Abstract Classes and Interfaces

Slides derived from the work of Dr. Amy McGovern and Dr. Deborah Trytten

Classes as Contracts

Recall: the public interface of a class is a **promise** to users of the class

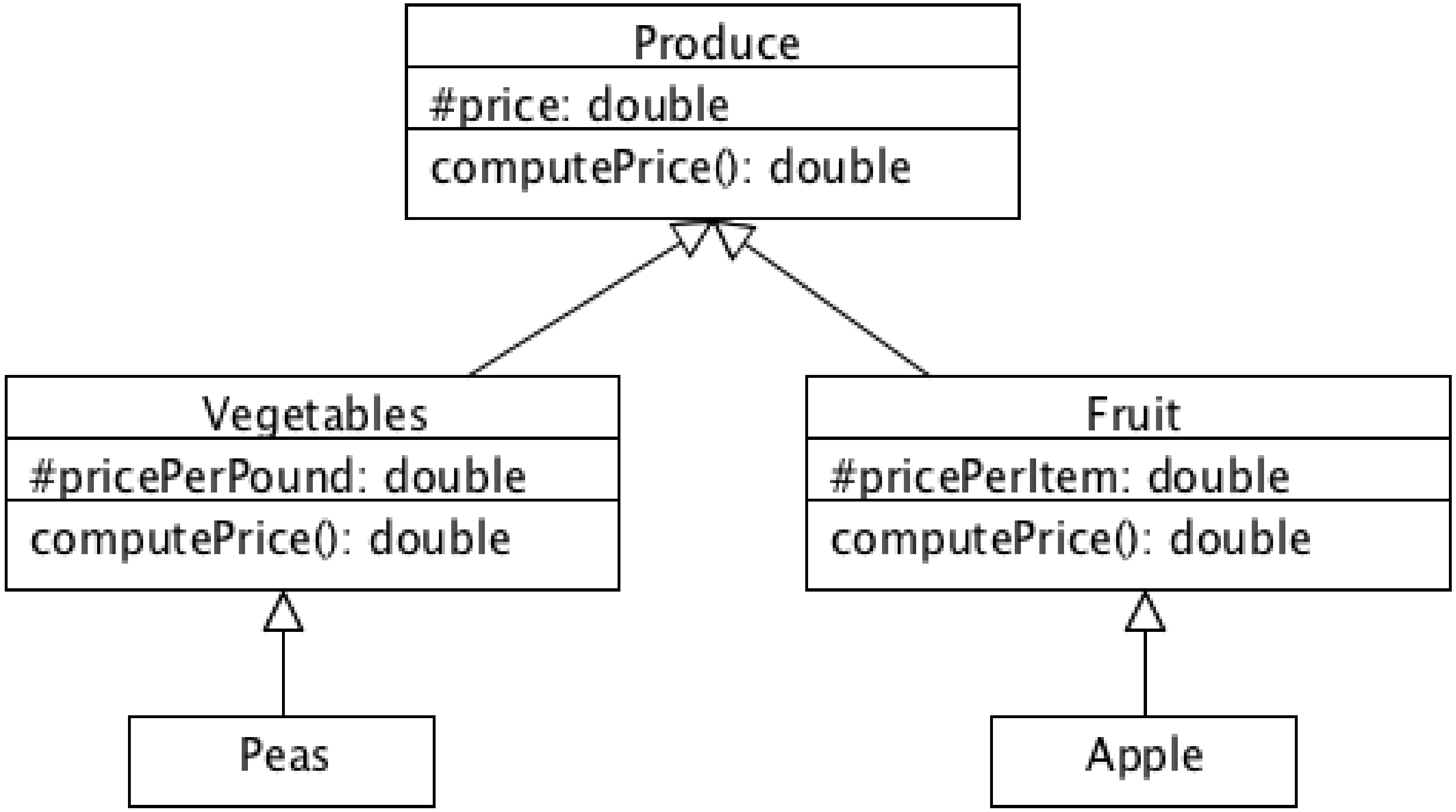
- Guarantee that certain methods will be available and that they have specific prototypes
- Guarantee that certain instance variables can be accessed (though we should generally not be using public instance variables)

Classes as Contracts

These promises extend into the class hierarchy

- The superclass makes certain promises about available methods and instance variables
- These promises must be kept by all child classes
  - But: the implementation of these promises can be overridden

Inheritance example



Classes as Contracts

Sometimes, a superclass needs to make a promise, but cannot provide an implementation

- Declare methods as **abstract**
- Declare class as abstract

Example

```
public abstract class Produce {
     protected double price;

     protected Produce(){
          price = 5;
     }

     public abstract double computePrice(double number);

     public String toString(){
          return "Produce: $" + price;
     }
}
```

Note: no method body for abstract method!

Abstract Classes

- Cannot be instantiated!

  No: `Produce p = new Produce();`

Abstract Classes

- Cannot be instantiated!

  No: `Produce p = new Produce();`

- … but can be used as types:

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

Very powerful: we can write methods that know how to interface with abstract types

Abstract Classes

Can also create arrays:

```
ArrayList<Produce> L =
    new ArrayList<Produce>();
L.add(new Apple(0.5));
L.add(new Orange(2.5));
```

Abstract Classes

A class that extends an abstract class must:

- Implement all abstract methods
- or also be abstract

Properties of Abstract Classes

Must provide constructors

- These constructors are protected or private (remember, we cannot create an instance of an abstract object)
- Child classes can reference these constructors with super()

Abstract Classes: Best Practices

In the abstract superclass:

- Provide as many method implementations as possible
- These implementations may call abstract methods
  - The methods will ultimately be implemented by the child classes (or grandchildren, etc).
  - It is these **concrete classes** that ensure all methods are implemented

Java API Examples

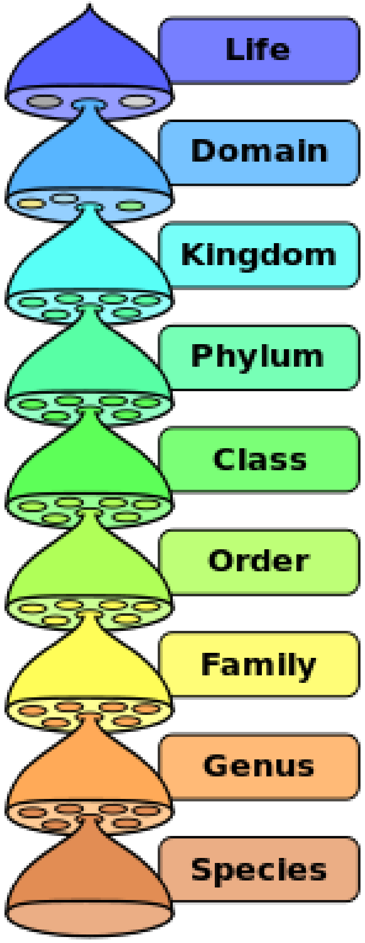
- ArrayList

- What do the class hierarchies look like?
- What aspects of the superclasses are abstract?

Example

- Kingdom: Animalia
- Phylum: Chordata
- Class: Mammalia
- Order: Carnivora
- Family: Felidae
- Genus: Panthera
  - Species: leo
  - Species: pardus
  - Species: tigris
  - Species: onca

- What is UML?
- Make an ArrayList of large cats at a zoo (give them names)



Pictures and classification from wikipedia

http://en.wikipedia.org/wiki/Kingdom_(biology)

Multiple Inheritance

Example: we might want to make a superclass of Cloneable objects

- A clone of an object is equal in content but distinct in memory footprint
- Clone of not just the object’s memory, but of all of its component objects (and their components, etc.)
- Cloneable requires the implementation of a clone() method that produces the copy

But there is a problem…

But there is a problem…

Java restriction:

- If a class inherits from the Cloneable class, then it cannot inherit from any other class
- Not allowing multiple inheritance solves some serious language design problems, but it is limiting

Java’s Workaround: The Interface

- An interface defines no implementation – only a set of abstract methods
- All checks can be made at compile time, so the runtime cost is low

Interfaces: Syntax

```
public interface InterfaceName{
    public abstract int methodName()
    :
}

public class ClassName implements IntefaceName{
  public int methodName(){
    :  // Concrete implementation
  }
  :
}
```

How Does this Fix Our Cloneable Problem?

How Does this Fix Our Cloneable Problem?

We can extend a different class and still make the same guarantees as those provided by Cloneable:

```
public class Apple extends Fruit implements Cloneable
{
        :
        :
}
```

A class can implement any number of interfaces

Comparable<T>

This interface requires only one method:

```
   int compareTo(T object)
```

- T is a placeholder for any class name
- Returns
  - negative number if **this** < object
  - zero if they are equal
  - positive number if **this** > object
- Defines a *Natural Ordering* of objects of class T
  - Basis for using generic sorting methods

Example

- Person class: first name, last name, ID number
- Implement comparable to sort by last name then by first name

- Show use with Collections.sort()

Abstract Classes & Interfaces

Similarities:

- Have missing methods that must be implemented by the child/implementing classes
- Cannot be instantiated
- Can be used as reference types

Abstract Classes & Interfaces

Differences:

- Interfaces have no constructors
- Interfaces can only define public static and final variables
- A class can implement multiple interfaces
- Abstract classes can implement some methods

Abstract Classes & Interfaces

Best practices:

- Use interfaces when you can
- Use inheritance when you are adding new functionality to a class that already implements some functionality

- Inheritance: “is-a” relationship
- Interface: can be “is-a”, “has-a” or “does-a”

Comparable Limitation

Comparable Limitation

- Our class can only implement on compareTo() method
- But – depending on the context, we may want to be able to sort our data in different ways

Comparator<T>

- Our solution is to implement a separate class that extends Comparator<T>
- Must implement:

```
int compare(T o1, T o2)
```

- Allows us to define many different ways to sort

Example with Person …

Abstract Classes and Interfaces

Act as a contract to the outside world

- A method can accept an object that implements interface Foo
- This method can then assume that any object provides certain methods
- Checks are made at compile time
  - Much easier to debug than at run time!