UBC OKANAGAN

COSC 121
Computer Programming II

Polymorphism

Part 2/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Outline

Previously:

- Multiple classes in one file
- Polymorphism
  - Rule 1: reference of supertype referring to subtype
  - Rule 2: can only access class members to known to reference
  - Rule 3: dynamic binding

Today: *more on polymorphism*

- Generic programming
- `instanceof` operator
- Object casting
- `Object`'s `equals` method

Polymorphism *part B*

Previously: The 3 Rules

RULE #1 A reference of a supertype can be used to refer to an object of a subtype.(not vice versa).

- `Human h = new Student;` //any student is also a human, but not vise versa
- *Useful: A reference of* ***`Object`*** *class can refer to any object.*

RULE #2 You can only access class members known to the reference variable

RULE #3 When invoking a method using a reference variable x, the method in the object referenced by x is executed, regardless of the type of x.

- If the method is not found in the object, parent classes are searched for the method (dynamic binding)

Generic Programming

Writing methods that are used generically for different types of arguments.

- Here we use polymorphism
- If a method's parameter type is a supertype (e.g., Object), you may pass an object to this method of any of the parameter's subtype (e.g., Student or String).
- When an object is used in the method, the particular implementation of the method of that object that is invoked is determined dynamically.

```
public class DynamicBindingTest2 {
    public static void main(String[] args) {
        m(new GradStudent());
        m(new Student());
        m(new Human());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}
```

```
class Human extends Object {
    public String toString() {return "Human";}
}

class Student extends Human {
    public String toString() {return "Student";}
}

class GradStudent extends Student {
}
```

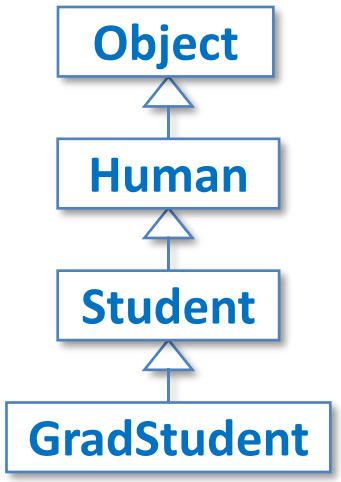
Method Matching vs. Dynamic Binding

Matching a method signature and dynamically binding a method implementation are two issues.

- **Dynamic binding** is used for overridden method and happens during the runtime. Here, Java tries to bind the method call to a method definition up a class-inheritance hierarchy
  - first searches a class for the method's implementation then its parent(s).
- **Static Binding:** In case of methods overloading, the compiler finds a matching method according method signature at compilation time.
  - Method signature = parameter type, number of parameters, and order of the parameters.

Method Matching vs. Dynamic Binding

```
public class DynamicBinding {
    public static void main(String[] args) {
        m(new GradStudent());
        m(new Student());
        m(new Human());
        m(new Object());
    }
    private static void m(Object object) {
        System.out.println(object.toString());
    }
    private static void m(Student s){
        System.out.println("No dynamic binding");
    }
}
class Human{
    public String toString(){return "Human";}
}
class Student extends Human{
    public String toString(){return "Student";}
}
class GradStudent extends Student{
}
```



Output

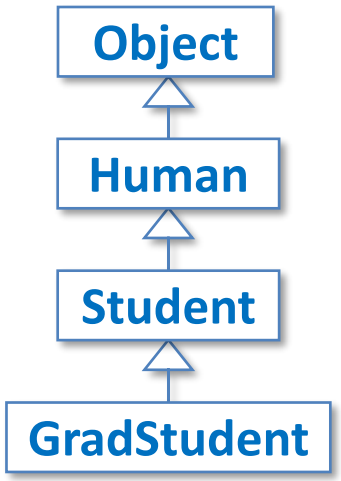
```
No dynamic binding
No dynamic binding
Human
java.lang.Object@4cc7014c
```

Method matching is based the argument type, i.e. the objects in this example. Then dynamic binding is used when method is not found in object.

Method Matching vs. Dynamic Binding

```
public class DynamicBinding2 {
    public static void main(String[] args) {
        Object j1 = new GradStudent();
        Object j2 = new Student();
        Object j3 = new Human();
        Object j4 = new Object();
        m(j1);
        m(j2);
        m(j3);
        m(j4);
    }
    private static void m(Object object) {
        System.out.println(object.toString());
    }
    private static void m(Student s){
        System.out.println("No dynamic binding");
    }
}
```

```
class Human{
    public String toString(){return "Human";}
}
class Student extends Human{
    public String toString(){return "Student";}
}
class GradStudent extends Student{
}
```



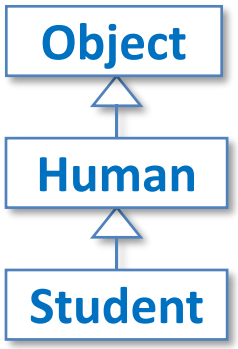
Output

```
Student
Student
Human
java.lang.Object@5b2b6037
```

Method matching is based the argument type, i.e. the references j1-j4 in this example, not their objects.

What is printed?

A. 1) object
B. 2) human
C. 2) student
D. 3) human
E. 3) student




```
public class Main {
   public static void main(String[] args) {
      Human h = new Student();
      print(h);
   }
   static void print(Object x) {
      System.out.print("1) object");
   }
   static void print(Human x) {
      System.out.print("2) ");
      x.foo();
   }
   static void print(Student x) {
      System.out.print("3) ");
      x.foo();
   }
}
```

```
class Human{
   void foo() {System.out.println("human");}
}
class Student extends Human{
   void foo() {System.out.println("student");}
}
```

Exercise

For each of the following, state whether the code is valid or invalid. Justify.

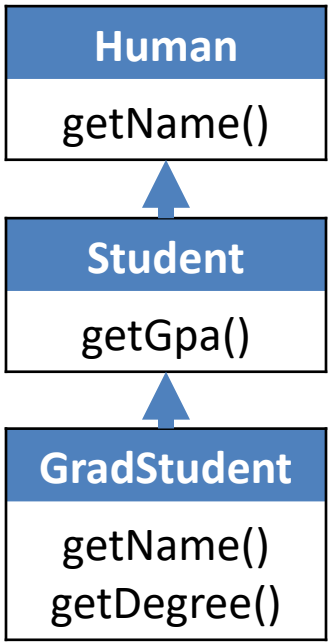
Q1: Human h = new GradStudent();

```
h.getName();     // Valid according to the 3 rules; getName from GradStudent is used
h.getDegree();   // Invalid. Violates rule #2; getDegree() is not part of Human.
h.getGpa();      // Invalid. Violates rule #2
```

Q2: Student s = new GradStudent();

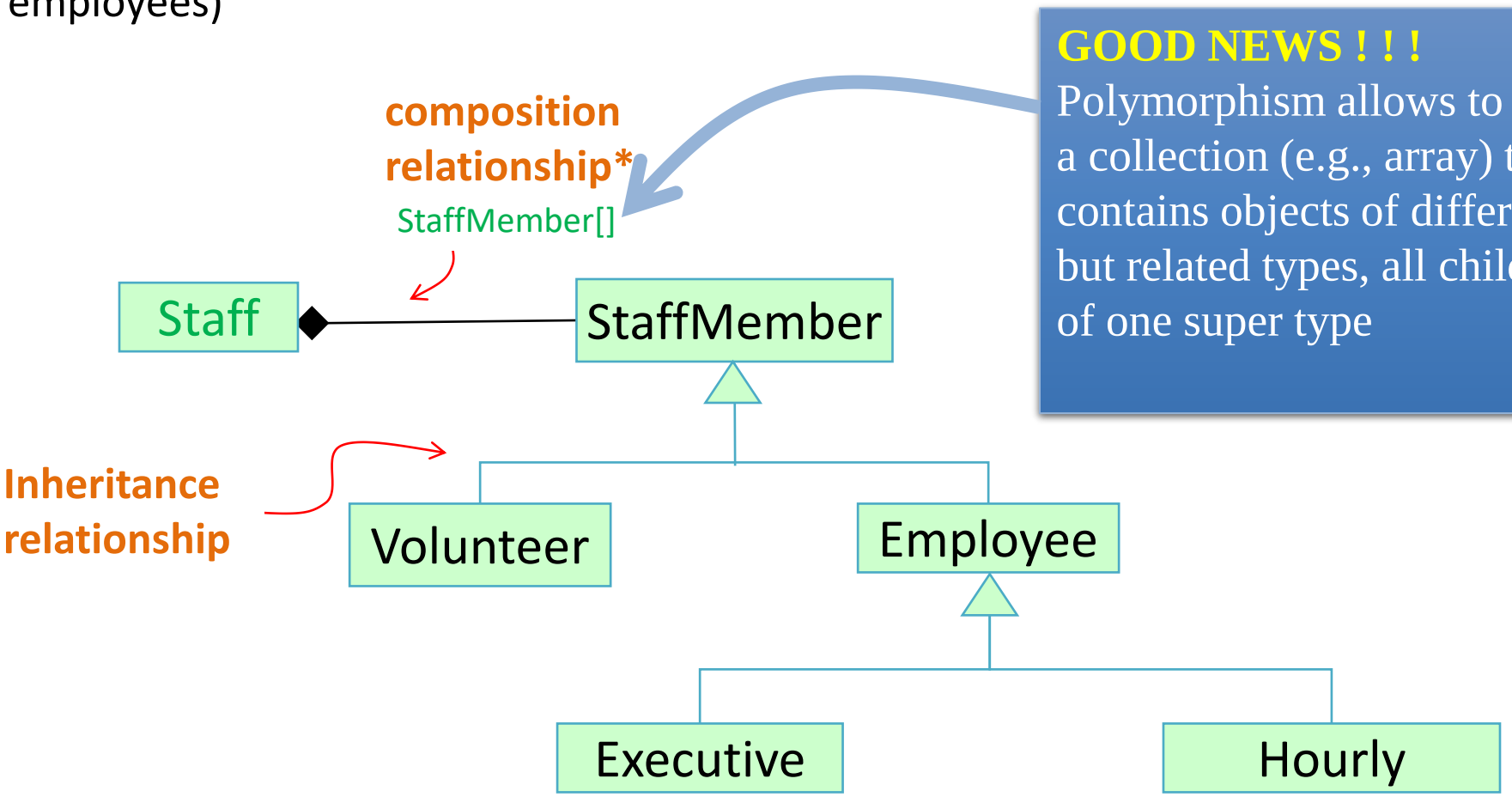
```
s.getName();     // Valid according to the 3 rules; getName() from GradStudent is used
s.getDegree();   // Invalid. Violates rule #2 (getDegree() is not part of Student)
s.getGpa();      // Valid according to the 3 rules; getGpa() from Student is used
```

Q2: Student s = new Human(); //Invalid –rule #1



Example

Write Java code to implement the following class diagram. It is required that `Staff` class will have a list of all its members (1 Volunteer, and 2 Executives and 3 Hourly employees)



GOOD NEWS ! ! !
Polymorphism allows to have a collection (e.g., array) that contains objects of different but related types, all children of one super type

* The composition relationship above means that Staff is composed of StaffMembers

Example, cont.

```
public class Staff {
    private StaffMember[] staffList;
    public Staff(){
            staffList = new StaffMember[6];
            staffList[0] = new Executive();
            staffList[1] = new Executive();
            staffList[2] = new Hourly();
            staffList[3] = new Hourly();
            staffList[4] = new Hourly();
            staffList[5] = new Volunteer();
            //other code
    }
}
```

```
public class StaffMember {
    //code
}
```

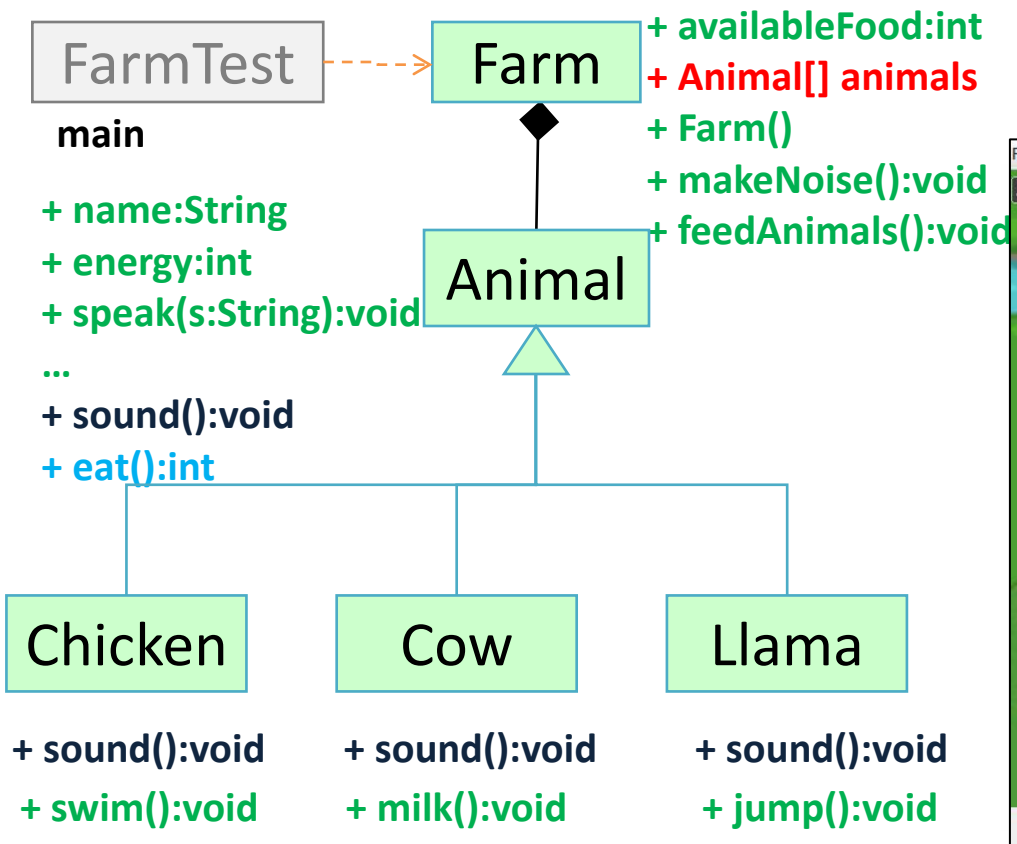
```
public class Volunteer extends StaffMember{
    //code
}
```

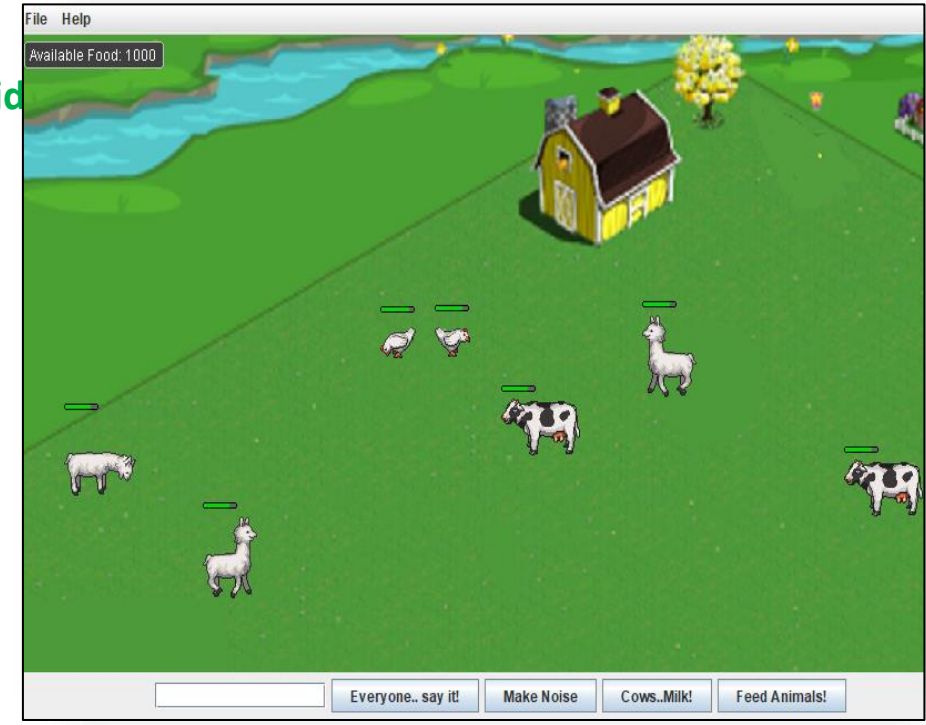
```
public class Employee extends StaffMember{
    //code
}
```

```
public class Executive extends Employee{
    //code
}
```

```
public class Hourly extends Employee{
    //code
}
```

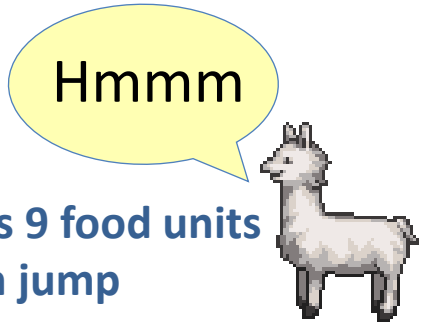
Project P1


FarmTest
main
Farm
+ availableFood:int
+ Animal[] animals
+ Farm()
+ makeNoise():void
+ feedAnimals():void
Animal
+ name:String
+ energy:int
+ speak(s:String):void
...
+ sound():void
+ eat():int
Chicken
+ sound():void
+ swim():void
Cow
+ sound():void
+ milk():void
Llama
+ sound():void
+ jump():void


File Help
Available Food: 1000
Everyone.. say it!
Make Noise
Cows..Milk!
Feed Animals!

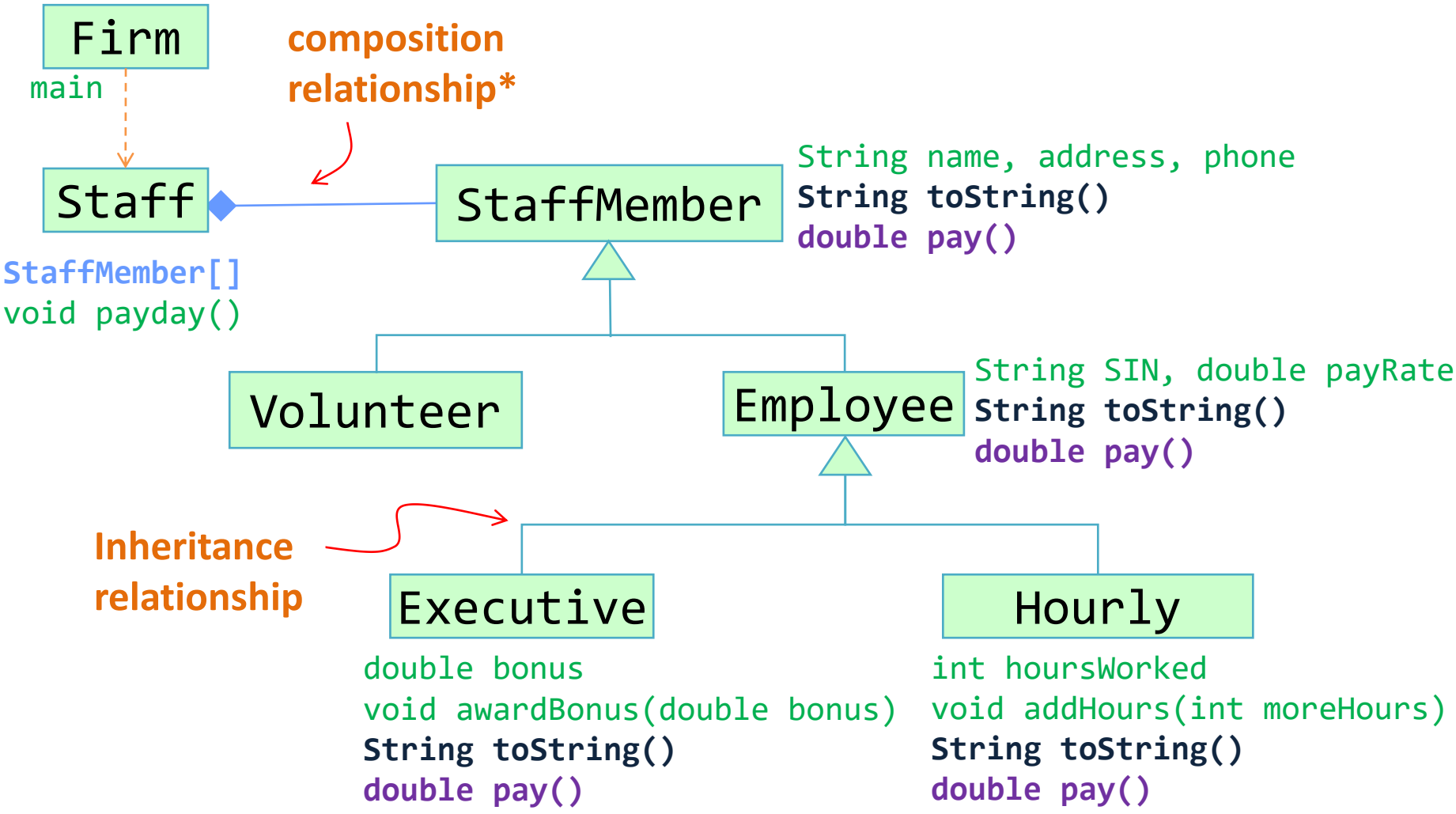

Cluck
Eats 5 food units
Can swim


Moo
Eats 20 food units
Gives milk


Hmmm
Eats 9 food units
Can jump

ToDo as a LAB Exercise E2

Write Java code to implement the following class diagram



* The composition relationship above means that Staff is composed of StaffMembers

Summary of Some Uses of Polymorphism

Potential Benefits of Polymorphism:

1- **Generic programming**:

Writing methods that are used generically (by means of polymorphism) for a wide range of object arguments

Example: a method that displays the name of an item

```
public void displayName(Human h){
        //arguments could be of the type: Human, Student, and GradStudent (rule #1)
    System.out.println(h.getName());    //see rule #3
}
```

2- **Creating an array that contains objects of different but related types, all children of one super type**

Example:

```
Human[] humanBeing = Human[3];
humanBeing[0] = new Human();
humanBeing[1] = new Student();
humanBeing[2] = new GradStudent();
```

Having all objects in one array allows to process them in a for loop

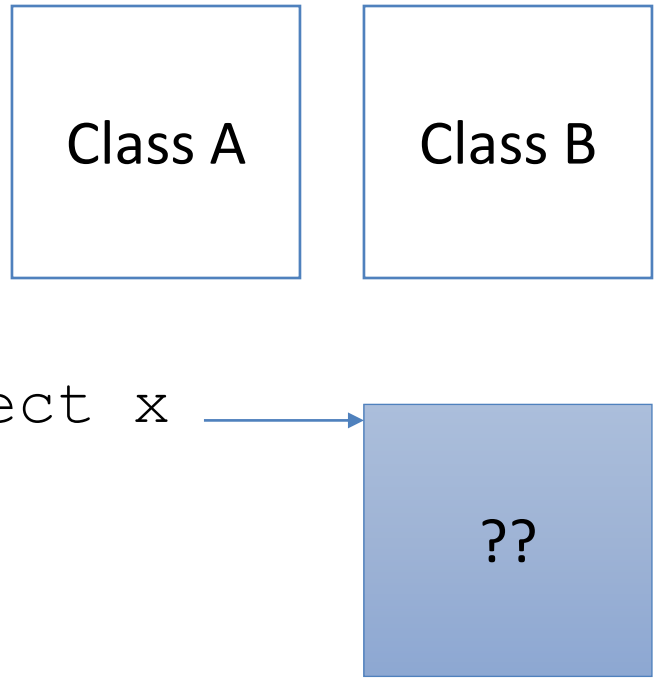
```
Ex:  for(Human h: humanBeing){
         System.out.println(h.getName());
     }
```

instanceof

`instanceof` Operator

Use the **`instanceof`** **operator** to test whether an **object** is an instance of a class.

Imagine that you have two classes, A and B. And you have a reference x of a supertype that points to an object of either A or B, but you don't know to which one. You may use this code:



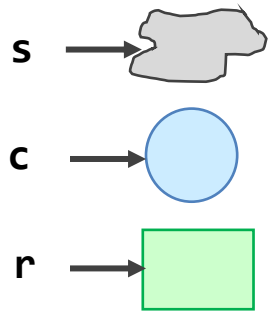

```
if(x instanceof A)
     System.out.print("Class A");
else
     System.out.print("Class B");
```

instanceof and inheritance

An object is an instance of its own class type and also of all its parent classes.

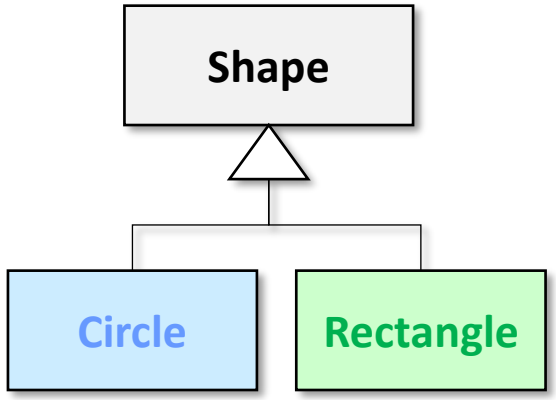
Example:

```
Shape s = new Shape();
Shape c = new Circle();
Shape r = new Rectangle();
```




```
s instanceof Circle   //false
c instanceof Circle   //true
r instanceof Circle   //false

s instanceof Shape    //true
c instanceof Shape    //true
r instanceof Shape    //true
```



Casting Objects and the `equals` Method

Implicit Objects

Casting objects refers to that one object reference can be typecast into another object reference.

Implicit casting:

```
Object obj = new Student(); // Implicit casting – this is Rule#1
```

The above statement is legal because an instance of Student is automatically an instance of Object.

Explicit Casting

Suppose you want to assign the **reference obj** to a variable of a **Student type** as follows:

```
Object obj = new Student(); //implicit casting,no error
Student s = obj;            //compile error
```

Why error ?

- Object obj is not necessarily an instance of Student. Even though you can see that obj is really a Student object, the compiler is not so clever to know it.

To fix this error

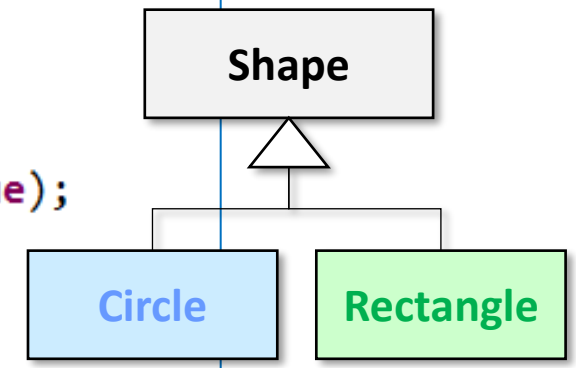
- tell the compiler that obj is a Student object, use explicit casting:

```
Student s =(Student)obj;//Explicit casting. No syntax error!
```

Example: Objects Casting + `instanceof`

The displayShapeInfo method displays the diameter if the object is a circle and the height & width if the object is a rectangle

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        // Display circle and rectangle properties
        Circle c = new Circle(1, "red", false);
        Rectangle r = new Rectangle(1, 1, "black", true);
        displayShapeInfo(c);
        displayShapeInfo(r);
    }
    public static void displayShapeInfo(Shape shp) {
        System.out.println(shp.getColor());
        if(shp instanceof Circle){
            Circle c = (Circle)shp;
            System.out.println("Diameter: " + c.getDiameter());
        } else if(shp instanceof Rectangle){
            Rectangle r = (Rectangle)shp;
            System.out.println("Width: " + r.getWidth());
            System.out.println("Height: " + r.getHeight());
        }
    }
}
```



Exercise

Consider the code below

```
class Human {}
class Student extends Human {}
```

Assume that

```
Human h1 = new Human();
Human h2 = new Student();
```

Which of these statements causes a compilation or a runtime error?

```
a) Student s = h2;
b) Student s = (Student) h2;
c) Student s = (Student) h1;
d) Student s = (Student) new Human();
e) Human h = h1;
```

CAUTION

For the casting to be successful, you must make sure that the **object to be cast is an instance of the subclass**.

If the superclass object is not an instance of the subclass, a runtime ClassCastException occurs.

Example:

- This code will generate a runtime error:

```
Object obj = new Object();
Student s = (Student)obj;   //runtime error. no compile error
```

The `Object`'s `equals` Method

The `==` operator is used to compare **the references** of the objects. Comparing two references for equality **does not** compare the contents of the objects referenced.

`public boolean equals(Object O)` is a method provided by the `Object` class. The default implementation uses == operator to compare two objects as follows:

```
public boolean equals(Object obj) {
  return this == obj;
}
```

It is expected that programmers will **override** this method so that it is used to compare the values of two objects.

The `Object`'s `equals` Method

Try the following code with the current implementation of the Circle class (slide 6):

```
Circle c1 = new Circle(5.2);
Circle c2 = new Circle(5.2);
System.out.println(c1.equals(c2));
```

The output is false as the two references c1 and c2 point to different objects. However, we can override the equals method in the Circle class to compare the values in the objects instead. The simplest implementation is as follows:

```
public boolean equals(Object o) {
    if (o instanceof Circle) {
        Circle c = (Circle)o;
        return this.radius == c.radius;
    } else
        return false;
}
```

and the value of c1.equals(c2) would be true.