

Object-oriented programming

International Baccalaureate · IB Diploma · Computer Science · HL

Scope and prerequisites

This handout teaches Theme B3, Object-oriented programming, for the course first assessed in 2027, using Python, with Java shown where its syntax differs. Sections marked **AHL** are for Higher Level only.

Prerequisites: B2 Programming (variables, selection, loops, functions and parameters). The examples are fictional models, not real bank systems.

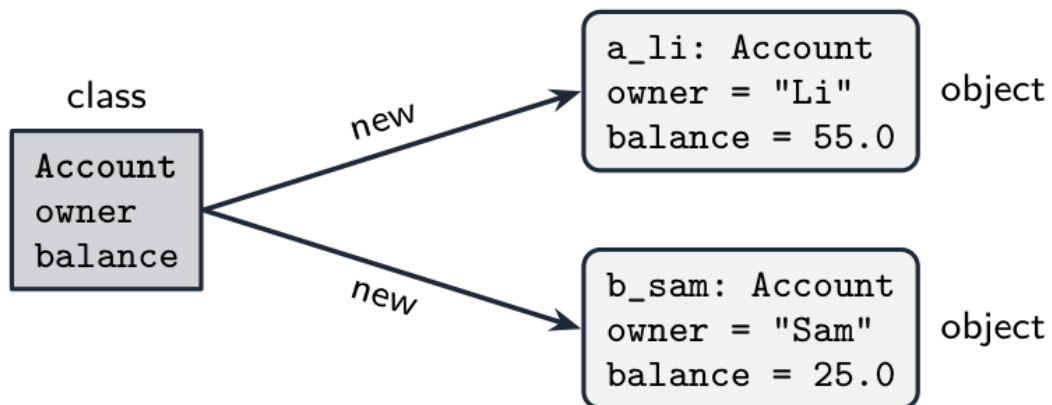
Classes and objects

■ Why model the world as objects?

A school app tracks thousands of students. Each one has a name, a year group and a list of grades, and each can enrol on a course or receive a grade. Writing separate variables for every student is impossible. Instead we write the design once, as a class, and make one object per student.

■ Class, object and instance

A **class** is a blueprint: it defines the **attributes** (the data each object holds) and the **methods** (what each object can do). An **object** is one thing built from that blueprint, an **instance** of the class. Making an object is **instantiation**



One class, many objects: the same attributes, each with its own values

- An object's **state** is the current values of its attributes; its **behaviour** is what its methods do.

- Objects of a class normally share its defined interface and may hold different instance values; Python also allows instance attributes to be added dynamically.

■ Writing a class in Python

```
class Account:
    def __init__(self, owner, balance):    # the constructor
        self.owner = owner                # instance attributes
        self.balance = balance

    def deposit(self, amount):
        self.balance = self.balance + amount

    def get_balance(self):
        return self.balance

a_li = Account("Li", 40.0)                # instantiation
b_sam = Account("Sam", 20.0)
a_li.deposit(15.0)
print(a_li.get_balance(), b_sam.get_balance())    # 55.0 20.0
```

- The object-initialisation step is often taught with the **constructor**. In normal Python construction, `__new__` creates the instance and `__init__` initialises its state. In Java it is a method with the class's name: `public Account(String owner, double balance)`.
- `self` is the object the method is working on: `a_li.deposit(15.0)` changes `a_li`'s balance only.
- A variable holds a **reference** to an object. After `c = a_li`, both names refer to the **same** object: a deposit through `c` is seen through `a_li`.

■ Instance and static members

An **instance attribute** belongs to one object. A **static** (class) attribute belongs to the class and is shared by every object.

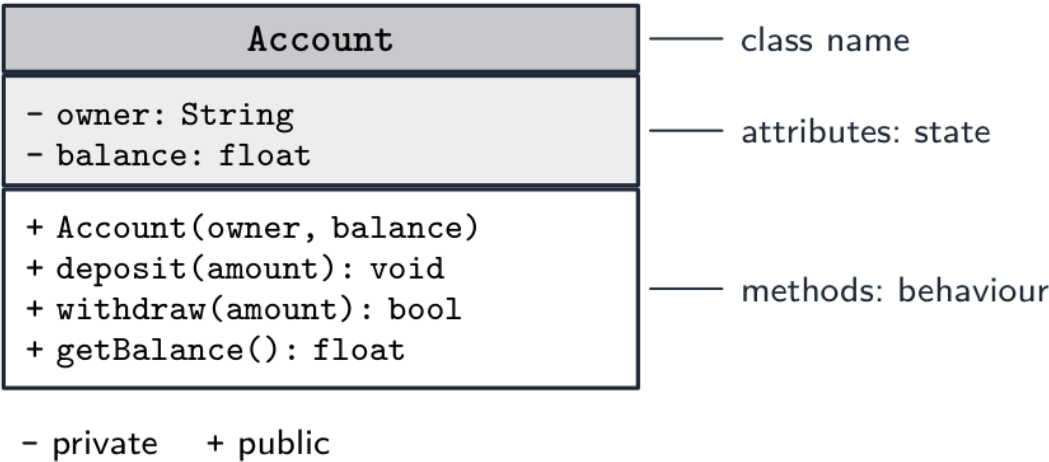
```
class Account:
    count = 0                                # static: one value for the whole
    ↪ class
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance
        Account.count = Account.count + 1
```

After two accounts are made, `Account.count` is 2. In Java a static member is marked **static** and is called on the class, such as `Math.sqrt(9)`.

Encapsulation

■ Hiding the state

Encapsulation bundles the data and the methods that act on it into one class, and hides the data so it can be changed only through those methods. The class then protects its own rules, such as “the balance is never negative”.



A UML class diagram: name, attributes, methods; - private, + public

Access	Java	Python convention	Who can use it
private	private	leading _ marks non-public use by convention	Java restricts access; Python’s single underscore does not enforce privacy
public	public	plain name	any code

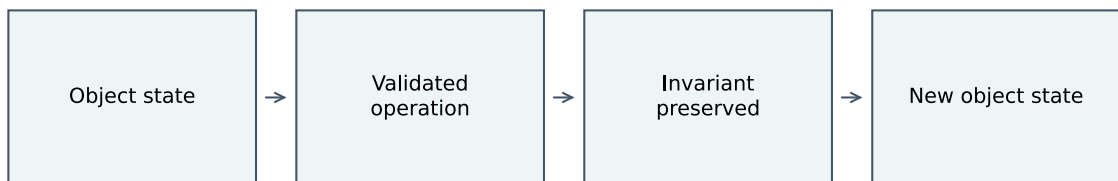
■ Accessors and mutators

Private attributes are read through an **accessor** (getter) and changed through a **mutator** (setter), which can check the new value first.

```
class Account:
    def __init__(self, owner, balance):
        self._owner = owner
        self._balance = max(0.0, balance)

    def get_balance(self):                # accessor
        return self._balance

    def withdraw(self, amount):          # mutator with validation
        if amount <= 0 or amount > self._balance:
            return False
        self._balance = self._balance - amount
        return True
```



A validated operation keeps the rule true: a rejected withdrawal leaves the state unchanged

Trace: `acc = Account("Li", 15.0)`; `acc.withdraw(30.0)` returns **False** and the balance stays 15.0; `acc.withdraw(15.0)` returns **True** and the balance becomes 0.0.

Advantages of encapsulation:

- **Data integrity:** invalid values are rejected in one place.
- **Maintenance:** the inside of a class can change without breaking code that uses only its public methods.
- **Teamwork:** a programmer can use a class from its public interface alone.

Designing with classes

■ Choosing attributes and methods

To design a class from a scenario, pick out the **nouns** that are data (attributes) and the **verbs** that are actions (methods).

A library lends books. Each book has a title, an ISBN and whether it is on loan. A book can be lent and returned.

Attributes (private)	Methods (public)
title: String	lend(): bool
isbn: String	give_back(): void
on_loan: bool	is_on_loan(): bool

lend returns **False** if the book is already on loan, so the rule "a book cannot be lent twice" lives inside the class.

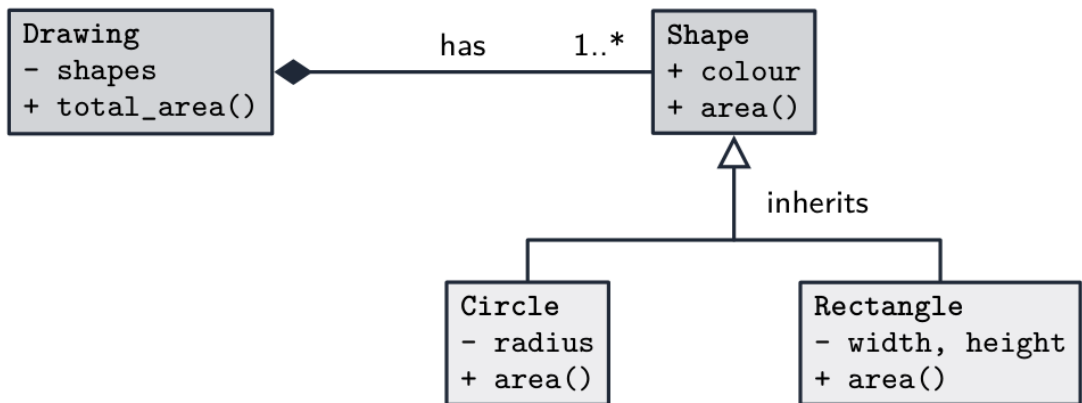
■ Advantages and limits of OOP

Advantages	Disadvantages
models real things, so designs are easy to discuss	more code and planning for a small program
classes are reused in other programs	objects can use more memory
encapsulation protects data	a poor class design is hard to change later

AHL: Inheritance and polymorphism

■ Inheritance

Inheritance lets a **subclass** take the attributes and methods of a **super-class** and add or change some. It models an "is a" relationship: a circle **is a** shape.



Hollow triangle: inherits (“is a”). Filled diamond: composition (“has a”)

```

import math

class Shape:
    def __init__(self, colour):
        self.colour = colour
    def area(self):
        return 0.0

class Circle(Shape):
    def __init__(self, colour, radius):
        super().__init__(colour)      # run the superclass constructor
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2      # overrides Shape.area

class Rectangle(Shape):
    def __init__(self, colour, width, height):
        super().__init__(colour)
        self.width, self.height = width, height
    def area(self):
        return self.width * self.height
  
```

- `super()` calls the superclass’s version, so the colour code is written once.
- In Java: `class Circle extends Shape`.

■ Polymorphism

Polymorphism means one call works on objects of different classes, each running its own version. A subclass method with the same name and parameters as the superclass's **overrides** it.

```
shapes = [Circle("red", 1.0), Rectangle("blue", 2.0, 3.0)]
for s in shapes:
    print(round(s.area(), 2))    # 3.14, then 6.0
```

The loop does not need to know which kind of shape it holds. A new subclass such as `Triangle` works with the same loop unchanged.

Overloading is different: several methods in one class share a name but take different parameters, such as two Java constructors `Account(String owner)` and `Account(String owner, double balance)`. Python does not overload; it uses default parameter values instead.

■ Abstract classes

An **abstract class** cannot be instantiated: it exists to be inherited. An **abstract method** must be implemented before a subclass can be concrete; Java abstract declarations have no body, while a Python abstract method may supply a body for reuse. `Shape` should be abstract, since a plain shape has no sensible area.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

`Shape()` now raises an error; `Circle` must define `area` or it cannot be instantiated either. In Java: `abstract class Shape` and `abstract double area();`.

■ Composition and aggregation

Relationship	Meaning	Example	UML
inheritance	"is a"	a <code>Circle</code> is a <code>Shape</code>	hollow triangle
composition	"has a"; the part dies with the whole	a <code>Drawing</code> has its <code>Shapes</code>	filled diamond
aggregation	"has a"; the part lives on alone	a <code>Team</code> has <code>Players</code>	hollow diamond

Prefer composition when the relationship is “has a”: a car has an engine, it is not a kind of engine. Deep inheritance chains are hard to change, because a change to a superclass affects every subclass.

■ Design patterns

A **design pattern** is a tested solution to a common design problem.

Pattern	Problem it solves	Idea
singleton	exactly one object is needed, such as a settings store	the class makes its single instance and always returns it
factory	the code should not depend on which subclass is made	one method decides which subclass to create
observer	several objects must react when one changes	the subject keeps a list of observers and notifies each

Practice

1. Explain the difference between a class and an object, using a phone model as an example.
2. For the **Account** class with **deposit**, state the output of: `x = Account("Ana", 10.0), y = Account("Bo", 10.0), x.deposit(5.0), print(x.get_balance(), y.get_balance())`.
3. `p = Account("Kai", 30.0), q = p, q.deposit(20.0)`. State `p.get_balance()` and explain.
4. Explain why **balance** is made private and changed only through **withdraw**.
5. Draw the UML class diagram for a **Thermostat** with a private **target** (an integer), a constructor, and public methods **raise_by(d)**, **lower_by(d)** and **get_target()**.
6. From the scenario "A cinema has screens. Each screen has a number and a number of seats; seats can be booked and cancelled", identify two attributes and two methods of **Screen**.
7. State **one** advantage and **one** disadvantage of OOP.
8. **AHL** A **Dog** and a **Cat** both inherit from **Animal**, which has **speak()**. Explain how polymorphism lets one loop make every animal in a list speak.
9. **AHL** Distinguish overriding from overloading.
10. **AHL** State whether each is inheritance, composition or aggregation: (a) a **Car** and its **Engine**, made and scrapped together; (b) a **Truck** and **Vehicle**; (c) a **Club** and its **Members**, who can leave and join other clubs.

Answers.

1. The class is the design (the phone model): it says what every phone has and can do. An object is one actual phone built from it, with its own state, such as its own battery level.

2. 15.0 10.0: each object has its own balance.
3. **50.0**: `q = p` copies the reference, so `p` and `q` are the same object.
4. Encapsulation: no code outside the class can set the balance to an invalid value; `withdraw` checks every change, so the rule "never negative" always holds.
5. Three compartments: `Thermostat`; - `target: int`; + `Thermostat(target)`, + `raise_by(d): void`, + `lower_by(d): void`, + `get_target(): int`.
6. Attributes: `number`, `seats` (or `booked`). Methods: `book()`, `cancel()`.
7. Advantage: e.g. classes can be reused, or data is protected by encapsulation. Disadvantage: e.g. more planning and code for a small program.
8. Each subclass overrides `speak()`. The loop calls `a.speak()` on each `Animal`; at run time each object runs its own class's version, so dogs bark and cats meow without the loop testing the type.
9. Overriding: a subclass redefines a superclass method with the same name and parameters. Overloading: methods in one class share a name but take different parameters.
10. (a) composition; (b) inheritance (a truck **is a** vehicle); (c) aggregation.

Exam tips

- Use the words exactly: a **class** is the blueprint, an **object** is an instance, **instantiation** makes it.
- In a UML class diagram, keep three compartments and mark each member - private or + public, with types.
- When you "explain encapsulation", say both parts: data and methods bundled together, and the data hidden behind methods that validate it.
- In code questions, remember that assigning an object variable copies the reference, not the object.
- Mutators normally return nothing (`void`); accessors return a value and change nothing.
- **AHL** For inheritance answers, name the "is a" test; for composition, the "has a" test.
- **AHL** Polymorphism answers need an example: the same call, different classes, each running its own overridden method.