritance

- New class keyword: **extends**
  - Defines the inheritance relationship
  - UML: Arrow with open head 
- Class A extends class B:
  - Inherits everything from class B **AND** allows us to add to it

Sharing Data Between Classes

Another way to share data is inheritance

- New method/data visibility keyword: **protected**
  - This data item/method is visible both inside the class and to classes that extend this class
  - Also visible to other classes in the same package
  - # in UML (as opposed to + or -)

Example: Online Ordering for Amazon

Consider the following product types and create a hierarchy:

- Product
- Downloadable software
- Software with media
- Book

What is the UML?

Where Do These Properties Belong in the Hierarchy?

- Price
- URL for downloading software
- Name of item
- Author
- ISBN
- Delivery method
- Shipping costs

Terminology

- Subclass can be called:
  - Child class
- Superclass can also be called:
  - Parent class
  - Base class

Terminology

- Subclasses get direct access to all of the public and protected data and methods from superclass
  - May have to implement methods again if we need more specific behavior

Consider equals()

Have you noticed that equals() works in a class, even if you didn't put it there?

```
public class Equalizer
{
    private int data;

    public Equalizer(int data)
    {
        this.data = data;
    }

}
```

Consider equals()

How does the program find an equals method in the Equalizer class?

Consider equals()

How does the program find an equals method in the Equalizer class?

- It is defined in the Object class:

  public boolean equals(Object o)

Consider equals()

Exercise:

- Demonstrate that this method is not working properly
  - Why?
- Fix it and demonstrate it
- Draw UML of Equalizer, both before and after

How about toString()

- What does toString() do? Or hashCode()?

Modeling Relationships

- The relationship represented by aggregation (with the diamond in UML) is “has-a”
- The relationship represented by inheritance (with the open headed arrow in UML) is “is-a”
  - More specialized classes are lower in the hierarchy

Modeling Relationships

Exercises:

- Example: Shape, Circle, Square, Ellipse, Rectangle, Quadrilateral
- Example: Student, Name, Address, City, State, Country, First Name, Last Name, Middle Name

Inheritance Can be Bad if Done Incorrectly

- Inheritance is widely used in Java
  - And all OOP languages
- Works fabulously in GUI components, and collections
- Inheritance breaks encapsulation if we use the *protected* keyword
- Aggregation/composition do not break encapsulation

Private or Protected Data?

Choosing private or protected can be a tough call

- If everything is private:
  - Inheritance doesn't provide the subclass itself with anything it can't get through composition
  - However: the "user" of a class does get to see a consistent interface between the super and child classes

Private or Protected Data?

Choosing private or protected can be a tough call

- If everything is protected
  - Classes become closely coupled
    - Changes in one are likely to causes changes in the other
  - Bad for maintenance ($$$)
  - These effects can be mitigated somewhat through the use of multiple packages

Private or Protected Data?

Choosing private or protected can be a tough call

- My take: stick with private

Administrivia

- Lab 4
- Project 1

Specification to Implementation

- There is a direct translation from UML to the skeleton of the class
  - Class/instance variables
  - Method prototypes
- Then, look to our specification document and any method-level documentation that we provide for a discussion about **what** the methods do

Specification to Implementation

- For the projects, and even the labs: get used to shifting your focus between different levels of the problem
- In general, when you are working on one class, you have to put the rest of the implementation out of your head
  - Worry about what ***this*** class is supposed to provide as an interface and how this should be implemented

A/B example

Implementing Inheritance: Instance Methods and Variables

- super.methodName() to explicitly call public or protected methods in the superclass
  - For a given class, remember that there is exactly one superclass because Java does not allow multiple inheritance

- super.instanceVariableName to refer to public or protected instance variables from the superclass

Implementing Inheritance: Constructor

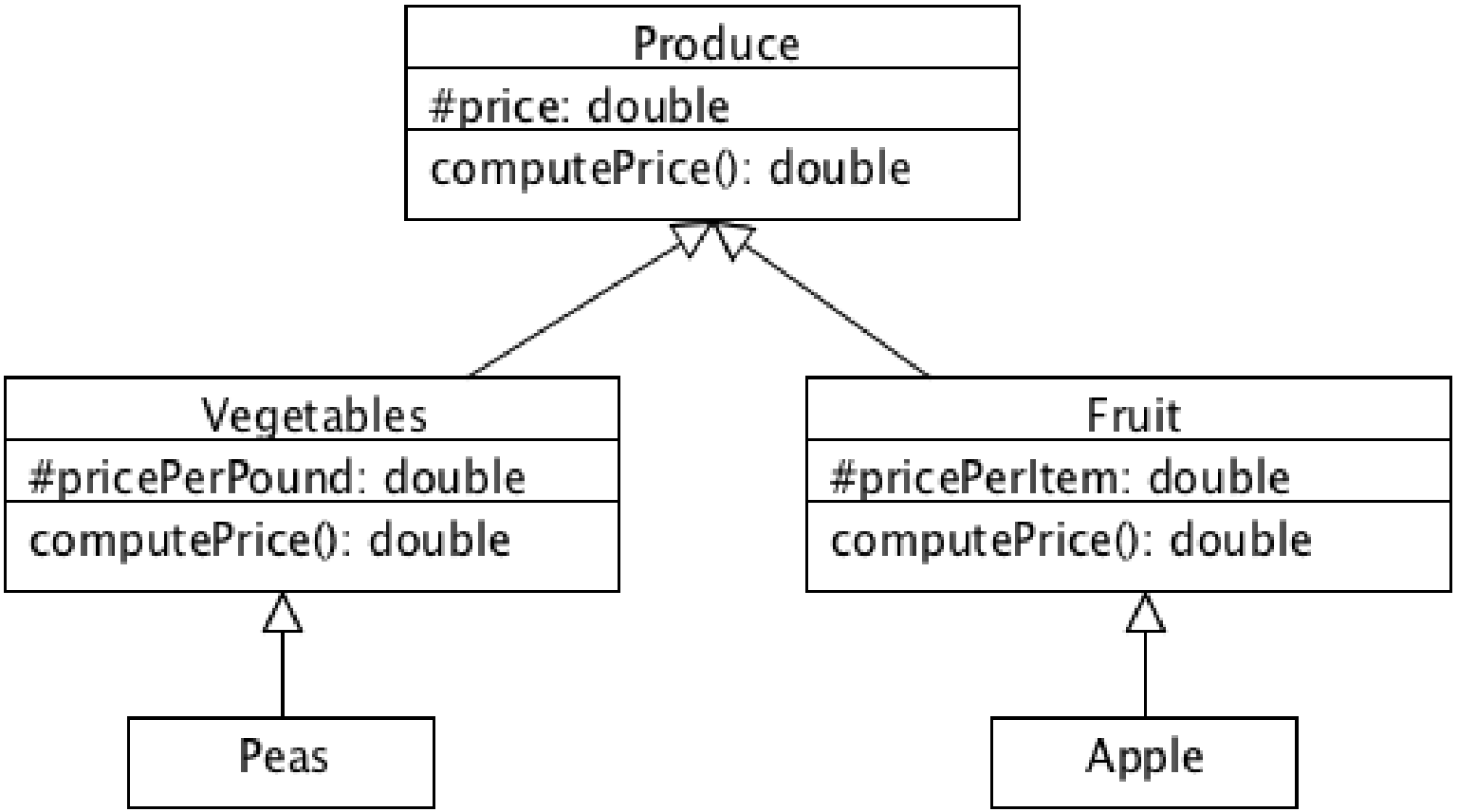
- Constructors are not inherited
- But: can use super() to call the superclass constructor
  - If used, it must be first statement in subclass constructors
  - Can call any of the constructors associated with the superclass
- Most constructors call other constructors…

Compiler

If you don’t use super(), compiler adds implicitly for you

- Why?

Inheritance example



Polymorphism

A variable of a super type can really be an instantiation of the sub type

```
Produce pr = new Apple();
```

This is called “Upcasting”

```
// We get Apple.computePrice()
//    from this call.
pr.computePrice();
```

Polymorphism: Methods

- Calling methods: Java Virtual Machine will select method based on **object type at run time** (not the type of the reference)
  - Search order: constructed class if available, then parent, then grandparent, etc.

```
Produce pr = new Apple();
pr.toString();              // Calls Apple.toString()
```

- Exercise: show example with Produce hierarchy

Polymorphism: Variables

- References to instance/class variables **are decided at compile time**
- When an instance/class variable is accessed, the **compiler** starts looking for the variable starting with the **class of the reference t**ype
  - If not found, then the parent class is checked
  - If not found, then the grandparent class is checked…

A/B example revisit

Administriva

- Project 1 due Wednesday
  - Code reviews: get them done as early as possible
- Lab 5 coming soon
  - Those who attend lecture will be given priority during Friday office hours

Overriding Methods

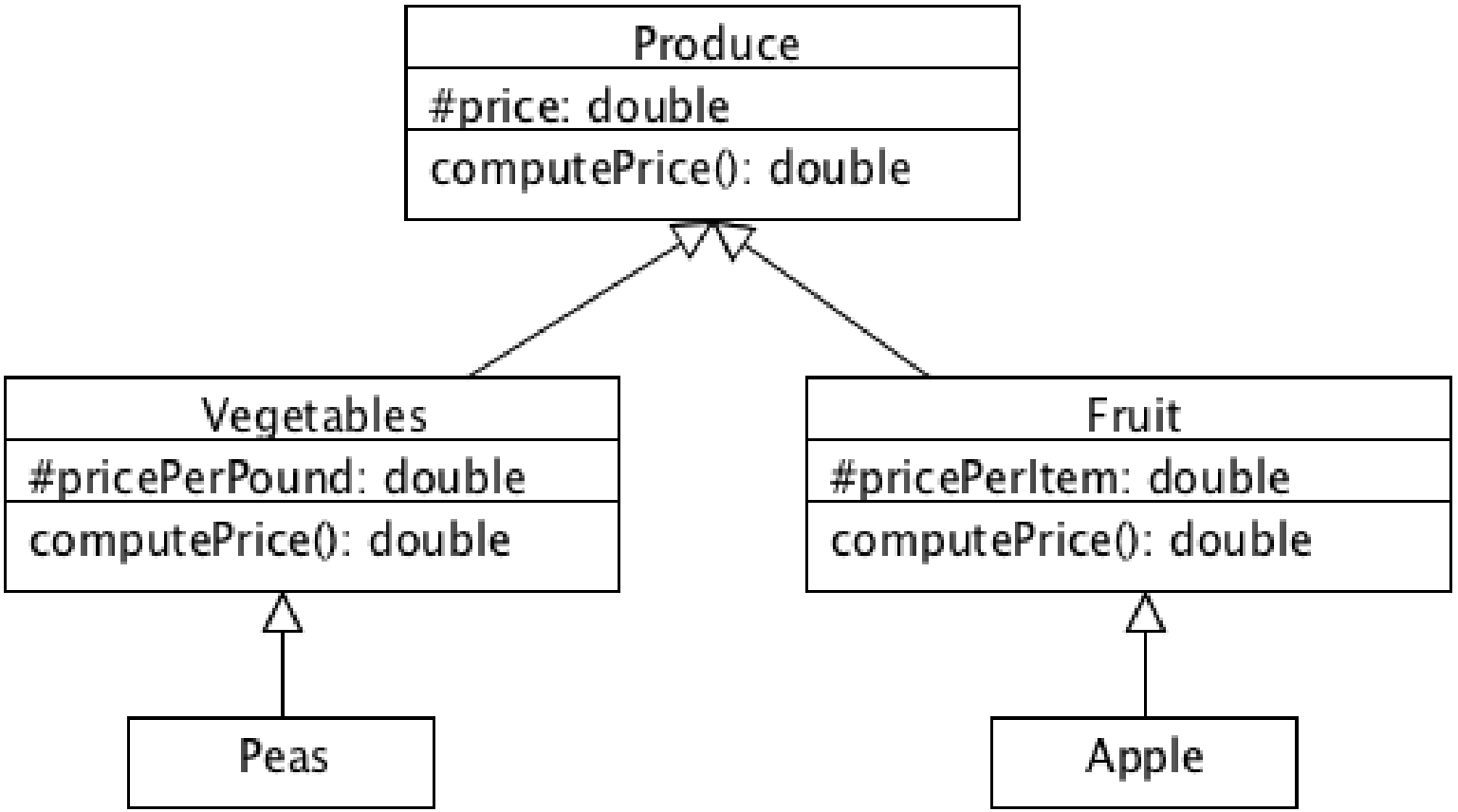
When a subclass implements a method that is identical to one in the superclass, it **overrides** the superclass method

- Superclass method must be public or protected
- Same name
- Same parameters
- Return values: new method must return a subclass of the original method's return type
- Static methods cannot be overridden

A/B example

- Dynamic binding

Inheritance Example



Recall: Upcasting

A variable of a super type can really be an instantiation of the sub type

```
Produce pr = new Apple();
```

This is called “Upcasting”

```
// We get Apple.computePrice()
//    from this call.
pr.computePrice();
```

Upcasting

Upcasting works by default because every Apple is guaranteed to do everything that a Produce object does

- This is true for any inheritance relationship: the child class is guaranteed to do everything that the parent class provides

Down-Casting

The other way can be made to work, but we need to be explicit:

```
Apple a = pr;   // Compiler disallows

Apple a = (Apple) pr;   // Allowed
```

- Forces java to treat the object as if it is the subclass
- Lets you access subclass methods
- If you improperly cast an object, you will receive Exceptions when you try to access the object

Casting and instanceof

instanceof will tell you whether an instance is a member of a class:

```
if (pr instanceof Apple) {
  Apple a = (Apple) pr;
  // Use a….

}
```

ArrayList example

Exercise: make an ArrayList of Produce and Fruit

- What can go in each?
- Printing out the lists

Immutable Classes and Inheritance

- It is possible to make a class so that it cannot be extended

```
public final class ClassName
```

- This must be done with all immutable classes
  - Why?
- Again, if unsure, make class final
  - Can always remove it later
  - Once you let people extend a class, you can't make changes or risk breaking their code