INTRODUCTION TO COMPUTER SCIENCE: PROGRAMMING METHODOLOGY

Lecture 7 Object Oriented Programming Features

Prof. Wei Cai

School of Science and Engineering

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

TOPICS OF OBJECT-ORIENTED PROGRAMMING

- Part I
  - Call-by-Reference / Call-by-Value
  - Encapsulation
- Part II
  - Inheritance
  - Overriding
  - Object Class
- Part III
  - Polymorphism
  - Dynamic Binding

THE PROBLEM OF CALL-BY-REFERENCE / CALL-BY-VALUE

Circle.py

```
import math

class Circle:
    # Construct a circle object
    def __init__(self, radius = 1):
        self.radius = radius

    def getPerimeter(self):
        return 2 * self.radius * math.pi

    def getArea(self):
        return self.radius * self.radius * math.pi

    def setRadius(self, radius):
        self.radius = radius
```

```
Radius                  Area

1                       3.141592653589793
2                       12.566370614359172
3                       29.274333882308138
4                       50.26548245743669
5                       79.53981633974483

Radius is 6
n is 5
```

Test.py

```
from Circle import Circle

def main():
    # Create a Circle object with radius 1
    myCircle = Circle()

    # Print areas for radius 1, 2, 3, 4, and 5
    n = 5
    printAreas(myCircle, n)

    # Display myCircle.radius and times
    print("\nRadius is", myCircle.radius)
    print("n is", n)

# Print a table of areas for radius
def printAreas(c, times):
    print("Radius \t\tArea")
    while times >= 1:
        print(c.radius, "\t\t", c.getArea())
        c.radius = c.radius + 1
        times = times - 1

main() # Call the main function
```

THE PROBLEM OF CALL-BY-REFERENCE / CALL-BY-VALUE

```
def main():
    # Create a Circle object with radius 1
    myCircle = Circle()

    # Print areas for radius 1, 2, 3, 4, and 5
    n = 5
    printAreas(myCircle, n)

    # Display myCircle.radius and times
    print("\nRadius is", myCircle.radius)
    print("n is", n)
```

```
Radius                  Area

1                       3.141592653589793
2                       12.566370614359172
3                       29.274333882308138
4                       50.26548245743669
5                       79.53981633974483

Radius is 6
n is 5
```

```
# Print a table of areas for radius
def printAreas(c, times):
    print("Radius \t\tArea")
    while times >= 1:
        print(c.radius, "\t\t", c.getArea())
        c.radius = c.radius + 1
        times = times - 1
```

CALL-BY-REFERENCE / CALL-BY-VALUE

- **Call-by-Value**
  - Copies the value of an argument into the formal parameter of that function
  - Changes made to the parameter of the main function do not affect the argument

- **Call-by-Reference**
  - Copies the reference (address) of an argument into the formal parameter.
    - The reference is used to access the actual argument used in the function call.
  - Changes made in the parameter alter the passing argument.

```
class B:
        def __init__(self):
                self.data = 0

def f(a, b, c):
        a = 1
        b.data = 1
        c = B()
        c.data = 1

a = 0
b = B()
c = B()
f(a,b,c)
print(a, b.data, c.data)
```

IS PYTHON CALL-BY-VALUE OR CALL-BY-REFERENCE?

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). [1] When a function calls another function, a new local symbol table is created for that call.

From: https://docs.python.org/3.4/tutorial/controlflow.html

DISCUSSIONS OVER CALL-BY-VALUE/CALL-BY-REFERENCE

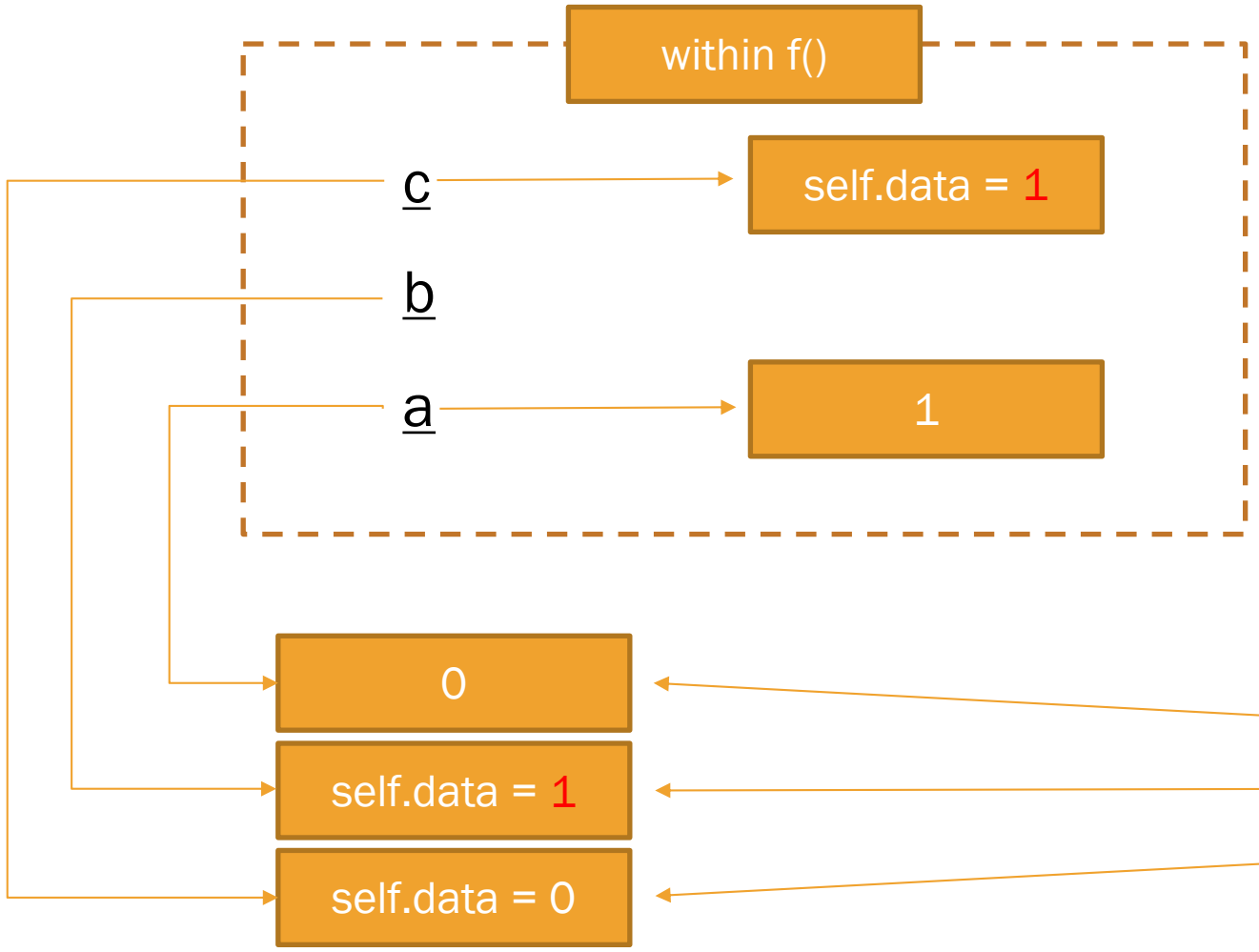
- https://www.quora.com/Are-arguments-passed-by-value-or-by-reference-in-Python
- https://www.zhihu.com/question/20591688

- Pass-by-value for immutable object
- Pass-by-reference for mutable object

Personal Tips

Think about the definition of = in Python

CALL-BY-REFERENCE / CALL-BY-VALUE




```
class B:
        def __init__(self):
                self.data = 0

def f(a, b, c):
        a = 1
        b.data = 1
        c = B()
        c.data = 1
```

```
a = 0
b = B()
c = B()
f(a,b,c)
print(a, b.data, c.data)
```

```
0 1 0
```

CALL-BY-REFERENCE / CALL-BY-VALUE

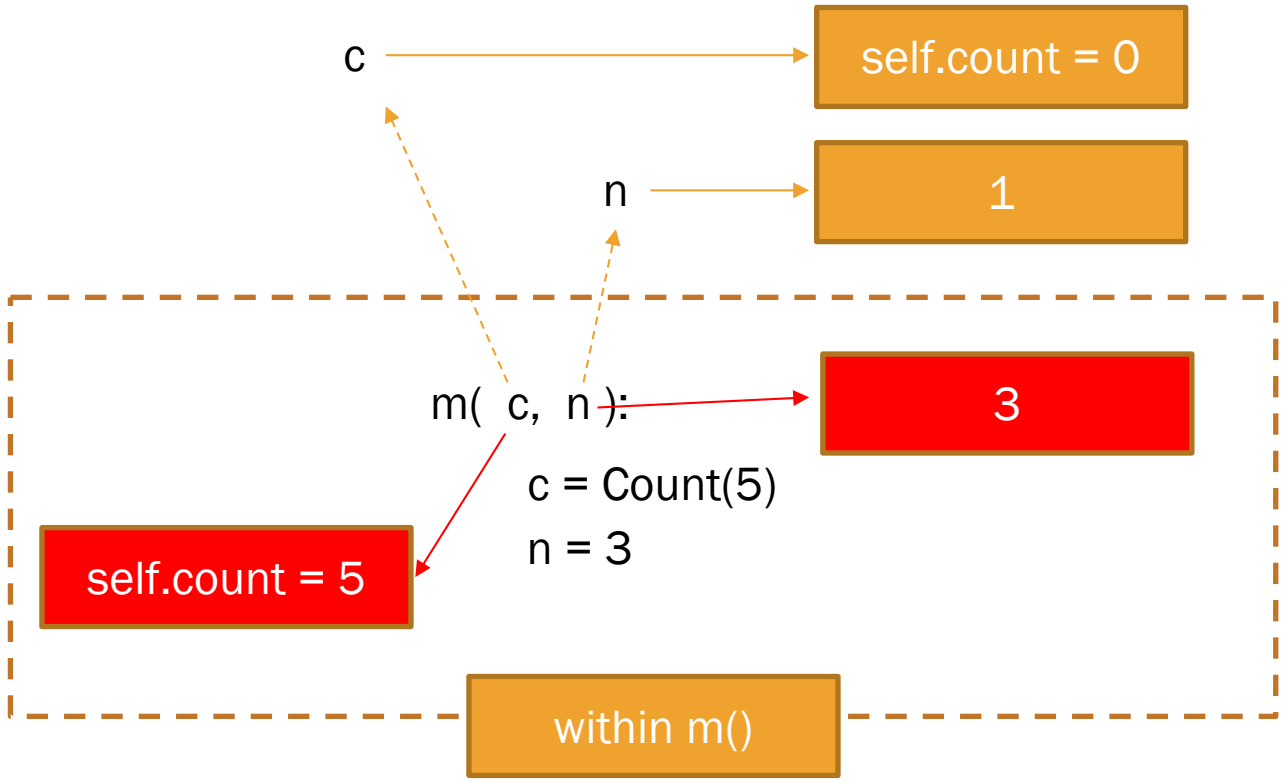
- List, Dictionary, Tuple

```
def f(myList, myDict,myTuple):
        myList[0]=0
        myList=[2,3,4]
        myDict["key1"] = "value2"
        myTuple = (2,3,4)

myList = [1,2,3]
myDict = {"key1":"value1"}
myTuple = (1,2,3)

f(myList, myDict,myTuple)

print(myList)
print(myDict)
print(myTuple)
```

```
[0, 2, 3]
{'key1': 'value2'}
(1, 2, 3)
```

EXAMPLE

```
class Count:
    def __init__(self, count = 0):
        self.count = count

def main():
    c = Count()
    n = 1
    m(c, n)

    print("count is", c.count)
    print("n is", n)

def m(c, n):
    c = Count(5)
    n = 3

main() # Call the main function
```



What would be the output of the above program?

```
count is 0
n is 1
```

ISSUE OF DIRECT ACCESS TO DATA FIELDS

- data may be tampered with

  c.radius = -1

  tv1.channel = 125

- classes become difficult to maintain and vulnerable to bugs
  - Suppose you want to modify the Circle class to ensure that the radius is non-negative after other programs have already used the class
  - You have to change not only the Circle class but also the programs that use it, because the clients may have modified the radius directly

Test.py

```
from Circle import Circle

def main():
    # Create a Circle object with radius 1
    myCircle = Circle()

    # Print areas for radius 1, 2, 3, 4, and 5
    n = 5
    printAreas(myCircle, n)

    # Display myCircle.radius and times
    print("\nRadius is", myCircle.radius)
    print("n is", n)

# Print a table of areas for radius
def printAreas(c, times):
    print("Radius \t\tArea")
    while times >= 1:
        print(c.radius, "\t\t", c.getArea())
        c.radius = c.radius + 1
        times = times - 1

main() # Call the main function
```

HOW TO PROTECT PRIVACY！！

- Data Hiding
  - Prevent other programmers from directly accessing the data fields of your class is a common industrial practice
- This can be done by defining private data fields
- In Python
  - name of private data fields: two leading underscores
  - name of private method: two leading underscores

PRIVATE DATA FIELDS

- Private data fields and methods can be accessed within a class, but they cannot be accessed outside the class
- Define some methods to allow access to private data fields

Test.py

```
import math

class Circle:
        def __init__(self, radius = 1):
                self.__radius=radius
        def getRadius(self):
                return self.__radius
        def getPerimeter(self):
                return 2*self.__radius*math.pi
        def getArea(self):
                return self.__radius**2*math.pi
        def setRadis(self, radius):
                self.__radius=radius

c = Circle(5)
print(c.getRadius(),"\t",c.getArea(),"\t",c.getPerimeter())
c.setRadis(10)
print(c.getRadius(),"\t",c.getArea(),"\t",c.getPerimeter())
```

PRACTICE

```
class A:
    def __init__(self, i):
        self.__i = i

def main():
    a = A(5)
    print(a.__i)

main() # Call the main function
```

- What is the problem with this program?

PRACTICE

```
def main():
    a = A()
    a.print()

class A:
    def __init__(self, newS = "Welcome"):
        self.__s = newS

    def print(self):
        print(self.__s)

main() # Call the main function
```

- Is the above code correct?
- If yes, what would be the output?

Welcome

PRACTICE

```
class A:
    def __init__(self, on):
        self.__on = not on

def main():
    a = A(False)
    print(a.on)

main() # Call the main function
```

- Is the above code correct?
- If not, how do we fix it?

DESCRIPTIVE: NAMING STYLES

- Single Pre Underscore: _variable
  - for internal use, but it doesn't stop you from accessing them
  - effects the names that are imported from the module
    - doesn't import the names which starts with a **single pre underscore**
- Single Post Underscore: variable_
  - used to avoid conflicts with the Python Keywords
- Double Pre Underscores: __variable
  - used for the name mangling
  - private
  - to avoid the overriding of the variable in subclasses
- Double Pre and Post Underscores: __variable__
  - magic methods
  - it's better to stay away from them

ABSTRACTION

- separate the implementation of a part of code from the usage of that code
- makes your code easy to maintain, debug and reuse

- In software engineering, there are many levels of abstraction

- Function Abstraction
  - separating the implementation of a function from its usage

EXAMPLE:

```
# Return the gcd of two integers
def gcd(n1, n2):
    gcd = 1 # Initial gcd is 1
    k = 2   # Possible gcd

    while k <= n1 and k <= n2:
        if n1 % k == 0 and n2 % k == 0:
            gcd = k # Update gcd
        k += 1

    return gcd # Return gcd

# Prompt the user to enter two integers
n1 = eval(input("Enter the first integer: "))
n2 = eval(input("Enter the second integer: "))

print("The greatest common divisor for", n1,
    "and", n2, "is", gcd(n1, n2))
```

CASE STUDY:

PRINTING FIRST 50 PRIME NUMBERS

```
def printPrimeNumbers(numberOfPrimes):
    NUMBER_OF_PRIMES = 50 # Number of primes to display
    NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
    count = 0 # Count the number of prime numbers
    number = 2 # A number to be tested for primeness

    # Repeatedly find prime numbers
    while count < numberOfPrimes:

        #Determine whether a number is a prime number
        isPrime = True
        divisor = 2
        while (divisor<=number/2):
            if number%divisor ==0:
                isPrime = False
                break
            divisor +=1


        # Print the prime number and increase the count
        if isPrime==True:
            count += 1 # Increase the count

            print(number, end = " ")
            if count % NUMBER_OF_PRIMES_PER_LINE == 0:
                # Print the number and advance to the new line
                print()

        # Check if the next number is prime
        number += 1

printPrimeNumbers(20)
```

```
def isPrime(number):
    divisor = 2
    while divisor <= number / 2:
        if number % divisor == 0:
            return False
        divisor +=1
    return True

def printPrimeNumbers(numberOfPrimes=50):
    NUMBER_OF_PRIMES_PER_LINE = 10
    count = 0
    number = 2
    while count<numberOfPrimes:
        if isPrime(number):
            count+=1
            print(number, end=" ")
            if count % NUMBER_OF_PRIMES_PER_LINE == 0:
                print()
        number+=1

print("The first 50 prime numbers are:")
printPrimeNumbers();
```

Write and maintain isPrime()

Programmer 1

Write and maintain printPrimeNumbers()

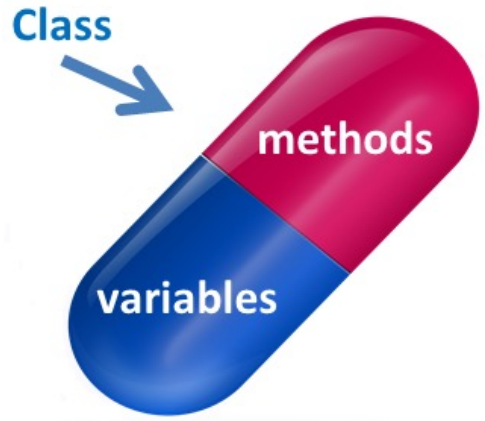
Programmer 2

CLASS ABSTRACTION

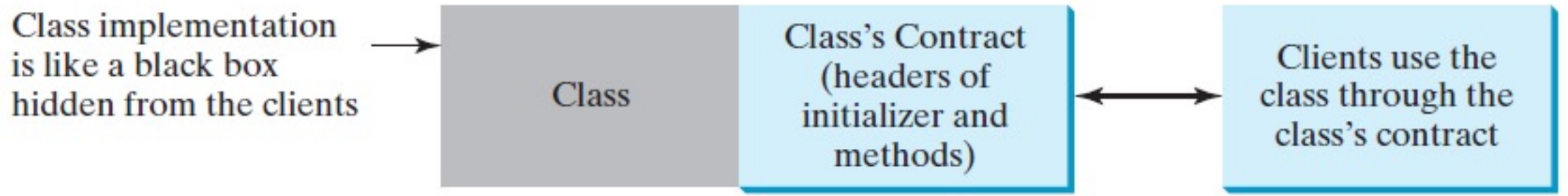
- Separating class implementation from the use of a class
- The class implementation details are invisible from the user
- The class's collection of methods, together with the description of how these methods are expected to behave, serves as the class's contract with the client

CLASS ENCAPSULATION

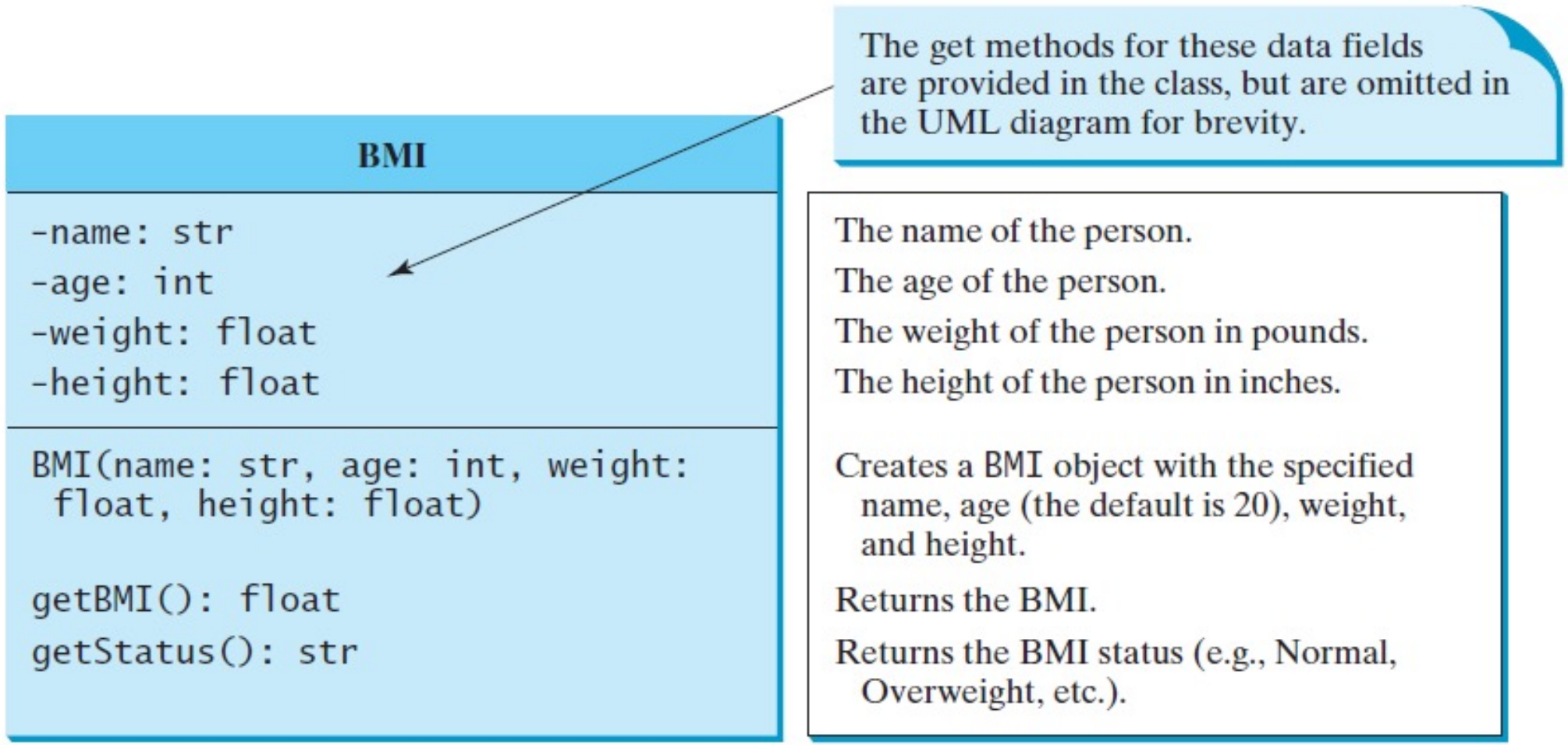
- Combines data and methods into a single object and hides the data fields and method implementation from the user



- The user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user.



EXAMPLE – BMI CALCULATION CONTRACT



THE CODE TO USE BMI CLASS

- We can use the BMI class if you have its contract
- You don't need to know the details about how it is implemented!!

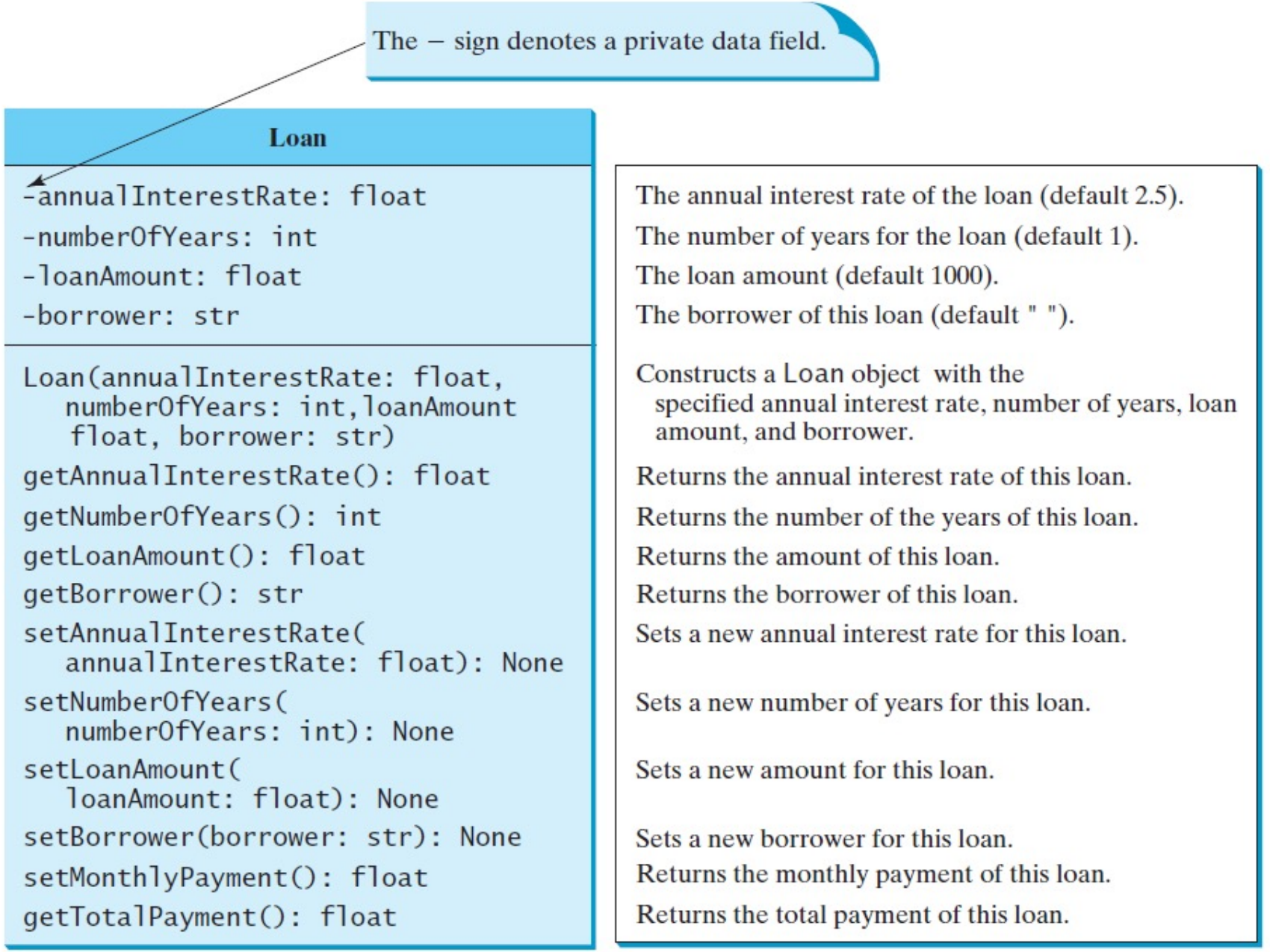
```
from BMI import BMI

def main():
    bmi1 = BMI("John Doe", 18, 145, 70)
    print("The BMI for", bmi1.getName(), "is",
        bmi1.getBMI(), bmi1.getStatus())

    bmi2 = BMI("Peter King", 50, 215, 70)
    print("The BMI for", bmi2.getName(), "is",
        bmi2.getBMI(), bmi2.getStatus())

main() # Call the main function
```

THE BMI CLASS

```
class BMI:
    def __init__(self, name, age, weight, height):
        self.__name = name
        self.__age = age
        self.__weight = weight
        self.__height = height

    def getBMI(self):
        KILOGRAMS_PER_POUND = 0.45359237
        METERS_PER_INCH = 0.0254
        bmi = self.__weight * KILOGRAMS_PER_POUND /  \
           ((self.__height * METERS_PER_INCH) * \
           (self.__height * METERS_PER_INCH))
        return round(bmi * 100) / 100
```

```
    def getStatus(self):
        bmi = self.getBMI()
        if bmi < 18.5:
            return "Underweight"
        elif bmi < 25:
            return "Normal"
        elif bmi < 30:
            return "Overweight"
        else:
            return "Obese"

    def getName(self):
        return self.__name

    def getAge(self):
        return self.__age

    def getWeight(self):
        return self.__weight

    def getHeight(self):
        return self.__height
```

EXAMPLE - LOAN



```
from Loan import Loan

def main():
    # Enter yearly interest rate
    annualInterestRate = eval(input
        ("Enter yearly interest rate, for example, 7.25: "))

    # Enter number of years
    numberOfYears = eval(input(
        "Enter number of years as an integer: "))

    # Enter loan amount
    loanAmount = eval(input(
        "Enter loan amount, for example, 120000.95: "))

    # Enter a borrower
    borrower = input("Enter a borrower's name: ")

    # Create a Loan object
    loan = Loan(annualInterestRate, numberOfYears,
        loanAmount, borrower)

    # Display loan date, monthly payment, and total payment
    print("The loan is for", loan.getBorrower())
    print("The monthly payment is",
        format(loan.getMonthlyPayment(), ".2f"))
    print("The total payment is",
        format(loan.getTotalPayment(), ".2f"))

main() # Call the main function
```

```
Enter yearly interest rate, for example, 7.25: 2.5 Enter
Enter number of years as an integer: 5 Enter
Enter loan amount, for example, 120000.95: 1000 Enter
Enter a borrower's name: John Jones Enter
The loan is for John Jones
The monthly payment is 17.75
The total payment is 1064.84
```

EXAMPLE OF LOAN CLASS

```
class Loan :
    def __init__(self, annualInterestRate = 2.5,
        numberOfYears = 1, loanAmount = 1000, borrower = " "):
        self.__annualInterestRate = annualInterestRate
        self.__numberOfYears = numberOfYears
        self.__loanAmount = loanAmount
        self.__borrower = borrower

    def getAnnualInterestRate(self):
        return self.__annualInterestRate

    def getNumberOfYears(self):
        return self.__numberOfYears

    def getLoanAmount(self):
        return self.__loanAmount

    def getBorrower(self):
        return self.__borrower

    def setAnnualInterestRate(self, annualInterestRate):
        self.__annualInterestRate = annualInterestRate

    def setNumberOfYears(self, numberOfYears):
        self.__numberOfYears = numberOfYears

    def setLoanAmount(self, loanAmount):
        self.__loanAmount = loanAmount

    def setBorrower(self, borrower):
        self.__borrower = borrower

    def getMonthlyPayment(self):
        monthlyInterestRate = self.__annualInterestRate / 1200
        monthlyPayment = \
          self.__loanAmount * monthlyInterestRate / (1 - (1 /
          (1 + monthlyInterestRate) ** (self.__numberOfYears * 12)))
        return monthlyPayment

    def getTotalPayment(self):
        totalPayment = self.getMonthlyPayment() * \
            self.__numberOfYears * 12
        return totalPayment
```

PRACTICE

(The Rectangle class) Following the example of the Circle class, design a class named Rectangle to represent a rectangle. The class contains:

- Two data fields named width and height.
- A constructor that creates a rectangle with the specified width and height. The default values are 1 and 2 for the width and height, respectively.
- A method named getArea() that returns the area of this rectangle.
- A method named getPerimeter() that returns the perimeter.

ANSWER: RECTANGLE CLASS

```
class Rectangle:
    # Construct a rectangle object
    def __init__(self, width = 1, height = 2):
        self.width = width
        self.height = height

    def getArea(self):
        return self.width * self.height

    def getPerimeter(self):
        return 2 * (self.width + self.height)
```

ANSWER: MAIN() FUNCTION

```
def main():
    # Create a rectangle with width 4 and height 40
    r1 = Rectangle(4, 40)
    print("The width of the rectangle is", r1.width)
    print("The height of the rectangle is", r1.height)
    print("The area of the rectangle is", r1.getArea())
    print("The perimeter of the rectangle is", r1.getPerimeter())

    # Create a rectangle with width 3.5 and height 35.9
    r1 = Rectangle(3.5, 35.9)
    print("The width of the rectangle is", r1.width)
    print("The height of the rectangle is", r1.height)
    print("The area of the rectangle is", r1.getArea())
    print("The perimeter of the rectangle is", r1.getPerimeter())

main()
```

PRACTICE

(The Stock class) Design a class named Stock to represent a company's stock that contains:

- A private string data field named symbol for the stock's symbol.
- A private string data field named name for the stock's name.
- A private float data field named previousClosingPrice that stores the stock price for the previous day.
- A private float data field named currentPrice that stores the stock price for the current time.
- A constructor that creates a stock with the specified symbol, name, previous price, and current price.
- A get method for returning the stock name.
- A get method for returning the stock symbol.
- Get and set methods for getting/setting the stock's previous price.
- Get and set methods for getting/setting the stock's current price.
- A method named getChangePercent() that returns the percentage changed from previousClosingPrice to currentPrice.

ANSWER: STOCK CLASS

```
class Stock:
    # Construct a stock object
    def __init__(self, name, symbol, previousPrice, currentPrice):
        self.__name = name
        self.__symbol = symbol
        self.__previousPrice = previousPrice
        self.__currentPrice = currentPrice

    def getName(self):
        return self.__name

    def getSymbol(self):
        return self.__symbol

    def getPreviousPrice(self):
        return self.__previousPrice

    def getCurrentPrice(self):
        return self.__currentPrice

    def setPreviousPrice(self, previousPrice):
        self.__previousPrice = previousPrice

    def setCurrentPrice(self, currentPrice):
        self.__currentPrice = currentPrice

    def getChangePercent(self):
        return format((self.__currentPrice - self.__previousPrice) * 100 / self.__previousPrice, "5.2f") + "%"
```

ANSWER: MAIN() FUNCTION

```
def main():
    # Create a stock
    stock = Stock("Intel", "INTC", 20.5, 20.35)
    print("The price change is", stock.getChangePercent())

main()
```

PRACTICE

(*Algebra: quadratic equations*) Design a class named `QuadraticEquation` for a quadratic equation $ax^2 + bx + c = 0$. The class contains:

- The private data fields `a`, `b`, and `c` that represent three coefficients.
- A constructor for the arguments for `a`, `b`, and `c`.
- Three `get` methods for `a`, `b`, and `c`.
- A method named `getDiscriminant()` that returns the discriminant, which is $b^2 - 4ac$.
- The methods named `getRoot1()` and `getRoot2()` for returning the two roots of the equation using these formulas:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `0` if the discriminant is negative.

```
import math

class QuadraticEquation:
    def __init__(self, a, b, c):
        self.__a = a
        self.__b = b
        self.__c = c

    def getA(self):
        return self.__a

    def getB(self):
        return self.__b

    def getC(self):
        return self.__c

    def getDiscriminant(self):
        return self.__b * self.__b - 4 * self.__a * self.__c

    def getRoot1(self):
        if self.getDiscriminant() < 0:
            return 0
        else:
            return (-self.__b + self.getDiscriminant()) / (2 * self.__a)

    def getRoot2(self):
        if self.getDiscriminant() < 0:
            return 0
        else:
            return (-self.__b - self.getDiscriminant()) / (2 * self.__a)
```

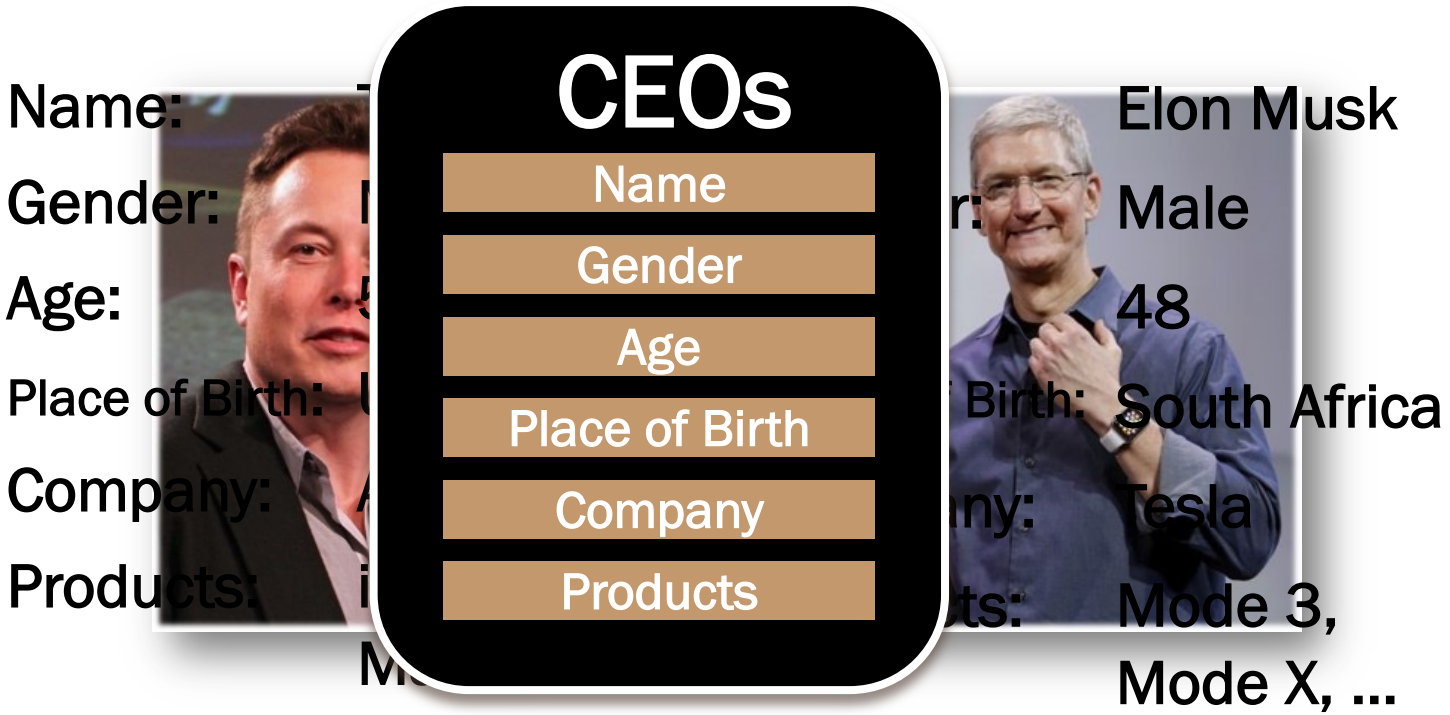
```
def main():
    a, b, c = eval(input("Enter a, b, c: "))
    equation = QuadraticEquation(a, b, c)
    discriminant = equation.getDiscriminant()

    if discriminant < 0:
        print("The equation has no roots")
    elif discriminant == 0:
        print("The root is", equation.getRoot1())
    else: # (discriminant >= 0)
        print("The roots are", equation.getRoot1(), "and", equation.getRoot2())

main()
```

Part II

- Inheritance
- Overriding
- Object Class

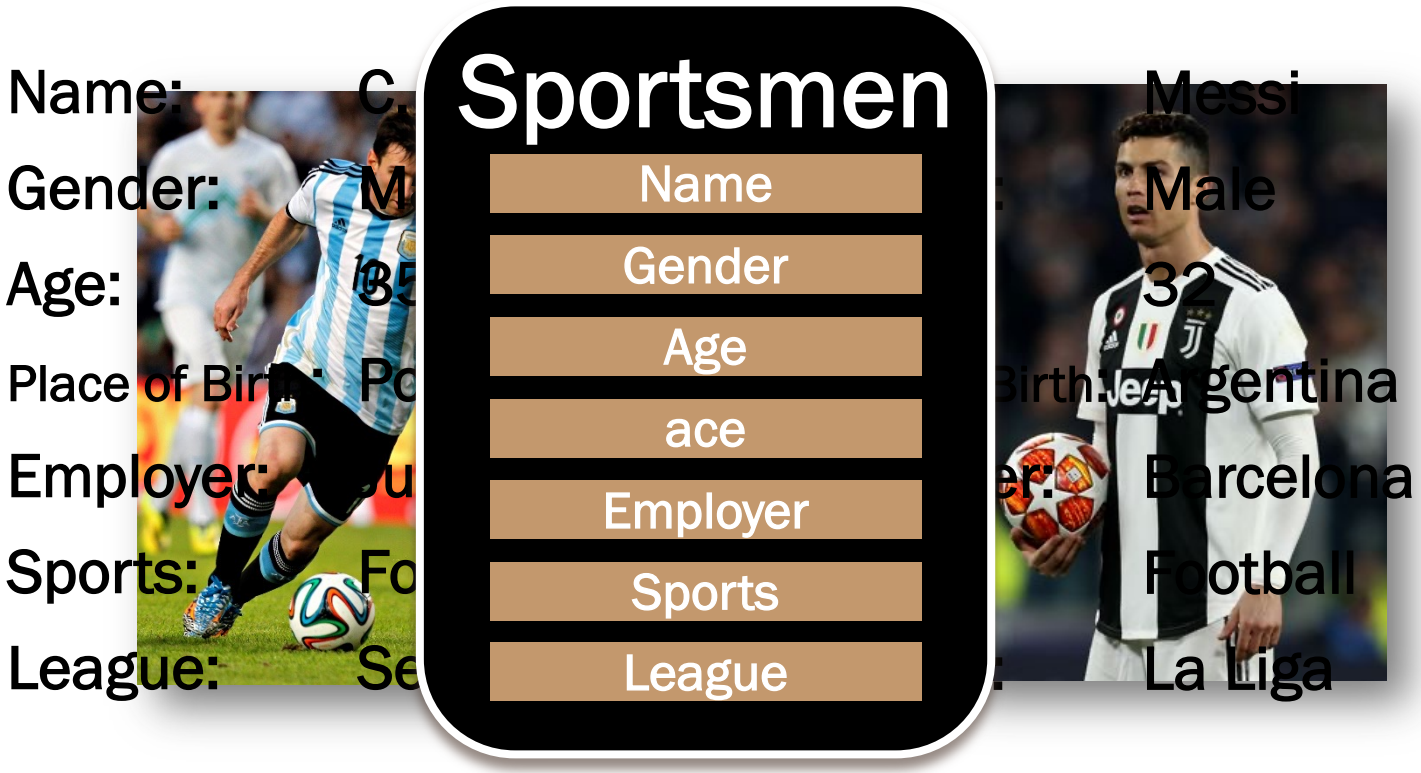
CONCEPT OF CLASS

- Example 1



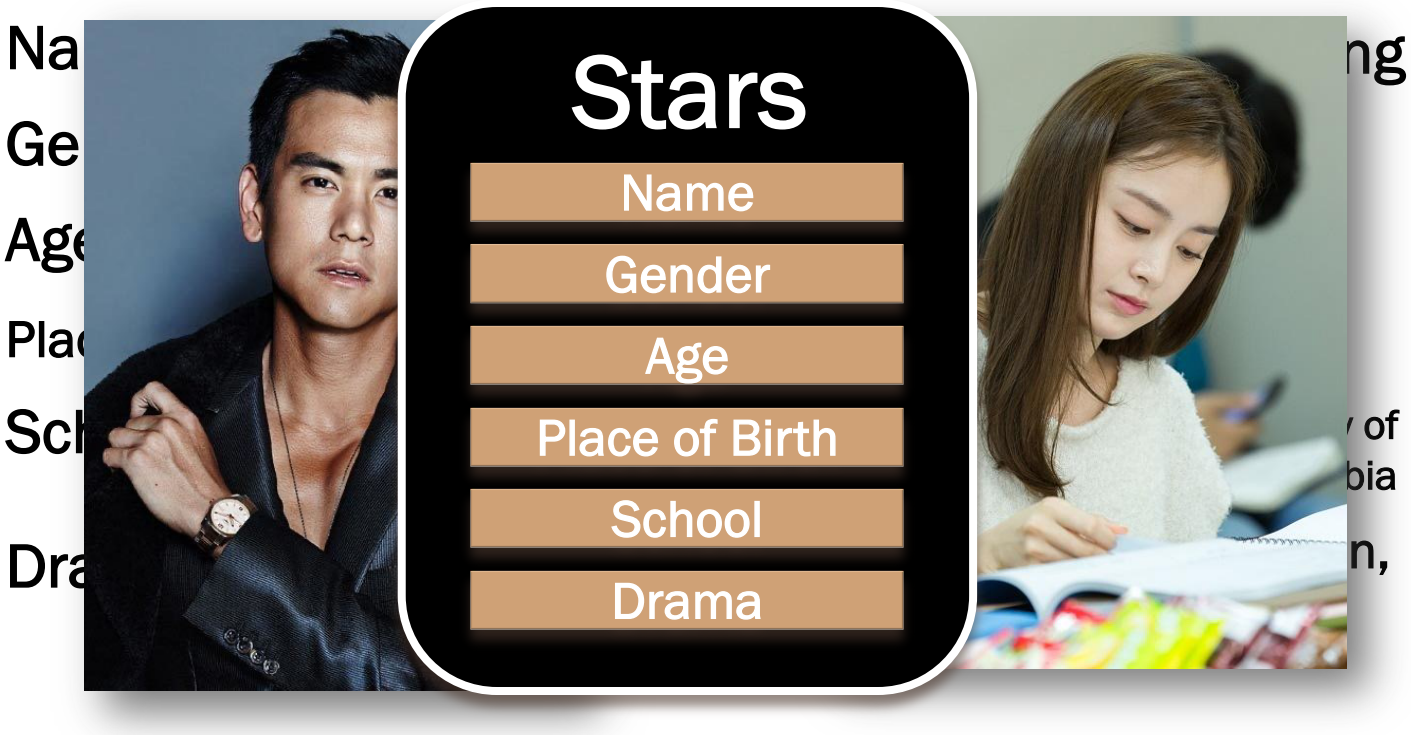
CONCEPT OF CLASS (CONT.)

- Example 2



CONCEPT OF CLASS (CONT.)

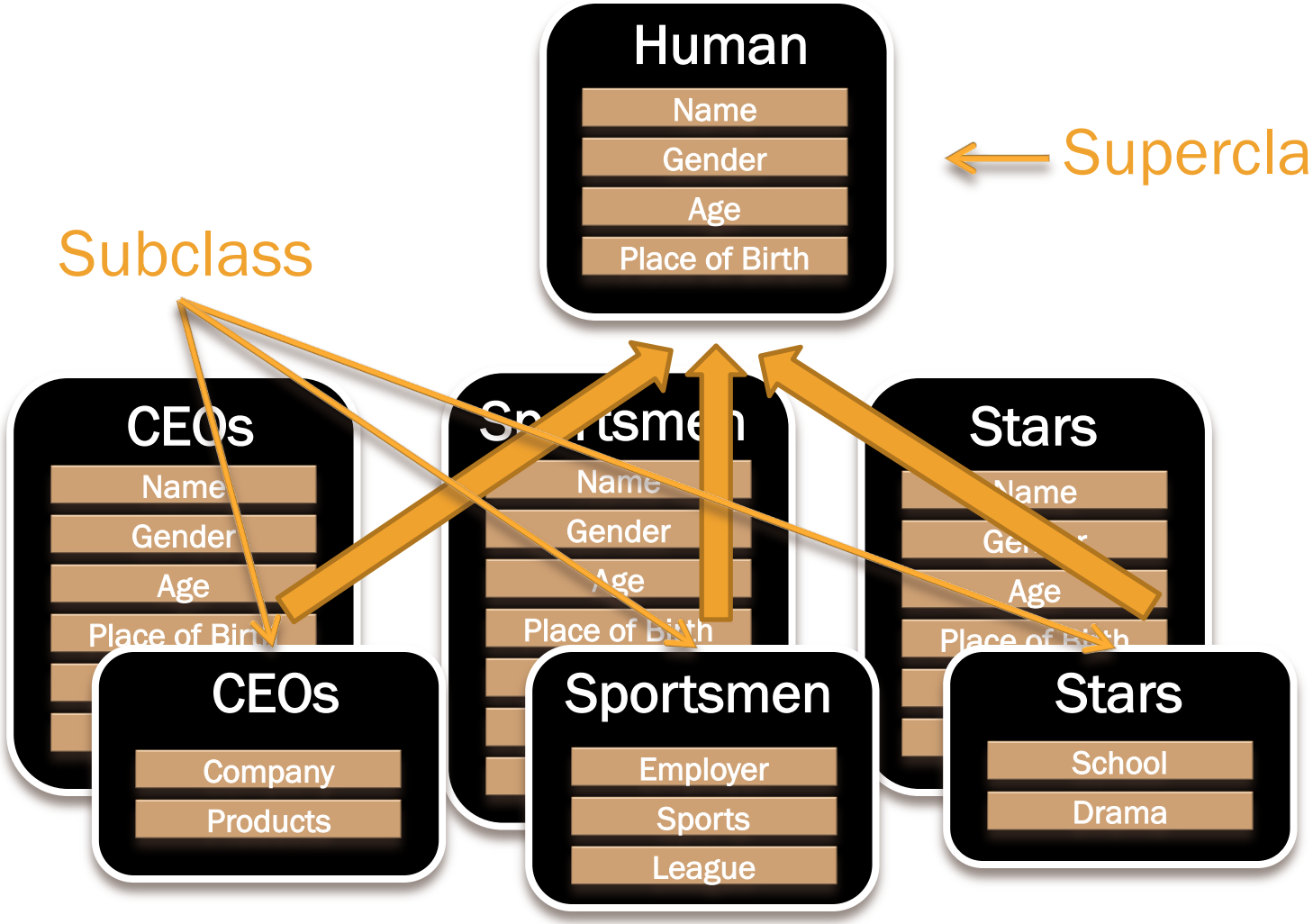
- Example 3



INHERITANCE

Subclass will inherit most of the attributes from his superclass

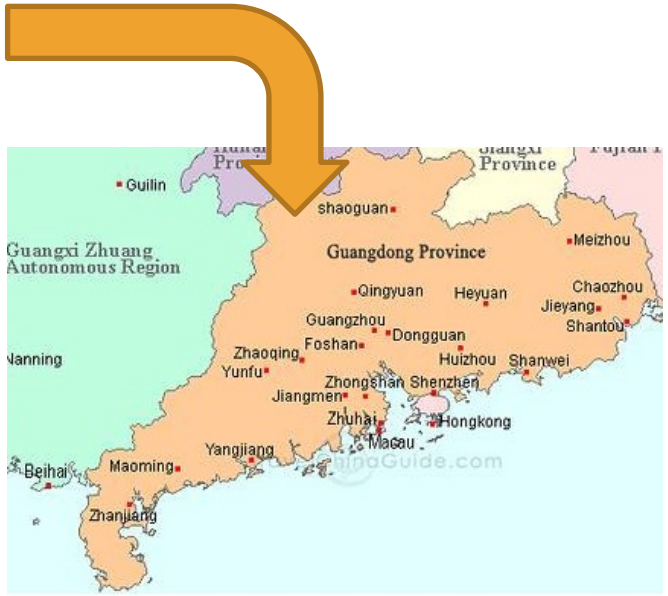
- Higher Abstraction



INHERITANCE: SUPERCLASS AND SUBCLASS

- Adding an important and powerful feature for reusing software
- Enables you to define a superclass and later extend it to subclasses
- Superclass
  - A general class generalizes common properties and behaviours of different classes
- Subclass
  - The specialized class inherits the properties and methods from the general class

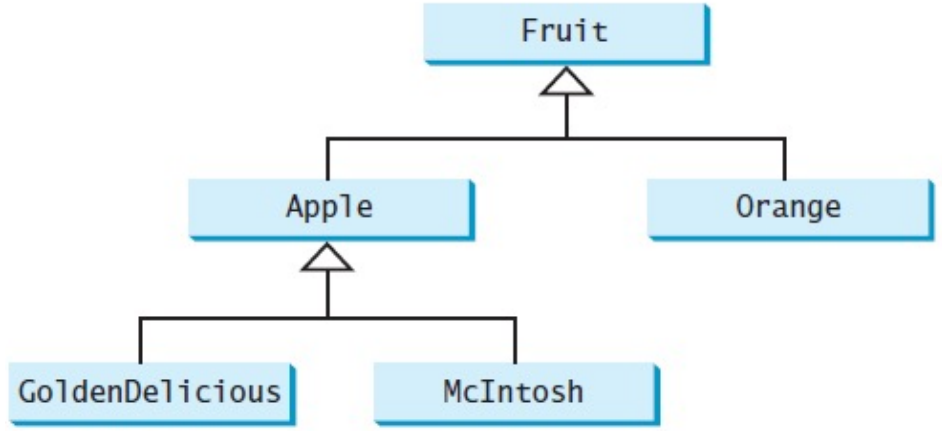
EXAMPLE: HUMAN -> CHINESE -> CANTONESE



香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

EXAMPLE: HUMAN -> STAR -> MOVIE STAR

QUESTIONS



Assume that the following statements are given:

```
goldenDelicious = GoldenDelicious()
orange = Orange()
```

(a) Is goldenDelicious a subclass of Fruit?
(b) Is goldenDelicious a subclass of Orange?
(c) Is goldenDelicious a subclass of Apple?
(d) Is goldenDelicious a subclass of GoldenDelicious?
(e) Is goldenDelicious a subclass of McIntosh?
(f) Is orange a subclass of Orange?
(g) Is orange a subclass of Fruit?
(h) Is orange a subclass of Apple?
(i) Suppose the method makeAppleCider is defined in the Apple class. Can goldenDelicious invoke this method? Can orange invoke this method?
(j) Suppose the method makeOrangeJuice is defined in the Orange class. Can orange invoke this method? Can goldenDelicious invoke this method?

EXAMPLE OF INHERITANCE

- A subclass inherits accessible data fields and methods from its superclass, but it can also have other data fields and methods

Test.py

```
class A:
    def __init__(self,value):
        self.__v = value
    def try (self,x):
        print(self.__v+x)


class B(A):
    def __init__(self, value, newvalue):
        super().__init__(value)
        self.__nv = newvalue
    def getNV(self):
        return self.__nv

b = B(2,3)
b.try (3)
print(b.getNV())
```

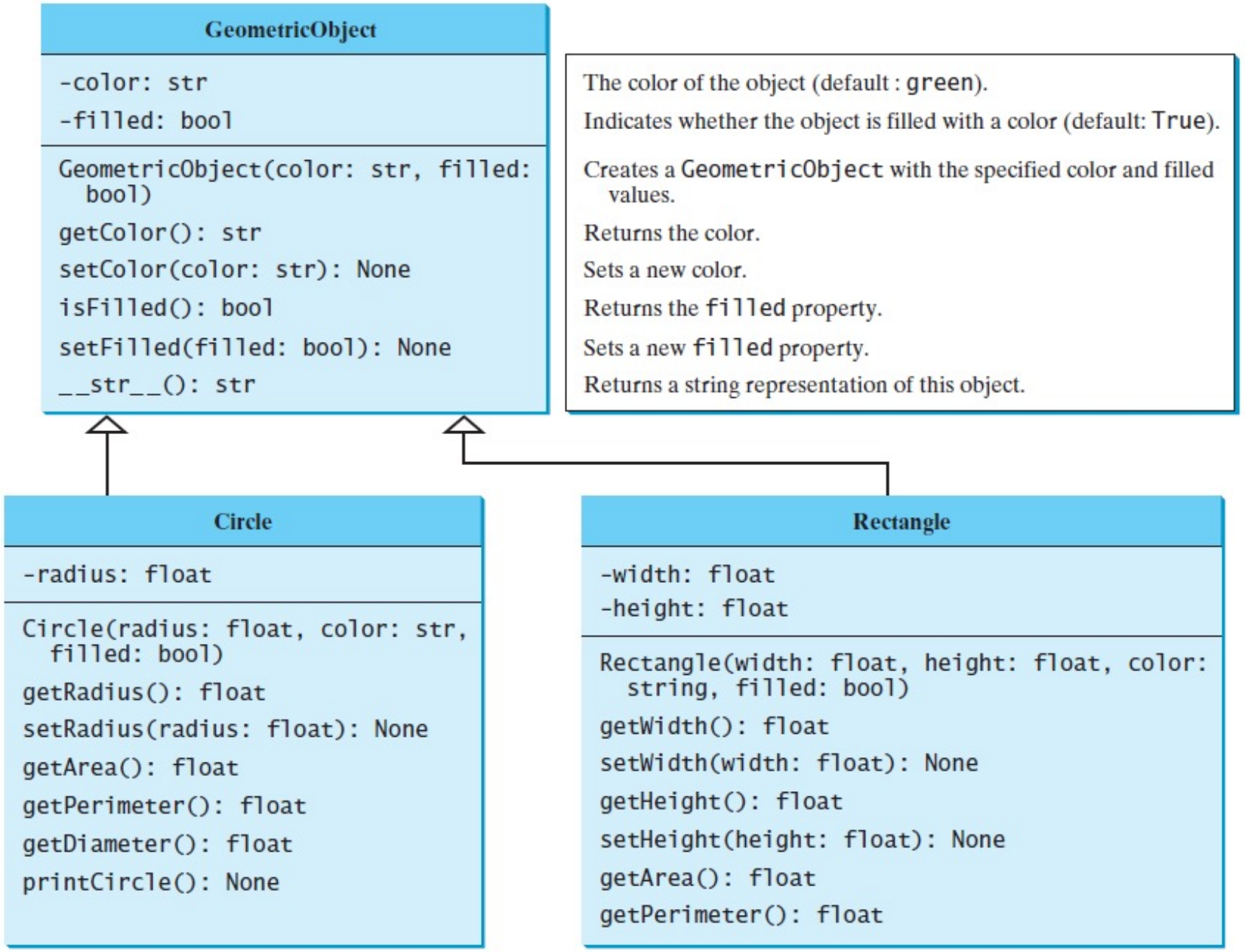
or super(B, self).__init__(value)

or A.__init__(self, value)

Result:

```
5
3
```

GEOMETRIC OBJECT AND TWO OF ITS SUBCLASSES



THE CODE FOR GEOMETRICOBJECT

```
class GeometricObject:                                          GeometricObject class
    def __init__(self, color = "green", filled = True):        initializer
        self.__color = color                                    data fields
        self.__filled = filled

    def getColor(self):                                         getColor
        return self.__color

    def setColor(self, color):                                  setColor
        self.__color = color

    def isFilled(self):                                         isFilled
        return self.__filled

    def setFilled(self, filled):
        self.__filled = filled

    def __str__(self):
        return "color: " + self.__color + \
            " and filled: " + str(self.__filled)
```

THE CODE FOR CIRCLE CLASS

- A subclass inherits accessible data fields and methods from its superclass, but it can also have other data fields and methods

```
from GeometricObject import GeometricObject
import math # math.pi is used in the class

class Circle(GeometricObject):
    def __init__(self, radius):
        super().__init__()
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def setRadius(self, radius):
        self.__radius = radius

    def getArea(self):
        return self.__radius * self.__radius * math.pi

    def getDiameter(self):
        return 2 * self.__radius

    def getPerimeter(self):
        return 2 * self.__radius * math.pi

    def printCircle(self):
        print(self.__str__() + " radius: " + str(self.__radius))
```

Circle class inherits the methods getColor, setColor, isFilled, setFilled, and __str__

The printCircle method invokes the __str__() method defined to obtain properties defined in the superclass

THE CODE FOR RECTANGLE CLASS

```
from GeometricObject import GeometricObject

class Rectangle(GeometricObject):
    def __init__(self, width = 1, height = 1):
        super().__init__()
        self.__width = width
        self.__height = height

    def getWidth(self):
        return self.__width

    def setWidth(self, width):
        self.__width = width

    def getHeight(self):
        return self.__height

    def setHeight(self, height):
        self.__height = self.__height

    def getArea(self):
        return self.__width * self.__height

    def getPerimeter(self):
        return 2 * (self.__width + self.__height)
```

extend superclass
initializer
superclass initializer

methods

THE CODE FOR TESTING CIRCLE AND RECTANGLE

```
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    circle = Circle(1.5)
    print("A circle", circle)
    print("The radius is", circle.getRadius())
    print("The area is", circle.getArea())
    print("The diameter is", circle.getDiameter())

    rectangle = Rectangle(2, 4)
    print("\nA rectangle", rectangle)
    print("The area is", rectangle.getArea())
    print("The perimeter is", rectangle.getPerimeter())

main() # Call the main function
```

```
A circle color: green and filled: True
The radius is 1.5
The area is 7.06858347058
The diameter is 3.0

A rectangle color: green and filled: True
The area is 8
The perimeter is 12
```

SOME MORE INFORMATION ABOUT SUPER AND SUB-CLASS

- A subclass is not a subset of its superclass; In fact, a subclass usually contains more information and methods than its superclass
- Inheritance models the is-a relationships, but not all is-a relationships should be modelled using inheritance
- Do not blindly extend a class just for the sake of reusing methods.
  - For example, it makes no sense for a Tree class to extend a Person class, even though they share common properties such as height and weight.
  - A subclass and its superclass must have the is-a relationship

PRACTICE

```
class A:
    def __init__(self, i = 0):
        self.i = i

class B(A):
    def __init__(self, j = 0):
        self.j = j

def main():
    b = B()
    print(b.i)
    print(b.j)

main() # Call the main function
```

super().__init__()

What is the problem with the above code?

OVERRIDING METHODS

- A subclass inherits methods from a superclass
- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass.
- This is referred to as method overriding

EXAMPLE

- The __str__() method in the GeometricObject class returns the string describing a geometric object. This method can be overridden to return the string describing a circle

```
class Circle(GeometricObject):
    # Other methods are omitted

    # Override the __str__ method defined in GeometricObject
    def __str__(self):
        return super().__str__() + " radius: " + str(radius)
```

__str__ in superclass

- The __str__() method is defined in the GeometricObject class and modified in the Circle class. Both methods can be used in the Circle class. To invoke the __str__ method defined in the GeometricObject class from the Circle class, use super().__str__()

PRACTICE

What would be the output of the following program?

Result:

```
4
0
```

```
class A:
    def __init__(self, i = 0):
        self.i = i

    def m1(self):
        self.i += 2

class B(A):
    def __init__(self, j = 0):
        super().__init__(3)
        self.j = j

    def m1(self):
        self.i += 1

def main():
    b = B()
    b.m1()
    print(b.i)
    print(b.j)

main() # Call the main function
```

NAME MANGLING WITH SUPERCLASS

```
class Test:
    def __init__(self):
        self.foo = 1
        self._bar = 2
        self.__baz = 3

class ExtendedTest(Test):
    def __init__(self):
        super().__init__()
        self.foo = 'overridden'
        self._bar = 'overridden'
        self.__baz = 'overridden‘
    def printbaz(self):
        print(self.foo)
        print(self._bar)
        print(self.__baz)
```

```
e = ExtendedTest()
e.printbaz()
print(dir(e))
print(e._Test__baz)
print(e._ExtendedTest__baz)
```

```
overridden
overridden
overridden
['_ExtendedTest__baz', '_Test__baz', '__class__',
'__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_bar', 'foo', 'printbaz']
3
overridden
```

THE OBJECT CLASS

- Every class in Python is descended from the object class

- The object class is defined in the Python library. If no inheritance is specified when a class is defined, its superclass is object by default

METHODS OF THE OBJECT CLASS: __STR__()

- The __str__() method returns a string description for the object
  - Usually you should override the __str__() method so that it returns an informative description for the object

WHAT IS THE OUTPUT OF THIS PROGRAM?

```
class A:
    def __init__(self, i = 0):
        self.i = i

    def m1(self):
        self.i += 1

    def __str__(self):
        return 'The content of this object is:'+str(self.i)

x = A(8)
print(x)
```

Result:

The content of this object is: 8

METHODS OF THE OBJECT CLASS: __NEW__()

- The __new__() method is automatically invoked when an object is constructed.
  - This method then invokes the __init__() method to initialize the object.
  - Normally you should only override the __init__() method to initialize the data fields defined in the new class

WHAT IS THE OUTPUT OF THIS PROGRAM?

```
class A:
    def __new__(self):
        print("A's __new__() invoked")

    def __init__(self):
        print("A's __init__() invoked")

class B(A):
    def __new__(self):
        print("B's __new__() invoked")

    def __init__(self):
        print("B's __init__() invoked")

def main():
    b = B()
    a = A()

main() # Call the main function
```

Result:

```
B's __new__() invoked
A's __new__() invoked
```

WHAT IS THE OUTPUT OF THIS PROGRAM?

```
class A:
    def __new__(self):
        self.__init__(self)
        print("A's __new__() invoked")

    def __init__(self):
        print("A's __init__() invoked")

class B(A):
    def __new__(self):
        self.__init__(self)
        print("B's __new__() invoked")

    def __init__(self):
        print("B's __init__() invoked")

def main():
    b = B()
    a = A()

main() # Call the main function
```

Result:

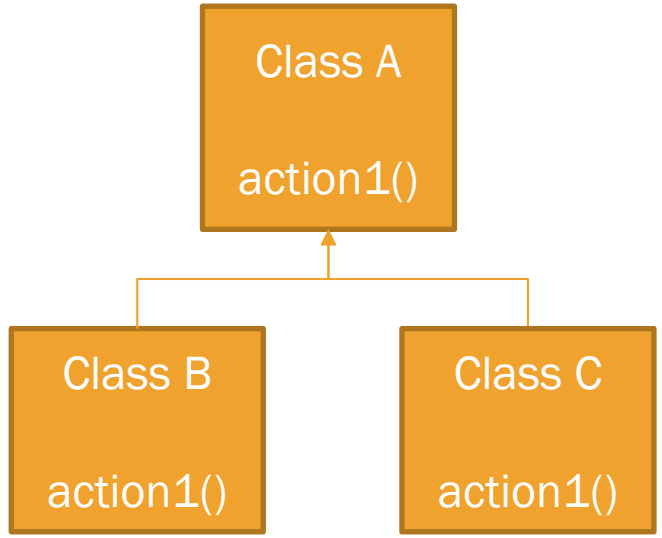
```
B's __init__() invoked
B's __new__() invoked
A's __init__() invoked
A's __new__() invoked
```

Part III

- Polymorphism
- Dynamic Binding

POLYMORPHISM AND DYNAMIC BINDING

- A subclass can override methods of its superclass
  - A method may be implemented in several classes along the inheritance chain
- Polymorphism
  - the objects of different classes (share some common members) can be passed as arguments to the same function
- Dynamic Binding
  - Python decides which method is invoked at runtime




```
def f(o):
    o.action1()
```

```
a = A()     b = B()     c = C()
f(a)        f(b)        f(c)
```

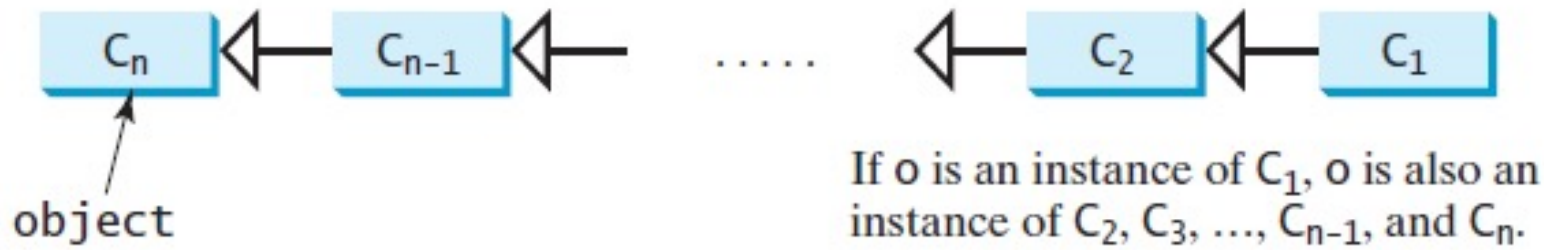
EXAMPLE

```
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    # Display circle and rectangle properties
    c = Circle(4)
    r = Rectangle(1, 3)
    displayObject(c)
    displayObject(r)
    print("Are the circle and rectangle the same size?",
        isSameArea(c, r))

# Display geometric object properties
def displayObject(g) :
    print(g.__str__())

# Compare the areas of two geometric objects
def isSameArea(g1, g2) :
    return g1.getArea() == g2.getArea()

main() # Call the main function
```

Result:

```
color: green and filled: True radius: 4
color: green and filled: True width: 1 height: 3
Are the circle and rectangle the same size? False
```

DYNAMIC BINDING



- That is, C_n is the most general class, and C_1 is the most specific class
- In Python, C_n is the object class
- If o invokes a method p, Python searches the implementation for the method p in C_1, C_2, …, C_{n-1}, and C_n, in this order, until it is found

EXAMPLE

- What would be the output of this program?

```
4
In C2, the f is: 4
```

```
class C1:
    def __init__(self):
        self.f  = 1

    def output(self):
        print('In C1, the f is:',self.f)

class C2(C1):
    def __init__(self):
        self.f  = 2

    def output(self):
        print('In C2, the f is:',self.f)

class C3(C2):
    def __init__(self):
        self.f  = 3

class C4(C3):
    def __init__(self):
        self.f = 4

a=C4()
print(a.f)
a.output()
```

EXAMPLE

What if replace __str__() by __str()?

Student
Student

Student
Graduate Student

```
class Student:
    def __str__(self):
        return "Student"

    def printStudent(self):
        print(self.__str__())

class GraduateStudent(Student):
    def __str__(self):
        return "Graduate Student"

a = Student()
b = GraduateStudent()
a.printStudent()
b.printStudent()
```

PRACTICE

What would be the outputs?

```
class Person:
    def getInfo(self):
        return "Person"

    def printPerson(self):
        print(self.getInfo())

class Student(Person):
    def getInfo(self):
        return "Student"

Person().printPerson()
Student().printPerson()
```

(a)

```
Person
Student
```

```
class Person:
    def __getInfo(self):
        return "Person"

    def printPerson(self):
        print(self.__getInfo())

class Student(Person):
    def __getInfo(self):
        return "Student"

Person().printPerson()
Student().printPerson()
```

(b)

```
Person
Person
```

QUESTION

- Suppose you want to modify the displayObject function to perform the following tasks:
  - Display the area and perimeter of a Circle or Rectangle instance
  - Display the diameter if the instance is a Circle, and the width and height if the instance is a Rectangle

```
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    # Display circle and rectangle properties
    c = Circle(4)
    r = Rectangle(1, 3)
    displayObject(c)
    displayObject(r)
    print("Are the circle and rectangle the same size?",
        isSameArea(c, r))

# Display geometric object properties
def displayObject(g):
    print(g.__str__())

# Compare the areas of two geometric objects
def isSameArea(g1, g2):
    return g1.getArea() == g2.getArea()

main() # Call the main function
```

DOES THIS PROGRAM WORK?

```
def displayObject(g):
    print("Area is", g.getArea())
    print("Perimeter is", g.getPerimeter())
    print("Diameter is", g.getDiameter())
    print("Width is", g.getWidth())
    print("Height is", g.getHeight())
```

ISINSTANCE() FUNCTION

- The isinstance() function can be used to determine whether an object is an instance of a class

- This function determines whether an object is an instance of a class by using the following syntax

```
isinstance(object, ClassName)
```

```python
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    # Display circle and rectangle properties
    c = Circle(4)
    r = Rectangle(1, 3)
    print("Circle...")
    displayObject(c)
    print("Rectangle...")
    displayObject(r)

# Display geometric object properties
def displayObject(g):
    print("Area is", g.getArea())
    print("Perimeter is", g.getPerimeter())

    if isinstance(g, Circle):
        print("Diameter is", g.getDiameter())
    elif isinstance(g, Rectangle):
        print("Width is", g.getWidth())
        print("Height is", g.getHeight())

main() # Call the main function
```

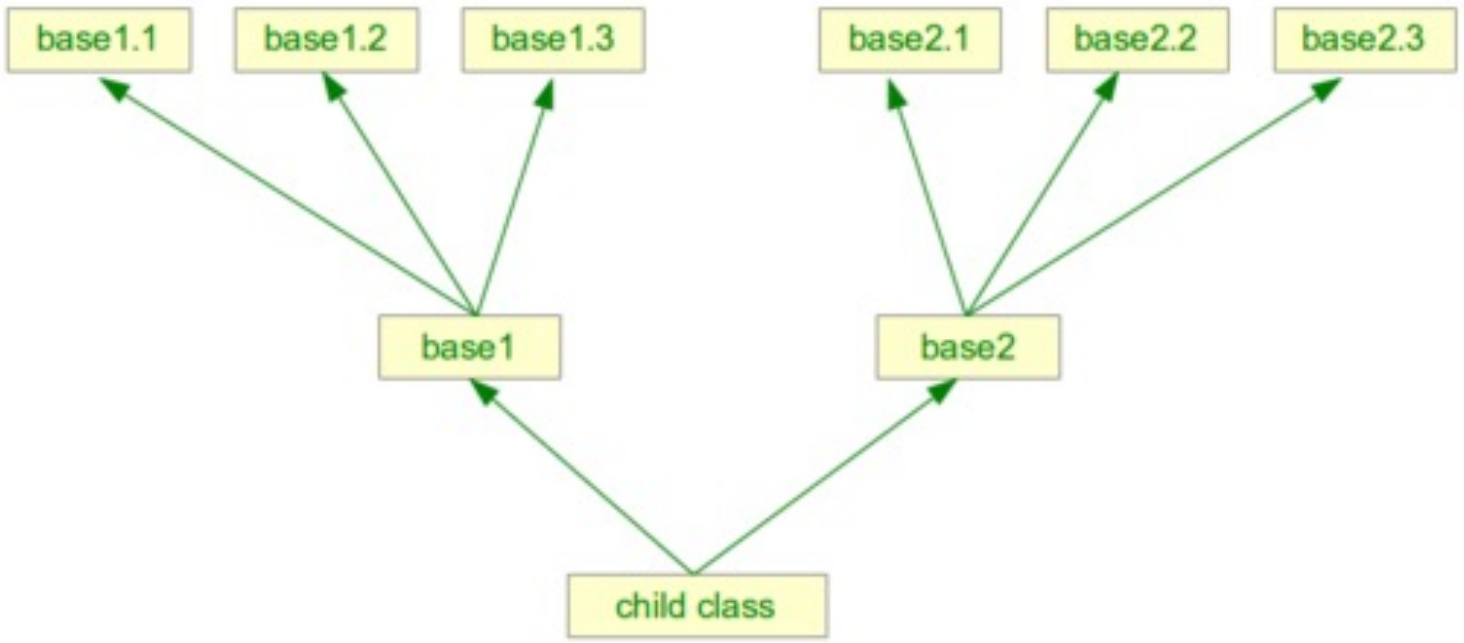
Result:

```
Circle...
Area is 50.26548245743669
Perimeter is 25.132741228718345
Diameter is 8
Rectangle...
Area is 3
Perimeter is 8
Width is 1
Height is 3
```

MULTIPLE INHERITANCE

- In Python, we can define new class from multiple classes
- This is called multiple inheritance
- Multiple inheritance is a feature in which a class can inherit data fields and methods from more than one parent class

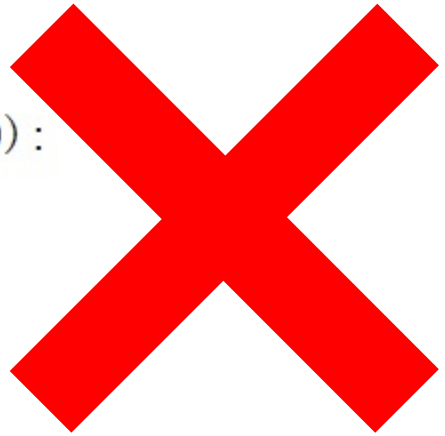
INHERITANCE TREE



The inheritance relationship in Python can be represented by a tree structure

EXAMPLE

```
class A():
    def __init__(self,a=100):
        self.a=a

class B():
    def __init__(self,b=200):
        self.b=b

class C(A,B):
    def __init__(self,a,b,c=300):
        super().__init__(a)
        super().__init__(b)
        self.c=c

    def output(self):
        print(self.a)
        print(self.c)
        print(self.b)

def main():
    c = C(1,2,3)
    c.output()

main()
```



EXAMPLE

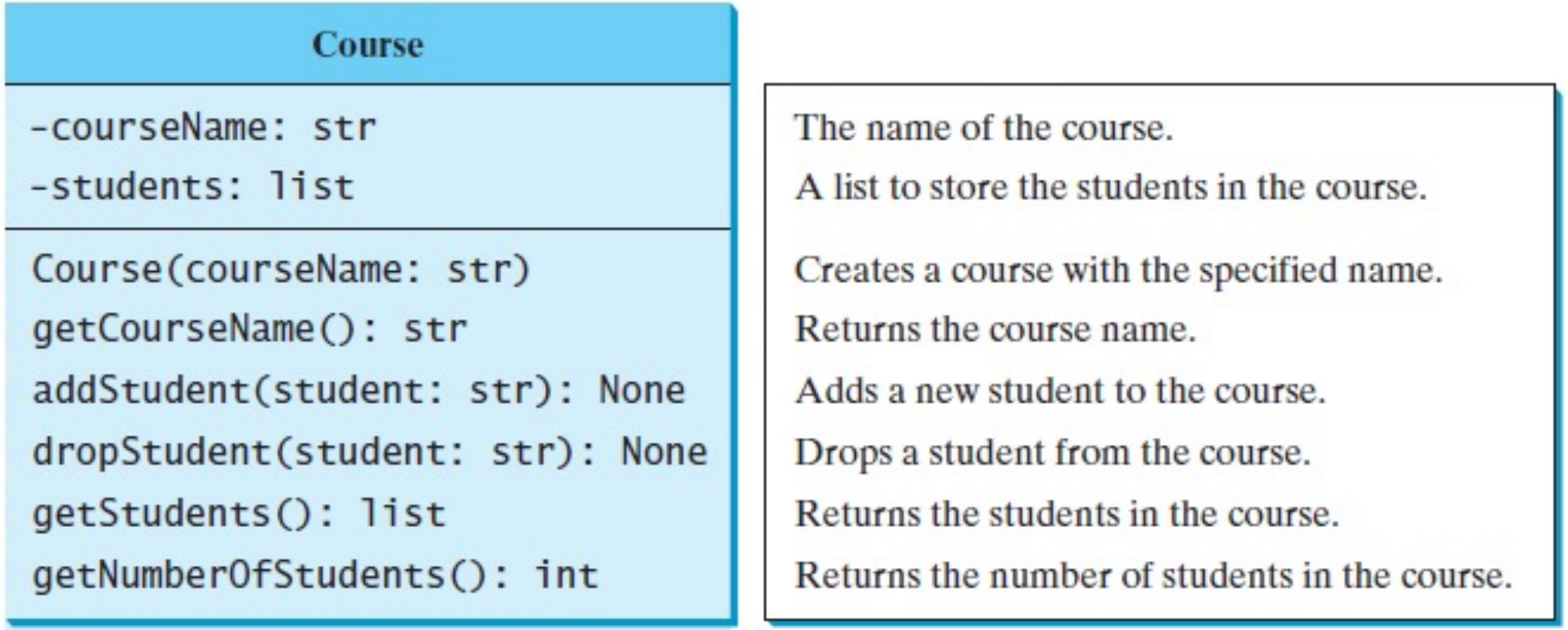
```
class A():
    def __init__(self,a=100):
        self.a=a

class B():
    def __init__(self,b=200):
        self.b=b

class C(A,B):
    def __init__(self,a,b,c=300):
        A.__init__(self,a)
        B.__init__(self,b)
        self.c=c

    def output(self):
        print(self.a)
        print(self.c)
        print(self.b)

def main():
    c = C(1,2,3)
    c.output()

main()
```

STAR CLASSES

```
class star:
    def __init__(self,name,gender):
        print('Star init...')
        self.name = name
        self.gender = gender

class popStar(star):
    def __init__(self,albumList):
        print('popStar init...')
        star.__init__(self,'Stephen Chow','Male')
        self.albumList = albumList

    def outputAlbum(self):
        print(self.name,'has published the following albums:')
        for key in self.albumList:
            print('The sold copies of album <'+key+'> is',self.albumList[key])

class movieStar(star):
    def __init__(self,movieList):
        print('movieStar init...')
        star.__init__(self,'Stephen Chow','Male')
        self.movieList = movieList

    def outputMovie(self):
        print(self.name,'has participated in the following movies:')
        for key in self.movieList:
            print('The income of movie <'+key+'> is',self.movieList[key])
```

```
class superStar(movieStar,popStar):
    def __init__(self,albumList,movieList):
        print('superStar init...')
        popStar.__init__(self,albumList)
        movieStar.__init__(self,movieList)

    def careerPerformance(self):
        print(self.name+' has a very successful career.')
        popStar.outputAlbum(self)
        movieStar.outputMovie(self)

def main():
    albumList = {'I am a singer':100,'Hahaha':200,'Gee':300}
    movieList = {'Kungfu':2000000,'Shaolin Soccer':20000000,'Mermeid':230909230}
    s = superStar(albumList,movieList)
    s.careerPerformance()

main()
```

PRACTICE: COURSE CLASS



Course.py

```
class Course:
    def __init__(self, courseName):
        self.__courseName = courseName
        self.__students = []

    def addStudent(self, student):
        self.__students.append(student)

    def getStudents(self):
        return self.__students

    def getNumberOfStudents(self):
        return len(self.__students)

    def getCourseName(self):
        return self.__courseName

    def dropStudent(student):
        print("Left as an exercise")
```

register.py

```
from Course import Course

def main():

    course1 = Course("Data Structures")
    course2 = Course("Database Systems")

    course1.addStudent("Peter Jones")
    course1.addStudent("Brian Smith")
    course1.addStudent("Anne Kennedy")

    course2.addStudent("Peter Jones")
    course2.addStudent("Steve Smith")

    print("Number of students in course1:",
        course1.getNumberOfStudents())
    students = course1.getStudents()
    for student in students:
        print(student, end = ", ")

    print("\nNumber of students in course2:",
        course2.getNumberOfStudents())

main() # Call the main function
```

Result:

```
Number of students in course1: 3
Peter Jones, Brian Smith, Anne Kennedy,
Number of students in course2: 2
```