CMSC 206

Inheritance, Abstract Classes, and Interfaces

Class Reuse

- We have seen how classes (and their code) can be reused with composition.
  - An object has another object as one (or more) of its instance variables.

- Composition models the “has a” relationship.
  - A Person has a String (name)
  - A Car has an Engine
  - A Book has an array of Pages

Object Relationships

- An object can be a specialized version of another object.
  - A Car is a Vehicle
  - A Triangle is a Shape
  - A Doctor is a Person
  - A Student is a Person

  This kind of relationship is known as the "is a type of" relationship.

- In OOP, this relationship is modeled with the programming technique known as **inheritance**.

- Inheritance creates new classes by adding code to an existing class. The existing class is reused without modification.

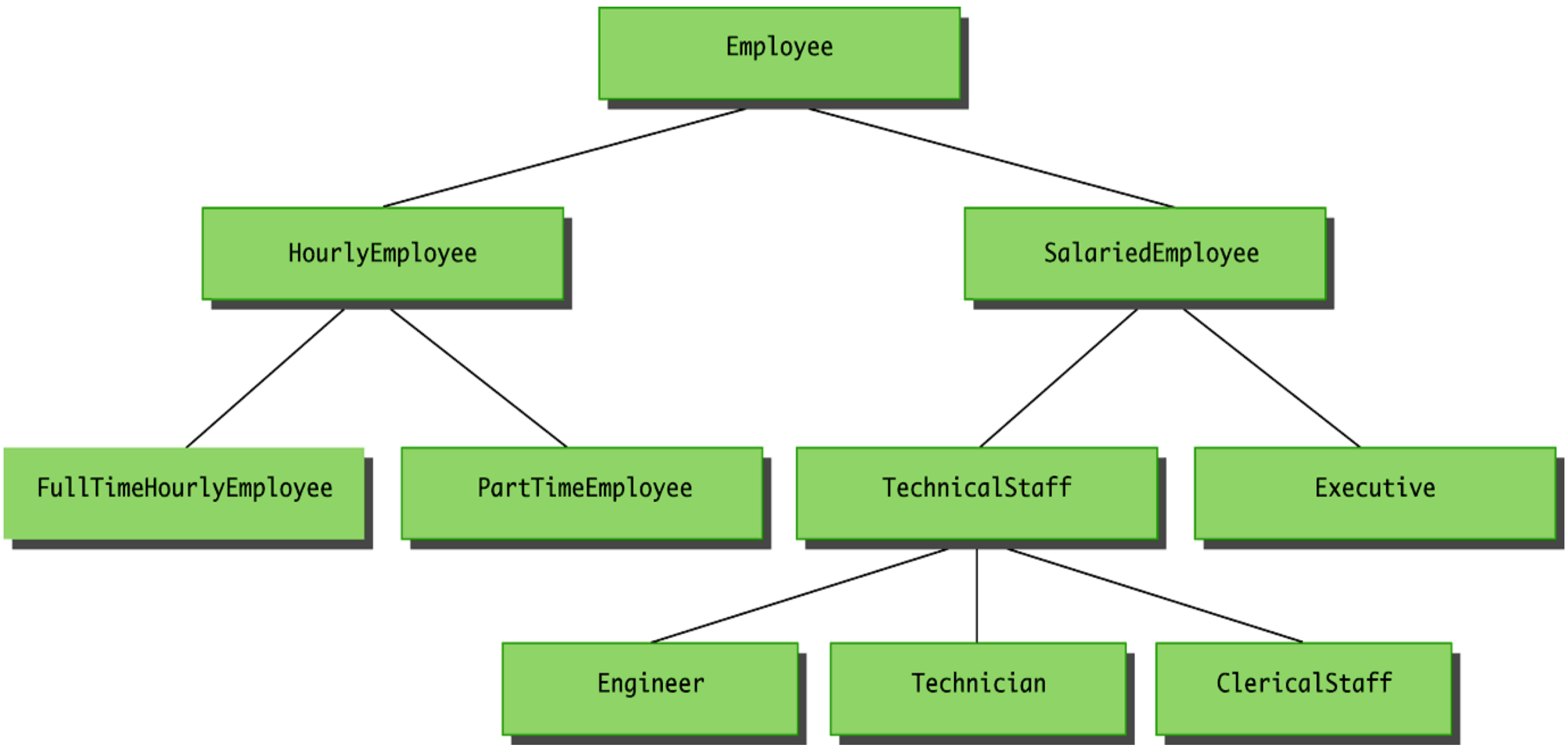
Introduction to Inheritance

- *Inheritance* is one of the main techniques of OOP.

- Using inheritance
  - a very general class is first defined,
  - then more specialized versions of the class are defined by
    - adding instance variables and/or
    - adding methods.
  - The specialized classes are said to *inherit* the methods and instance variables of the general class.

A Class Hierarchy

- There is often a natural hierarchy when designing certain classes.



Derived Classes

- All employees have certain characteristics in common:
  - a name and a hire date
  - the methods for setting and changing the names and hire dates
- Some employees have specialized characteristics:
  - Pay
    - hourly employees are paid an hourly wage
    - salaried employees are paid a fixed wage
  - Calculating wages for these two different groups would be different.

Inheritance and OOP

- Inheritance is an abstraction for
  - sharing similarities among classes (name and hireDate), and
  - preserving their differences (how they get paid).
- Inheritance allows us to group classes into families of related types (Employees), allowing for the sharing of common operations and data.

General Classes

- A class called `Employee` can be defined that includes all employees.
  - This class can then be used as a foundation to define classes for hourly employees and salaried employees.
    - The `HourlyEmployee` class can be used to define a `PartTimeHourlyEmployee` class, and so forth.

The Employee Class

```
/**
 Class Invariant: All objects have a name string and hire date.
 A name string of "No name" indicates no real name specified yet.
 A hire date of Jan 1, 1000 indicates no real hire date specified yet.
*/
public class Employee
{
    private String name;
    private Date hireDate;

   // no-argument constructor
    public Employee( )
    {
         name = "No name";
         hireDate = new Date("Jan", 1, 1000); //Just a placeholder.
    }
   // alternate constructor
    public Employee(String theName, Date theDate)   { /* code here */ }

   // copy constructor
    public Employee(Employee originalObject)        { /* code here */ }
```

(continued)

Employee Class

```
// some accessors and mutators
 public String getName( )                    { /* code here */ }
 public Date getHireDate( )                  { /* code here */ }
 public void setName(String newName)         { /* code here */ }
 public void setHireDate(Date newDate)       { /* code here */ }

// everyone gets the same raise
public double calcRaise( )
     { return 200.00; }

// toString and equals
 public String toString( )                   { /* code here */ }
 public boolean equals(Employee otherEmployee)
     { /* code here */ }

}  // end of Employee Class
```

Derived Classes

- Since an hourly employee "**is an**" employee, we want our class **`HourlyEmployee`** to be defined as a **derived** class of the class **`Employee`**.
  - A derived class is defined by adding instance variables and/or methods to an existing class.
  - The class that the derived class is built upon is called the **base class.**
  - The phrase **`extends BaseClass`** must be added to the derived class definition:

    ```
    public class HourlyEmployee extends Employee
    ```

- In OOP, a base class/derived class relationship is alternatively referred to by the term pairs:
  - **superclass/subclass**
  - **parent class/child class**

HourlyEmployee Class

```
/**
 Class Invariant: All objects have a name string, hire date,
nonnegative  wage rate, and nonnegative number of hours worked. */

public class HourlyEmployee extends Employee
{
   // instance variables unique to HourlyEmployee
    private double wageRate;
    private double hours; //for the month

   // no-argument Constructor
   public HourlyEmployee( )                                   { /* code here */}

   // alternative constructor
   public HourlyEmployee(String theName, Date theDate,
      double theWageRate, double theHours)                    { /* code here */}

   // copy constructor
    public HourlyEmployee(HourlyEmployee originalHE)          { /* code here */}
```

(continued)

HourlyEmployee Class

```
    // accessors and mutator specific to HourlyEmployee

    public double getRate( )                        { /* code here */ }
    public double getHours( )                       { /* code here */ }
    public void setHours(double hoursWorked)        { /* code here */ }
    public void setRate(double newWageRate)         { /* code here */ }

    // toString and equals specific for HourlyEmployee
    public String toString( )                       { /* code here */ }
    public boolean
         equals(HourlyEmployee otherHE)  { /* code here */ }

}  // end of HourlyEmployee Class
```

Inherited Members

- The derived class **inherits** all of the
  - public methods (and private methods, indirectly),
  - public and private instance variables, and
  - public and private static variables

  from the base class.

- Definitions for the inherited variables and methods <u>do not</u> appear in the derived class's definition.
  - The code is reused without having to explicitly copy it, unless the creator of the derived class redefines one or more of the base class methods.

- All instance variables, static variables, and/or methods defined directly in the derived class's definition are *added* to those inherited from the base class

Using HourlyEmployee

```
public class HourlyEmployeeExample
{
   public static void main(String[] args)
   {
      HourlyEmployee joe =
      new HourlyEmployee("Joe Worker", new Date(1, 1, 2004), 50.50, 160);

      // getName is defined in Employee
      System.out.println("joe's name is " + joe.getName( ));

      // setName is defined in Employee
      System.out.println("Changing joe's name to Josephine.");
      joe.setName("Josephine");

      // setRate is specific for HourlyEmployee
      System.out.println("Giving Josephine a raise");
      joe.setRate( 65.00 );

      // calcRaise is defined in Employee
      double raise = joe.calcRaise( );
      System.out.println("Joe's raise is " + raise );
   }
}
```

Overriding a Method Definition

- A derived class can change or **override** an inherited method.
- In order to override an inherited method, a new method definition is placed in the derived class definition.
- For example, perhaps the HourlyEmployee class had its own way to calculate raises. It could override Employee's calcRaise( ) method by defining its own.

Overriding Example

```
public class Employee
{
    ....
    public double calcRaise() { return 200.00; }
}

public class HourlyEmployee extends Employee
{
    . . . .
    // overriding calcRaise – same signature as in Employee
    public double calcRaise() { return 500.00; }
}
```

Now, this code

```
HourlyEmployee joe = new HourlyEmployee();
double raise = joe.calcRaise();
```

invokes the **overridden** `calcRaise()` method in the HourlyEmployee class rather than the `calcRaise()` method in the Employee class

To override a method in the derived class, the overriding method must have the same method signature as the base class method.

Overriding Versus Overloading

- Do not confuse *overriding* a method in a derived class with *overloading* a method name.
  - When a method in a derived class has the *same signature* as the method in the base class, that is overriding.
  - When a method in a derived class or the same class has a *different signature* from the method in the base class or the same class, that is overloading.
  - Note that when the derived class overrides or overloads the original method, it still inherits the original method from the base class as well (we' ll see this later).

The `final` Modifier

- If the modifier **`final`** is placed before the definition of a *method*, then that method <u>may not</u> be overridden in a derived class.

- It the modifier **`final`** is placed before the definition of a *class*, then that class <u>may not</u> be used as a base class to derive other classes.

Pitfall: Use of Private Instance Variables from a Base Class

- An instance variable that is private in a base class is not accessible *by name* in a method definition of a derived class.
  - An object of the **`HourlyEmployee`** class cannot access the private instance variable **`hireDate`** by name, even though it is inherited from the **`Employee`** base class.
- Instead, a private instance variable of the base class can only be accessed by the public accessor and mutator methods defined in that class.
  - An object of the **`HourlyEmployee`** class can use the **`getHireDate()`** or **`setHireDate()`** methods to access **`hireDate`**.

Encapsulation and Inheritance Pitfall: Use of Private Instance Variables from a Base Class

- If private instance variables of a class were accessible in method definitions of a derived class, …
    - then anytime someone wanted to access a private instance variable, they would only need to create a derived class, and access the variables in a method of that class.
- This would allow private instance variables to be changed by mistake or in inappropriate ways.

Pitfall: Private Methods Are Effectively Not Inherited

- The private methods of the base class are like private variables in terms of not being directly available.
- A private method is completely unavailable, unless invoked indirectly.
  - This is possible only if an object of a derived class invokes a public method of the base class that happens to invoke the private method.
- This should not be a problem because private methods should be used only as helper methods.
  - If a method is not just a helper method, then it should be public.

`Protected` Access

- If a method or instance variable is modified by `protected` (rather than `public` or `private`), then it can be accessed *by name*
  - Inside its own class definition
  - Inside any class derived from it
  - In the definition of any class in the same package
- The `protected` modifier provides very weak protection compared to the `private` modifier
  - It allows direct access to any programmer who defines a suitable derived class
  - Therefore, instance variables should normally **not** be marked `protected`

“`Package`” Access

- If a method or instance variable has no visibility modifier (`public` `private`, or `protected`), it is said to have “package access”, and it can be accessed *by name*
  - Inside its own class definition
  - In the definition of any class in the same package
  - *BUT NOT inside any class derived from it*
- So, the implicit “package” access provides slightly stronger protection than the `protected` modifier, but is still very weak compared to the `private` modifier
  - By design, it is used when a set of classes closely cooperate to create a unified interface
  - By default, it is used by novice programmers to get started without worrying about visibility modifiers or packages

Inherited Constructors?

An Employee constructor cannot be used to create HourlyEmployee objects. Why not?

We must implement a specialized constructor for HourlyEmployees. But how can the HourlyEmployee constructor initialize the private instance variables in the Employee class since it doesn't have direct access?

The **`super`** Constructor

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class
  - In order to invoke a constructor from the base class, it uses a special syntax:

```
public DerivedClass(int p1, int p2, double p3)
{
  super(p1, p2);
  derivedClassInstanceVariable = p3;
}
```

  - In the above example, `super(p1, p2);` is a call to the base class constructor

The **`super`** Constructor

- A call to the base class constructor can never use the name of the base class, but uses the keyword **`super`** instead
- A call to **`super`** must always be the first action taken in a constructor definition

The `super` Constructor

- If a derived class constructor does not include an invocation of `super`, then the no-argument constructor of the base class will automatically be invoked
  - This can result in an error if the base class has not defined a no-argument constructor
- Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, an explicit call to `super` should almost always be used.

HourlyEmployee Constructor

```
public class HourlyEmployee extends Employee
{
   private double wageRate;
   private double hours;     // for the month

   // the no-argument constructor invokes
   // the Employee (super) no-argument constructor
   // to initialize the Employee instance variables
   // then initializes the HourlyEmployee instance variables

   public HourlyEmployee()
    {
        super();
        wageRate = 0;
        hours = 0;
    }
```

HourlyEmployee Constructor

```
// the alternative HourlyEmployee constructor invokes an
// appropriate Employee (super) constructor to initialize
// the Employee instance variables (name and date), and then
// initializes the HourlyEmployee rate and hours

public HourlyEmployee(String theName, Date theDate,
                      double theWageRate, double theHours)
 {
      super(theName, theDate);
      if ((theWageRate >= 0) && (theHours >= 0))
      {
          wageRate = theWageRate;
          hours = theHours;
      }
      else
      {
           System.exit(0);
      }
 }
```

Review of Rules For Constructors

- Constructors can chain to other constructors:
  - in own class, by invoking `this(…)`;
  - in parent class, by invoking `super(…)`;
- If there is an explicit call to `this(…)` or `super(…)`, it must be the very first statement in the body
  - It must come even before any local variable declarations
- You can call either this() or super(), but not both
- If you don't have explicit call to this() or super(), an implicit call to a no-arg super() is implicitly inserted
- Implied by above rules:
  At least one constructor will be called at each class level up the inheritance hierarchy, all the way to the top (Object)

Access to a Redefined Base Method

- Within the definition of a method of a derived class, the base class version of an overridden method of the base class can still be invoked
  - Simply preface the method name with super and a dot

```
// HourlyEmployee's toString( ) might be
public String toString( )
{
  return (super.toString() + "$" + getRate( ));
}
```

- However, using an object of the derived class outside of its class definition, there is no way to invoke the base class version of an overridden method

You Cannot Use Multiple `supers`

- It is only valid to use `super` to invoke a method from a direct parent
  - Repeating `super` will not invoke a method from some other ancestor class
- For example, if the `Employee` class were derived from the class `Person`, and the `HourlyEmployee` class were derived form the class `Employee` , it would not be possible to invoke the `toString` method of the `Person` class within a method of the `HourlyEmployee` class

```
super.super.toString() // ILLEGAL!
```

- Ensures that each class has *complete* control over its interface

Base/Derived Class Summary

Assume that class D (Derived) is derived from class B (Base).

1. Every object of type D **is a** B, but not vice versa.

2. D is a more specialized version of B.

3. **Anywhere an object of type B can be used, an object of type D can be used just as well**, but not vice versa.

(Adapted from: ***Effective C++***, 2nd edition, pg. 155)

Tip: Static Variables Are Inherited

- Static variables in a base class are inherited by any of its derived classes
- The modifiers `public`, `private`, and `protected` have the same meaning for static variables as they do for instance variables

The Class **`Object`**

- In Java, every class is a descendent of the class ***`Object`***
  - **`Object`** is the root of the entire Java class hierarchy
  - Every class has **`Object`** as its ancestor
  - Every object of every class is of type **`Object`**, as well as being of the type of its own class (and also all classes in between)
- If a class is defined that is not explicitly a derived class of another class, it is by default a derived class of the class **`Object`**

The Class `Object`

- The class `Object` is in the package `java.lang` which is always imported automatically
- Having an `Object` class enables methods to be written with a parameter of type `Object`
  - A parameter of type `Object` can be replaced by an object of any class whatsoever
  - For example, some library methods accept an argument of type `Object` so they can be used with an argument that is an object of any class
  - Recall the ArrayList class (an old form of it) we studied earlier: the store and retrieve methods were declared to work on instances of type `Object`

The Class `Object`

- The class `Object` has some methods that every Java class inherits
  - For example, the `equals` and `toString` methods
- Every object inherits these methods from some ancestor class
  - Either the class `Object` itself, or a class that itself inherited these methods (ultimately) from the class `Object`
- However, these inherited methods should be overridden with definitions more appropriate to a given class
  - Some Java library classes assume that every class has its own version of such methods

The Right Way to Define **equals**

- Since the **equals** method is always inherited from the class **Object**, methods like the following simply overload it:

```
public boolean equals(Employee otherEmployee)
  { . . . }
```

- However, this method should be **overridden**, not just overloaded:

```
public boolean equals(Object otherObject)
  { . . . }
```

The Right Way to Define `equals`

- The overridden version of `equals` must meet the following conditions
  - The parameter `otherObject` of type `Object` must be type cast to the given class (e.g., `Employee)`
  - However, the new method should only do this if `otherObject` really is an object of that class, and if `otherObject` is not equal to `null`
  - Finally, it should compare each of the instance variables of both objects

A Better **equals** Method for the Class **Employee**

```
public boolean equals(Object otherObject)
{
  if(otherObject == null)
    return false;
  else if(getClass( ) != otherObject.getClass( ))
    return false;
  else
  {
    Employee otherEmployee = (Employee)otherObject;
    return (name.equals(otherEmployee.name) &&
      hireDate.equals(otherEmployee.hireDate));
  }
}
```

The `getClass()` Method

- Every object inherits the same `getClass()` method from the `Object` class
  - This method is marked `final`, so it cannot be overridden
- An invocation of `getClass()` on an object returns a representation *only* of the class that was used with `new` to create the object
  - The results of any two such invocations can be compared with `==` or `!=` to determine whether or not they represent the exact same class

```
(object1.getClass() == object2.getClass())
```

Why `equals()` Must be Overridden

Imagine we have:

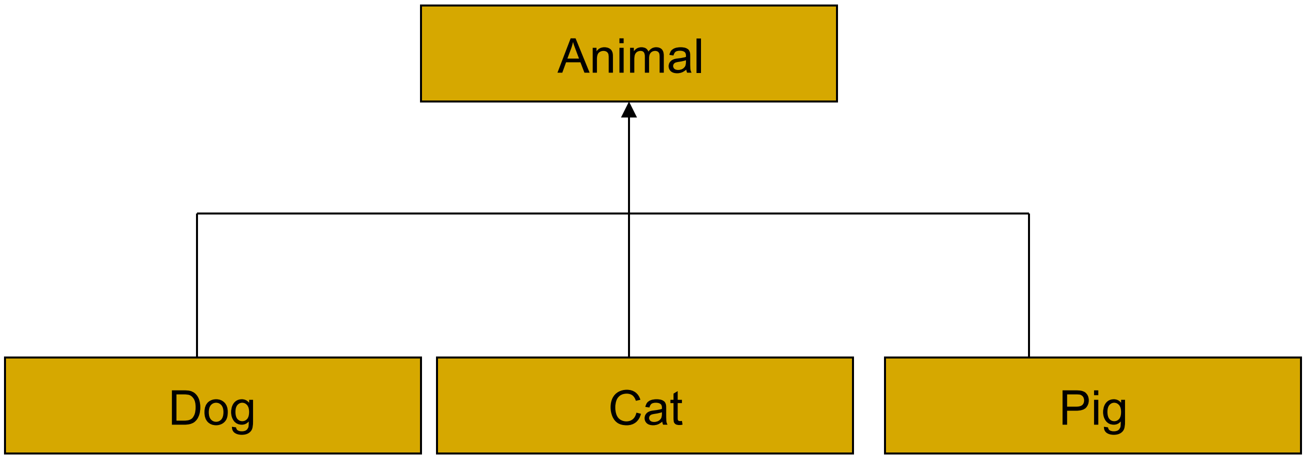
```
public class Point {
  public int x, y;
  … // Stuff here like constructors, etc.
  public boolean equals(Point otherPt) {
    return (x == otherPt.x && y == otherPt.y);
  }
}
public class Point3D extends Point {
  public int z;
  public boolean equals(Point3D otherPt) {
    return (x == otherPt.x && y == otherPt.y && z == otherPt.z);
  }
}
  …
  Point pt2d = new Point(1.0, 2.0);
  Point3D pt3d = new Point3D(1.0, 2.0, 3.0);
  if (pt3D.equals(pt2D))
    System.out.println("pt2d and pt3D equal");
```

What will it print out?

Basic Class Hierarchy Design

- How many levels of classes should we create?
  - Two extremes:
    - MovableThing -> A1981BlueMiataWithBlackVinylTop vs.
    - Vehicle->Car->Car2Door->Convertible2Door->Miata->BlueMiata->…
    - or something in between, perhaps? Yes…
- Create intermediate classes where you do—or might later—want to make a distinction that splits the tree
- It is easier to create than take away intermediate classes.
- What to put at a given level?
  - Maximize abstracting out common elements
  - But, think about future splits, and what is appropriate at given level

Animal Hierarchy



Animals That Speak

```
public class Animal
{
   public void speak( int x )
   { System.out.println(“ Animal “ + x );}
}
public class Dog extends Animal
{
   public void speak (int x )
   { System.out.println( “Dog “ + x ); }
}
public class Cat extends Animal
{
   public void speak (int x )
   { System.out.println( “Cat “ + x ); }
}
public class Pig extends Animal
{
   public void speak (int x )
   { System.out.println( “Pig “ + x ); }
}
```

The ZooDemo Class

In the ZooDemo, we ask each Animal to say hello to the audience.

```
public class ZooDemo
{
   // Overloaded type-specific sayHello method
   // for each kind of Animal

   public static void sayHello( Dog d, int i )
        { d.speak( i ); }

   public static void sayHello( Cat c, int i )
        { c.speak( i ); }

   public static void sayHello( Pig p, int i)
        { p.speak( i ); }
```

(continued)

The ZooDemo Class

```
   public static void main( String[ ] args )
   {
        Dog dusty = new Dog( );
        Cat fluffy = new Cat( );
        Pig sam = new Pig( );

        sayHello( dusty, 7 );
        sayHello( fluffy, 17 );
        sayHello( sam, 27 );
   }
} // end Zoo Demo

//------- output -----
Dog 7
Cat 17
Pig 27
```

Problems with ZooDemo?

- The ZooDemo class contains a type-specific version of sayHello for each type of Animal.
- What if we add more types of Animals?
- Wouldn't it be nice to write just one sayHello method that works for all animals?
- This is called Polymorphism

New ZooDemo

```
public class ZooDemo
{
   // One sayHello method whose parameter
   // is the base class works for all Animals

   public static void sayHello( Animal a, int x )
        { a.speak( x ); }

   public static void main( String[ ] args )
   {
        Dog dusty = new Dog( );
        Cat fluffy = new Cat( );
        Pig sam = new Pig( );

        sayHello( dusty, 7 );
        sayHello( fluffy, 17 );
        sayHello( sam, 27 );
   }
}
```

Introduction to Abstract Classes

- An *abstract method* is like a placeholder for a method that will be fully defined in a descendent class.
  - It postpones the definition of a method.
  - It has a complete method heading to which the modifier `abstract` has been added.
  - It cannot be private.
  - It has no method body, and ends with a semicolon in place of its body.

```
public abstract double getPay();
public abstract void doIt(int count);
```

  - The body of the method is defined in the derived classes.
- The class that contains an abstract method is called an *abstract class.*

Abstract Class

- A class that has at least one abstract method is called an *abstract class.*

- An abstract class must have the modifier `abstract` included in its class heading.

```
public abstract class Employee
{
  private instanceVariables;
  . . .
  public abstract double getPay();
  . . .
}
```

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods.

- If a derived class of an abstract class adds to or does not define all of the abstract methods,
  - it is abstract also, and
  - must add **`abstract`** to its modifier.

- A class that has no abstract methods is called a *concrete class.*

Abstract Employee Class

```
public abstract class Employee
{
    private String name;
    private Date hireDate;
     public abstract double getPay( );

     // constructors, accessors, mutators, equals, toString

     public boolean samePay(Employee other)
     {
          return(this.getPay() == other.getPay());
     }

}
```

Pitfall: You Cannot Create Instances of an Abstract Class

- An abstract class can only be used to derive more specialized classes.
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker
- An abstract class constructor cannot be used to create an object of the abstract class.
  - However, a derived class constructor will include an invocation of the abstract class constructor in the form of **`super`**.

An Abstract Class Is a Type

- Although an object of an abstract class cannot be created, it is perfectly fine to have a parameter of an abstract class type.
  - This makes it possible to plug in an object of any of its descendent classes.

- It is also fine to use a variable of an abstract class type, as long is it names objects of its concrete descendent classes only.

Additional Topics/Questions

- Are constructors inherited?
- What happens when a child redefines an instance variable?
  - Variables do not overload or override: they "hide"
  - What happens if:
    - parent: "public int x", child: "public String x"
    - parent: "public int x:, child: "private int x"
    - → then: child-of-child: "x = 42"
- Do private methods inherit/obey polymorphism?
- Can a child class define a private method with the same signature as an inherited method?

Additional Topics/Questions

- What happens when a parent's method is called?
  - Recall: parent method can be triggered through inheritance, or via super.someMethod()
  - What happens w/call to myOverriddenMethod() in parent?
  - What happens w/call to private method in parent?
    - …when child has same-named private method?
    - …when child has same-named public method?

Classes and Methods

- When a class defines its methods as public, it describes how the class user interacts with the method.
- These public methods form the class' **`interface`** .
- An abstract class contains one or more methods with only an interface – no method body is provided.
- Java allows us to take this concept one step further.

Interfaces

- An interface is something like an extreme abstract class.
- All of the methods in an interface are abstract – they have no implementations.
- An interface
  - has no instance variables.
  - Only defines methods.
  - is NOT a class.
  - is a type that can be satisfied by any class that implements the interface

Interfaces

- The syntax for defining an interface is similar to that of defining a class
  - Except the word `interface` is used in place of `class`
- An interface specifies a set of methods that any class that implements the interface must have
  - It contains method headings (and optionally static final constant definitions) only
  - It contains no instance variables nor any complete method definitions

Interfaces

- An interface and all of its method headings should be declared public
- When a class implements an interface, it must make all the methods in the interface public.
- Because an interface is a type, a method may be written with a parameter of an interface type
  - That parameter will accept as an argument any class that implements the interface

Implementing an Interface

- To create a class that implements all the methods defined in an interface, use the keyword **implements**.
- Whereas **interface** defines the headings for methods that must be defined, a class that **implements** the interface defines how the methods work.

The Animal Interface

```
public interface Animal
{
  public void eat( );
}
```

Yes, animals do more than eat, but we’re trying to make this a simple example.

Interfaces

- To *implement an interface*, a concrete class must do two things:

1. It must include the phrase

    ```
    implements Interface_Name
    ```

    at the start of the class definition

    – If more than one interface is implemented, each is listed, separated by commas

2. The class must implement *all* the method headings listed in the definition(s) of the interface(s)

Implementing Animal

```
// Lion and Snake implement the required eat( ) method
public class Lion implements Animal
{
   public void eat()
        { System.out.println("Lions Devour"); }
}
public class Snake implements Animal
{
   public void eat()
        { System.out.println( "Snakes swallow whole"); }
}
```

Implementing Animal

```
// Dog implements the required eat( ) method and has
// some of its own methods and instance variables
public class Dog implements Animal {
   private String name;
   Dog(String newName)
        {name = newName;}
   public void eat()
        {System.out.println("Dog chews a bone");}
}

// Poodle is derived from Dog, so it inherits eat( )
// Adds a method of its own
public class Poodle extends Dog
{
   Poodle( String name )
        { super(name); }  // call Dog constructor

   public String toString( )
        { return "Poodle"; }
}
```

Implementing Animal

```
// Using classes that implement Animal
public class Jungle {
   public static void feed( Animal a )
        { a.eat(); }

   public static void main( String[] args ){
        Animal[ ] animals = {
                new Lion( ),
                new Poodle( "Fluffy“ ),
                new Dog( "Max“ ),
                new Snake( )
        };
        for (int i = 0; i < animals.length; i++)
                feed( animals[ i ] );
   }
}

// --- Output
```

Lions Devour
Dog chews a bone
Dog chews a bone
Snakes swallow whole

Extending an Interface

- An new interface can add method definitions to an existing interface by **extending** the old

```
interface TiredAnimal extends Animal
{
   public void sleep( );
}
```

The TiredAnimal interface includes both eat( ) and sleep( );

Implementing Multiple Interfaces

- Recall the Animal interface from earlier

```
public interface Animal
{
   public void eat( );
}
```

- Define the Cat interface

```
public interface Cat
{
   void purr( );      // public by default;
}
// since a Lion is an Animal and a Cat, Lion may wish
// to implement both interfaces
public class Lion implements Animal, Cat
{
   public void eat( ) {System.out.println("Big Gulps");}
   public void purr( ) {System.out.println("ROOOAAAR!");}
}
```

Just separate the Interface names with a comma

Inconsistent Interfaces

- In Java, a class can have only one base class
  - This prevents any inconsistencies arising from different definitions having the same method heading
- In addition, a class may implement any number of interfaces
  - Since interfaces do not have method bodies, the above problem cannot arise
  - However, there are other types of inconsistencies that can arise

Inconsistent Interfaces

- When a class implements two interfaces:
  - Inconsistency will occur if the interfaces contain methods with the same name but different return types
- If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is illegal

The `Comparable` Interface

- The **`Comparable`** interface is in the **`java.lang`** package, and so is automatically available to any program
- It has only the following method heading that must be implemented (note the Object parameter)

```
public int compareTo(Object other);
```

- It is the programmer's responsibility to follow the semantics of the **`Comparable`** interface when implementing it
- When implementing **`compareTo`**, you would of course overload it by using an appropriate parameter type

The `Comparable` Interface Semantics

- The method **`compareTo()`** must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other

- If the parameter **`other`** is not of the same type as the class being defined, then a **`ClassCastException`** should be thrown

The `Comparable` Interface Semantics

- Almost any reasonable notion of "comes before" is acceptable
  - In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable
- The relationship "comes after" is just the reverse of "comes before"

compareTo for Person

```
public class Person implements Comparable
{
   private String name;
   ...
   public int compareTo( Object obj )
   {
        Person p = (Person)obj;
        return name.compareTo(p.name);
   }
   ....
}
```

If **obj** is not a **Person** object a **ClassCastException** will be thrown

Comparing the names using String's **compareTo** method

Using Comparable

```
// prints the index of the smallest Integer in an array
// Note use of Integer, not int
public class FindSmallest {
   public static void main( String[ ] args)
   {
        // find the smallest Integer in an array
        // Integer (implements Comparable )
        int index = 0;          // index of smallest value
        Integer[ ] values = {
               new Integer(144), new Integer(200), new Integer(99),
               new Integer(42),  new Integer(132) };
        for (int i = 1; i < values.length; i++)
        {
               if ( values[ i ].compareTo( values[ index ] ) < 0 )
                       index = i;
        }
        System.out.println("Index of smallest value is " + index);
   }
}
```