Polymorphism

- Literal meaning of Polymorphism is “many forms”. The ability to assume several different forms or shapes.
- Enables you to “program in the general” rather than “program in the specific.”
- Polymorphism enables you to write programs that process objects that share the same superclass as if they’re all objects of the superclass; this can simplify programming.
- Polymorphism is applied to two classes with inheritance relationship (i.e. Superclass and subclass relationship)
- The ability to use a superclass object variable to call various subclasses’ methods seamlessly and the proper behavior is automatically invoked.

Polymorphism Example (1)

- Example: Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes Fish, Frog and Bird represent the three types of animals under investigation.
  - Each class extends superclass Animal, which contains a method move and maintains an animal's current location
  - A program maintains an Animal array containing references to objects of the various Animal subclasses. To simulate the animals' movements

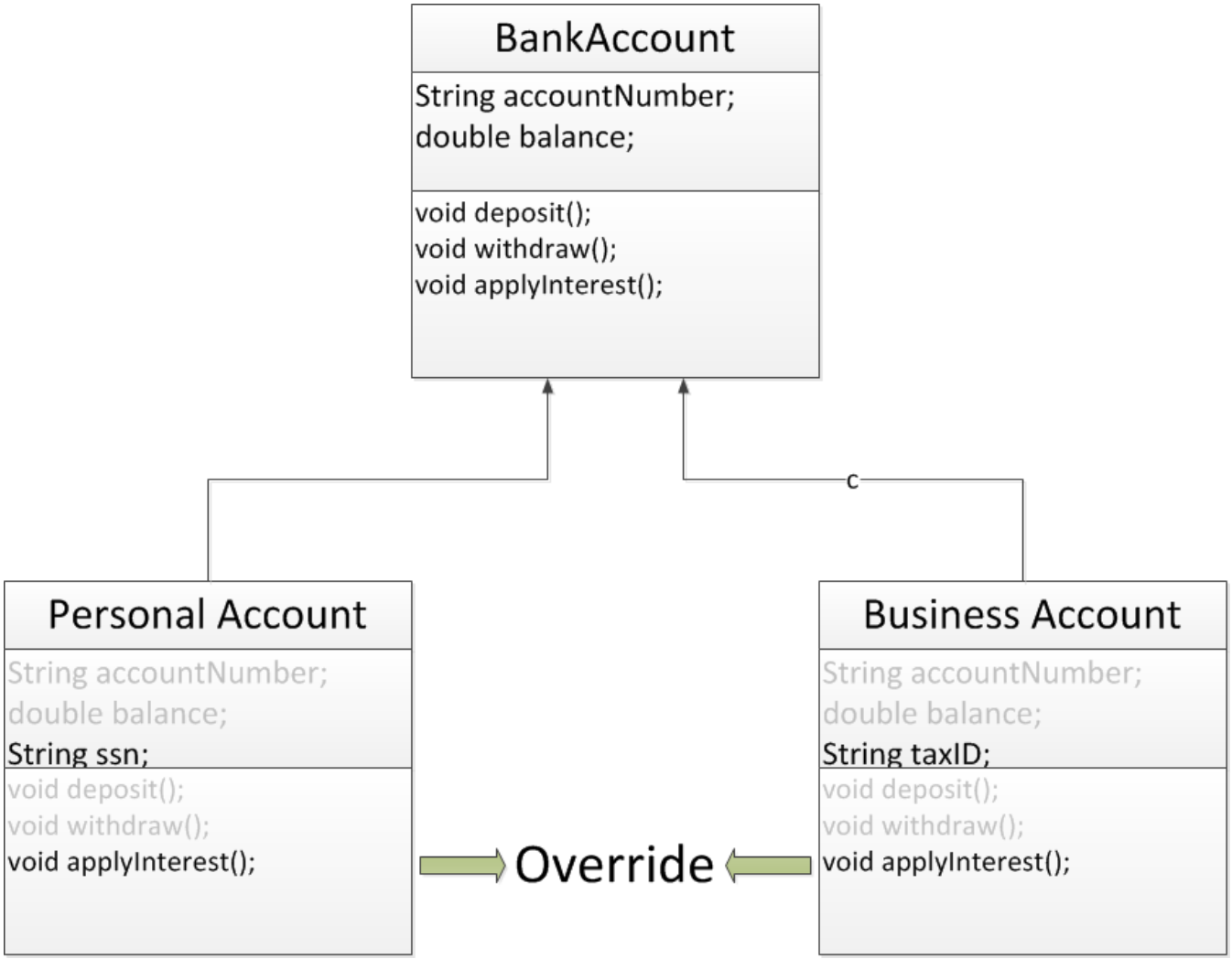
Polymorphism Example (1)

- Each specific type of `Animal` responds to a `move` message in a unique way:
  - a `Fish` might swim three feet
  - a `Frog` might jump five feet
  - a `Bird` might fly ten feet.
- The program issues the same `move` method to each animal object, but each object knows how to modify its location appropriately for its specific type of movement.
- Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of polymorphism.
- The same method to a variety of objects has "many forms" of results—hence the term polymorphism.

Polymorphism Example (2)

- Imagine an application where there are two different type of bank accounts: Personal Account and Business Account. The way the interest is calculated for each bank account is different. Personal accounts accrue fixed interest but Business accounts accrue variable interest based on the market value.
- By **overriding** the “applyInterest” methods in the Personal Account and Business Account subclasses, the interest can be calculated properly based on the type of account

Polymorphism Example (2)



Polymorphism Example (2)

- Polymorphism allows us to define an array of Bank Account and create an object of any of the subclasses (Personal Account or Business Account) for each element in the array and call the **applyInterest** method and expect the interest is calculated correctly base on each type of account

```
BankAccount[] accounts = new BankAccount[2];

accounts[0] = new BusinessAccount();
accounts[1] = new PersonalAccount();

accounts[0].applyInterest();
accounts[1].applyInterest();
```

Benefits of Polymorphism

- With polymorphism, we can design and implement systems that are easily *extensible*
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

When Polymorphism Works

- Polymorphism conditions that should be met:
  - The method call for a derived class object must be through a variable of a base class type
  - The method in the subclass must override the method in the superclass

Demonstrating Polymorphic Behavior

- In the next example, we aim a superclass reference at a subclass object.
  - Invoking a method on a subclass object via a superclass reference invokes the subclass functionality
  - The type of the referenced object, not the type of the variable, determines which method is called
- This example demonstrates that an object of a subclass can be treated as an object of its superclass, enabling various interesting manipulations.
- A program can create an array of superclass variables that refer to objects of many subclass types.
  - Allowed because each subclass object *is an* object of its superclass.

Demonstrating Polymorphic Behavior

- A superclass object cannot be treated as a subclass object, because a superclass object is *not* an object of any of its subclasses.
- The *is-a* relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.

Demonstrating Polymorphic Behavior

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.

- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type.
  - If that class contains the proper method declaration (or inherits one), the call is compiled.

- At execution time, the type of the object to which the variable refers determines the actual method to use.
  - This process is called dynamic binding.

Employee Example

- Simple payroll example that calculate the weekly income of employees based on the number of hours worked per week.
- There are two types of Employees: Contractor and Exempt.
- Contractors get paid 1.5 times for overtime hours
- Exempt Employee are not paid for the first 5 hours of overtime and the remainder is paid at normal hourly rate.

Employee Class

```
public class Employee {
    String name;
    float hourlyRate;
    float weeklyIncome;

    Employee(String name, float hourlyRate)
    {
        this.name = name;
        this.hourlyRate = hourlyRate;
    }

    //blank implementation
    void calculateWeeklyIncome(float hoursWorkedPerWeek)
    {

    }

    @Override
    public String toString()
    {
        return name + "'s weekly income for is $" + weeklyIncome;
    }
}
```

Exempt Class

```
public class Exempt extends Employee{

    //Constructor
    Exempt(String name, float hourlyRate)
    {
        super(name, hourlyRate);
    }

    //Calculate Weekly income based on the hours worked per week
    @Override
    void calculateWeeklyIncome(float hoursWorkedPerWeek)
    {
        //The first 5 hours of overtime is not paid
        //and the remainder of overtime is paid at normal hourly rate
       if (hoursWorkedPerWeek <= 45)
        {
            super.calculateWeeklyIncome(40);
        }
        else
        {
            weeklyIncome = (hourlyRate * (hoursWorkedPerWeek - 5));
        }
    }
}
```

Contractor Class

```
public class Contractor extends Employee{

    //Constructor
    Contractor(String name, float hourlyRate)
    {
        super(name, hourlyRate);
    }

    //Calculate Weekly income based on the hours worked per week
    @Override
    void calculateWeeklyIncome(float hoursWorkedPerWeek)
    {
        //weekly income is calculated for 1.5 times the hourly rate for overtime
        if (hoursWorkedPerWeek <= 40)
        {
            super.calculateWeeklyIncome(hoursWorkedPerWeek);
        }
        else
        {
            weeklyIncome = (hourlyRate * 40) + (hoursWorkedPerWeek - 40) * hourlyRate * 1.5f;
        }
    }

}
```

Payroll Main Class

```
public class PayrollMain {
    public static void main(String[] args)
    {

        // array of Employee              name        , hourly rate in $
        Employee[] staff = {new Exempt ("Alin Smith", 42.0f),
                            new Contractor ("Mary Jones", 55.0f)
                           };

        for (int i = 0 ; i < staff.length ; i++)
        {
        // calculate weekly income based on 50 hours of work per week
        staff[i].calculateWeeklyIncome(50.0f);
        System.out.println(staff[i]);
        }
    }
}
```