# Object-Oriented Programming: Inheritance

Chapter 11

## TOPICS

## Introduction

## How Can We Classify Vehicles?

Into what broad categories can we classify vehicles?

## EXAMPLES OF VEHICLES

## Categories of Vehicles

Into what broad categories can we classify vehicles?

**Medium** it travels in/on – land, sky, water

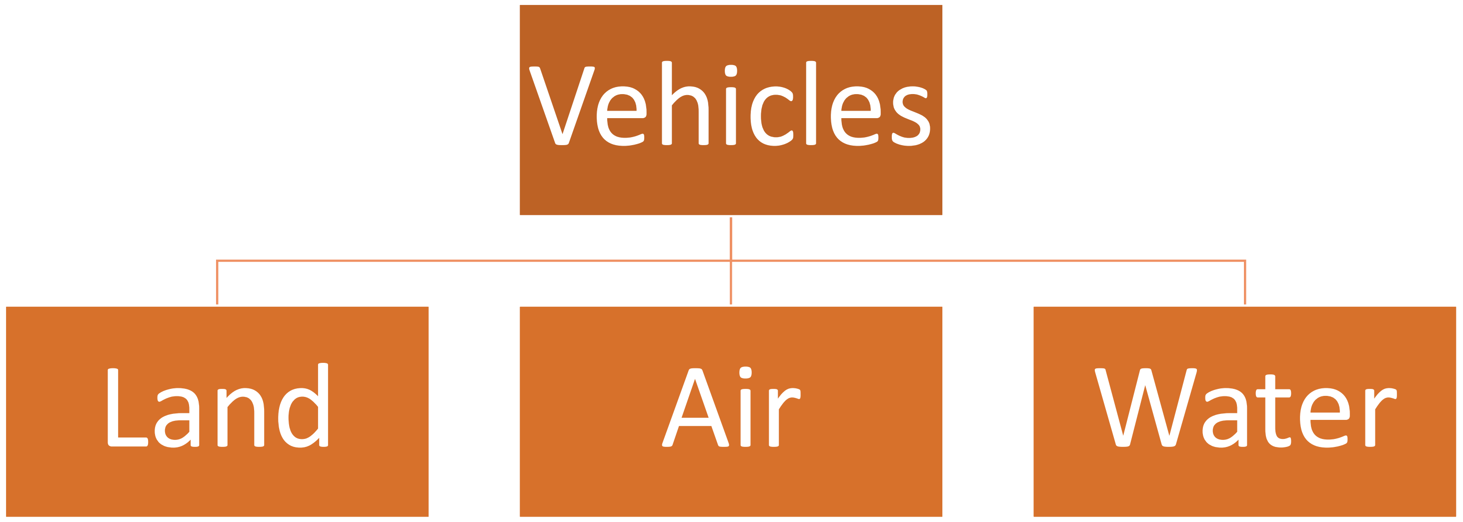
**Method of interaction** with medium – wheels, skis, wings, propellers, continuous tracks (like a tank)

Means of **propulsion** – engine, human, horse

**Other ways** – means of steering, size, ride in or on, electric vs. gas-powered, etc.

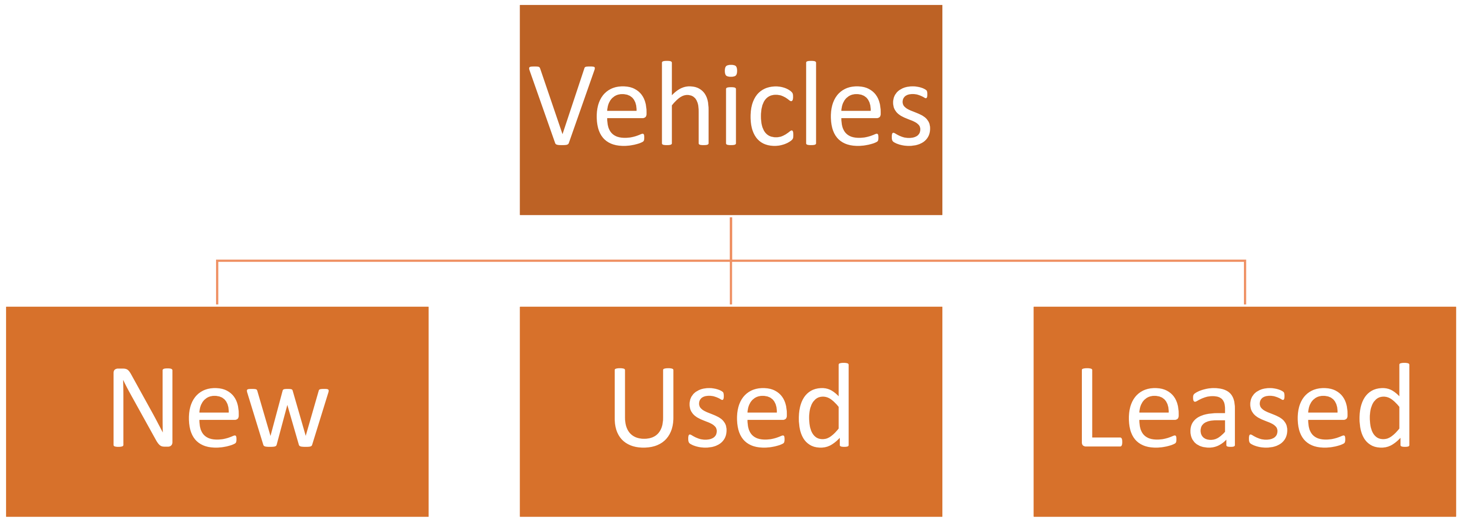
## Categories of Vehicles – Example 1

One example of broad categories to organize our vehicles into a hierarchy:



## Categories of Vehicles – Example 2

Might need different categories for different purposes:



## Properties of Vehicles

What are some properties of a vehicle?

## Properties of Vehicles – Examples

What are some properties of a vehicle?

| | |
|---|---|
| Engine size | Color |
| Number of doors | Year |
| Number of wheels | Battery lifespan |
| Number of passengers | Suspension |
| Type of brakes | Type of steering mechanism |
| Gas mileage | Weight class |

## Behaviors of Vehicles

What are some behaviors of a vehicle?

## Behaviors of Vehicles – Examples

What are some behaviors of a vehicle?

| | |
|---|---|
| Steer | Raise ladder |
| Turn | Play radio |
| Accelerate | Lift a heavy weight |
| Brake | Check engine |
| Shoot rocket | Deploy air bag |
| Land | Fold up (Transformers only) |

## The Turn Behavior

Consider the "Turn" behavior

What must happen to make each type of vehicle turn?

| | | |
|---|---|---|
| Pickup truck | *vs.* | Hook & ladder fire truck |
| Rowboat | *vs.* | Motorboat |
| Jet (in the air) | *vs.* | Glider (in the air) |

What is similar about how these vehicles move?  What is different?

## Designing a New Vehicle

If you were going to design a new kind of vehicle, what could you do to save time and money, and ensure that this new vehicle is viable?

In other words, if your assignment is “Design a new vehicle”…

**What question(s) would you ask about this new vehicle?**

**What would you do to save time with your design?**

## Designing a New Vehicle – Questions & Time-Saving Ideas

**What question(s) would you ask about this new vehicle?**

You might ask how this new vehicle is similar to, or different from, other vehicles that already exist.

**What would you do to save time with your design?**

You might use existing designs that are similar as a starting point, so that you [literally] don’t have to reinvent the wheel.

## Inheritance Basics and Definitions

## Inheritance is Software Reuse

Form of **software reuse**

A new class is created by:

1. Using an existing class, and

2. Adding to it with new or modified capabilities

## Benefits of Reusing Software

What are the benefits of reusing existing software?

## Reusing Software Saves Time & Effort

Can **save time** by basing new classes on existing proven and debugged classes

Increases likelihood that system will be **implemented and maintained effectively**

But we don't want to reuse by copying and pasting code

Why not?

## Copying and Pasting is not a Desirable Form of Reuse

Prone to errors in the process of copying and pasting

Maintenance and updating is difficult if the code is copied into many places

Can make programs very large and difficult to understand

## New Classes can Reuse Code via Inheritance

Can use existing code without copying it

Can add to the existing code

## Inheritance in C++ – Terms to Remember

New class **is derived from** an existing class

Existing class is the **base class**

New class is the **derived class**

## Creating a New Class using Inheritance

General format:

**class** *derivedClassName* **: public** *baseClassName*

This goes in the header file.

Java syntax: public class subClassName extends superClassName

## Derived Class

Can **add** its own data members and member functions

Is **more specific** than its base class

- Represents more specialized group of objects.

Exhibits behaviors of its base class **AND** can add behaviors that are specific to the derived class

Can be a base class of future derived classes

**Inheritance = specialization**

## Base Class

**Direct base class** – base class from which the derived class explicitly inherits

**Indirect base class** – any class above the direct base class in the class hierarchy

## Relationships between Classes

**Is-a** relationship represents **inheritance**

- Object of a derived class can also be treated as an object of its base class

**Has-a** relationship represents **composition**

- Object contains references to other objects

## Base Classes and Derived Classes

Base classes = more general

Derived classes = more specific

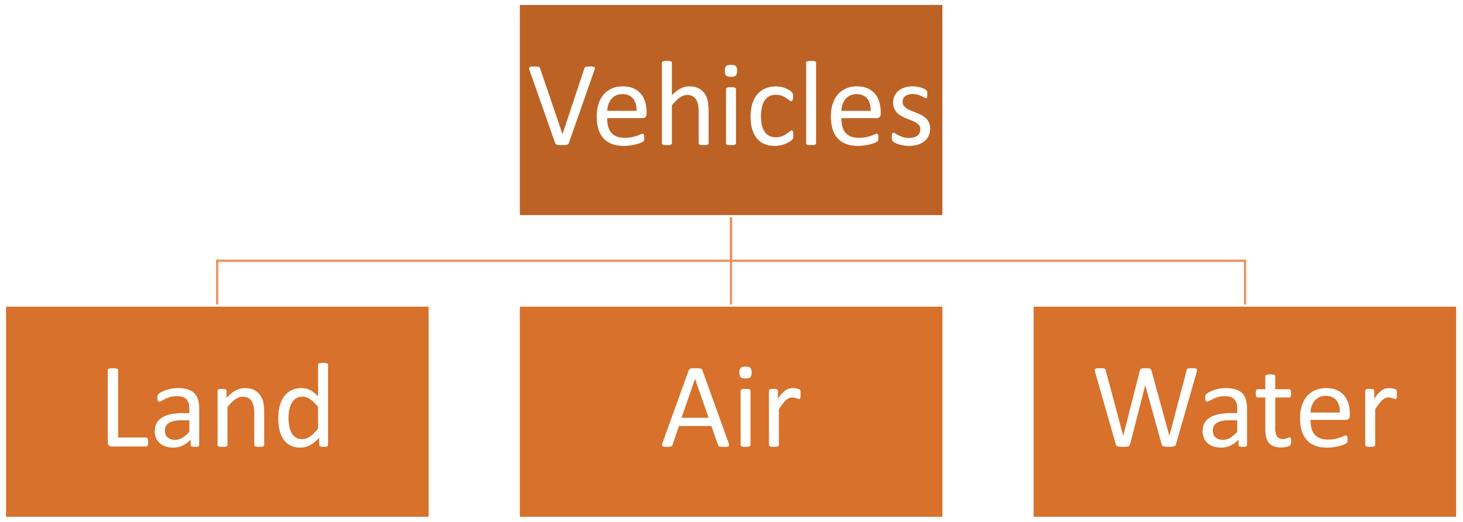
Set of base class objects is larger than the set of its derived class objects because:

Every derived class object **is an object of its base class**, and

One base class can have many derived classes

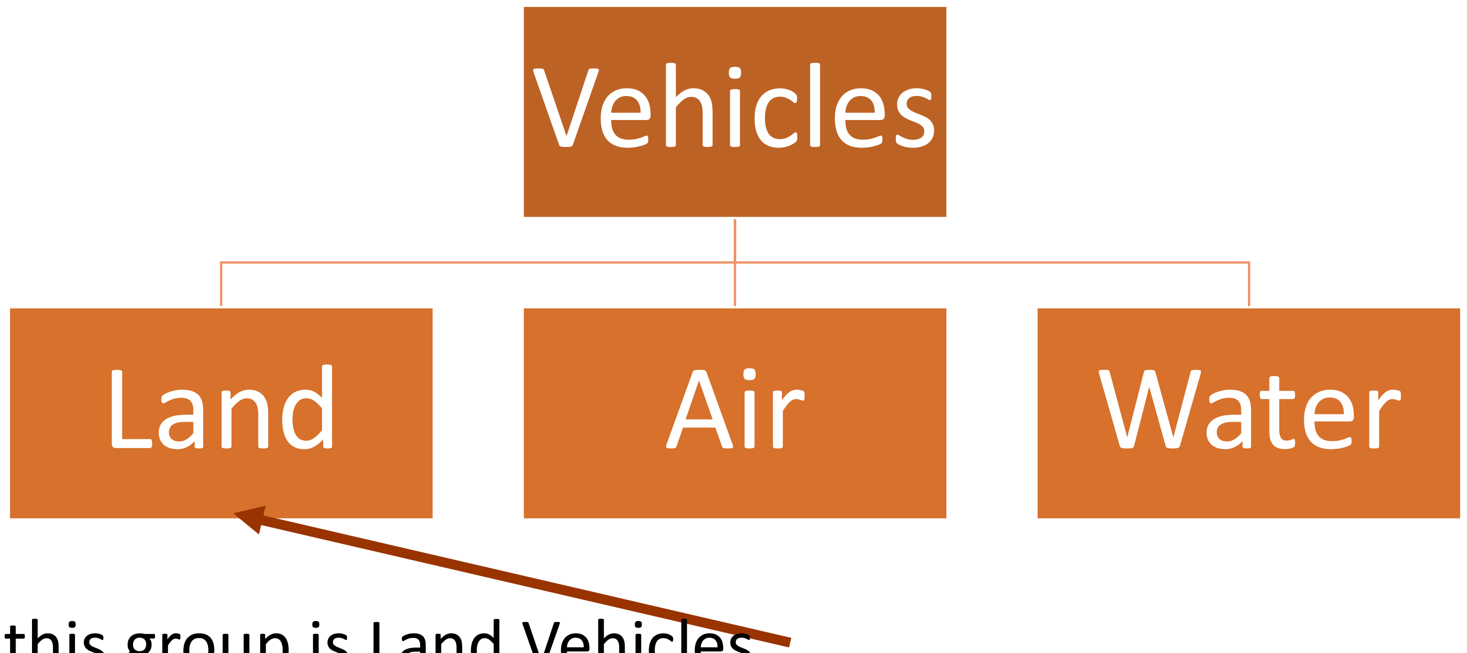
## Base Classes and Derived Classes – Remember the Vehicles?

Which of these are Vehicles in this example hierarchy?



## Base Classes and Derived Classes – Only Some are Land Vehicles

All of these are Vehicles



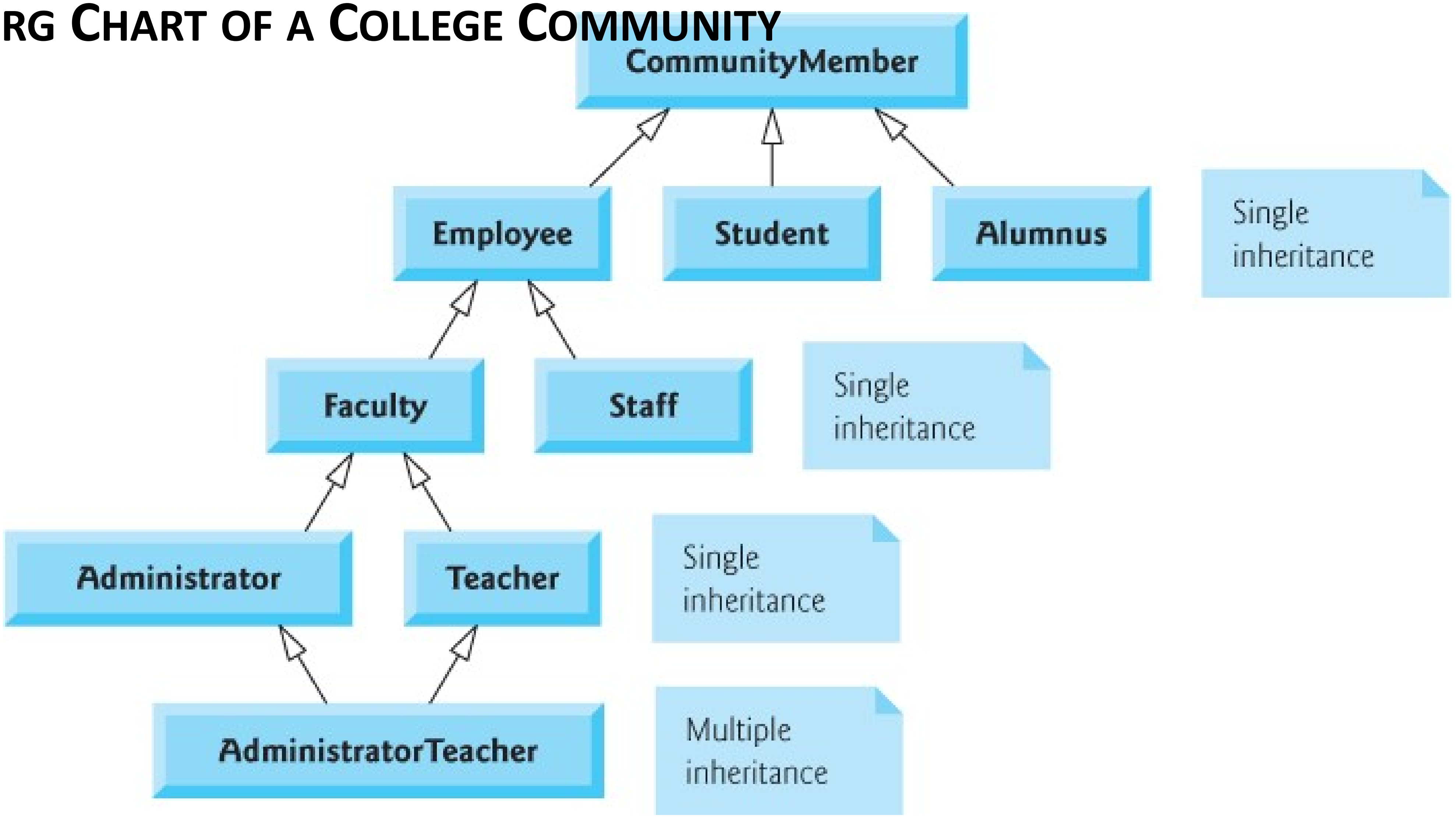
But only this group is Land Vehicles

## INHERITANCE HIERARCHY

A base class exists in a hierarchical relationship with its derived classes.

Next slide shows a sample university community class hierarchy – also called an **inheritance hierarchy**

## ORG CHART OF A COLLEGE COMMUNITY



## Hierarchy Example

Each arrow = **is-a relationship**

Follow arrows upward

- Employee **is a** CommunityMember
- Teacher **is a** Faculty member

CommunityMember is the **direct base class** of Employee, Student and Alumnus classes

- And is an **indirect base class** of all other classes

Starting from bottom, follow arrows and apply the is-a relationship up to topmost base class

## Remember the Has-a Relationship

Not every class relationship is inheritance

**Has-a** relationship:

- Create classes by **composition** of existing classes

Cannot say that an Employee **is** a BirthDate or that an Employee **is** a TelephoneNumber

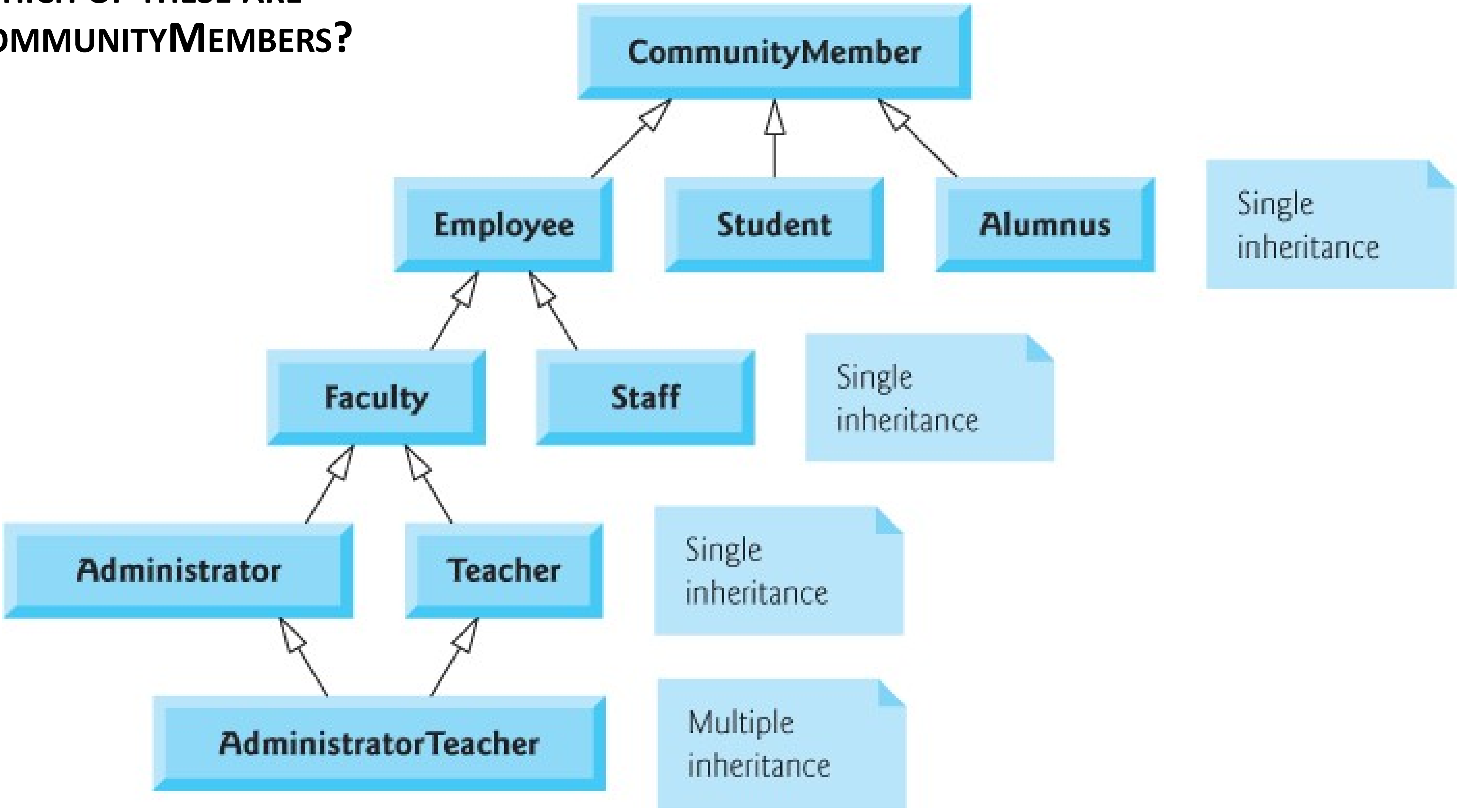
Employee **has a** BirthDate, and an Employee **has a** TelephoneNumber

## Objects of a Common Base Class

**All objects that derive from the same base class can be treated as objects of that base class**

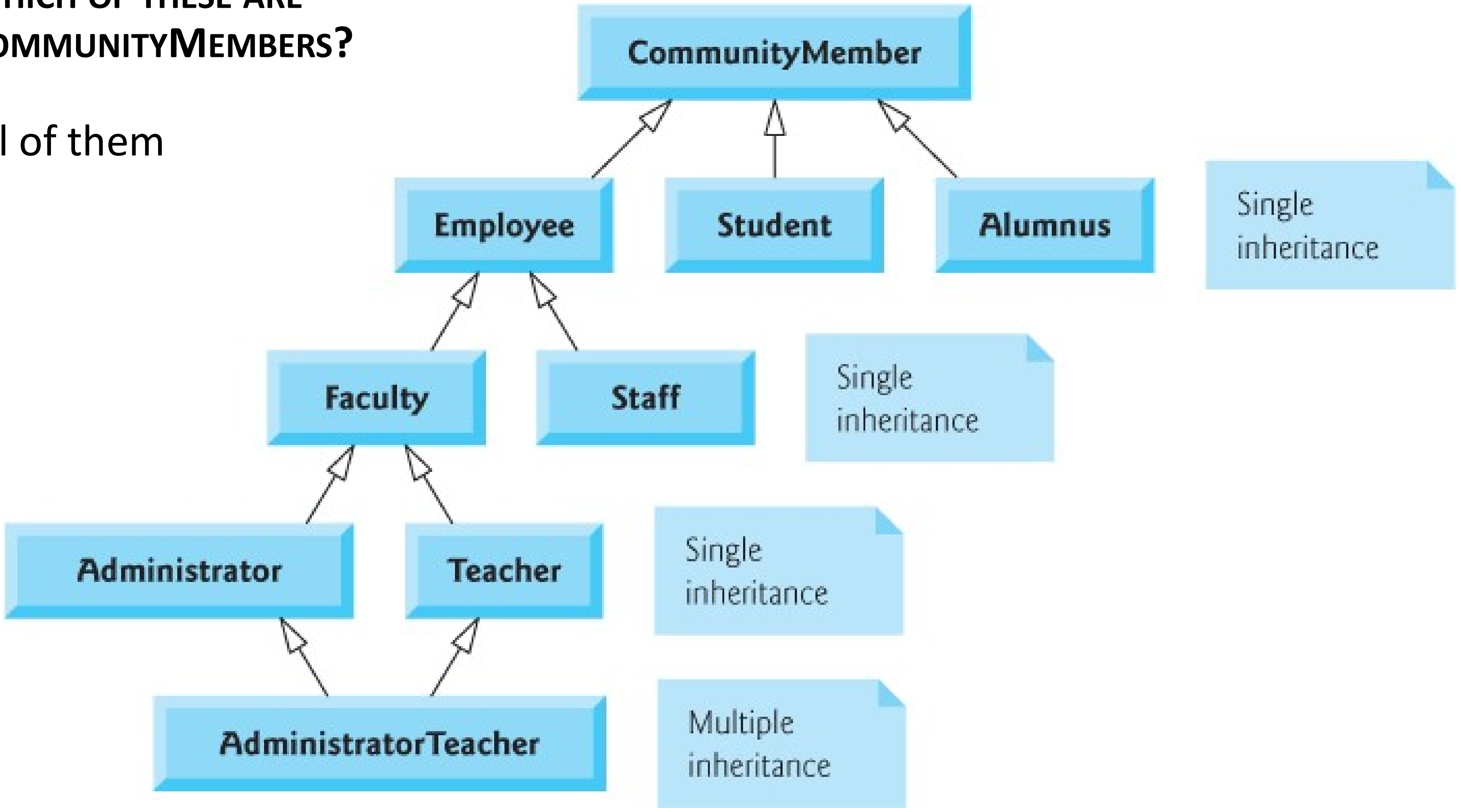
A function that accepts a CommunityMember object will accept an Alumnus object

## WHICH OF THESE ARE COMMUNITYMEMBERS?

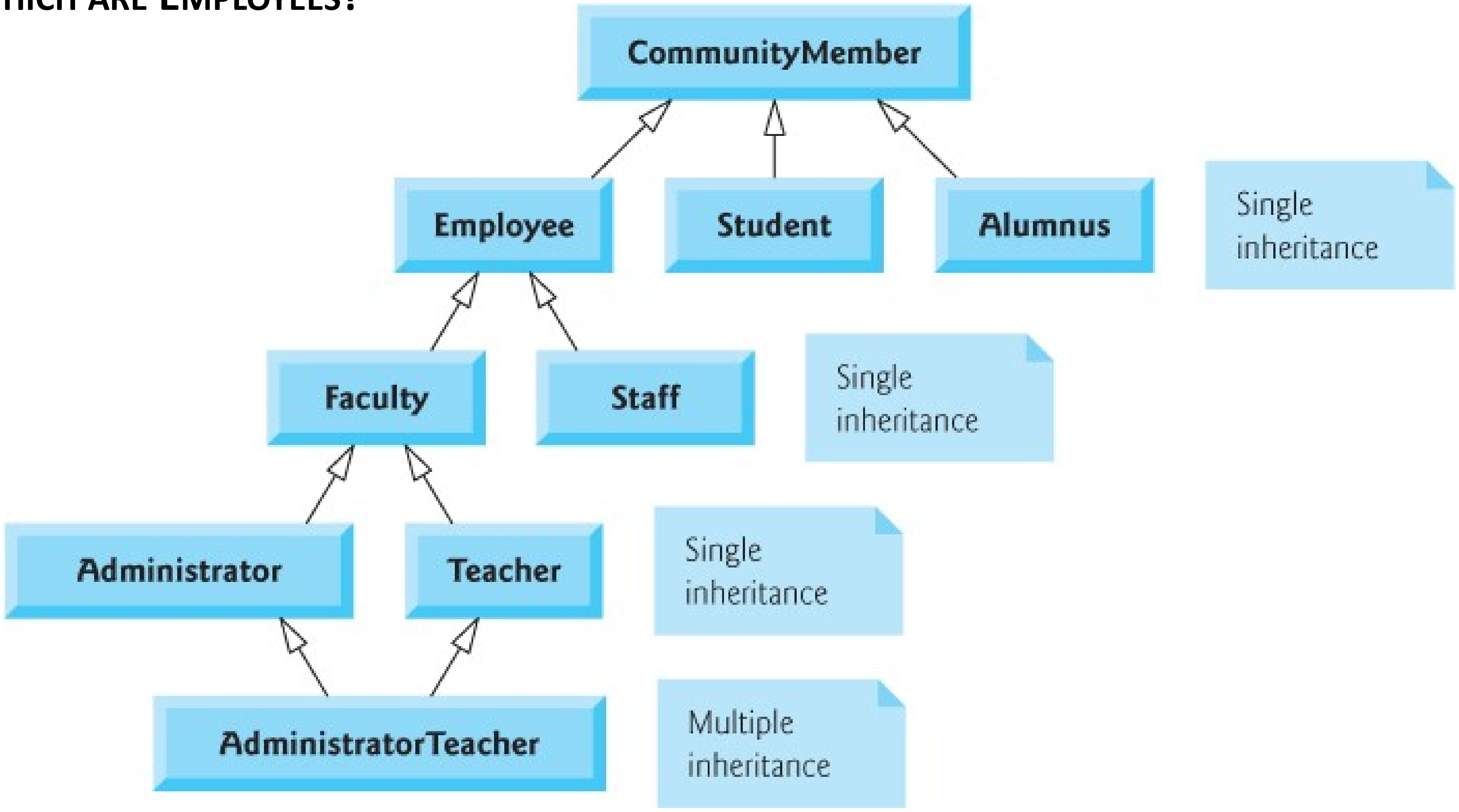


## WHICH OF THESE ARE COMMUNITYMEMBERS?

All of them

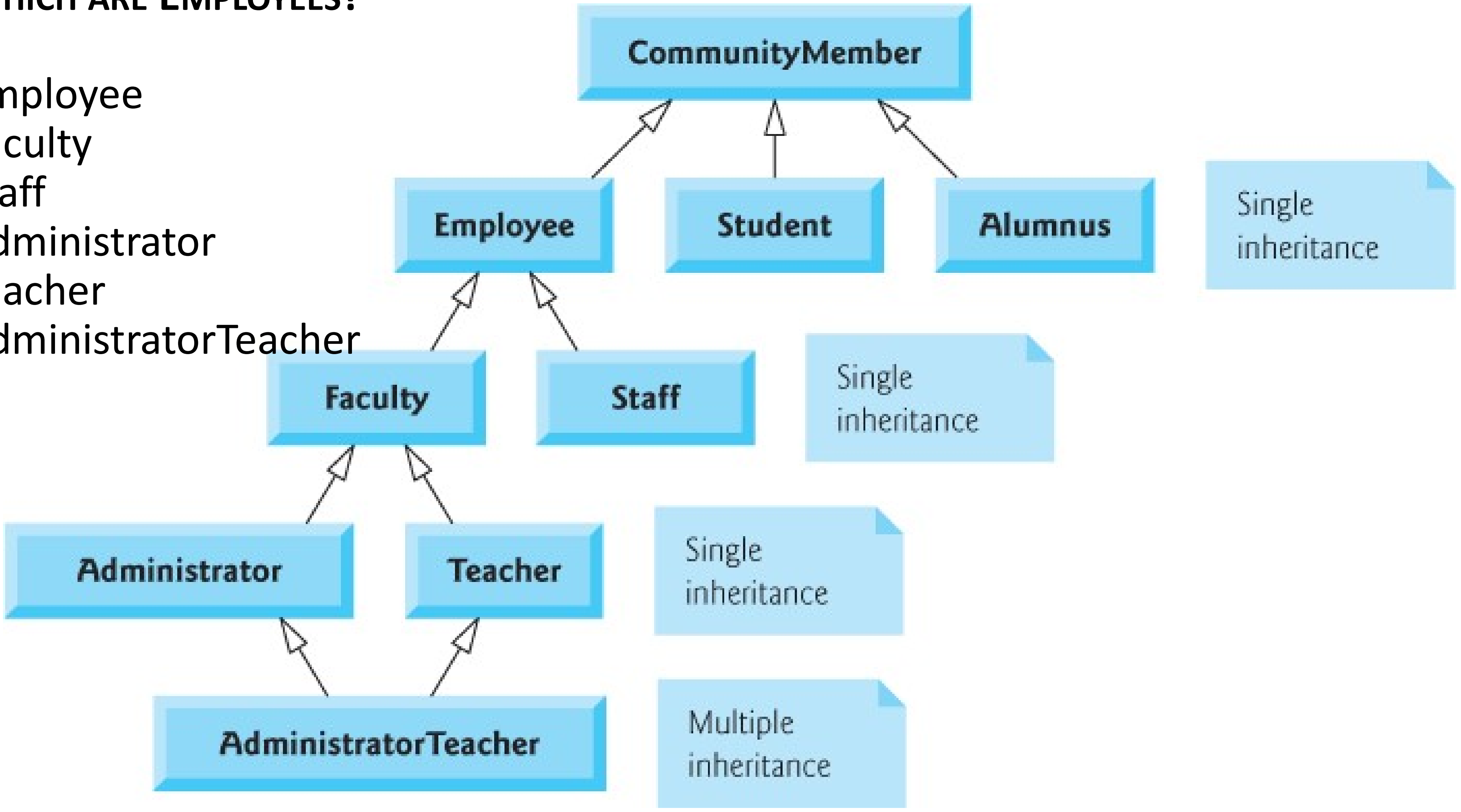


## Which are Employees?

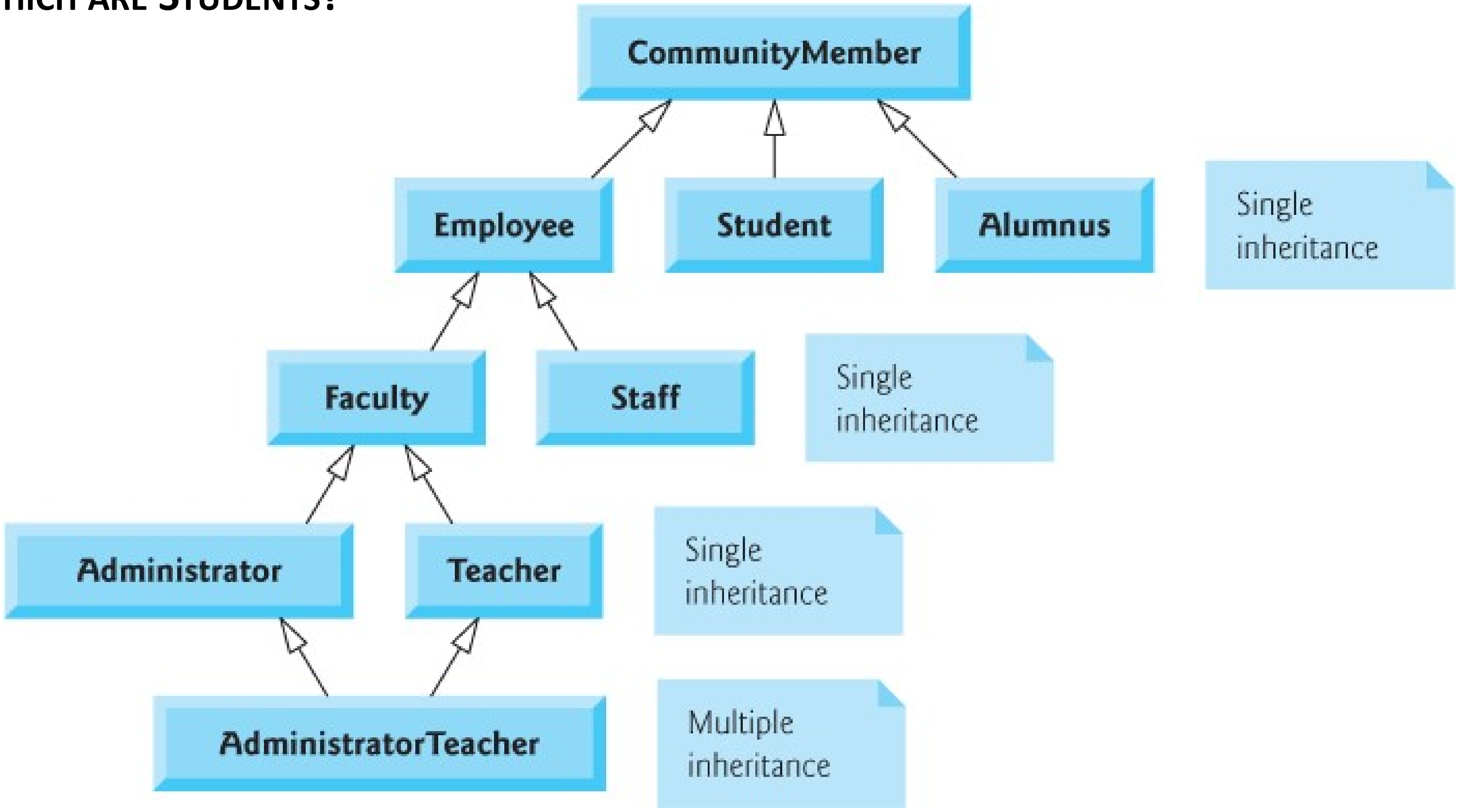


## WHICH ARE EMPLOYEES?

Employee
Faculty
Staff
Administrator
Teacher
AdministratorTeacher

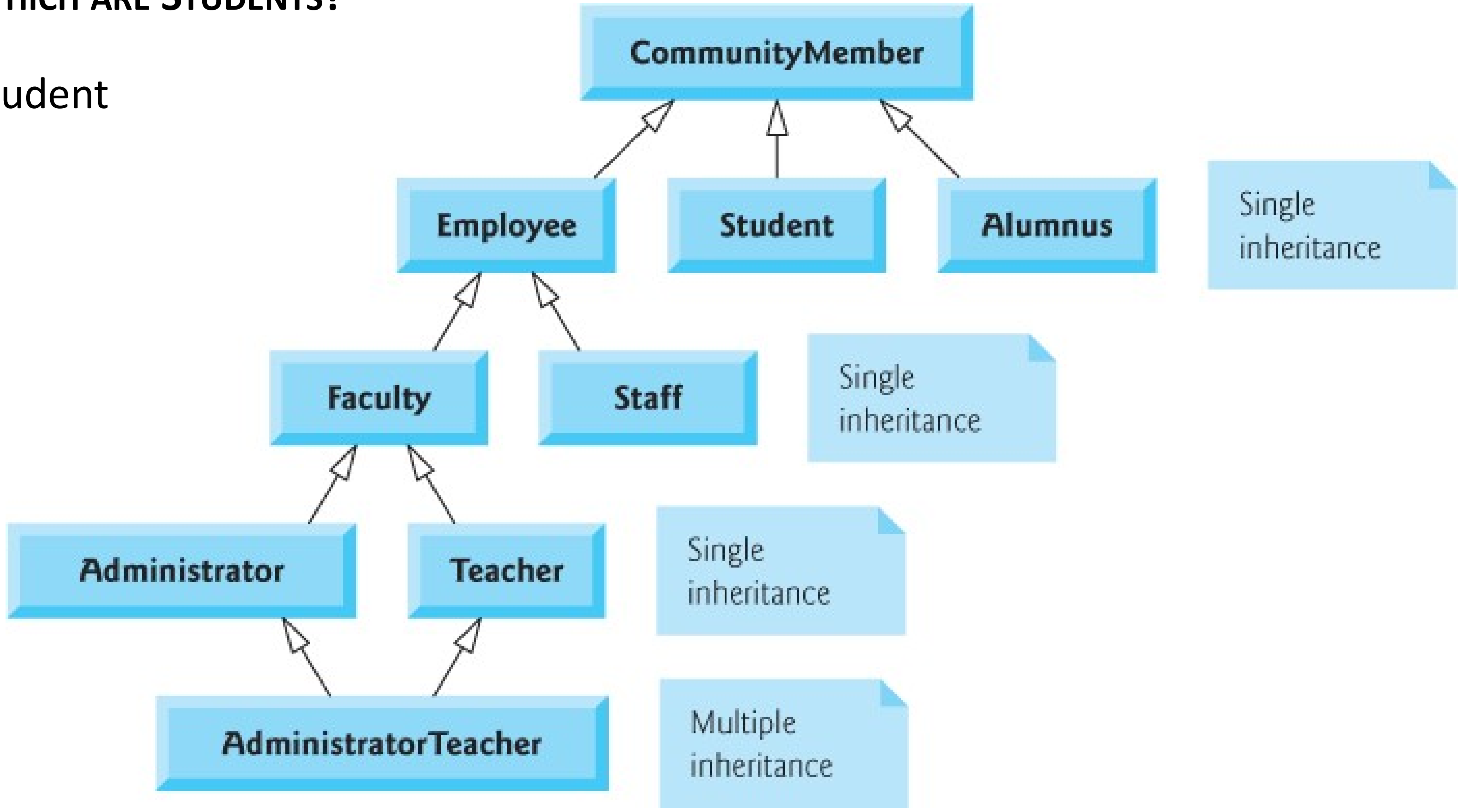


## Which are Students?

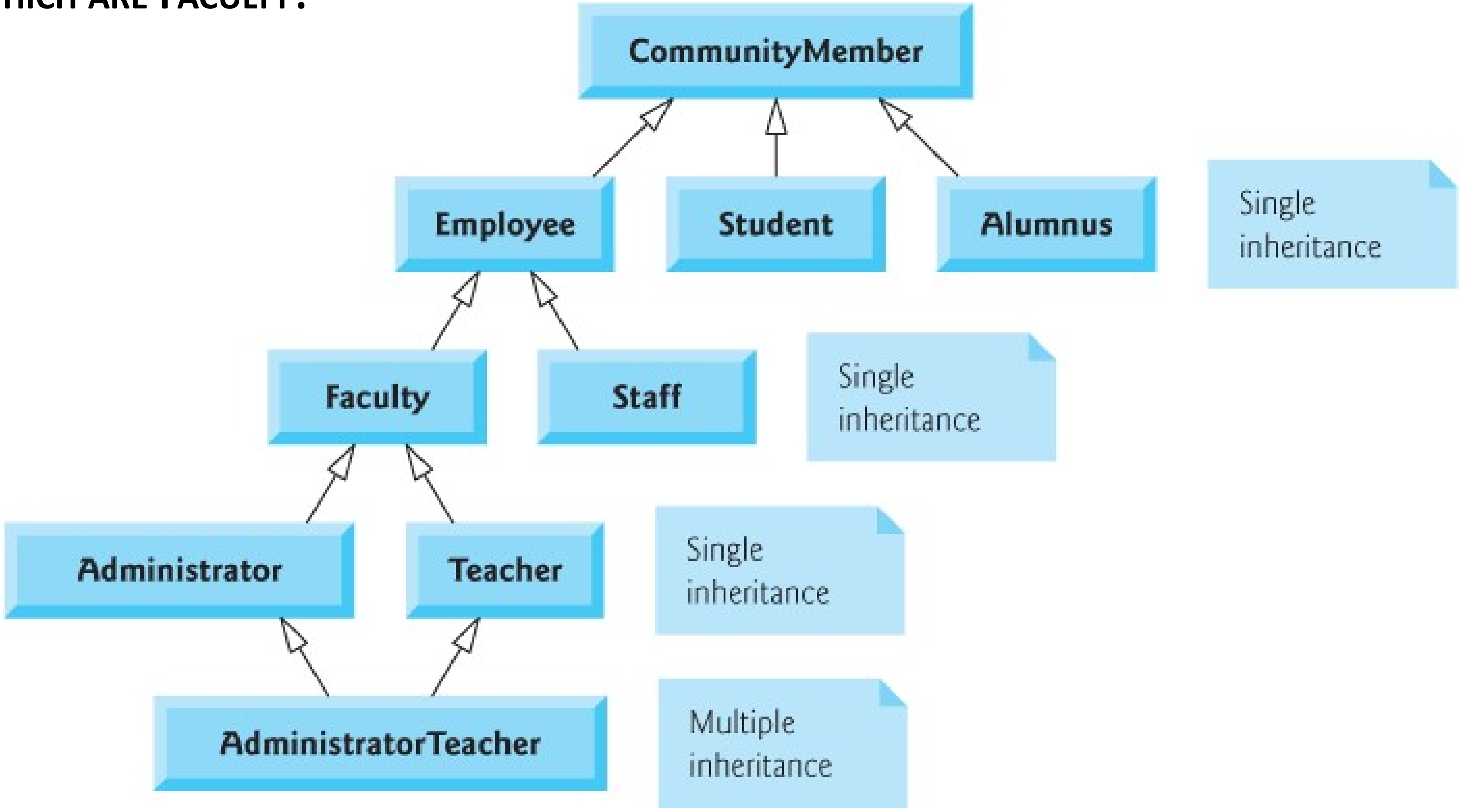


## WHICH ARE STUDENTS?

Student

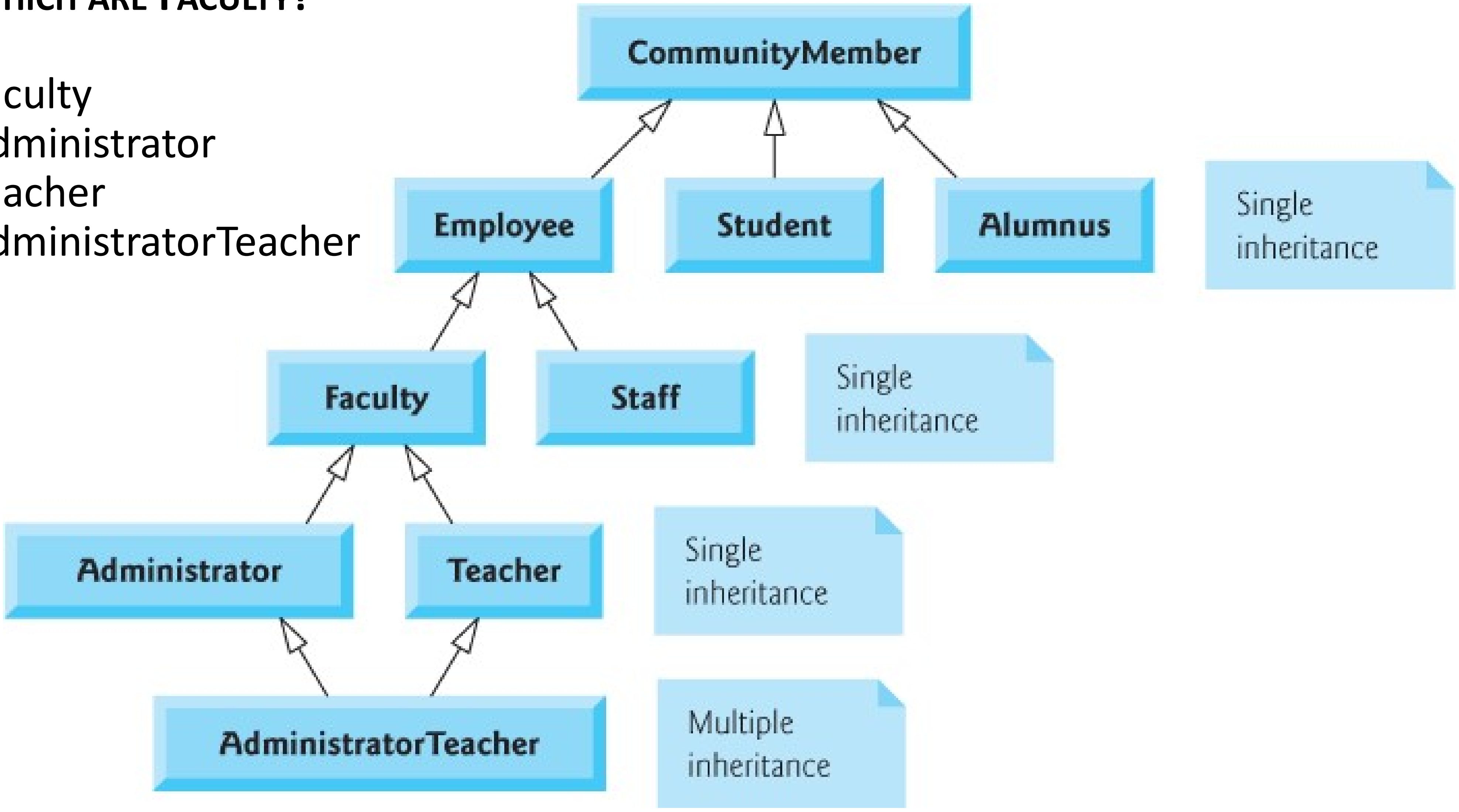


## Which are Faculty?



## WHICH ARE FACULTY?

Faculty
Administrator
Teacher
AdministratorTeacher



## Types of Inheritance

### Single inheritance

- Class is derived from only one base class

### Multiple inheritance

- Class is derived from more than one base class
- In the CommunityMember example, the AdministratorTeacher class derives from both the Administrator Class and the Teacher class
- Generally discouraged
- Chapter 23 has more on multiple inheritance

### Public / Private / Protected

## Types of Inheritance: Public, Private, and Protected

## Access Modifiers We've Seen So far

**public** members of a class:

- Accessible to any program that has a handle (name, reference, or pointer) to an object of that class, or to one of its derived classes

**private** members of a class:

- Accessible only within class' body, and in friends of that class

## A Third Type – protected Members

**protected** – intermediate level of access between public and private

A class' protected members are accessible:

1. Within class' body

2. By members and friends of the class

3. By members and friends of its derived classes

## Public Inheritance

Base class members retain their original member access when they become members of derived class, i.e.:

- Public in base stay public in derived
- Protected in base stay protected in derived

Derived class can manipulate private members of base class through the functions inherited from base class

Friend functions, constructors & destructors not inherited

## Access in Derived Classes

Remember what the members of a class are:

- Data members
- Member functions

**Public** and **protected** class members stay public or protected when they become members of a derived class

A class' **private** members are hidden in its derived classes

- Can be accessed through public or protected functions inherited from base class

## Inheriting Functions

Derived class inherits functions from base class

May inherit unneeded functions, or functions it should not have, or it needs a customized version of the function

Derived class can **redefine** the base class function with its own implementation

We’ll see this later in chapter examples.

## Accessing Base Class Functions

Derived class functions can refer to variables and functions inherited from base class by using just the variable or function name

- But only if that variable or function is public or protected…

If derived class function **redefines** inherited base class function:

Base class function can still be accessed with the **name of the base class and the scope resolution operator ::**

## Summary of Class Access Types

**Public** members are accessible:

- Within the **class** itself
- **Outside** the class, via a **handle** to the **class**
- **Outside** the class, via a **handle** to a **derived class** of the class

**Protected** members are accessible:

- Within the **class** itself
- Within a **derived class** of the class
- Within **members and friends of its derived classes**

**Private** members are accessible:

- Within the **class** itself

## Relationship between Base Classes and Derived Classes – Examples

## Company Payroll Example 1 – Commission Employees

Commission employees are paid a percentage of their sales only

Fig11_04_06

- CommissionEmployee.h – specifies the class' public services, which are?
- CommissionEmployee.cpp – note constructor does not use member-initializer list – we'll see why in a bit
- Fig11_06.cpp – driver

Public services are the parts of the class that are **public**, that client code can use.

## Payroll Example 2 – Base Plus Commission Employee

Another type of employee may receive a base salary, and commissions on sales

So, a **BasePlusCommissionEmployee** class contains first name, last name, SSN, gross sales amount, commission rate and base salary.

All but the base salary are in common with class CommissionEmployee

## Payroll Example 2 – Base Plus Commission Employee Code

Fig11_07_09

- BasePlusCommissionEmployee.h
- BasePlusCommissionEmployee.cpp
- Fig11_09.cpp – driver class for this example

## BasePlusCommissionEmployee Public Services

BasePlusCommissionEmployee's public services:

- constructor
- earnings function
- toString function
- get and set for each data member

Most of these are also in common with CommissionEmployee

## BasePlusCommissionEmployee – Derived Class?

What is BasePlusCommissionEmployee’s base class? (Does it have one?)

BasePlusCommissionEmployee does not have a base class

- Note line 9 in its header does not specify another class

So it is not a derived class

## BasePlusCommissionEmployee Code – Data Members

Much of BasePlusCommissionEmployee's code is similar, or identical, to that of CommissionEmployee

| | CommissionEmployee | BasePlusCommissionEmployee |
|---|---|---|
| firstName | ✓ | ✓ |
| lastName | ✓ | ✓ |
| SSN | ✓ | ✓ |
| grossSales | ✓ | ✓ |
| commissionRate | ✓ | ✓ |
| baseSalary | X | ✓ |

## BasePlusCommissionEmployee Code – Sets and Gets

Much of BasePlusCommissionEmployee's code is similar, or identical, to that of CommissionEmployee

| | CommissionEmployee | BasePlusCommissionEmployee |
|---|---|---|
| set / get FirstName | ✓ | ✓ |
| set / get LastName | ✓ | ✓ |
| set / get SSN | ✓ | ✓ |
| set / get grossSales | ✓ | ✓ |
| set / get commissionRate | ✓ | ✓ |
| set / get baseSalary | X | ✓ |

## BasePlusCommissionEmployee Code – Other Functions

Much of BasePlusCommissionEmployee’s code is similar, or identical, to that of CommissionEmployee

| | **CommissionEmployee** | **BasePlusCommissionEmployee** |
|---|---|---|
| **constructor** | ✓ | Same as constructor for CommissionEmployee, plus sets baseSalary |
| **earnings** | ✓ | Same as earnings function of CommissionEmployee, plus adds baseSalary to total |
| **toString** | ✓ | Same as toString function of CommissionEmployee, plus outputs baseSalary |

## BasePlusCommissionEmployee Code

We literally copied CommissionEmployee’s code, pasted it into BasePlusCommissionEmployee, then modified the new class to include a base salary and functions that manipulate the base salary.

What is wrong with this approach?

## BasePlusCommissionEmployee Code – Copying/Pasting Isn't a Good Idea

We literally copied CommissionEmployee's code, pasted it into BasePlusCommissionEmployee, then modified the new class to include a base salary and functions that manipulate the base salary.

What is wrong with this approach?

- Often error prone
- Time consuming
- Maintenance nightmare

## Software Engineering Observation

Using inheritance:

- Common data members and member functions of all classes in the hierarchy are **declared in one place in the code—the base class**

When changes are made to these common features in the base class:

- **Derived classes inherit the changes**

Without inheritance:

- **Changes would need to be made to all the source code files** that contain a copy of the code in question

## Payroll Example 3 – Creating an Inheritance Hierarchy – Attempt 1

Fig11_10_11

- BasePlusCommissionEmployee.h
- BasePlusCommissionEmployee.cpp
- CommissionEmployee.h
- CommissionEmployee.cpp

BasePlusCommissionEmployee class **derives from** CommissionEmployee class

Therefore, a BasePlusCommissionEmployee object **is a** CommissionEmployee

## Payroll Example 3 – Creating an Inheritance Hierarchy – Attempt 1 – Base Class Header

Note that the **base class header** (CommissionEmployee.h) is included in the derived class header

- BasePlusCommissionEmployee.h (line 8)

This is necessary for three reasons:

- Compiler needs to know CommissionEmployee exists before line 10
- Compiler uses a class definition to determine the size of an object of that class, so it needs to know about the base class members too
- Compiler needs to know if derived class is using base class' members properly (e.g. calling gets correctly)

## Payroll Example 3 – Creating an Inheritance Hierarchy – Attempt 1 – More on Base Class Header

BasePlusCommissionEmployee.h (line 10):

**class BasePlusCommissionEmployee : public CommissionEmployee**

**Colon :** indicates inheritance

Keyword **public** indicates type of inheritance

Because it's a public derived class, BPCE inherits all members of class CE, except for constructor

- Constructors and destructors not inherited

## Payroll Example 3 – Creating an Inheritance Hierarchy – Attempt 1 – Data Member

Inheritance passes on class CommissionEmployee’s data members and member functions to BasePlusCommissionEmployee

BasePlusCommissionEmployee has its own data member: **baseSalary**

Employees who work only on commission don’t get a base salary, so the CommissionEmployee class doesn’t have this data member

## Payroll Example 3 – Creating an Inheritance Hierarchy – Attempt 1 – Derived Class

BasePlusCommissionEmployee.cpp (lines 14-15)

- Uses member initializer to pass arguments to base class constructor

Derived class constructor must **implicitly or explicitly call its base class constructor** to initialize the data members inherited from the base class.

- In this case, an explicit call

If no explicit call, attempt is made to **implicitly** call default constructor

- In this example, default constructor doesn’t exist
- If lines 14-15 were left out, it would not compile

## ACCESSING PRIVATE DATA MEMBERS

Only a base class' **public** and **protected** members are **directly** accessible in the derived class

- Directly means using the name of the member, instead of a get or set

Attempting to access base class' **private** data members is a compile error

**earnings** function in BasePlusCommissionEmployee.cpp (line 35)

- Tries to use private data members of CommissionEmployee – compiler error

**toString** function in BasePlusCommissionEmployee.cpp (47-48)

- Tries to use private data members of CommissionEmployee – compiler error

## Payroll Example 4 – Creating an Inheritance Hierarchy – Attempt 2

Fig11_12

CommissionEmployee.h – declares members **protected**

CommissionEmployee.cpp – same as previous example

BasePlusCommissionEmployee.cpp

- Unchanged, but now can access members inherited from CommissionEmployee
- Note explicit call to CommissionEmployee constructor (line 13)

## Payroll Example 4 – Creating an Inheritance Hierarchy – Attempt 2 – Uses Inheritance

Fig11_09.cpp

Identical to previous version, and gives same output

Only change is BasePlusCommissionEmployee is created using inheritance

Much shorter than earlier example

CommissionEmployee code defined in one place

## Notes on Using protected Data

Can improve performance, since derived classes can access base class data directly instead of via a get/set

But, derived class may use invalid data to set these base class members

derived class members functions more likely to be written to depend on base class implementation, rather than base class services

## PROTECTED VS. GET/SET

Inheriting protected data members slightly increases performance

- Can directly access them in derived class without overhead of a set or get function call

In most cases, it’s still better to use private data members to encourage proper software engineering, and leave code optimization issues to the compiler

- Code will be easier to maintain, modify and debug.

## Payroll Example 5 – Creating an Inheritance Hierarchy – Attempt 3

Hierarchy reengineered using good software engineering practices…

Class CommissionEmployee declares data members as private and provides public functions for manipulating these values.

Fig11_14_15

- CommissionEmployee.h / CommissionEmployee.cpp
- BasePlusCommissionEmployee.h / BasePlusCommissionEmployee.cpp
- Fig11_09.cpp

## Payroll Example 5 – Creating an Inheritance Hierarchy – Attempt 3 – Better SE Practices

**earnings** and **toString** functions use **getters** to obtain values of **its own data members**

Remember – using getters and setters even within a class' own functions is good software engineering

If internal representation of data changes (e.g., variable names) only getters and setters that directly manipulate data members need to change.

These changes occur solely within the base class—no changes to the derived class are needed.

**Localizing the effects of changes like this is good software engineering practice!**

## Payroll Example 5 – Creating an Inheritance Hierarchy – Attempt 3 – Base Class Versions Invoked

Inherits CommissionEmployee’s non-private functions and can access the private base class members via those functions.

BasePlusCommissionEmployee has changes that distinguish it from earlier version:

earnings and toString functions each **invoke their base class versions** and do not access data members directly

## NEW BASEPLUSCOMMISSIONEMPLOYEE – EARNINGS FUNCTION

earnings function **redefines** base class' earnings function

The new version calls CommissionEmployee's earnings function on line 33 using the syntax **CommissionEmployee::earnings()**

This call obtains the part of the earnings based on commission alone, and that value is then added to the earnings from base salary

## NEW BASEPLUSCOMMISSIONEMPLOYEE – TOSTRING FUNCTION

toString function **redefines** class CommissionEmployee's toString()

On line 39, part of the String is created via calling CommissionEmployee's toString function:

Uses the syntax **CommissionEmployee::toString()**

Combines that with the information specifically related to a BasePlusCommissionEmployee

## Common Programming Error

When a base class function is redefined in a derived class:

- Derived class version often calls base class version to do a portion of the work

Forgetting **base class name** and **scope resolution operator ::** when calling the base class function causes the derived class function to invoke itself, potentially creating infinite recursion ☹

## Good Software Engineering Practice

If a function performs all or some of the actions needed by another function, call that function rather than duplicate its code

## Constructors in Derived Classes

## Constructors in Derived Classes – A Chain of Calls

Instantiating a derived class object begins a chain of constructor calls

The derived class constructor, before performing its own tasks, **invokes its direct base class' constructor**

If the base class is derived from another class, the base class constructor invokes the constructor of the next class up the hierarchy, and so on.

Each base class' constructor manipulates the base class data members that the derived class object inherits.