CSC 133
Object-Oriented Computer Graphics Programming

OOP Concepts II – Inheritance & Polymorphism

Dr. Kin Chung Kwan

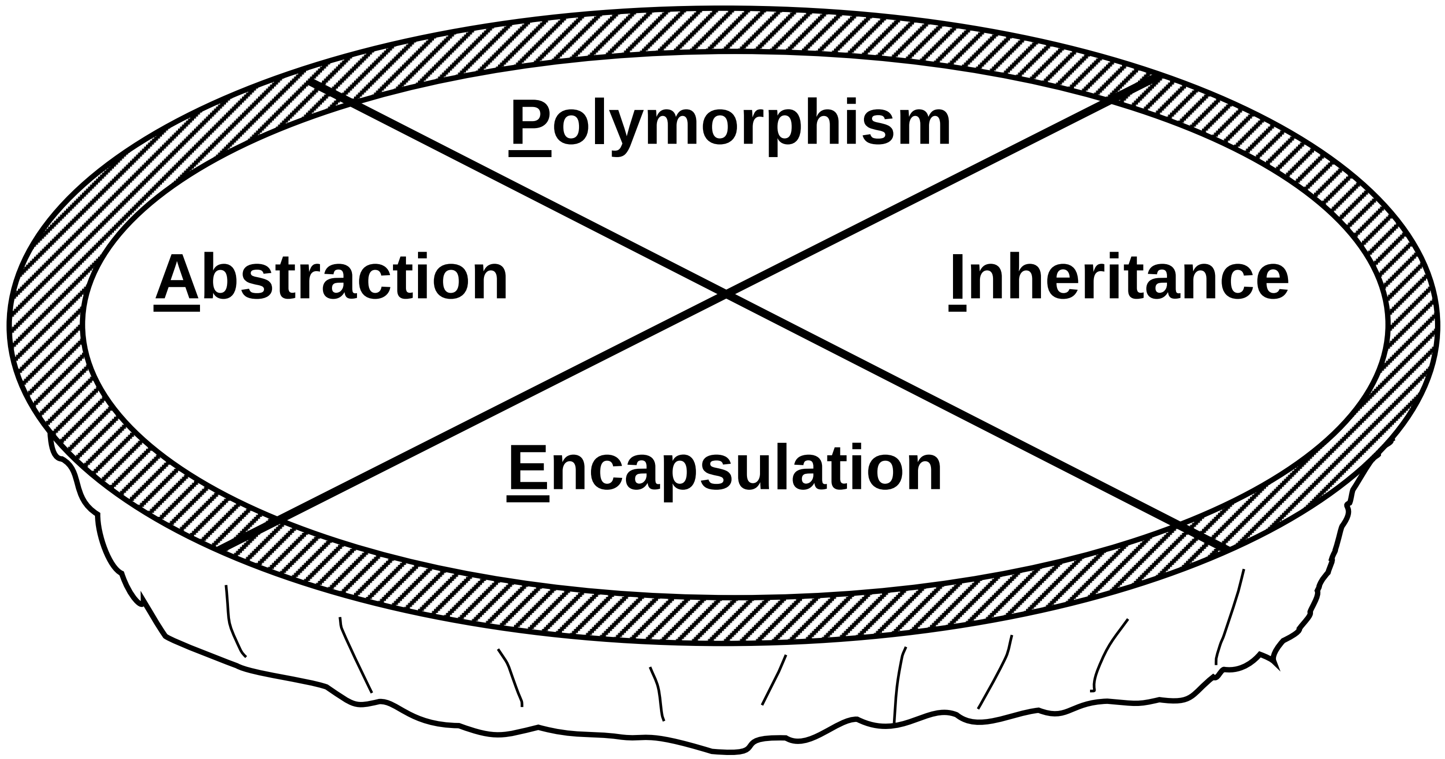
Spring 2023

Computer Science Department
California State University, Sacramento

SACRAMENTO STATE

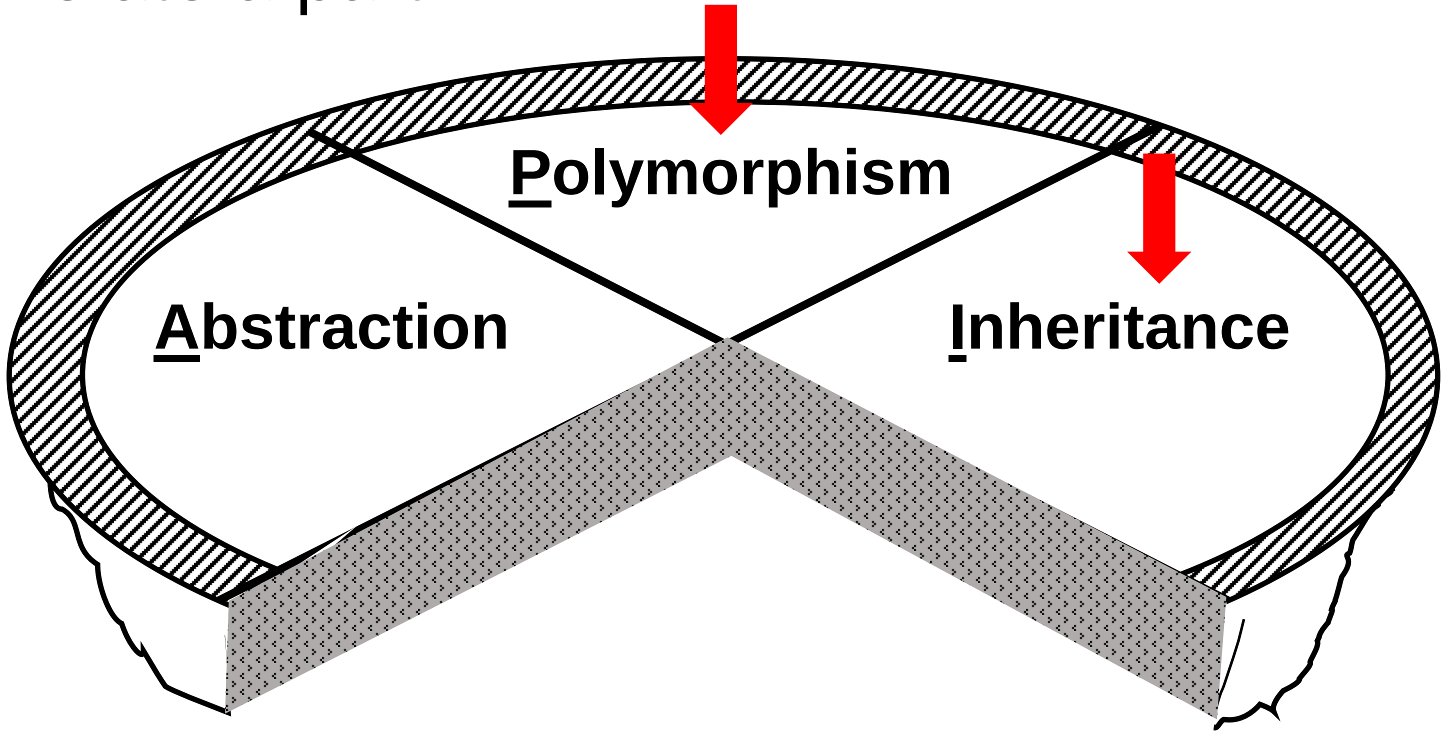
“A Pie”

Four distinct OOP Concepts (or Pillars)



Last week

We ate a part



Inheritance

It is the most useful thing in OO

What Is Inheritance?

- A specific kind of relationship between classes
- Various definitions:
    - Creation of a **hierarchy of classes**, where lower-level classes share properties of a common "parent class"
    - A mechanism for indicating that one class is "similar" to another but has specific differences
    - A mechanism for enabling properties (attributes and methods) of a "super class" to be propagated down to "sub classes"
    - Using a "base class" to define what characteristics are **common** to all instances of the class, then defining "derived classes" to define what is special about each subgrouping

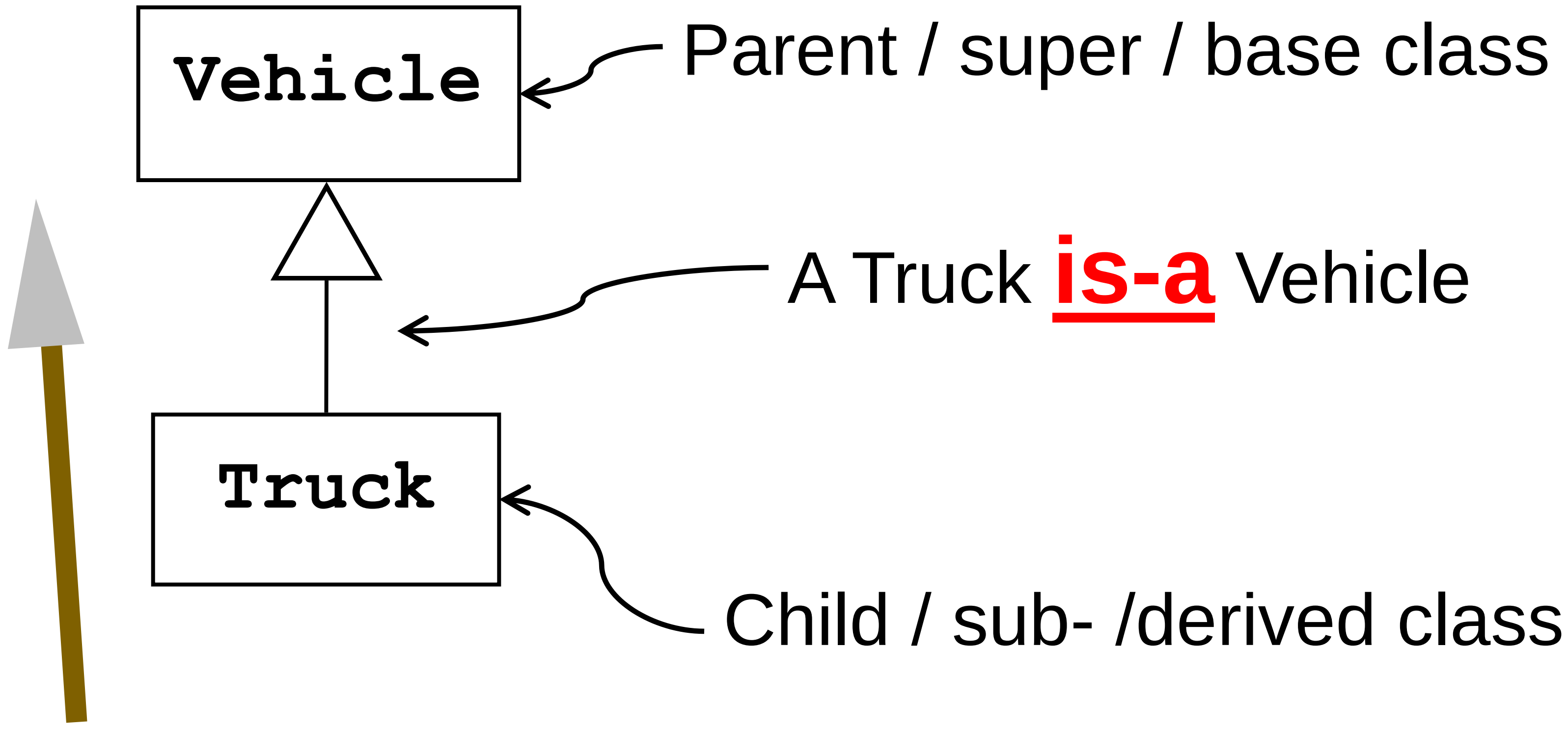
“IS-A” Relationship

“**is-a**” relationship.

- Child can do what parent do

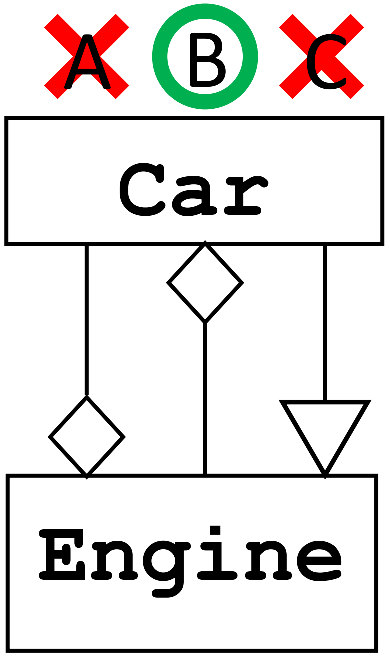
Teacher	Full-time teacher
Parent class	Child class
Is a class	is a teacher
Can teach	Can teach

Inheritance In UML



Question

If you can’t say “A is a B” (or “A is a kind of B”), it is **NOT** inheritance



An Engine “is a” Car ? X
A Car “is an” Engine ? X

A Car “<u>has-an</u>” Engine ✓

An Engine “<u>is a part of</u>” a Car ✓

Inheritance In Java

Specified with the keyword "extends" :

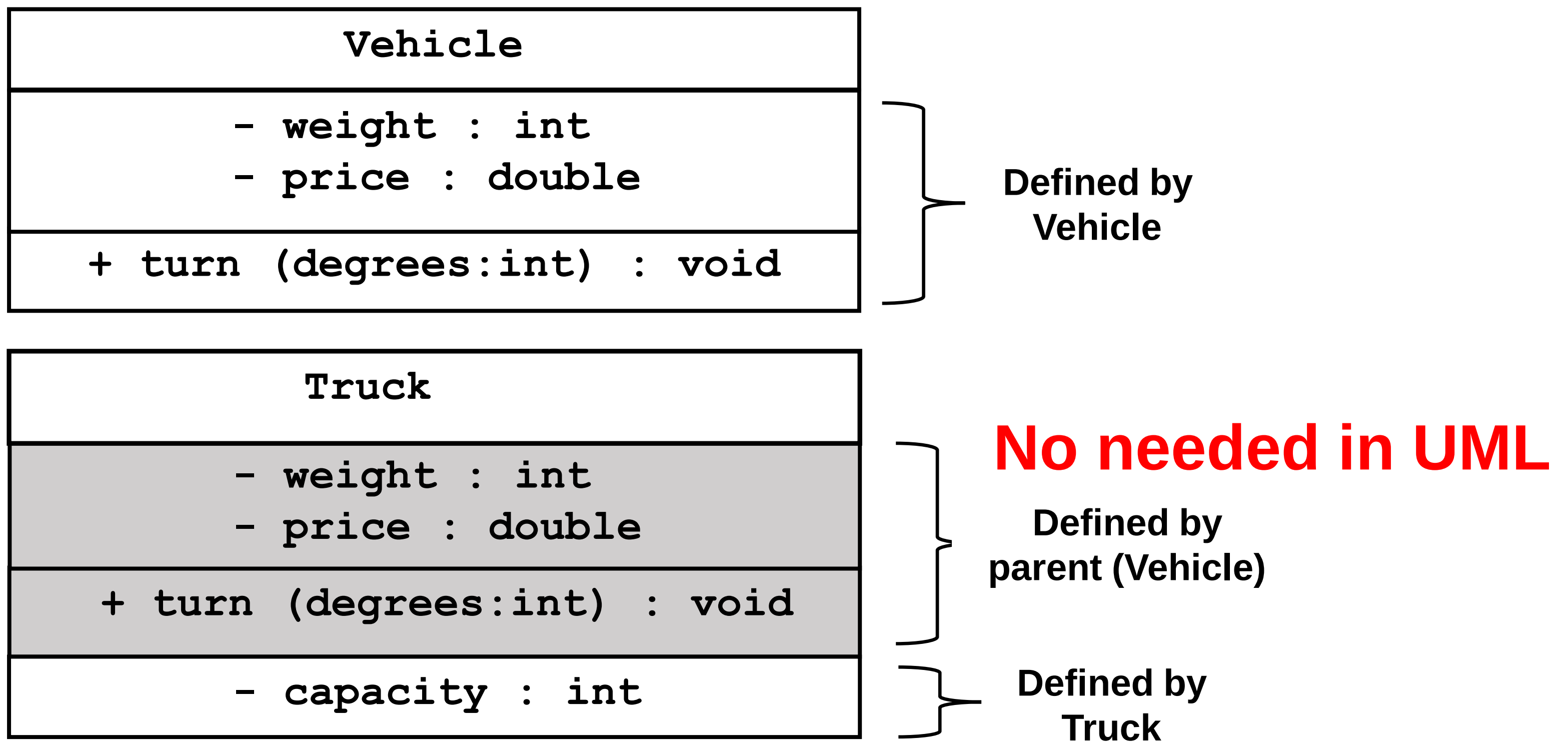
- Single "extends" allowed
- By default, extends Object

```
public class Vehicle {

  private int weight;
  private double price;
  //... other Vehicle data here

  public Vehicle ()
  { ... }

  public void turn (int direction)
  { ... }

  // ... other Vehicle methods here
}
```

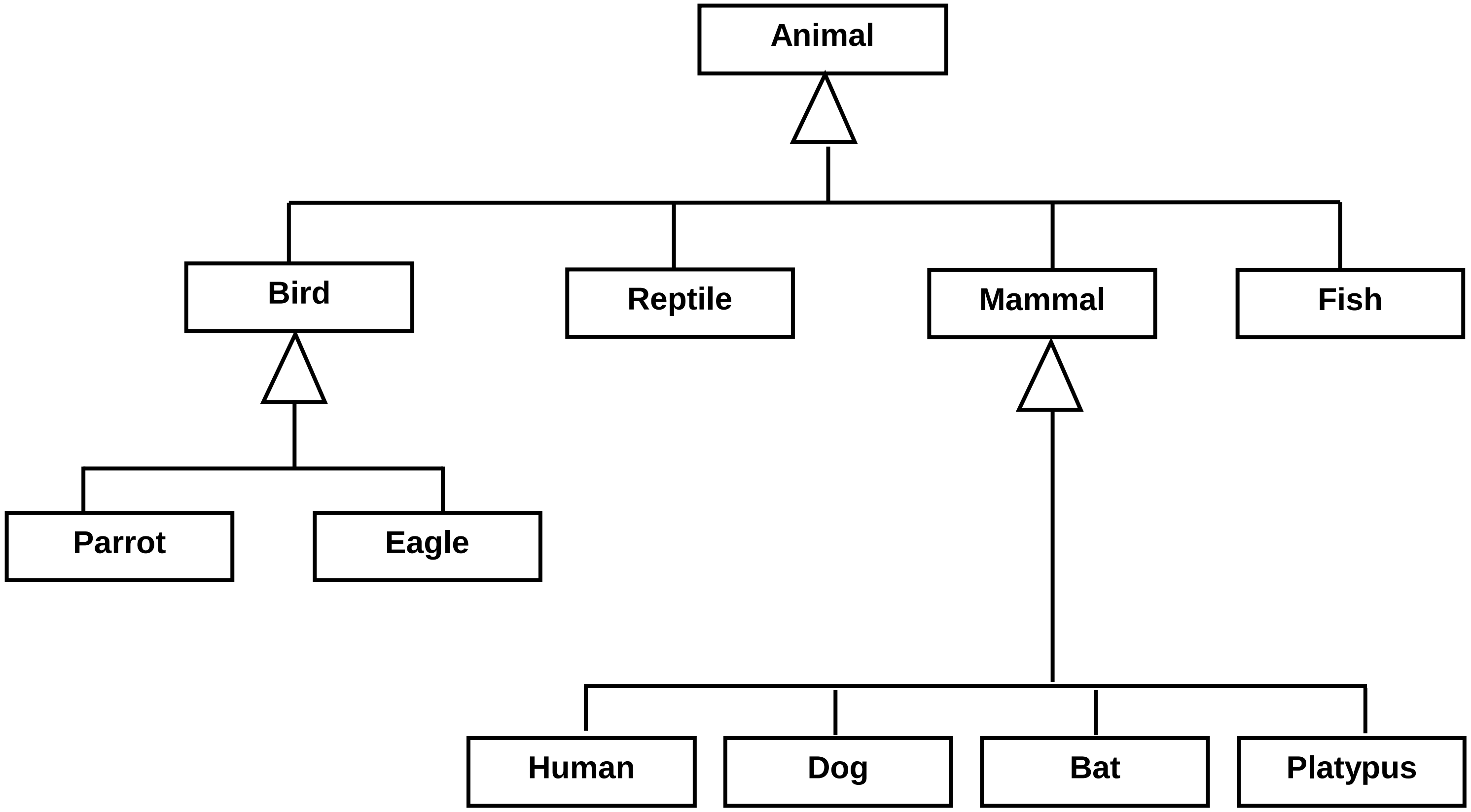
```
public class Truck extends Vehicle {
  private int capacity;
  //... other Truck data here

  public Truck ()
  { ... }

  // ... Truck-specific methods here
}
```

Effects of Inheritance

Child with have the codes of parent



Inheritance Hierarchies



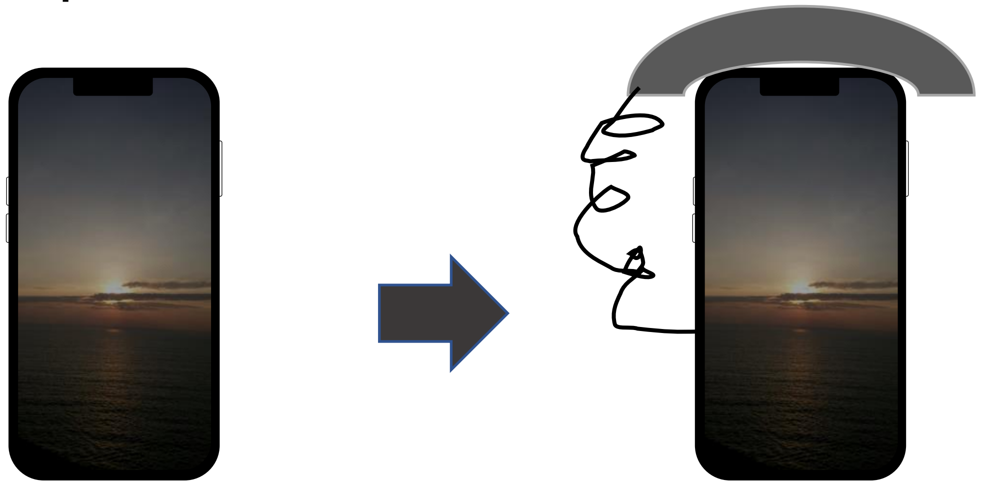
Typical Uses for Inheritance

- Extension
  - Define new behavior, and
  - Retaining existing behaviors
- Specialization
  - Modify existing behavior(s)
- Specification
  - Provide (“specify”) the implementation details of “abstract” behavior(s)

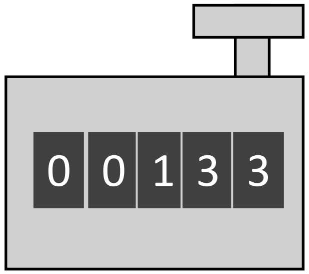
Inheritance for Extension

Define **new** behavior but

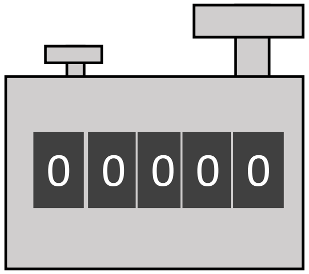
- Retains parent class's behaviors

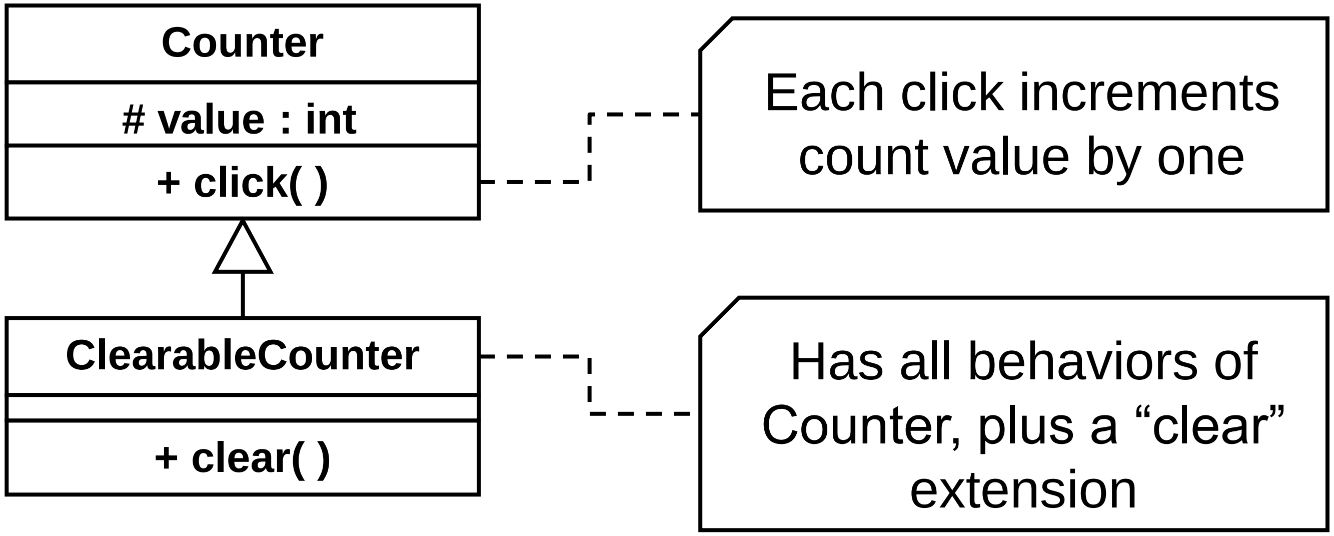


Extension Example



- Example: Counter
  - Parent class increments on each "click"
  - Extension adds support for "clearing" (resetting)




Counter
# value : int
+ click( )
ClearableCounter
+ clear( )
Each click increments count value by one
Has all behaviors of Counter, plus a "clear" extension

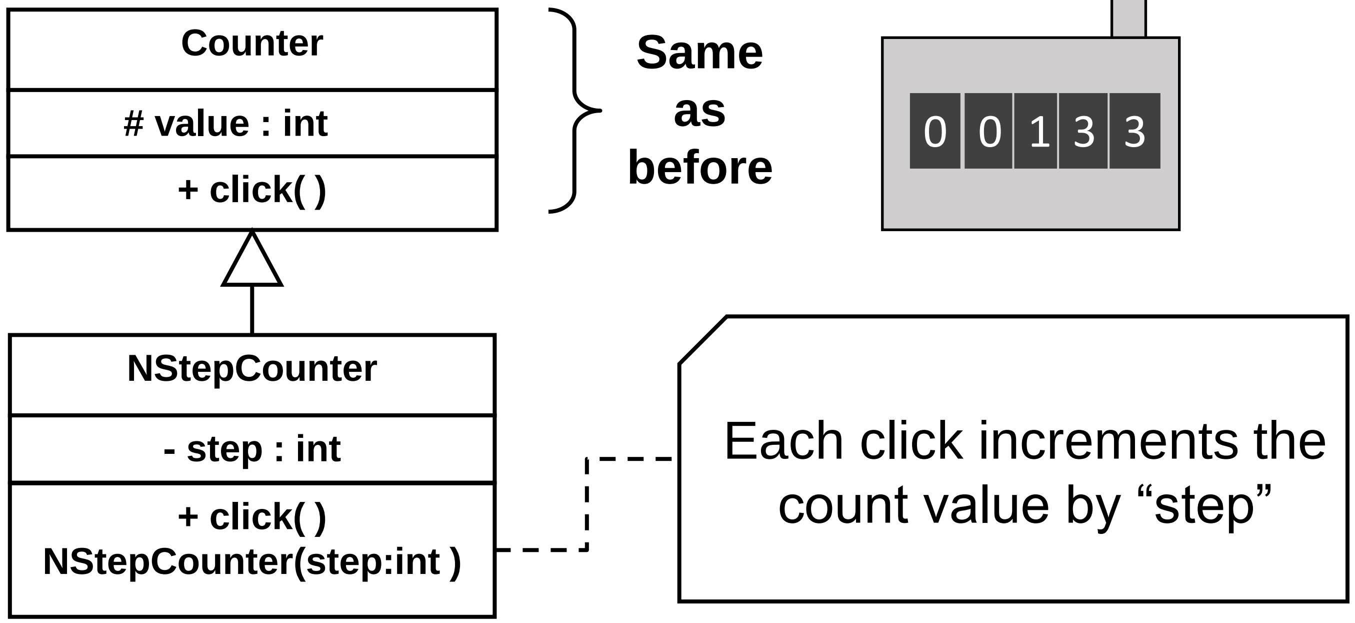
Code

```
 /** This class defines a counter which is incremented on each call
 to click().
  * The Counter has no ability to be reset.   */
 public class Counter {
   protected int value ;

   /** Increment the counter by one. */
   public void click() {
     value = value + 1;
   }
 }
/** This class defines a type with all the properties of a Counter, and
 *  which also has a "clear" function to reset the counter to zero. */
public class ClearableCounter extends Counter {

   // Reset the counter value to zero.  Note that this method can
   // access the "value" field in the parent because that field
   //  is defined as "protected".

  public void clear () {
    value = 0 ;
  }
}
```

Inheritance for Specialization

Modify **existing** behavior defined by parent

- Uses overriding to change the behavior
- Example: N-Step Counter



Inheritance for Specification

Used to specify (define) behavior **declared** (but not **defined**) by the parent

- Classes which declare but don't define behavior: Abstract Classes

- Methods which don't contain implementations: Abstract methods

This is abstraction in next lecture.

Method Overriding

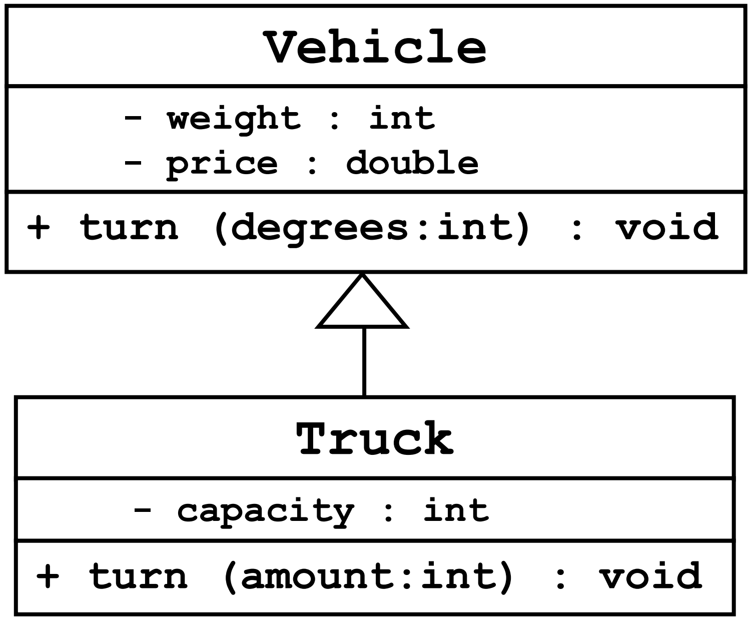
Method Overriding

- Inheritance leads to an interesting possibility:
  duplicate method declarations

- Occurs when a child class redefines an **inherited** method:
  - with same name, same parameters, same return type

- Child objects contain BOTH the parent method code and the child (overriding) method code

Method Overriding



Truck's turn(int) "***overrides***"

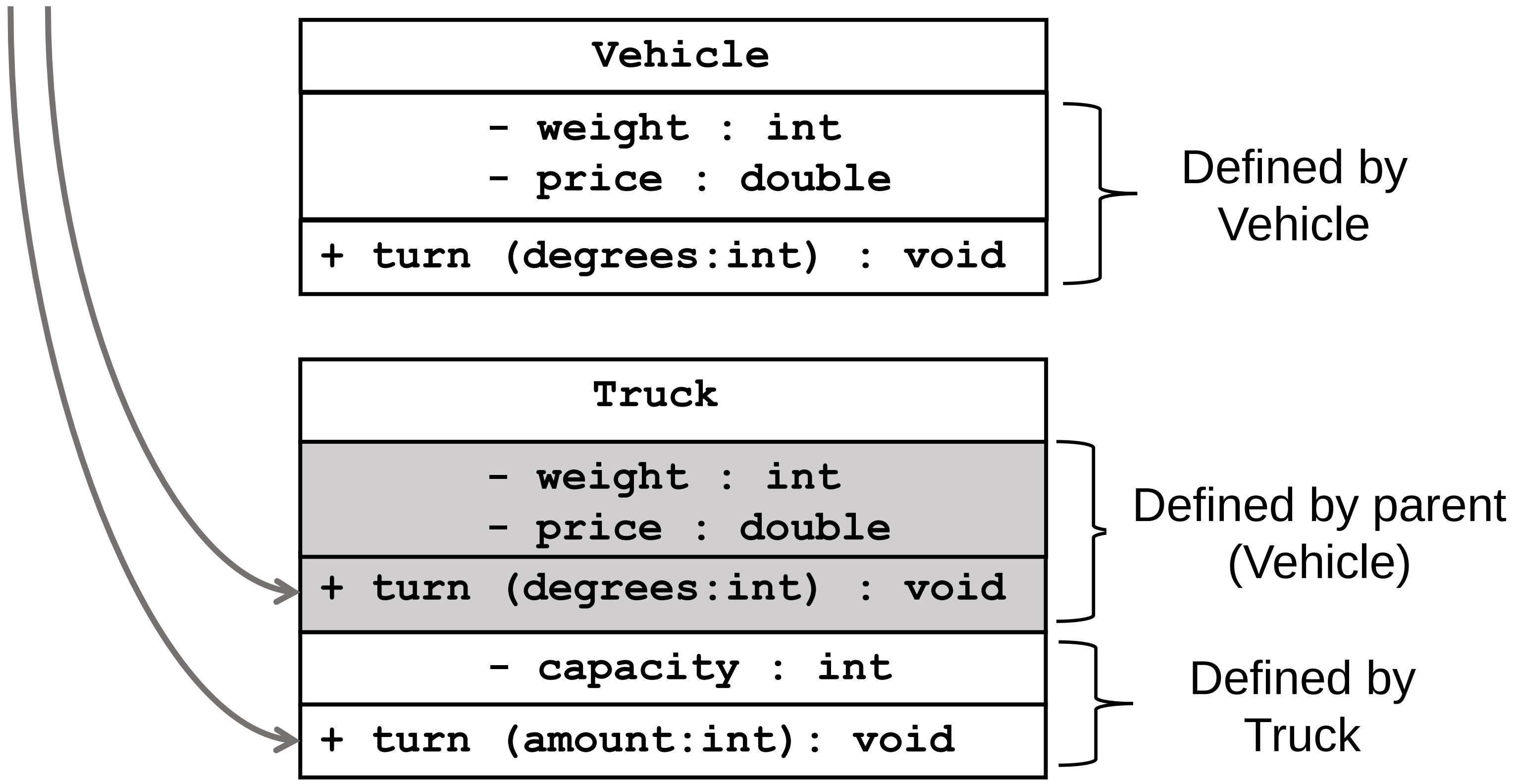
Vehicle's turn(int)

```
public class Vehicle {
   private int weight ;
   private double price ;

   public void turn (int degrees)
   { // some code to accomplish turning... }

   ...
}
```

```
public class Truck extends Vehicle {

   private int capacity;

   public void turn (int amount)
   { // different code to accomplish turning... }

   ...
}
```

Effects of Overriding

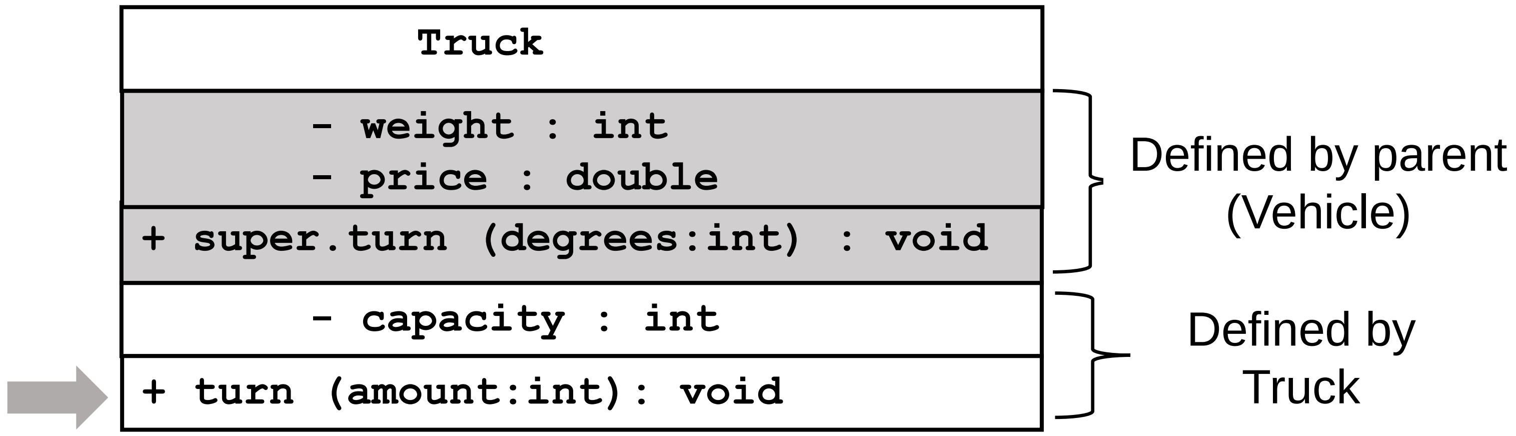
Two **`turn()`** bodies! Which one is invoked?



Function Calling

Call the method in child

- Always invokes the (overridden) child (for Java)
- Using "`super.xxx(...)`" to call the parent's one


Truck
- weight : int
- price : double
+ super.turn (degrees:int) : void
- capacity : int
+ turn (amount:int): void
Defined by parent (Vehicle)
Defined by Truck

Polymorphism

Polymorphism

Literally: from the Greek

poly *("many")* + ***morphos*** *("forms")*

Examples in nature:

- Carbon: graphite or diamond
- H_2O: water, ice, or steam
- Blood: A, B, AB, or O type

Polymorphism Example

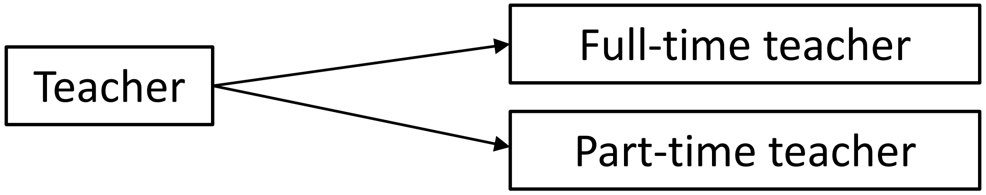
Same operation for various types of objects

```
kc.learnFrom( teacher )   vs   kc.learnFrom( student )
```

Same operation in a variety of ways

```
kc.teachCSC133( ) vs other.teachCSC133( )
```

A reference to different types



Overloading

Same name but different parameter types

- Not the same as “overriding”
- Can occur in the same class or split between parent/child classes

- Overloading examples, methods with:
  - Different numbers of parameters:

```
distance(p1);    distance(p1,p2);
```

  - Different parameter types:

```
computeStandings(int numTeams);
computeStandings(double average);
computeStandings(Hashtable teams);
```

Type of Polymorphism

1. **Static Polymorphism**

During compilation

- Polymorphic operator
- Polymorphic method

2. **Dynamic Polymorphism**

During runtime

- Polymorphic reference

Polymorphic Operator

- Static: detectable during compilation.
- The "+" can perform on different types of objects

```
int1 =  int2 + int3 ;
float1 =  float2 + float3 ;
```

- Coding:

```
int operator + (int obj) {...}
float operator + (float obj) {...}
```

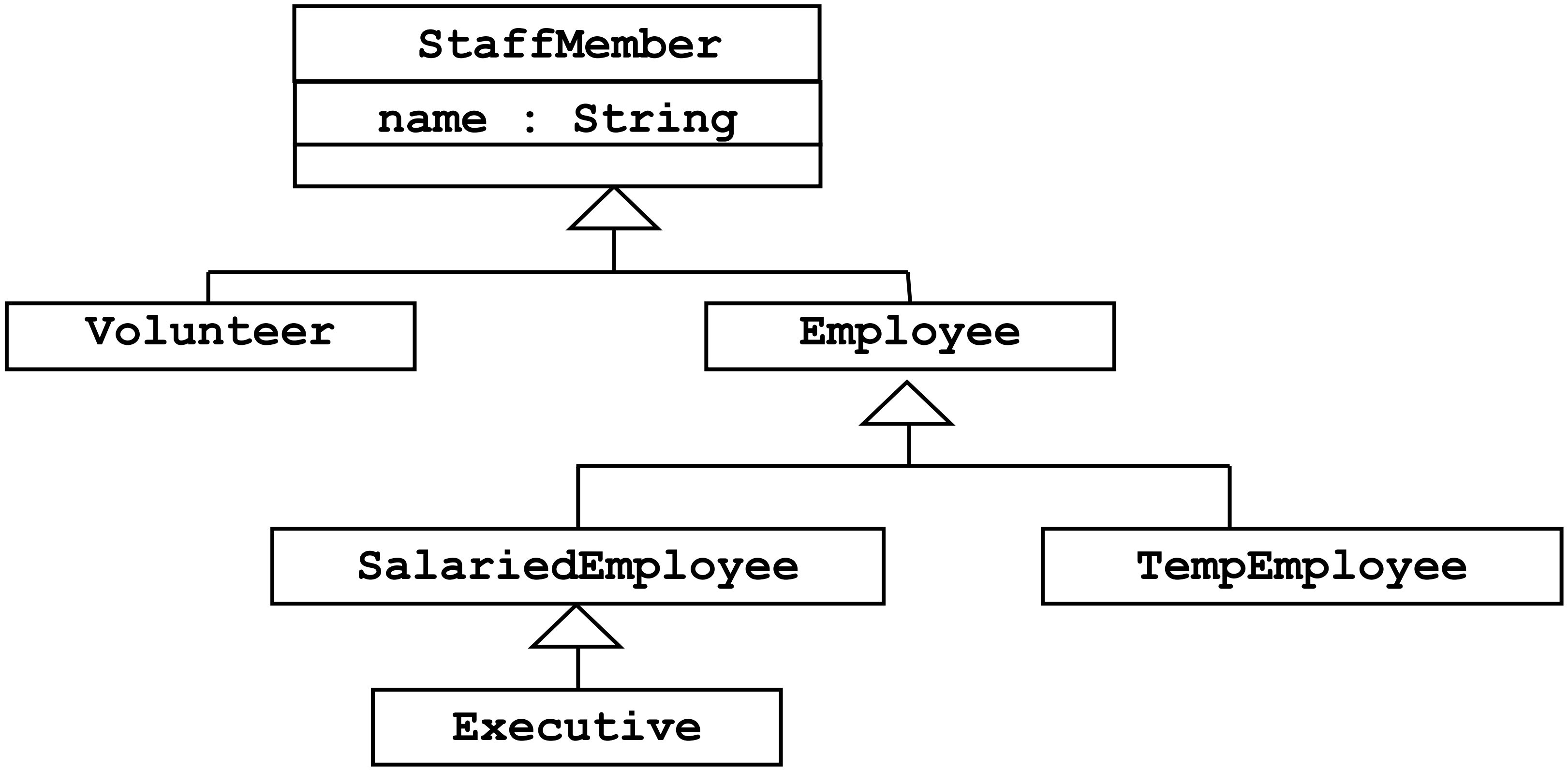
Polymorphic Methods

- Static: detectable during compilation.
- Same name, different parameters
- Example:

```
//return the distance to an origin
double distance (int x, int y) { . . . }
//return the distance between two points
double distance (int p1, int p2, int p3, int p4) { .
  . . }
//return the distance between two points
double distance (Point p1, Point p2) { . . . }
```

Dynamic Polymorphic

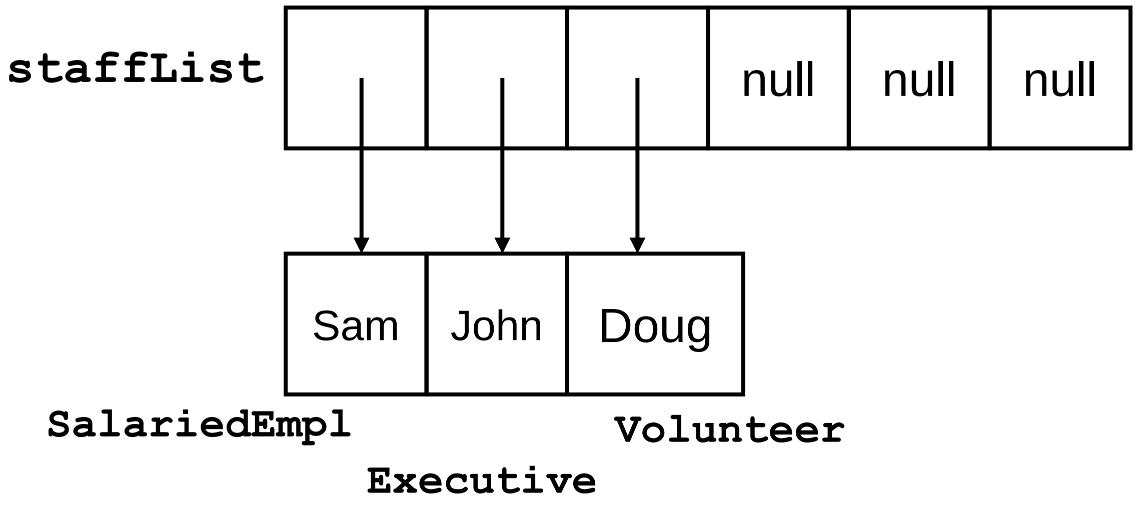
Consider the following class hierarchy:



Polymorphic References

A variable refer to different object types at runtime:

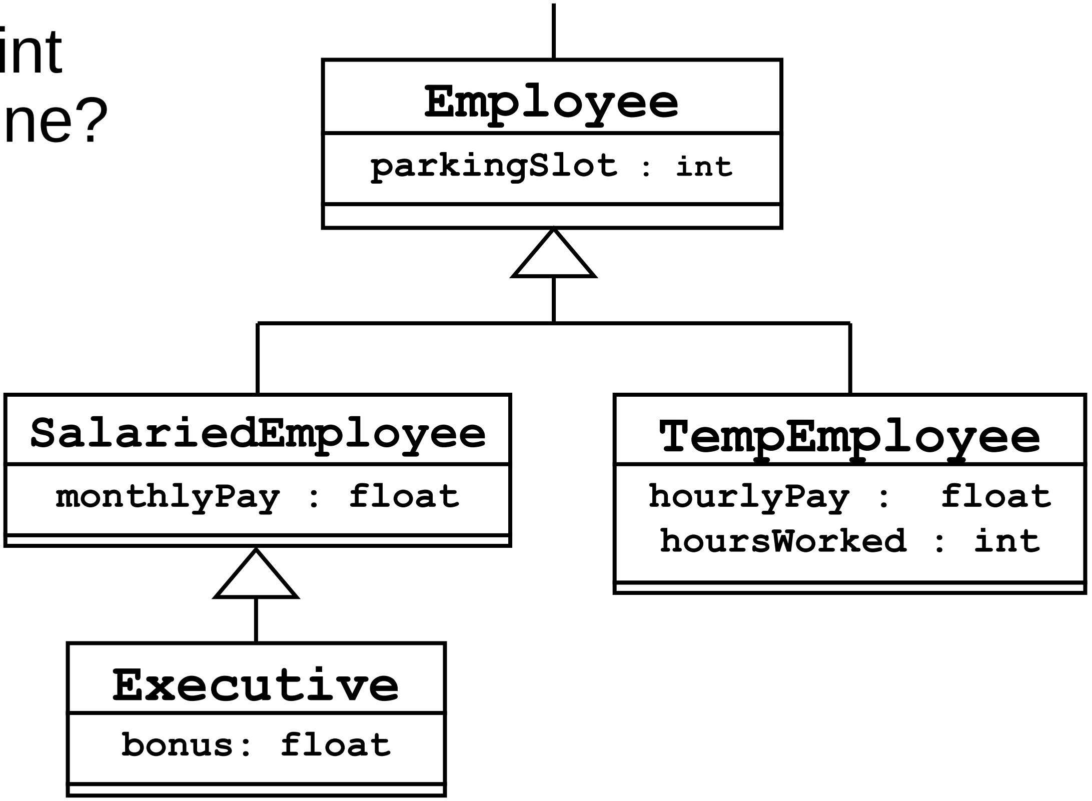
```
StaffMember [ ] staffList = new StaffMember[6];
staffList[0] = new SalariedEmployee (“Sam”);
staffList[1] = new Executive (“John”);
staffList[2] = new Volunteer (“Doug”);
```



Runtime Polymorphism

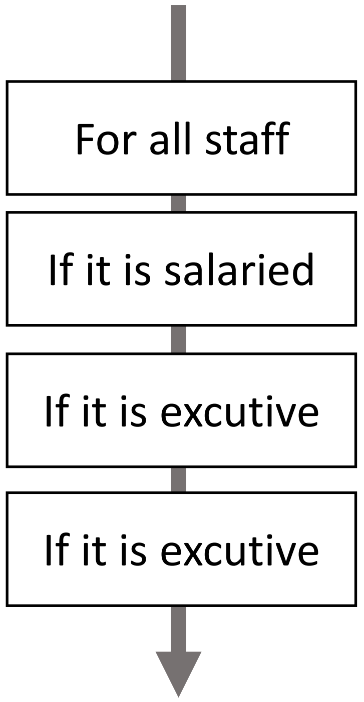
Consider this expanded version of the hierarchy

- What if we want to print paychecks for everyone?



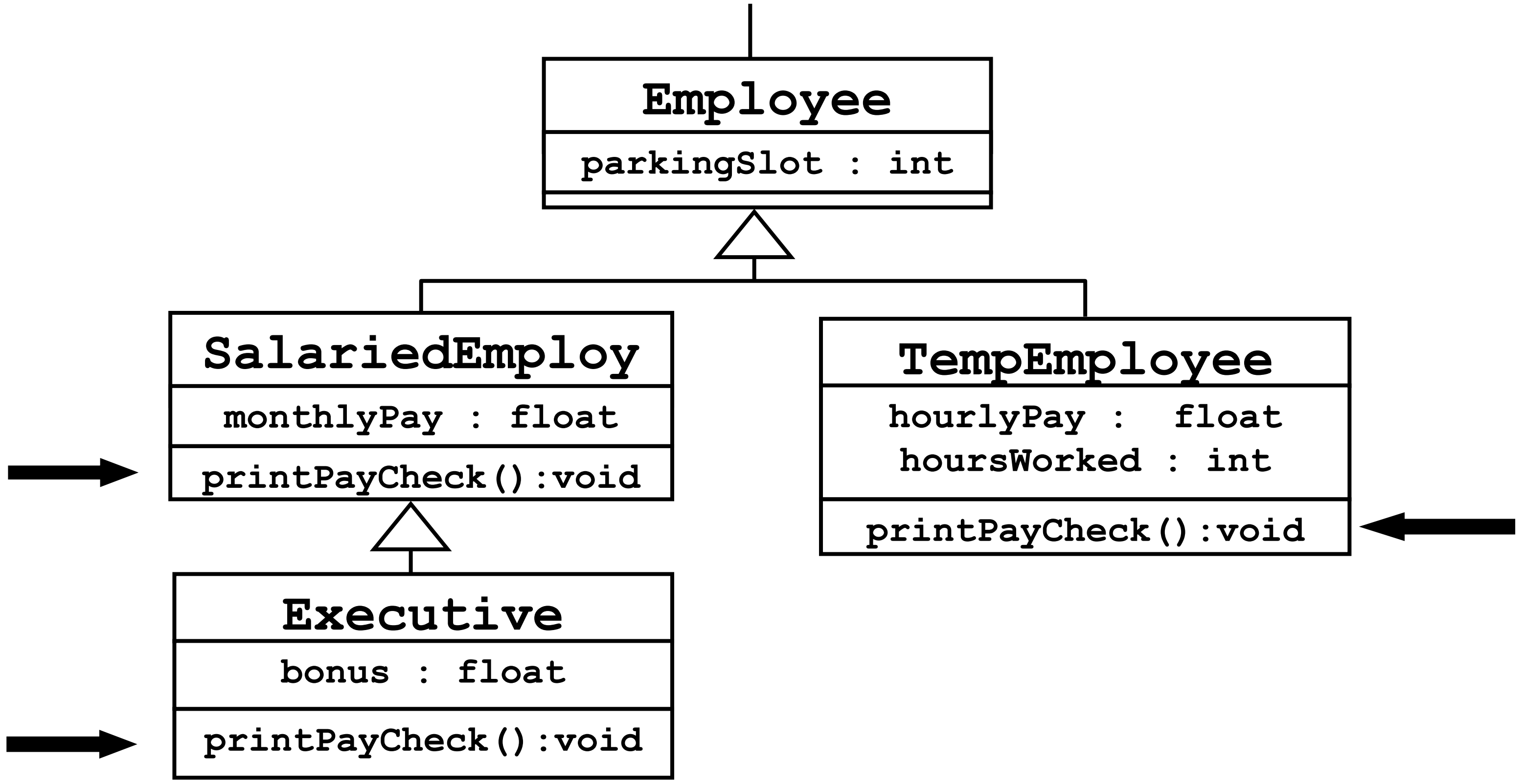
Printing Paychecks (traditional approach)

```
for (int i=0; i<staffList.length; i++) {
    String name = staffList[i].getName();
    float amount = 0;
    if (staffList[i] instanceof SalariedEmployee) {
        SalariedEmployee curEmp = (SalariedEmployee) staffList[i];
        amount = curEmp.getMonthlyPay();
        printPayCheck (name, amount);
    } else if (staffList[i] instanceof Executive) {
        Executive curExec = (Executive) staffList[i] ;
        amount = curExec.getMonthlyPay() + curExec.getBonus());
        printPayCheck (name, amount);
    } else if (staffList[i] instanceof TempEmployee) {
        TempEmployee curTemp = (TempEmployee) staffList[i] ;
        amount = curTemp.getHoursWorked()*curTemp.getHourlyPay();
        printPayCheck (name, amount);
    }
}
. . .
private void printPayCheck (String name, float amt) {
    System.out.println ("Pay To The Order Of:" + name + " $" + amt);
}
```



Polymorphism Solution

Computation should be encapsulated



Call Polymorphic Method

```
...
for (int i=0; i<staffList.length; i++) {
      staffList[i].printPayCheck() ;
}
...
```

Now, the Print method which gets invoked is:

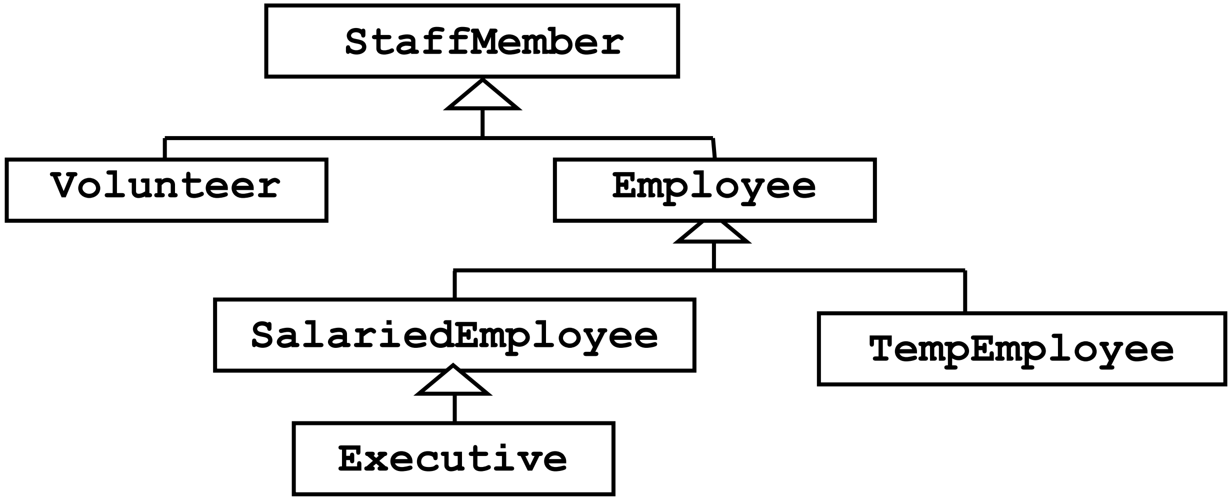
- determined at runtime, and
- depends on subtype

Maintainable and Extendable

Problem

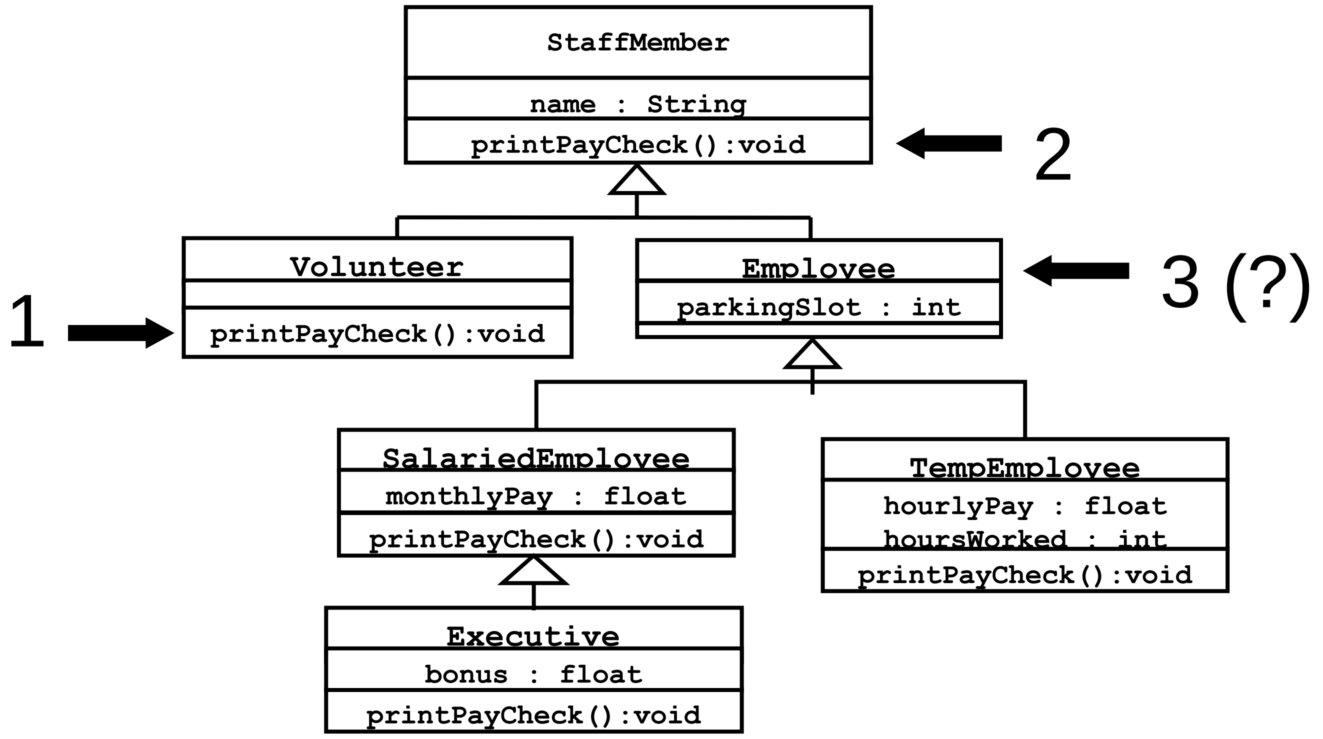
```
...
for (int i=0; i<staffList.length; i++) {
  staffList[i].printPayCheck() ;
}
...
```

What if `staffList[3]` is a volunteer?



Safety in Polymorphism

Ideally, **every** class should know how to deal with `printPayCheck()` messages:



Any Questions?

Free to Go!