

Polymorphism

Inheritance provides Power to OOP

Polymorphism provides flexibility through inheritance

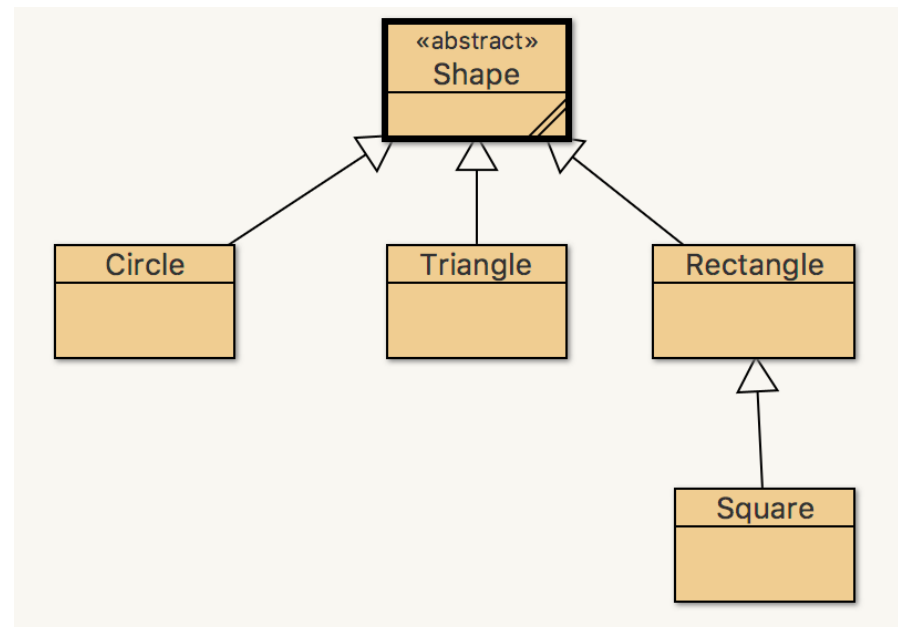
```
mirror_mod = modifier_ob.  
set mirror object to mirror  
mirror_mod.mirror_object  
operation == "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation == "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation == "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True  
selection at the end -add  
mirror_ob.select= 1  
modifier_ob.select=1  
context.scene.objects.active  
("Selected" + str(modifier_ob.  
mirror_ob.select = 0  
context.select  
context.objects.one.name]  
print("please select ex  
types.Operator):  
X mirror to the selected  
object.mirror_mirror_x"  
mirror X"  
context):  
context.active_object is not
```

Arrays of Shapes?

- Last time we saw the Shapes hierarchy and how to create a hierarchy of classes in order to reuse code.
- What if you want to create a **collection** of Shapes?
- All values in a particular array must have the same type or **be of compatible types!**

Can an array of Shape contain circles, triangles, and rectangles?

Yes! Due to polymorphism.



Polymorphism via Inheritance

```
Rectangle myShape = new Rectangle();  
myShape.calculateArea();  
Square perfect = new Square();  
myShape = perfect;  
myShape.calculateArea();
```

Class **Rectangle** has a method called `calculateArea()`, and the child class **Square** overrides it

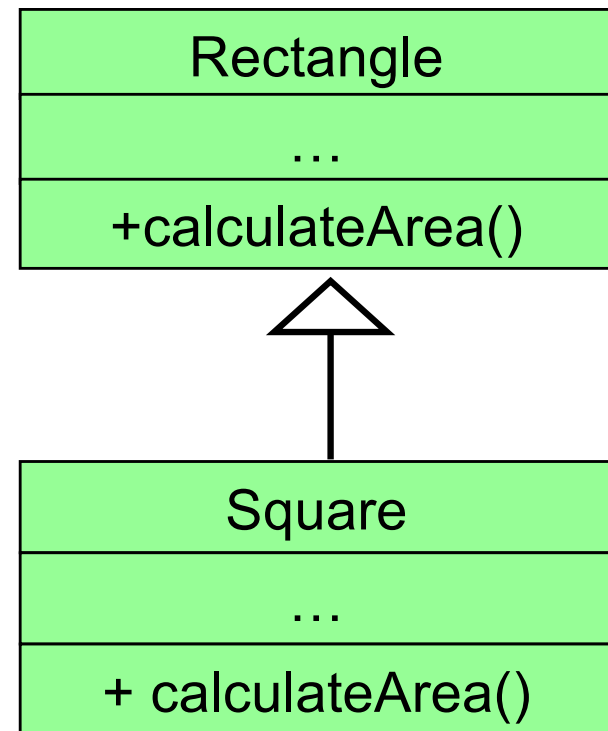
Now consider the following invocation

```
myShape.calculateArea();
```

Which `calculateArea()` is invoked?

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

If `myShape` refers to a **Square** object, it invokes the **Square** version of `calculateArea()`!

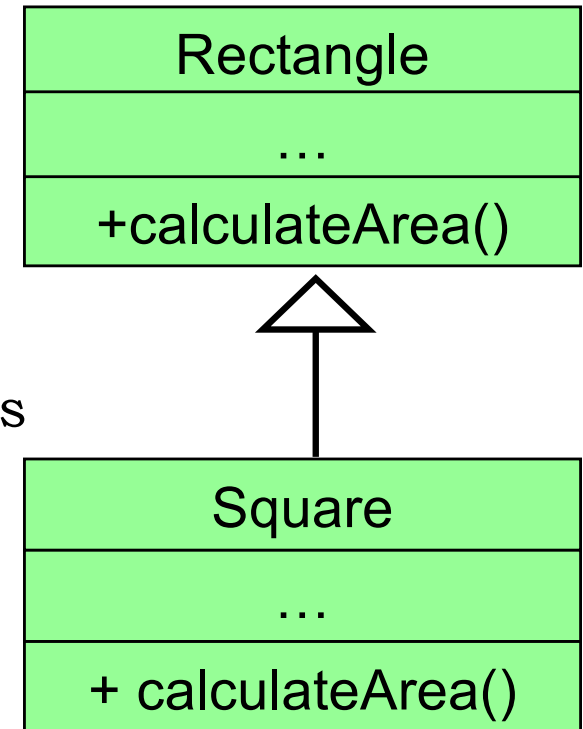


Static and Dynamic Binding

- Consider the following method invocation:

```
myShape. calculateArea();
```

- 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 (That's not happening in Java)



- Java defers method binding until **run** time: this is called **dynamic binding** or *late binding*
- Dynamic binding provides **flexibility** in program design

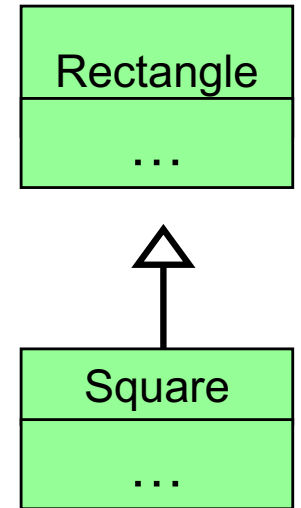


Polymorphism: "having many forms"

- 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

```
Rectangle myShape;
```

- Java allows this reference to point to a Rectangle object, or to any object of **any compatible type!**
- This **compatibility** can be established using **inheritance** or using **interfaces**

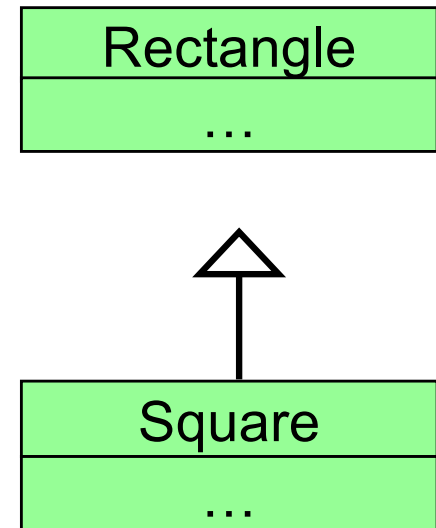


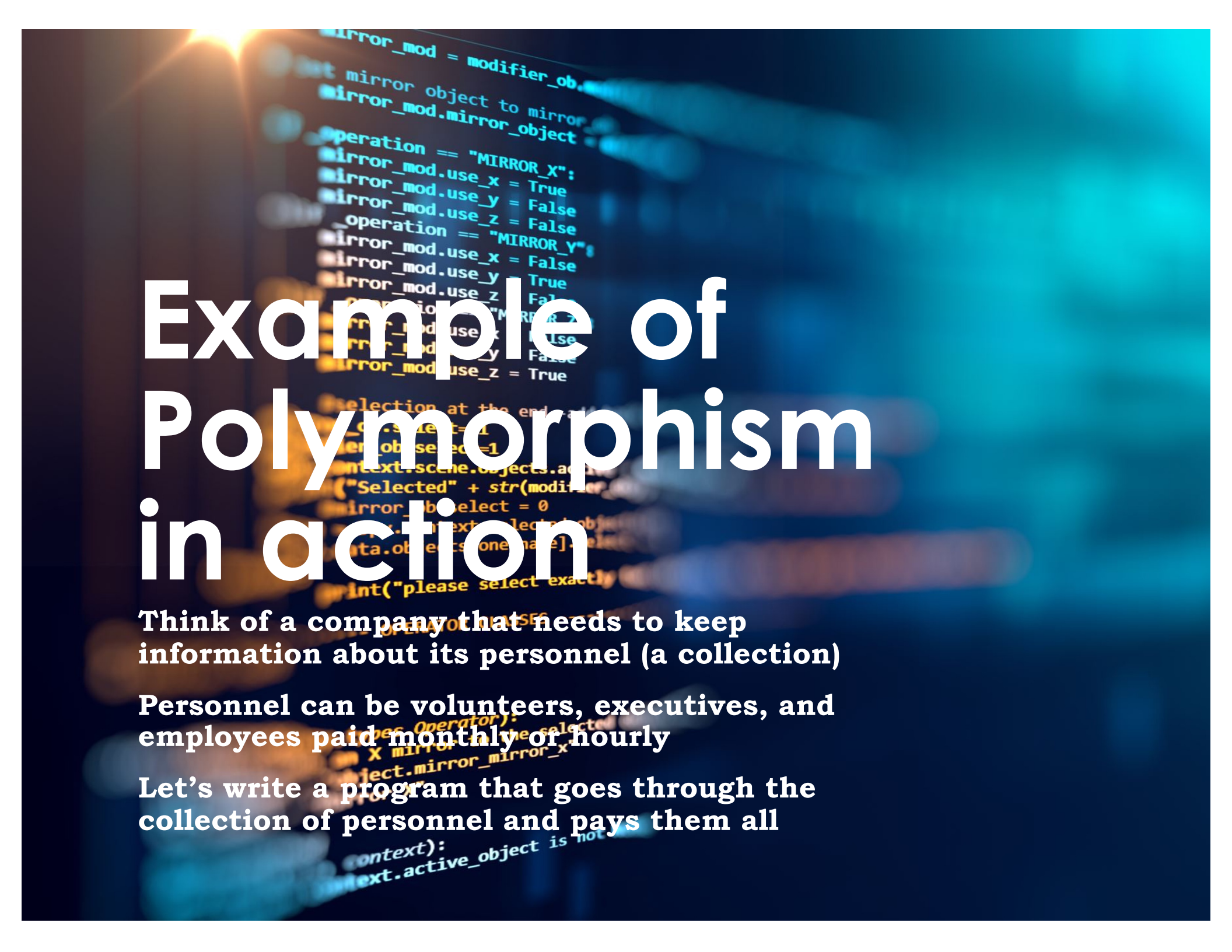
References and Inheritance

- Assigning a child object to a parent reference is considered to be a **widening conversion**, and can be performed by simple assignment
- Assigning a parent object to a child reference can be done also, but it is considered a **narrowing conversion** and must be done with a cast
- The widening conversion is the most useful

```
Rectangle wide =  
new Square();
```

```
Square narrow =  
(Square) new Rectangle();
```





Example of Polymorphism in action

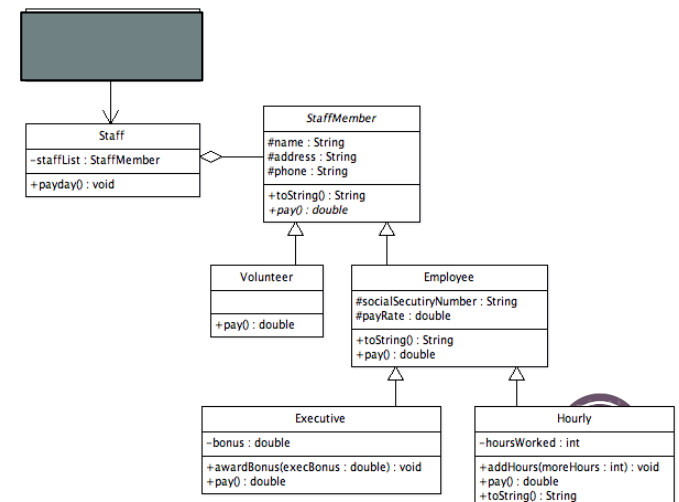
Think of a company that needs to keep information about its personnel (a collection)

Personnel can be volunteers, executives, and employees paid monthly or hourly

Let's write a program that goes through the collection of personnel and pays them all

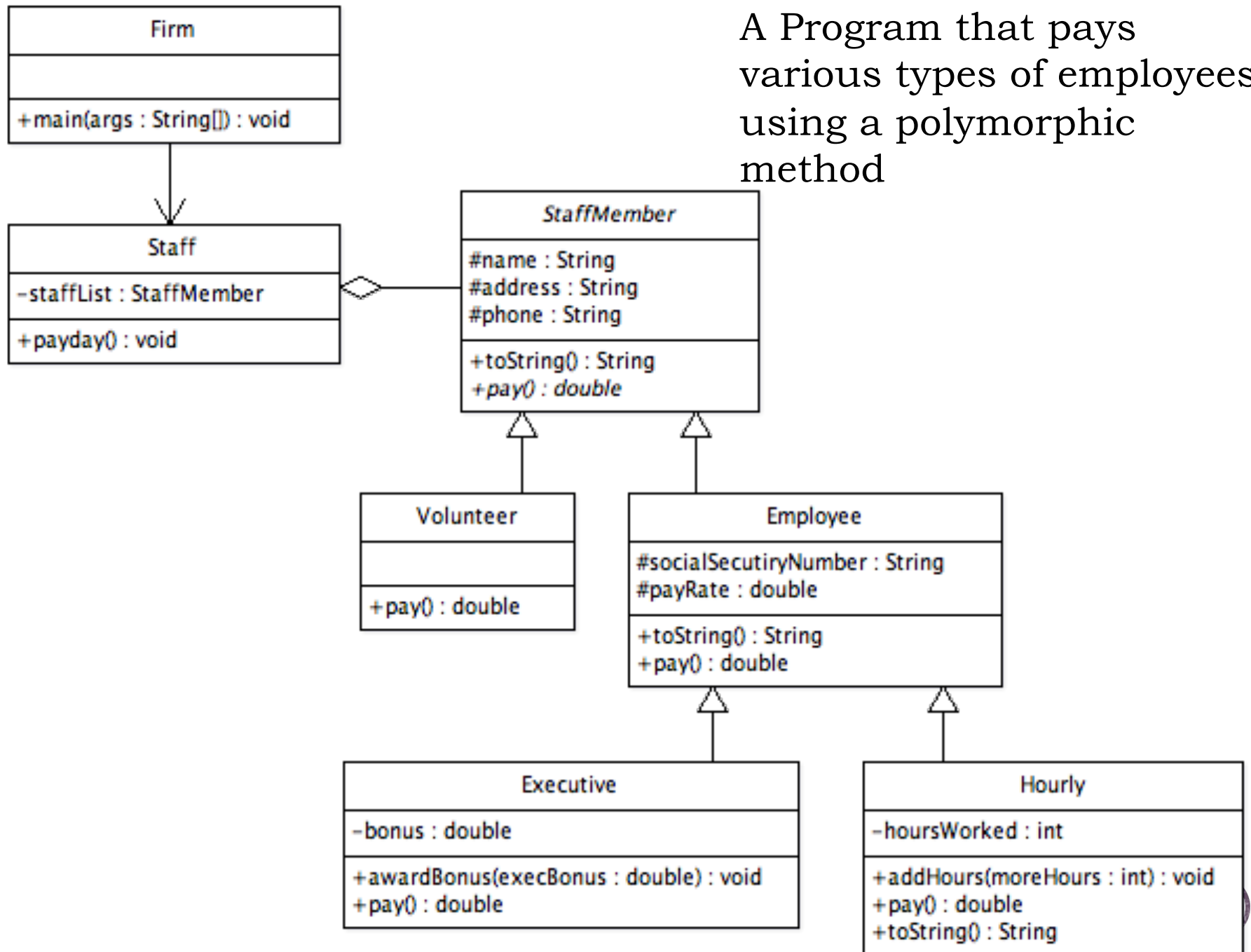
Firm.java

```
1  /**
2   * Demonstrates polymorphism via inheritance.
3   * @author Java Foundations
4   */
5  public class Firm {
6      /**
7       * Creates a staff of employees for a firm and pays them.
8       */
9      public static void main (String[] args) {
10         Staff personnel = new Staff();
11
12         personnel.payday();
13     }
14 }
15
```



Exploring the benefits and flexibility of polymorphism

A Program that pays various types of employees using a polymorphic method



Staff.java

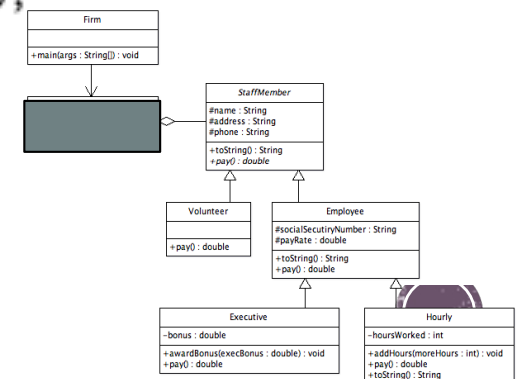
```
/**
 * Represents the personnel staff of a particular business.
 * @author Java Foundations
 */
public class Staff {
    private StaffMember[] staffList;

    /**
     * Constructor: Sets up the list of staff members.
     */
    public Staff () {
        staffList = new StaffMember[6];

        staffList[0] = new Executive ("Tony", "123 Main Line", "555-0469", "123-45-6789", 2423.07);
        staffList[1] = new Employee ("Paulie", "456 Off Line", "555-0101", "987-65-4321", 1246.15);
        staffList[2] = new Employee ("Vito", "789 Off Rocker", "555-0000", "010-20-3040", 1169.23);
        staffList[3] = new Hourly ("Michael", "678 Fifth Ave.", "555-0690", "958-47-3625", 10.55);
        staffList[4] = new Volunteer ("Adrianna", "987 Babe Blvd.", "555-8374");
        staffList[5] = new Volunteer ("Benny", "321 Dud Lane", "555-7282");

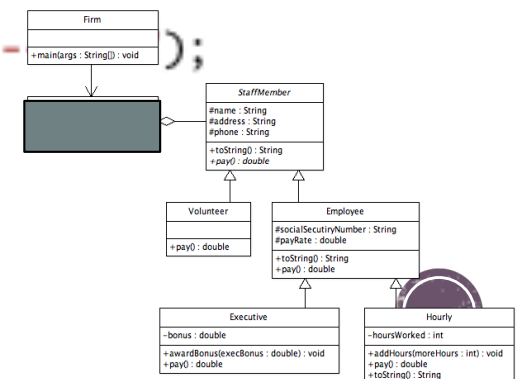
        ((Executive)staffList[0]).awardBonus (500.00);

        ((Hourly)staffList[3]).addHours (40);
    }
}
```



Staff.java

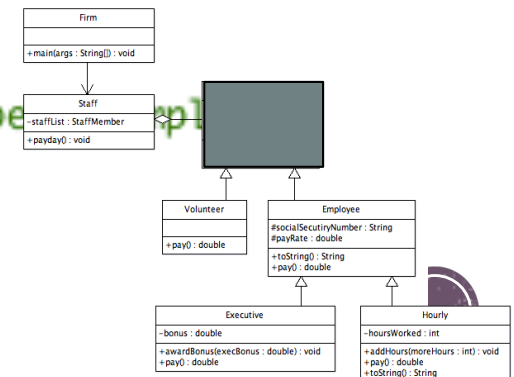
```
25
26  /**
27   * Pays all staff
28   */
29  public void payday () {
30      double amount;
31
32      for (int count=0; count < staffList.length; count++) {
33          System.out.println (staffList[count]);
34
35          amount = staffList[count].pay(); // polymorphic
36
37          if (amount == 0.0)
38              System.out.println ("Thanks!");
39          else
40              System.out.println ("Paid: " + amount);
41
42          System.out.println ("-----");
43      }
44  }
45 }
```



StaffMember.java

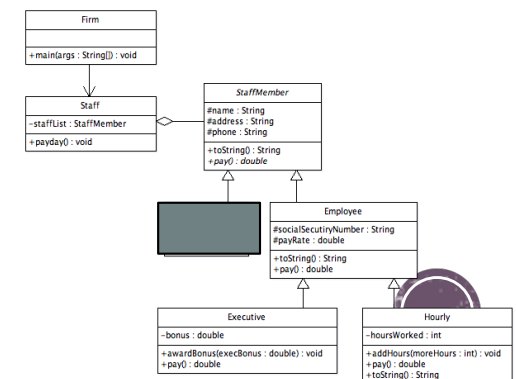
```
1  /**
2   * Represents a generic staff member
3   * @author Java Foundations
4   */
5  abstract public class StaffMember {
6      protected String name;
7      protected String address;
8      protected String phone;
9
10     /**
11     * Constructor: Sets up this staff member using the specified information.
12     */
13     public StaffMember (String eName, String eAddress, String ePhone) {
14         name = eName;
15         address = eAddress;
16         phone = ePhone;
17     }
18
19     /**
20     * Derived classes must define the pay method for each type
21     */
22     public abstract double pay();
23
24 }
```

For toString() see the book ...



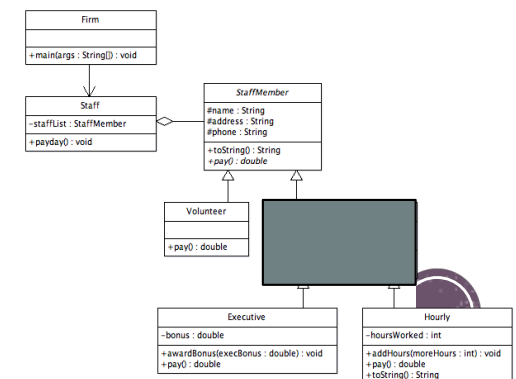
Volunteer.java

```
1 /**
2  * Represents a staff member that works as a volunteer.
3  * @author Java Foundations
4  */
5 public class Volunteer extends StaffMember {
6     /**
7      * Constructor: Sets up this volunteer using the specified information.
8      */
9     public Volunteer (String eName, String eAddress, String ePhone) {
10         super (eName, eAddress, ePhone);
11     }
12
13     /**
14      * @return a zero pay value for this volunteer.
15      */
16     public double pay() {
17         return 0.0;
18     }
19 }
--
```



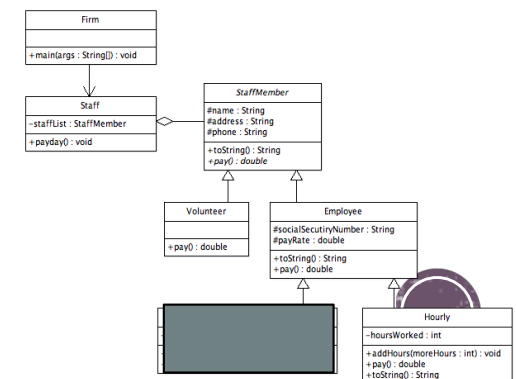
Employee.java

```
1  /**
2   * Represents a general paid employee.
3   * @author Java Foundations
4   */
5  public class Employee extends StaffMember {
6      protected String socialSecurityNumber;
7      protected double payRate;
8
9      /**
10     * Constructor: Sets up this employee with the specified information.
11     */
12     public Employee (String eName, String eAddress, String ePhone,
13                     String socSecNumber, double rate) {
14         super (eName, eAddress, ePhone);
15
16         socialSecurityNumber = socSecNumber;
17         payRate = rate;
18     }
19
20     /**
21     * @return the pay rate for this employee.
22     */
23     public double pay() {
24         return payRate;
25     }
26 }
```



Executive.java

```
1  /**
2   * Represents an executive staff member, who can earn a bonus.
3   * @author Java Foundations
4   */
5  public class Executive extends Employee {
6      private double bonus;
7
8      /**
9       * Constructor: Sets up this executive with the specified information
10     */
11     public Executive (String eName, String eAddress, String ePhone,
12                     String socSecNumber, double rate) {
13         super (eName, eAddress, ePhone, socSecNumber, rate);
14         bonus = 0; // bonus has yet to be awarded
15     }
16
17     /**
18     * Computes and returns the pay for an executive, which is the
19     * regular employee payment plus a one-time bonus.
20     */
21     public double pay() {
22         double payment = super.pay() + bonus;
23
24         bonus = 0;
25
26         return payment;
27     }
28 }
```



Hourly.java

```
1  /**
2   * Represents an employee that gets paid by the hour.
3   * @author Java Foundations
4   */
5  public class Hourly extends Employee {
6      private int hoursWorked;
7
8      /**
9       * Constructor: Sets up this hourly employee using the specified information.
10     */
11     public Hourly (String eName, String eAddress, String ePhone,
12                   String socSecNumber, double rate) {
13         super (eName, eAddress, ePhone, socSecNumber, rate);
14         hoursWorked = 0;
15     }
16
17     /**
18     * Computes and returns the pay for this hourly employee.
19     */
20     public double pay() {
21         double payment = payRate * hoursWorked;
22
23         hoursWorked = 0;
24
25         return payment;
26     }
27 }
```

