Object Oriented Programming II

Dr. Burak Kaleci

March 15, 2019

Content

Introduction

- Inheritance is a form of software reuse in which you create a class that absorbs an existing class's capabilities, then customizes or enhances them.
- When creating a class,instead of writing completely new data members and member functions, you can specify that the new class should inherit the members of an existing class.
- This **existing** class is called the **base class**,and the **new** class is called the **derived class**.

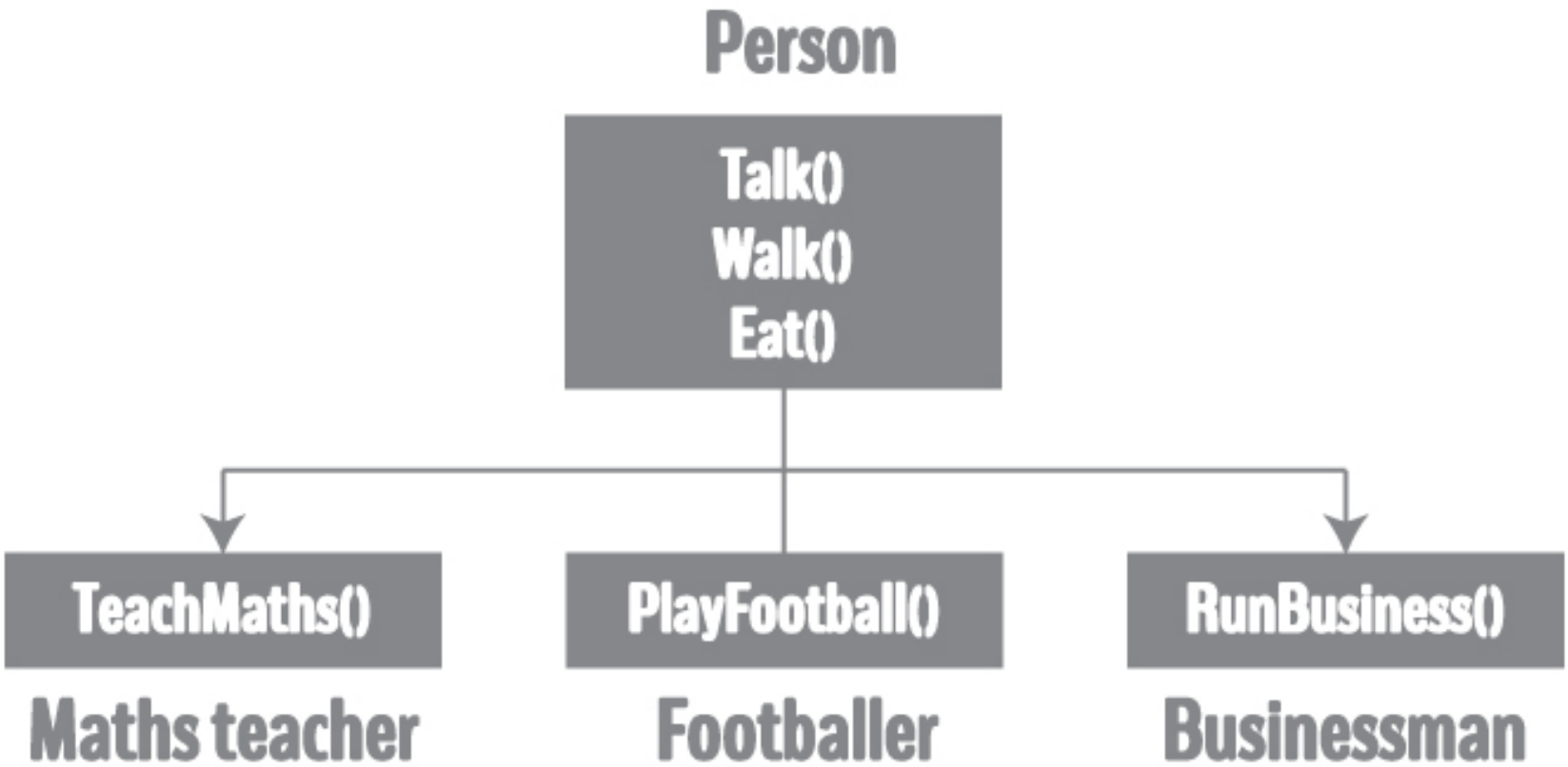
Introduction

- Suppose, in your game, you want three characters: a maths teacher, a footballer and a businessman.
- Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A maths teacher can teach maths, a footballer can play football and a businessman can run a business.
- You can individually create three classes who can walk, talk and perform their special skill as shown in the figure below.

Introduction

- In each of the classes, you would be copying the same code for walk and talk for each character.
- If you want to add a new feature - eat, you need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes.
- It'd be a lot easier if we had a Person class with basic features like talk, walk, eat, sleep, and add special skills to those features as per our characters. This is done using inheritance.

Introduction



Introduction

- Using inheritance, now you don't implement the same code for walk and talk for each class. You just need to inherit them.
- So, for Maths teacher (derived class), you inherit all features of a Person (base class) and add a new feature TeachMaths. Likewise, for a footballer, you inherit all the features of a Person and add a new feature PlayFootball and so on.
- This makes your code cleaner, understandable and extendable.

Introduction

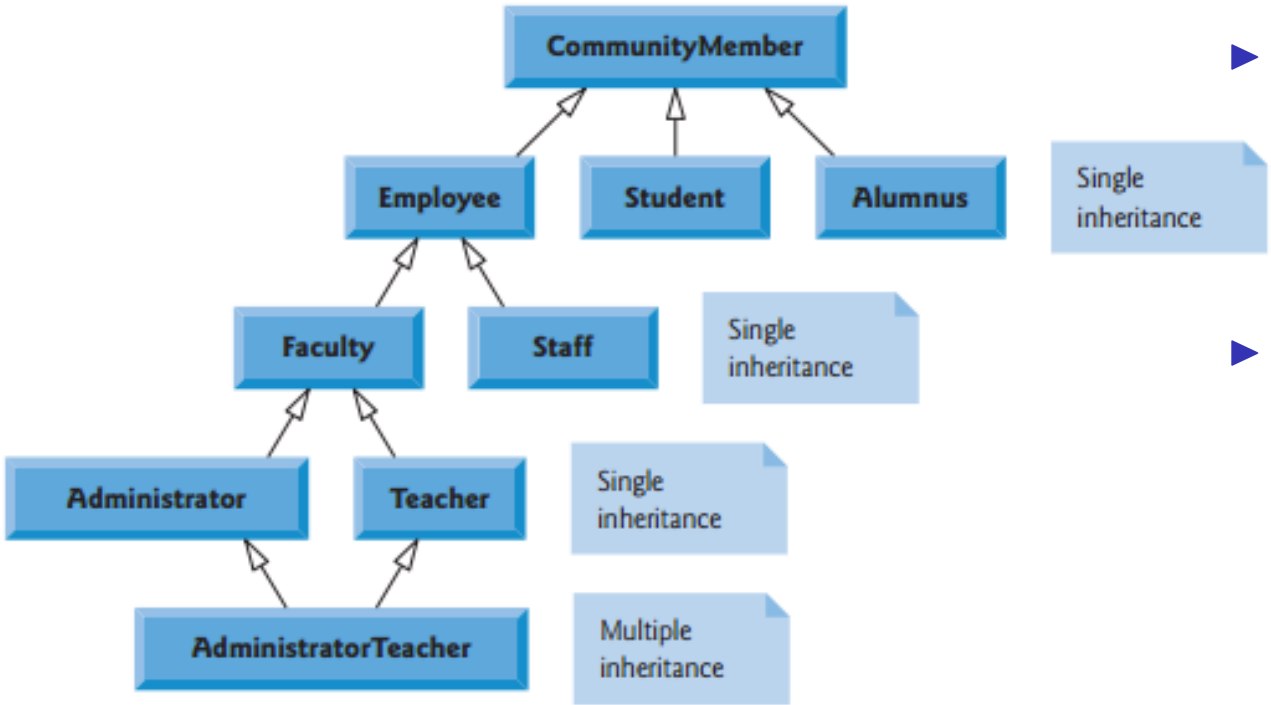
Figure lists several simple examples of base classes and derived classes. Base classes tend to be more general and derived classes tend to be more specific.

Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

Introduction

- Because every derived-class object is an object of its base class, and one base class can have many derived classes,the set of objects represented by a base class typically is larger than the set of objects represented by any of its derived classes.
- For example, the base class Vehicle represents all vehicles, including cars,trucks, boats, air planes, bicycles and soon.
- By contrast,derived class Car represents a smaller, more specific subset of all vehicles.
- Inheritance relationships form class hierarchies. A base class exists in a hierarchical relationship with its derived classes.
- A class becomes either a base class (supplying members to other classes), a derived class (inheriting its members from other classes), or both.

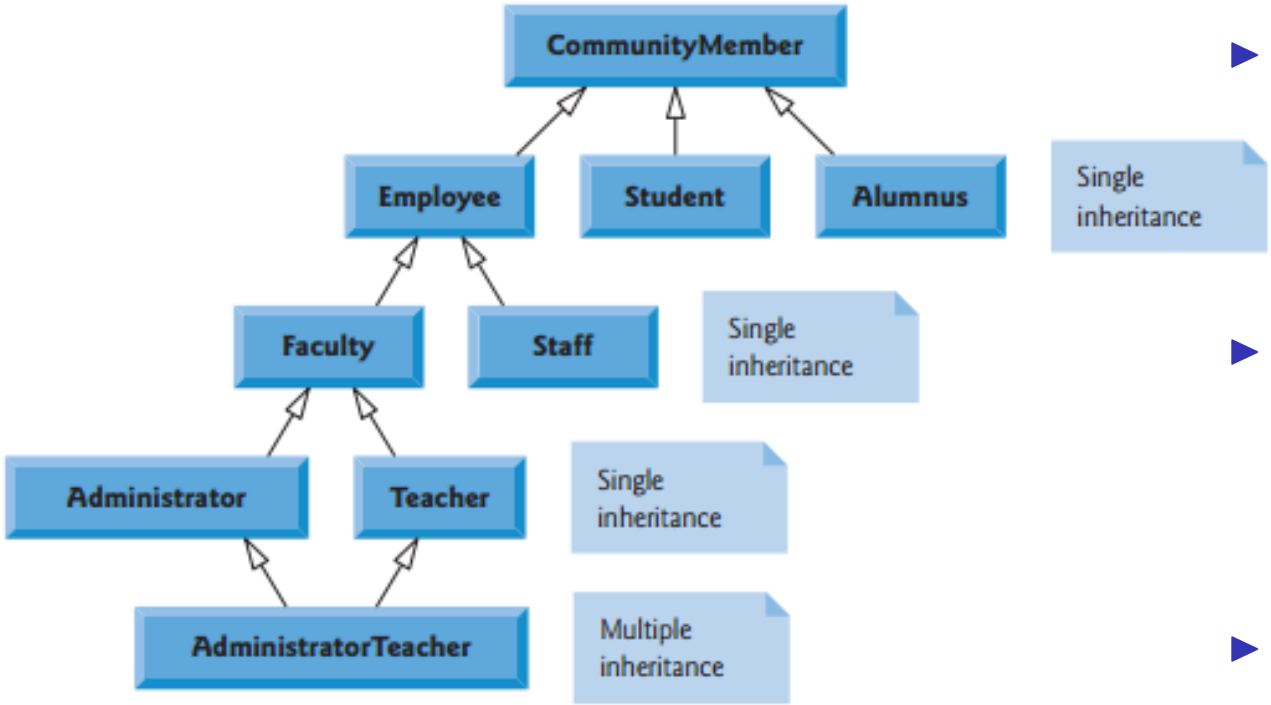
CommunityMember Class Hierarchy



- Let's develop a simple inheritance hierarchy with five levels.

- A university community has thousands of CommunityMembers.
- These CommunityMembers consist of Employees, Students and alumni (each of class Alumnus).
- Employees are either Faculty or Staff.

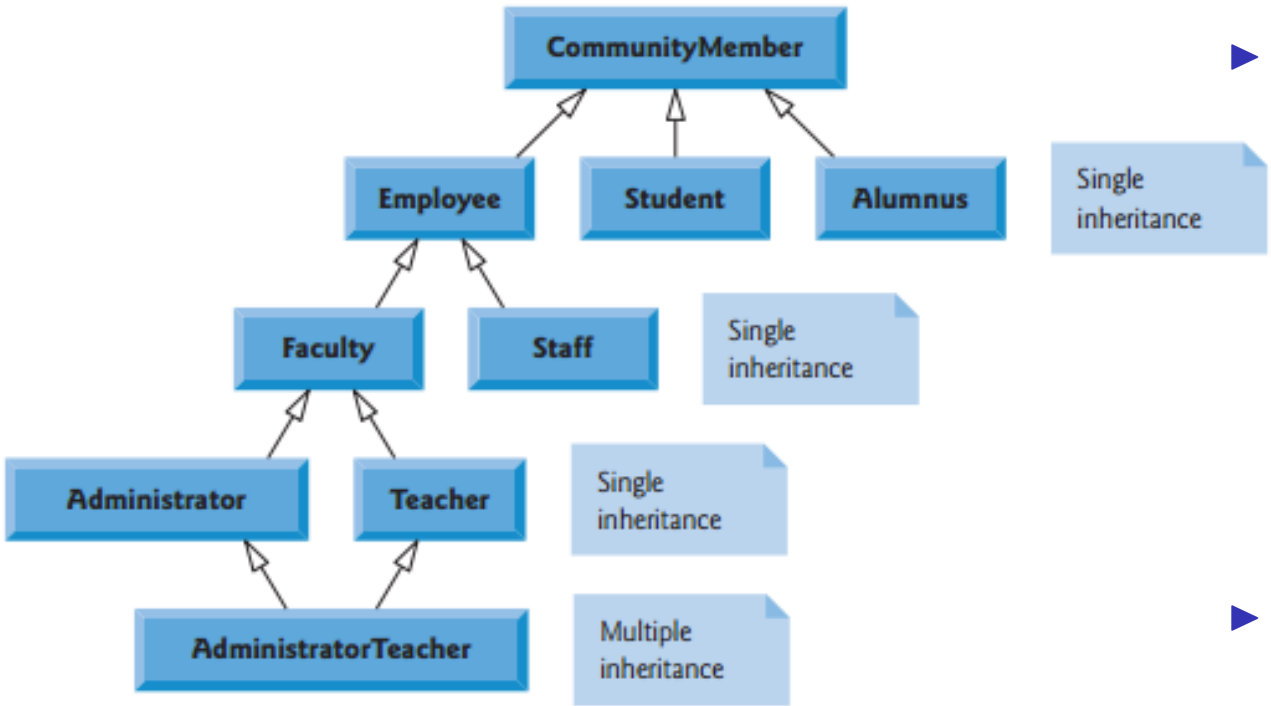
CommunityMember Class Hierarchy



- Faculty are either Administrators or Teachers.
- Some Administrators, however, are also Teachers.

- We've used multiple inheritance to form class AdministratorTeacher.
- **With single inheritance,a class is derived from one base class.**
- With **multiple inheritance**,a derived class **inherits from two or more** (possibly unrelated) **base classes**.

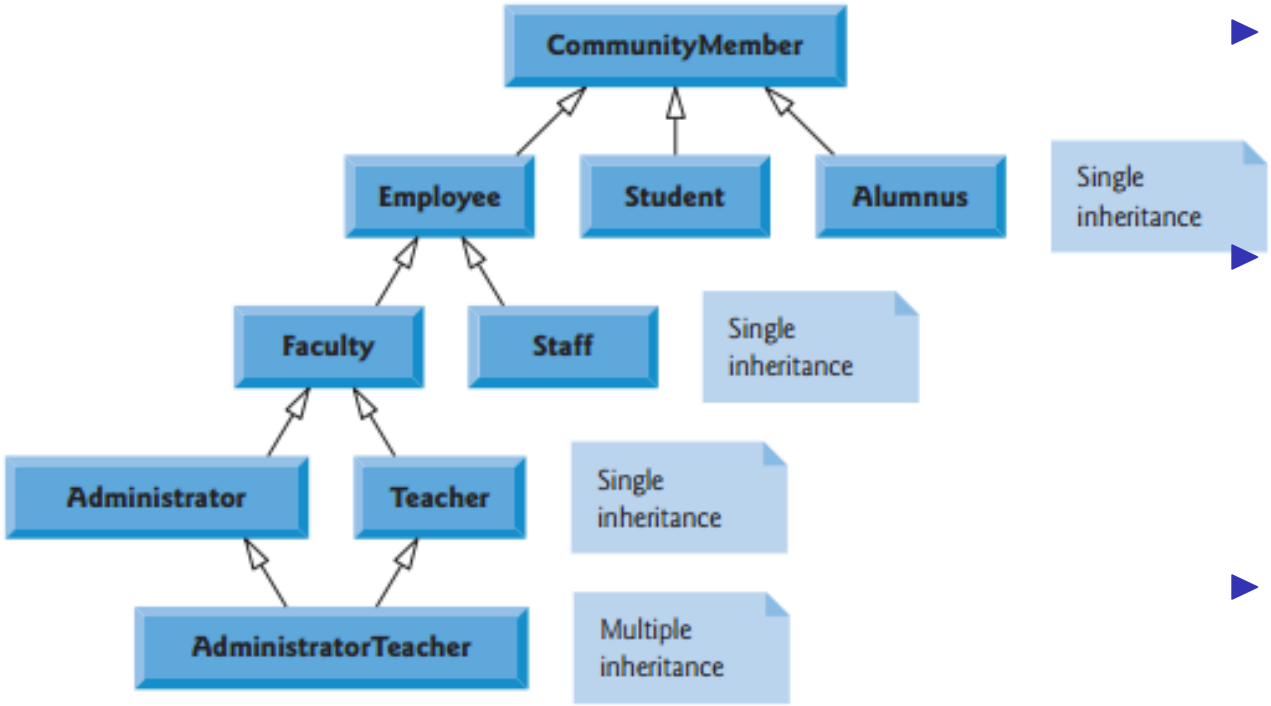
CommunityMember Class Hierarchy



- Each arrow in the hierarchy represents an **is-a relationship**.

- For example, as we follow the arrows in this class hierarchy, we can state "an Employee is a CommunityMember" and "a Teacher is a Faculty member."
- CommunityMember is the direct base class of Employee, Student and Alumnus.

CommunityMember Class Hierarchy

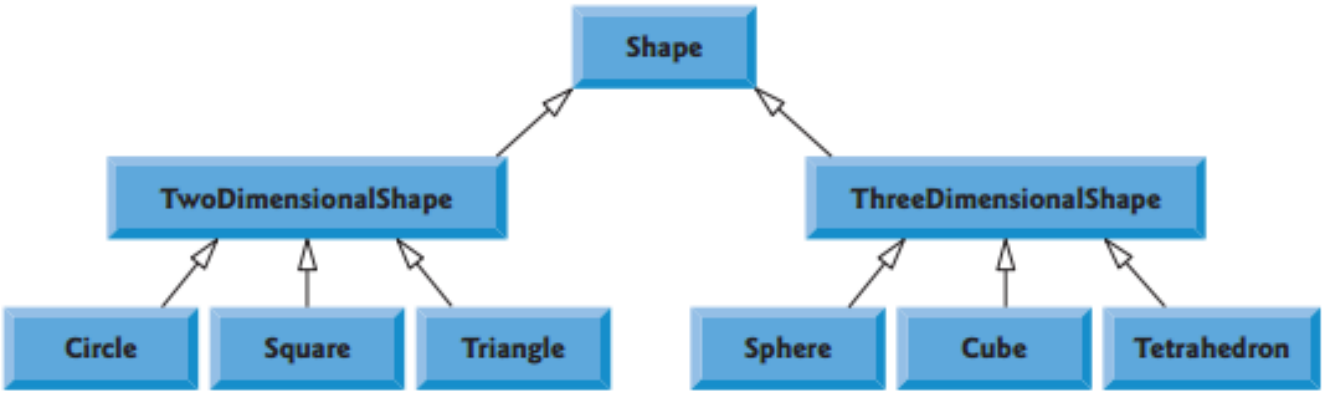


- An indirect base class is inherited from two or more levels up the class hierarchy.
- Starting from the bottom of the diagram, you can follow the arrows upwards and apply the is-a relationship to the top most base class.
- For example, an AdministratorTeacher is an Administrator, is a Faculty member, is an Employee and is a CommunityMember.

- In addition, CommunityMember is an indirect base class of all the other classes in the diagram.

Shape Class Hierarchy

- This hierarchy begins with **base class** Shape.
- Classes TwoDimensionalShape and ThreeDimensionalShape **derive from** base class Shape.
- The third level of this hierarchy contains more specific types of TwoDimensionalShapes and ThreeDimensionalShapes.
- We can follow the arrows from the bottom of the diagram upwards to the topmost base class in this hierarchy to identify several is-a relationships.
- For instance, a Triangle is a TwoDimensionalShape and is a Shape,while a Sphere is a ThreeDimensionalShape and is a Shape.



Superclass and SubClass

- **Superclasses are also called base classes,and subclasses are also called derived classes. Either set of terms is correct. For consistency, this text will use the terms superclass and subclass.**
- Inheritance involves a superclass and a subclass.
- The **superclass** is the general class and the **subclass** is the specialized class.
- You can think of the subclass as an extended version of the superclass.
- **The subclass inherits attributes and methods from the superclass without any of them having to be rewritten.**
- Furthermore, new attributes and methods may be added to the subclass, and that is what makes it a specialized version of the superclass.

An example

- Suppose we are developing a program that a car dealership can use to manage its inventory of used cars.
- The dealership's inventory includes three types of **automobiles**: **cars**, **pickup trucks**, and **sport-utility vehicles (SUVs)**.
- Regardless of the type, the dealership keeps the following data about each automobile:
  - Make
  - Year Model
  - Mileage
  - Price

An example

- Each type of vehicle that is kept in inventory has these general characteristics, plus its own specialized characteristics.
- For cars, the dealership keeps the following additional data:
  Number of doors (2 or 4)
- For pickup trucks, the dealership keeps the following additional data:
  Drive type(two-wheel drive or four-wheel drive)
- For SUVs, the dealership keeps the following additional data:
  Passenger capacity

An example

- In designing this program, one approach would be to write the following three classes:

  A Car class with data attributes for the make, year model, mileage, price, and the number of doors.

  A Truck class with data attributes for the make, year model, mileage, price, and the drive type.

  An SUV class with data attributes for the make, year model, mileage, price, and the passenger capacity.

- This would be an inefficient approach, however, because all three of the classes have a large number of common data attributes. As a result, the classes would contain a lot of duplicated code. In addition, if we discover later that we need to add more common attributes, we would have to modify all three classes.

- **A better approach would be to write an Automobile superclass to hold all the general data about an automobile, then write subclasses for each specific type of automobile.**

Automobile superclass

```
2  class Automobile:
3
4      def __init__(self,make, model, mileage, price):
5          self.__make = make
6          self.__model = model
7          self.__mileage = mileage
8          self.__price = price
9
10     def set_make(self,make):
11         self.__make = make
12
13     def set_model(self,model):
14         self.__model = model
15
16     def set_mileage(self,mileage):
17         self.__mileage = mileage
18
19     def set_price(self,price):
20         self.__price = price
21
22     def get_make(self):
23         return self.__make
24
25     def get_model(self):
26         return self.__model
27
28     def get_mileage(self):
29         return self.__mileage
30
31     def get_price(self):
32         return self.__price
```

- The Automobile class's __init__ method accepts arguments for the vehicle's make, model, mileage, and price. It uses those values to initialize the following data attributes:
  - __make
  - __model
  - __mileage
  - __price
- The methods that appear in lines 10 through 20 are **mutators** for each of the data attributes, and the methods in lines 22 through 32 are the **accessors**.

Automobile superclass

- The Automobile class is a complete class from which we can create objects.
- If we wish, we can write a program that imports the vehicle module and creates instances of the Automobile class.
- However, the Automobile class holds only general data about an automobile.
- It does not hold any of the specific pieces of data that the dealership wants to keep about cars, pickup trucks, and SUVs.
- To hold data about those specific types of automobiles, we will write subclasses that inherit from the Automobile class.

Car subclass

```
35 class Car(Automobile):
36
37     def __init__(self,make, model, mileage, price,doors):
38
39         Automobile.__init__(self,make,model,mileage,price)
40
41         self.__doors = doors
42
43     def set_doors(self,doors):
44         self.__doors = doors
45
46     def get_doors(self):
47         return self.__doors
```

- Take a closer look at the first line of the class declaration, in line 35:
  class Car(Automobile):
- This line indicates that we are defining a class named Car, and it inherits from the Automobile class.
- The Car class is the subclass, and the Automobile class is the superclass.

- If we want to express the relationship between the Car class and the Automobile class, we can say that **a Car is an Automobile**.
- Because the Car class extends the Automobile class, it inherits all of the methods and data attributes of the Automobile class.
- Look at the header for the __init__ method in line 37.
- Notice in addition to the required self parameter, the method has parameters named **make**, **model**, **mileage**, **price**, and **doors**.

Car subclass

- This makes sense because a Car object will have data attributes for the car's make, model, mileage, price, and number of doors.
- Some of these attributes are created by the Automobile class, however, so we need to call the Automobile class's __init__ method and pass those values to it. That happens in line 39:

  Automobile.__init__(self, make, model, mileage, price)
- This statement calls the Automobile class's __init__ method.
- Notice the statement passes the self variable, as well as the make, model, mileage, and price variables as arguments.
- When that method executes, it initializes the __make, __model, __mileage, and __price data attributes.
- Then, in line 41, the __doors attribute is initialized with the value passed into the doors parameter.

Car subclass driver program

```
1  # This program demonstrates the Car class.
2
3  import vehicles
4
5  def main():
6      # Create an object from the Car class.
7      # The car is a 2007 Audi with 12,500 miles, priced
8      # at $21,500.00, and has 4 doors.
9      used_car = vehicles.Car('Audi', 2007, 12500, 21500.0, 4)
10
11     # Display the car's data.
12     print('Make:', used_car.get_make())
13     print('Model:', used_car.get_model())
14     print('Mileage:', used_car.get_mileage())
15     print('Price:', used_car.get_price())
16     print('Number of doors:', used_car.get_doors())
17
18 # Call the main function.
19 main()
```

- Line 3 imports the vehicles module, which contains the class definitions for the Automobile and Car classes.
- Line 9 creates an instance of the Car class, passing 'Audi' as the car's make, 2007 as the car's model, 12,500 as the mileage, 21,500.0 as the car's price, and 4 as the number of doors.
- The resulting object is assigned to the used_car variable.
- The statements in lines 12 through 15 call the object's **get_make**, **get_model**, **get_mileage**, and **get_price** methods.
- Even though the Car class does not have any of these methods, it inherits them from the Automobile class.
- Line 16 calls the get_doors method, which is defined in the Car class.

Truck subclass

```
49 class Truck(Automobile):
50
51     def __init__(self,make, model, mileage, price,drive_type):
52
53         Automobile.__init__(self,make,model,mileage,price)
54
55         self.__drive_type = drive_type
56
57     def set_drive_type(self,drive_type):
58         self.__drive_type = drive_type
59
60     def get_drive_type(self):
61         return self.__drive_type
```

- The Truck class's __init__ method begins in line 49.
- Notice it takes arguments for the truck's make, model, mileage, price, and drive type.
- Just as the Car class did, the Truck class calls the Automobile class's __init__ method (in line 53) passing the make, model, mileage, and price as arguments.
- Line 87 creates the __drive_type attribute, initializing it to the value of the drive_type parameter.
- The set_drive_type method in lines 57 through 58 is the mutator for the __drive_type attribute, and the get_drive_type method in lines 60 through 61 is the accessor for the attribute.

SUV subclass

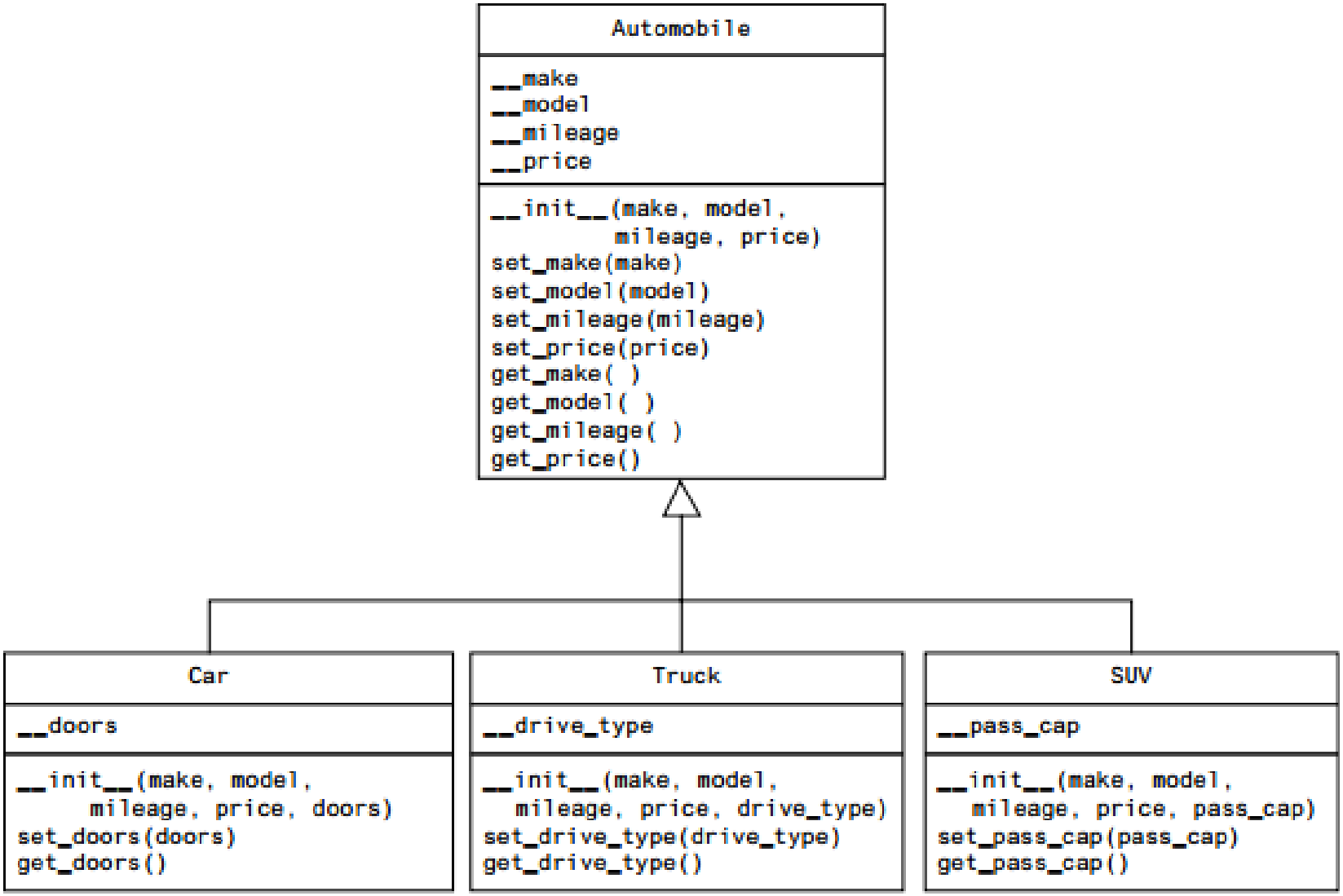
```
64 class SUV(Automobile):
65
66     def __init__(self,make, model, mileage, price,pass_cap):
67
68         Automobile.__init__(self,make,model,mileage,price)
69
70         self.__pass_cap = pass_cap
71
72     def set_pass_cap(self,pass_cap):
73         self.__pass_cap = pass_cap
74
75     def get_pass_cap(self):
76         return self.__pass_cap
```

- The SUV class's __init__ method begins in line 64.
- It takes arguments for the vehicle's make, model, mileage, price, and passenger capacity.
- Just as the Car and Truck classes did, the SUV class calls the Automobile class's __init__ method (in line 68) passing the make, model, mileage, and price as arguments.
- Line 70 creates the __pass_cap attribute, initializing it to the value of the pass_cap parameter.
- The set_pass_cap method in lines 72 through 73 is the mutator for the __pass_cap attribute, and the get_pass_cap method in lines 75 through 76 is the accessor for the attribute.

Driver program for car_truck_suv subclasses

```
2  import vehicles
3
4  def main():
5
6      car = vehicles.Car('BMW', 2001, 70000, 15000.0, 4)
7      truck = vehicles.Truck('Toyota', 2002, 40000, 12000.0, '4WD')
8      suv = vehicles.SUV('Volvo', 2000, 30000, 18500.0, 5)
9
10     print('USED CAR INVENTORY')
11     print('==================')
12
13     print('The following car is in inventory:')
14     print('Make:', car.get_make())
15     print('Model:', car.get_model())
16     print('Mileage:', car.get_mileage())
17     print('Price:', car.get_price())
18     print('Number of doors:', car.get_doors())
19     print()
20
21     print('The following pickup truck is in inventory.')
22     print('Make:', truck.get_make())
23     print('Model:', truck.get_model())
24     print('Mileage:', truck.get_mileage())
25     print('Price:', truck.get_price())
26     print('Drive type:', truck.get_drive_type())
27     print()
28
29     print('The following SUV is in inventory.')
30     print('Make:', suv.get_make())
31     print('Model:', suv.get_model())
32     print('Mileage:', suv.get_mileage())
33     print('Price:', suv.get_price())
34     print('Passenger Capacity:', suv.get_pass_cap())
35
36 main()
```

Inheritance in UML Diagrams



Introduction

- The term **polymorphism** refers to an object's ability to take different forms.
- It is a powerful feature of object-oriented programming.
- In this section, we will look at two essential ingredients of polymorphic behavior:
  - The ability to define a method in a superclass, then define a method with the same name in a subclass. When a subclass method has the same name as a superclass method, it is often said that the subclass method overrides the superclass method.
  - The ability to call the correct version of an overridden method, depending on the type of object that is used to call it. If a subclass object is used to call an overridden method, then the subclass's version of the method is the one that will execute. If a superclass object is used to call an overridden method, then the superclass's version of the method is the one that will execute.

Mammal Superclass

```
2  class Mammal:
3
4      def __init__(self,species):
5          self.__species = species
6
7      def show_species(self):
8          print('I am a', self.__species)
9
10     def make_sound(self):
11         print('Grrrrr')
```

- The Mammal class has three methods: __init__, show_species and make_sound.

- Here is an example of code that creates an instance of the class and calls these methods:

```
import animals
mammal =
animals.Mammal('regular mammal')
mammal.show_species()
mammal.make_sound()
```

- This code will display the following:

```
I am a regular mammal
Grrrrr
```

Dog Subclass

```
13 class Dog(Mammal):
14
15     def __init__(self):
16
17         Mammal.__init__(self,'Dog')
18
19     def make_sound(self):
20         print('Woof! Woof!')
```

- Even though the Dog class inherits the __init__ and make_sound methods that are in the Mammal class, those methods are not adequate for the Dog class.
- So, the Dog class has its own __init__ and make_sound methods, which perform actions that are more appropriate for a dog.
- We say that the __init__ and make_sound methods in the Dog class override the __init__ and make_sound methods in the Mammal class.
- Here is an example of code that creates an instance of the Dog class and calls the methods:

  import animals
  dog = animals.Dog()
  dog.show_species()
  dog.make_sound()
- This code will display the following:

  I am a Dog
  Woof! Woof!
- When we use a Dog object to call the show_species and make_sound methods, the versions of these methods that are in the Dog class are the ones that execute.

Cat Subclass

```
23 class Cat(Mammal):
24
25     def __init__(self):
26
27         Mammal.__init__(self,'Cat')
28
29     def make_sound(self):
30         print('Meow')
```

- The Catclass also overrides the Mammal class's __init__ and make_sound methods.

- Here is an example of code that creates an instance of the Cat class and calls the methods:

```
import animals
cat = animals.Cat()
cat.show_species()
cat.make_sound()
```

- This code will display the following:

```
I am a Cat
Meow
```

- When we use a Cat object to call the show_species and make_sound methods, the versions of these methods that are in the Cat class are the ones that execute.

The isinstance Function

```
1  # This program demonstrates polymorphism.
2
3  import animals
4
5  def main():
6      # Create a Mammal object, a Dog object, and
7      # a Cat object.
8      mammal = animals.Mammal('regular animal')
9      dog = animals.Dog()
10     cat = animals.Cat()
11
12     # Display information about each one.
13     print('Here are some animals and')
14     print('the sounds they make.')
15     print('-------------------------')
16     show_mammal_info(mammal)
17     print()
18     show_mammal_info(dog)
19     print()
20     show_mammal_info(cat)
21
22 # The show_mammal_info function accepts an object
23 # as an argument, and calls its show_species
24 # and make_sound methods.
25
26 def show_mammal_info(creature):
27     creature.show_species()
28     creature.make_sound()
29
30 # Call the main function.
31 main()
```

- Polymorphism gives us a great deal of flexibility when designing programs.
- For example, look at the following function:

```
def show_mammal_info(creature):
    creature.show_species()
    creature.make_sound()
```

- We can pass any object as an argument to this function, and as long as it has a show_species method and a make_sound method, the function will call those methods.
- In essence, we can pass any object that "is a" Mammal(or a subclass of Mammal) to the function.

The isinstance Function

```
1  def main():
2      # Pass a string to show_mammal_info …
3      show_mammal_info('I am a string')
4
5  # The show_mammal_info function accepts an object
6  # as an argument, and calls its show_species
7  # and make_sound methods.
8
9  def show_mammal_info(creature):
10     creature.show_species()
11     creature.make_sound()
12
13 # Call the main function.
14 main()
```

- But what happens if we pass an object that is not a Mammal, and not of a subclass of Mammal to the function?

- In line 3, we call the show_mammal_info function passing a string as an argument.
- When the interpreter attempts to execute line 10, however, an AttributeError exception will be raised because strings do not have a method named show_species.
- We can prevent this exception from occurring by using the built-in function **isinstance**.
- You can use the isinstance function to determine whether an object is an instance of a specific class, or a subclass of that class.
- Here is the general format of the function call:
  isinstance(object, ClassName)

The isinstance Function

```
1  # This program demonstrates polymorphism.
2
3  import animals
4
5  def main():
6      # Create an Mammal object, a Dog object, and
7      # a Cat object.
8      mammal = animals.Mammal('regular animal')
9      dog = animals.Dog()
10     cat = animals.Cat()
11
12     # Display information about each one.
13     print('Here are some animals and')
14     print('the sounds they make.')
15     print('-------------------------')
16     show_mammal_info(mammal)
17     print()
18     show_mammal_info(dog)
19     print()
20     show_mammal_info(cat)
21     print()
22     show_mammal_info('I am a string')
23
24 # The show_mammal_info function accepts an object
25 # as an argument and calls its show_species
26 # and make_sound methods.
27
28 def show_mammal_info(creature):
29     if isinstance(creature, animals.Mammal):
30         creature.show_species()
31         creature.make_sound()
32     else:
33         print('That is not a Mammal!')
34
35 # Call the main function.
36 main()
```

- In the general format, object is a reference to an object, and ClassName is the name of a class.
- If the object referenced by object is an instance of ClassName or is an instance of a subclass of ClassName, the function returns true. Otherwise, it returns false.
- In lines 16, 18, and 20 we call the show_mammal_info function, passing references to a Mammal object, a Dog object, and a Cat object.
- In line 22, however, we call the function and pass a string as an argument. Inside the show_mammal_info function, the if statement in line 29 calls the isinstance function to determine whether the argument is an instance of Mammal(or a subclass). If it is not, an error message is displayed.