Flexibility in data types

Consider the following method invocation:
```java
myShape.area();
```

At some point, this invocation is bound to the definition of the method that it invokes

- If this binding occurred statically at compile time, then that line of code would call the same method every time

- Java defers method binding until run time: this is called dynamic binding or late binding

- Dynamic binding provides flexibility in program design

- A polymorphic reference is a variable that can refer to different types of objects at different points in time

Suppose we create the following reference variable
```java
Rectangle myShape;
```

- Java allows this reference to point to an Rect object, or to any object of any compatible type

- This compatibility can be established using inheritance or using interfaces

Assignment is due Monday 11:59 PM!

Points off for javadoc, but do try it!

Study sample solutions in #2 (and in every assignment)!

Reading for today and Tuesday: Ch. 10 (with some exceptions)
An object reference can refer to an object of its class, or to an object of any class related to it by inheritance.

What determines which method is invoked?

Class Rectangle has a method called area(), and the child Class Square overrides it.

Now consider the following invocation:

```java
myShape.area();
```

Which area() is invoked?

If `myShape` refers to a Rectangle object, it invokes the Rectangle version of `area()`.

If `myShape` refers to a Square object, it invokes the Square version!

---

A Program that pays a set of diverse employees using a polymorphic method.

```java
/**
 * Demonstrates polymorphism via inheritance.
 * @author Java Foundations
 */
public class Firm {
   /**
    * Creates a staff of employees for a firm and pays them.
    */
   public static void main (String[] args) {
      Staff personnel = new Staff();
      personnel.payday();
   }
}
```
/*  
* Represents a generic staff member
*/
abstract public class StaffMember {
    protected String name;
    protected String address;
    protected String phone;
    
    /*
    * Constructor: Sets up this staff member using the specified information.
    */
    public StaffMember (String eName, String eAddress, String ePhone) {
        name = eName;
        address = eAddress;
        phone = ePhone;
    }
    
    /*
    * Derived classes must define the pay method for each type of staff member.
    */
    public abstract double pay();
    
    toString() omitted...
}

public class Volunteer extends StaffMember {
    /*
    * Constructor: Sets up this staff member using the specified information.
    */
    public Volunteer (String eName, String eAddress, String ePhone) {
        super (eName, eAddress, ePhone);
    }
    
    /*
    * Return a zero pay value for this volunteer.
    */
    public double pay() {
        return 0.0;
    }
}

public void payday () {
    double amount;
    
    for (int count=0; count < staffList.length; count++) {
        System.out.println (staffList[count]);
        amount = staffList[count].pay(); // polymorphic
        if (amount == 0.0)
            System.out.println ("Thanks!");
        else
            System.out.println ("Paid: " + amount);
        System.out.println ("---------------------------");
    }
}
 public class Employee extends StaffMember {
  protected String socialSecurityNumber;
  protected double payRate;

  /**
   * Constructor: Sets up this employee with the specified information.
   *
   * @param eName  Employee's name
   * @param eAddress  Employee's address
   * @param ePhone  Employee's phone number
   * @param socSecNumber  Employee's social security number
   * @param rate  Employee's pay rate
   */
  public Employee (String eName, String eAddress, String ePhone,
                   String socSecNumber, double rate) {
    super (eName, eAddress, ePhone);
    socialSecurityNumber = socSecNumber;
    payRate = rate;
  }

  /**
   * Return the pay rate for this employee.
   */
  public double payRate() {
    return payRate;
  }
}

 public class Executive extends Employee {
  private double bonus;

  /**
   * Constructor: Sets up this executive with the specified information
   *
   * @param eName  Executive's name
   * @param eAddress  Executive's address
   * @param ePhone  Executive's phone number
   * @param socSecNumber  Executive's social security number
   * @param rate  Executive's pay rate
   */
  public Executive (String eName, String eAddress, String ePhone,
                   String socSecNumber, double rate) {
    super (eName, eAddress, ePhone, socSecNumber, rate);
    bonus = 0; // bonus has yet to be awarded
  }

  /**
   * Computes and returns the pay for an executive, which is the
   * regular employee pay rate plus a one-time bonus.
   */
  public double pay() {
    double payment = super.pay() + bonus;
    bonus = 0;
    return payment;
  }
}

 public class Hourly extends Employee {
  private int hoursWorked;

  /**
   * Constructor: Sets up this hourly employee using the specified information.
   *
   * @param eName  Hourly employee's name
   * @param eAddress  Hourly employee's address
   * @param ePhone  Hourly employee's phone number
   * @param socSecNumber  Hourly employee's social security number
   * @param rate  Hourly employee's pay rate
   */
  public Hourly (String eName, String eAddress, String ePhone,
                 String socSecNumber, double rate) {
    super (eName, eAddress, ePhone, socSecNumber, rate);
    hoursWorked = 0;
  }

  /**
   * Computes and returns the pay for this hourly employee.
   */
  public double pay() {
    double payment = payRate * hoursWorked;
    hoursWorked = 0;
    return payment;
  }
}