Polymorphism

- Polymorphism means *having many forms*.
- How does this related to object oriented programming?
- Consider the following reference:

```java
ChessPiece bishop;
```

The bishop variable may be used to reference an object of type `ChessPiece` (an instantiation of the `ChessPiece` class).

- It is also possible however, to have this variable reference a compatible type. That is, the relationship between a reference variable and the object it refers to is more flexible, and need not be exactly the same type.

---

Binding

- Since a polymorphic reference can refer to different types of objects over time the specific method it invokes can also change from one invocation to the next.
- At a certain point, the method invocation is *bound* to the method definition, that is, a commitment is made to execute certain code.
- Though *binding* often occurs at compile-time, the binding of a method invocation to its definition cannot be made till run time for a polymorphic reference.
- Consider

```java
obj.doIt();
```

If `obj` is polymorphic, it can refer to different types of objects at different times, thus calling a different definition of `doIt()` each time it is invoked.

---

Late or Dynamic Binding

- For polymorphic references, the method definition that is used depends on the object that is being referred to by the reference variable at that moment.
- This binding decision cannot be made until run time and is thus called *late or dynamic* binding.
- Though it is less efficient for bindings to occur at run time rather than compile time, it is considered to be an acceptable overhead given the flexibility of a polymorphic reference.
- There are two ways to create a polymorphic reference:
  1. using inheritance
  2. using an interface
Polymorphism via Inheritance

- A reference variable of a particular type can refer to any object of that class or any of its subclasses.
- Example: if a Horse is a subclass of Mammal, then a Mammal reference can be used to refer to an object of the Horse class.

```
Mammal pet;
Horse stallion = new Horse();
pet = stallion; //a valid assignment
```

- The reverse is also allowed, but would require an explicit cast, and is generally considered to be more prone to problems (the Horse has all the functionality of a Mammal, but the reverse is not necessarily true).

Polymorphic Reference throughout a Hierarchy

- This relationship works throughout a class hierarchy.

```
        Animal
         |  
 Reptile  Bird
         |  
  Snake  Lizard  Parrot  Horse  Rat
```

![Figure 1: Mammal and Horse are both subclasses of Animal.](image)

- If Mammal is derived from Animal, the following assignment is valid:

```
Animal creature = new Horse();
```

- At any time, the creature reference may refer to an Animal, Mammal, or Horse object.
- If all three of these classes implement the method move in a different way, the invocation `creature.move();` cannot be determined until run time.

- Since Mammal and Animal are more general classes, they may be defined as abstract. They may still have polymorphic references.
- If the move method of the Mammal class is abstract, and is uniquely defined in the Horse, Dog and Whale class, a Mammal reference can be used to execute any of them.

Polymorphism and the Object Class

- Of course, this then means that an Object reference (the parent class of all Java objects) can be used to refer to any object.
- Because an ArrayList is designed to hold Object references, it can hold any type of object, and in fact, can hold several types of objects at one time.
Hierarchy of Employees

Firm
- main(args: String[]): void

Staff
- staffList: StaffMember[]
- payday(): void

Volunteer Employee
- socialSecurityNumber: String
- payRate: double

Executive
- pay(): double

Hourly
- pay(): double
- addHours(moreHours: int): void

StaffMember
- name: String
- address: String
- phone: String

The Firm class contains a main driver that creates a Staff of employees and invokes the payday method.

```java
public class Firm {
    public static void main(String[] args) {
        Staff personnel = new Staff();
        personnel.payday();
    }
}
```

The Staff class definition

- The Staff class maintains an array of StaffMember objects.

```java
abstract public class StaffMember {
    protected String name;
    protected String address;
    protected String phone;
    public StaffMember(String n, String a, String p) {
        name = n;
        address = a;
        phone = p;
    }
    public String toString() {
        String result = "Name: " + name + "\n";
        result += "Address: " + address + "\n";
        result += "Phone: " + phone;
        return result;
    }
    abstract public double pay();
}
```

- The StaffMember class is made abstract because it is generic and likely will not need to be instantiated.

```java
public class Executive extends StaffMember {
    public Executive(String n, String a, String p) {
        super(n, a, p);
        // Set initial bonus
        awardBonus(500.00);
    }
    public void addHours(int moreHours) {
        // Implement addHours for Executive
    }
}
```

- The pay method will differ for each subclass of StaffMember and is therefore made abstract, forcing it to be defined by the subclasses (unless they themselves are abstract).

```java
public class Hourly extends StaffMember {
    public Hourly(String n, String a, String p) {
        super(n, a, p);
    }
    public double pay() {
        // Implement pay for Hourly
    }
}
```
The Abstract pay Method

- By defining pay abstractly in a StaffMember, the payday method of Staff can polymorphically pay each employee.

```java
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("---------------------");
    }
}
```

The Volunteer Class

- The Volunteer Class inherits from the StaffMember class.
- Because it is not an abstract class, it must override the abstract method pay in StaffMember and provide a definition.

```java
public class Volunteer extends StaffMember
{
    public Volunteer (String n, String a, String p)
    {
        super(n, a, p);
    }
    public double pay()
    {
        return 0.0;
    }
}
```
- Since volunteers are not paid, the return amount of the pay method is zero.

The Employee Class

- The Employee class also inherits from StaffMember and overrides the pay method.

```java
public class Employee extends StaffMember {
    protected String socialSecurityNumber;
    protected double payRate;

    public Employee (String n, String a, String p,
                     String ssn, double r) {
        super(n, a, p);
        socialSecurityNumber = ssn;
        payRate = r;
    }

    public String toString() {
        String result = super.toString();
        result += "Social Security Number: "
                  + socialSecurityNumber;
        return result;
    }

    public double pay() {
        return payRate;
    }
}
```
- Unlike StaffMember, the Employee class can be instantiated (even though it too is somewhat generic).

The Hourly Class

- The Hourly class extends Employee, overriding both pay and toString methods.
- The method addHours is added but cannot be accessed polymorphically (an explicit cast is required) because it’s not in other classes of the hierarchy.

```java
public class Hourly extends Employee {
    private int hoursWorked;

    public Hourly (String n, String a, String p,
                   String sssn, double r) {
        super(n, a, p, ssn, r);
        hoursWorked = 0;
    }

    public void addHours(int moreHours) {
        hoursWorked += moreHours;
    }

    public double pay() {
        double payment = payRate * hoursWorked;
        hoursWorked = 0; //reset hours
        return payment;
    }

    public String toString() {
        String result = super.toString();
        result += "Current hours: "+ hoursWorked;
        return result;
    }
}
```
**The Executive Class**

- The Executive class also extend Employee.
- It provides the additional method awardBonus (which cannot be accessed polymorphically).

```java
public class Executive extends Employee {
    private double bonus;
    public Executive (String n, String a, String p, String ssn, double r) {
        super(n, a, p, ssn, r);
        bonus = 0;
    }
    public void awardBonus(double execBonus) {
        bonus = execBonus;
    }
    public double pay() {
        double payment = super.pay() + bonus;
        bonus = 0;
        return payment;
    }
}
```

**Summary of Firm Program**

- The Firm class contains a main driver that creates a Staff of employees and invokes the payday method.
- The Staff class maintains an array of objects that represent individual staff members.
- The staffList array is declared to hold StaffMember references, but is actually filled with objects created from the subclasses. It is, therefore, filled with polymorphic references.
- The payday method of the Staff class scans through the list of of employees, printing their information and invoking their pay methods.
- The pay method is polymorphic since each class has its own version.
- The StaffMember class is abstract and serves as the ancestor of all the employee classes.
- The method pay is abstract and requires a definition by the descendant classes.
- The essence of polymorphism: each class knows best how it should handle a specific behaviour.

---

**Polymorphism via Interfaces**

- An interface name can be used as the reference variable type when referring to any object of any class that implements that interface.
- Given the following declaration of an interface:
  ```java
  public interface Speaker {
      public void speak();
      public void announce (String str);
  }
  ```
  Speaker can be used to declare an object reference variable
  ```java
  Speaker current;
  ```
- The reference variable `current` can be used to refer to any object of any class that implements the Speaker interface.
- Given
  ```java
  public class Philosopher implements Speaker {
      ...
  }
  ```
  we can make the following assignment
  ```java
  current = new Philosopher();
  ```

**Non polymorphic methods**

- Suppose the Philosopher class has a method called pontificate(). The following would generate a compile error.
  ```java
  Speaker special = new Philosopher();
  special.pontificate(); //compile error
  ```
- The compiler can determine only that the object is a Speaker, and therefore can guarantee only that the object can respond to the `speak` and `announce` methods.
- We would amend the code with a cast as follows:
  ```java
  ((Philosopher).special).pontificate(); //compile error
  ```