CS1010X: Programming Methodology I
Chapter 12: Object-Oriented Programming

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 12: Object-Oriented Programming

Learning Objectives

By the end of this chapter, students should be able to:

- **Understand** OOP as the combination of **generic operations** (message passing) and **object-based abstractions** (state + behavior).
- **Implement** classes in Python, utilizing **methods**, **attributes**, and the **constructor**.
- **Apply** inheritance to factor out common logic into superclasses and specialize behavior in subclasses.
- **Demonstrate** polymorphism via method overriding and dynamic dispatch.
- **Analyze** multiple inheritance scenarios using the Method Resolution Order.

12.1 Object-Oriented Programming

Object-oriented programming (OOP) is a conceptual framework that combining two computational ideas that we have previously seen in **Chapters 10 and 11**:

1. **Generic operations** via **message passing**: The caller sends a request (message), and the object decides how to handle it.
2. **Object-based abstractions**: Bundling **state** (data) and **behavior** (functions) into a single logical unit.

Python has support for OOP, and thus handles the message passing protocol for us. Therefore, we only need to handle the object-based abstractions.

There are some important terminologies associated with object-oriented programming.

- **Class:** The blueprint or template. It defines the structure (attributes) and capabilities (methods) common to all objects of this type.
- **Instance:** A concrete object created from a class. It occupies memory.
- **Attribute:** A data variable stored on an instance (e.g. `self.balance`).
- **Method:** A function defined inside a class, designed to operate on the instance.

12.1.1 Example: Bank Account

This example demonstrates the encapsulation of state (`balance`) and the logic to modify it.

```
class BankAccount(object):
    def __init__(self, initial_balance):
        # The constructor: runs immediately when the object is created
        self.balance = initial_balance

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            return self.balance
```

```
10          else:
11              return "Money not enough"
12
13      def deposit(self, amount):
14          self.balance += amount
15          return self.balance
```

Using the Instance

```
1  ben_account = BankAccount(100)
2  print(ben_account.withdraw(40))   # 60
3  print(ben_account.withdraw(200))  # "Money not enough"
4  print(ben_account.deposit(20))    # 80
```

The line `self.balance = ...` creates a new variable that lives *inside* that specific object instance.

Now, the code also makes reference to an explicit `self` parameter.

- `self` is a reference to the **current instance** of the class. It allows the code to distinguish between *this* bank account and *another* bank account.
- **Why is it explicitly defined?** By defining `self` as an argument, it is clear that methods operate on specific instance data.

Mechanism: Method Binding

When you define a method, you must include `self` in the definition. However, when you *call* the method, you skip it. Python performs a "magic" translation behind the scenes called **Method Binding**.

The Code You Write:

```
ben_account.deposit(20)
```

What Python Actually Executes:

```
BankAccount.deposit(ben_account, 20)
```

The object before the dot (`ben_account`) is automatically passed as the first argument to the function defined in the class.

12.1.2 Dunder Methods

Python uses special methods enclosed in double underscores ("dunder methods") to integrate objects with built-in syntax.

Method	Purpose	Triggered By
`__init__`	Initialization	`obj = Class()`
`__str__`	User-friendly string	`print(obj)`, `str(obj)`
`__repr__`	Debug string	`repr(obj)`, interactive shell
`__len__`	Length	`len(obj)`
`__add__`	Addition	`obj1 + obj2`
`__eq__`	Equality	`obj1 == obj2`

12.2 Object-Oriented Principles

More rigorously, we shall examine the object-oriented programming principles, which rests on four fundamental pillars that guide the design of robust, scalable software.

The four key pillars of object-oriented programming are:

1. **Abstraction:** Expose *what* an object can do while hiding *how* it does it (Implementation Hiding).
2. **Encapsulation:** Bundle state and behavior together and restrict access to internal details (Information Hiding).
3. **Inheritance:** Reuse and refine behavior via an **is-a** relationship; ensures subtypes are usable wherever supertypes are expected (Liskov Substitution Principle).
4. **Polymorphism:** Allow the same code to exhibit different behaviors based on the runtime type of the objects (Dynamic Dispatch).

12.2.1 Abstraction (Implementation Hiding)

Abstraction is the process of hiding complex implementation details and showing only the essential features of an object. It allows the user to focus on *what* an object does rather than *how* it does it. We have seen the use of functions as a form of abstraction.

For example, if you are driving a car, to stop, you press the brake pedal. You do not need to understand hydraulic pressure, brake pads, or friction coefficients. The pedal is the **abstraction** (interface); the mechanics are the **implementation** (hidden details).

In Python, we achieve abstraction by defining public methods that act as a simple interface for complex internal logic. The user interacts only with the public method, and does not need to understand the internal workings of the function.

```python
class CoffeeMachine:
    def brew_cup(self):
        """The Abstraction: Simple public interface."""
        self._boil_water()
        self._grind_beans()
        self._pour_coffee()
        print("Coffee is ready!")
    # Internal details (Implementation) hidden from the user
    # denoted by the underscore convention
    def _boil_water(self):
        print("Boiling water...")
    def _grind_beans(self):
        print("Grinding beans...")
    def _pour_coffee(self):
        print("Pouring...")
# Usage
machine = CoffeeMachine()
# The user sends one simple message. The complexity is hidden.
machine.brew_cup()
```

Additional: Abstract Base Classes

In object-oriented languages like Java, there is the concept of abstract classes, which only serve as a template for functions that their children must implement. Python does not have interfaces or abstract classes. Instead, in Python, we use **Abstract Base Classes (ABCs)** from the `abc` module to strictly define these contracts.
An abstract class cannot be instantiated directly. It serves as a blueprint, enforcing that all subclasses implement specific methods.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        """Children MUST implement this method."""
        pass

    def description(self):
        # Concrete methods are allowed in abstract classes
        return "I am a shape."

# s = Shape() -> TypeError: Can't instantiate abstract class
    Shape

class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        # Must override abstract method to become concrete
        return 3.14 * self.r ** 2
```

12.2.2 Encapsulation (Information Hiding)

Encapsulation bundles data (state) and methods (behavior) into a single unit and restricts direct access to some of an object's components.

Python does not use strict access modifiers like `public`, `private`, or `protected`. Instead, it relies on **naming conventions**. However, these conventions are not enforced by the interpreter:

- **Public (`name`):** Accessible from anywhere.
- **Protected (`_name`):** Intended for internal use and subclasses only.
- **Private (`__name`):** Triggers **name mangling** (it is renamed to `_Class__name`) to make accidental access harder.

Good encapsulation follows the **Tell, Don't Ask** principle. Do not fetch state from an object to perform logic elsewhere; instead, tell the object to do the work itself.

A major sign of poor encapsulation is fetching data from an object, performing logic, and putting it back ("Asking"). Instead, you should **tell** the object what to do.

For example, imagine we have client code that deducts a monthly maintenance fee if the balance is low from our bank account above.

Approach 1: Asking (Brittle). The client code inspects the internal state directly.

```
# Client Code
if account.balance < 1000:
    account.balance -= 10  # Logic is leaked outside the class
```

If we decide later to change the implementation of `BankAccount` so that `balance` is no longer a simple number but is calculated on the fly from a list of transactions (see below), this client code will crash.

Approach 2: Telling (Robust). The logic is encapsulated inside the class. The client sends a message.

```
# Client Code
account.deduct_monthly_fee() # We tell the object what we want
```

Suppose we refactored the `BankAccount` class to use a **transaction history** instead of a simple float variable.

```
# REFACTORED CLASS implementation
class BankAccount:
    def __init__(self):
        self._transactions = []  # Internal representation changed!

    def deduct_monthly_fee(self):
        # The class updates its own complex internal state
        if sum(self._transactions) < 1000:
            self._transactions.append(-10)
```

As a result,

- The **Asking** code (`account.balance -= 10`) now **fails** with an `AttributeError` (you can't assign to a computed property or modify the missing attribute directly).
- The **Telling** code (`account.deduct_monthly_fee()`) **works perfectly** without any changes to the client code. The abstraction barrier protected the user from the internal change.

Avoid Anemic Models

Avoid exposing every field via trivial getters and setters (e.g. `get_x()`, `set_x()`). This breaks encapsulation and turns your class into a dumb data structure.

12.2.3 Inheritance

Inheritance allows a class to derive attributes and methods from another class. It is appropriate only when there is a strict **is-a** relationship (e.g. a `Ship` *is a* `MobileObject`).

Subclasses inherit all methods and properties by default, fulfilling the Don't Repeat Yourself (DRY) principle, but also can **override** them, providing extensibility.

How do we decide when to use inheritance?

1. **Identify Classes:** Look for the distinct "nouns" in your system (e.g. Ships, Torpedoes).
2. **Identify Commonality:** Do these objects share state (position, velocity) or behavior (move, update)?
3. **Refactor:** Extract this commonality into a **Base Class** (Superclass).

12.2.4 Inheritance vs. Composition

Composition vs Inheritance. When designing systems, you often face a choice: should class A inherit from class B, or should class A *contain* an instance of class B?

- **Inheritance ("Is-A"):** A rigid relationship. Subclasses inherit everything (even methods they do not want/need). Changes to the parent ripple down to all children.
- **Composition ("Has-A"):** A flexible, black-box relationship. Class A holds a reference to Class B and uses it. You can change the behavior at runtime by swapping the contained object.

Inheritance exposes the *entire* interface of the parent. This can violate encapsulation if the parent has methods that the child should not expose.

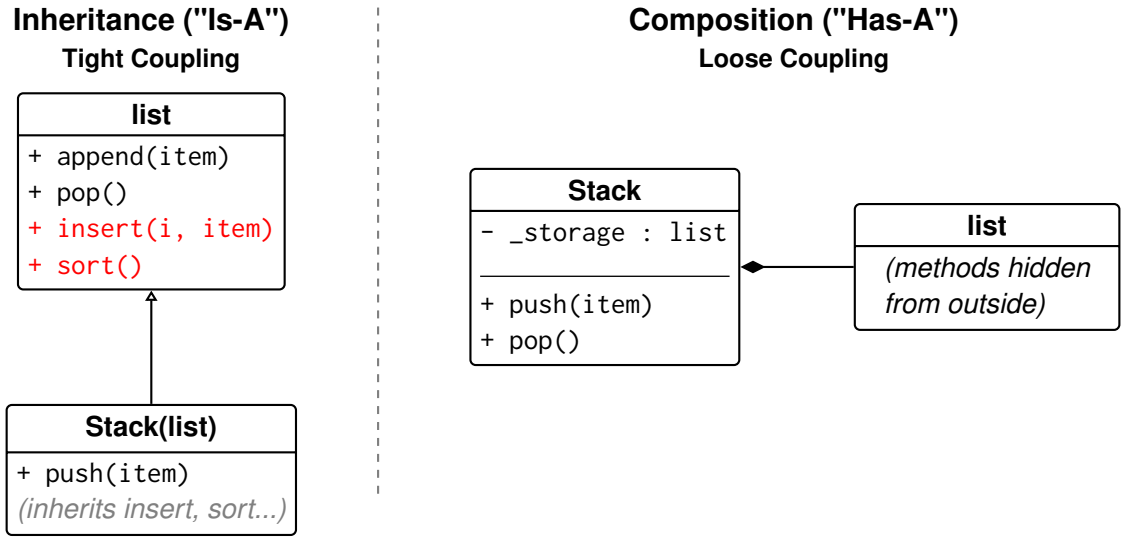


Figure 12.1: Inheritance vs Composition

Consider a stack implemention. A stack should only allow push and pop. If we inherit from Python's built-in `list`, we accidentally inherit methods like `insert()` and `sort()`, which break the "Last-In-First-Out" rule.

```
# BAD: Inheritance
# A Stack IS NOT a List (it behaves differently)
class Stack(list):
    def push(self, item):
        self.append(item)

s = Stack()
s.push(1)
s.insert(0, 99) # DANGEROUS! Inherited method breaks Stack logic.
```

Instead of *being* a list, the stack should *have* a list. We delegate the storage work to an internal list but only expose the methods we want.

```
# GOOD: Composition
# A Stack HAS A List
class Stack:
    def __init__(self):
        self._storage = []  # Internal component

    def push(self, item):
        self._storage.append(item)

    def pop(self):
        return self._storage.pop()

s = Stack()
s.push(1)
# s.insert(0, 99) # ERROR: Method does not exist. Safety ensured.
```

Composition vs Inheritance

Inheritance is best used only for **is-a** relationships where the subclass shares the exact interface of the parent. For everything else (code reuse, sharing logic), use Composition. It decouples your code and makes testing easier.

Suppose we want to model a universe with **ships** and **torpedoes**. Both have a `position` and `velocity`, but ships have extra health and torpedoes have a blast radius.

1. The Base Class (`MobileObject`)

```python
class MobileObject(object):
    def __init__(self, position, velocity):
        self.position = position
        self.velocity = velocity

    def move(self):
        # Common logic: update position based on velocity
        pass
```

2. The Subclass (`Ship`)

The `Ship` inherits the movement logic but adds specific combat capabilities.

```python
class Ship(MobileObject):
    def __init__(self, name, position, velocity, num_torps):
        # 1. Delegate initialization of common fields to the parent
        super().__init__(position, velocity)

        # 2. Initialize subclass-specific state
        self.name = name
        self.num_torps = num_torps

    def fire_torps(self):
        if self.num_torps > 0:
            self.num_torps -= 1
            print(f"{self.name} fired a torpedo!")
```

super(). We use `super().__init__` to call the parent's logic.

Why not just set `self.position` manually?

- **Maintainability:** If `MobileObject` changes (e.g. adds coordinate validation), `Ship` inherits that safety automatically.
- **Cooperation:** In multiple inheritance (see below), `super()` ensures every class in the hierarchy is initialized exactly once.

Liskov Substitution Principle (LSP)

Inheritance is powerful but dangerous. The LSP provides the safety rule:
Formal Definition: If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program.
The "Contract" Analogy: A subclass must honor the contract of the superclass.

- **Preconditions:** The subclass cannot be "pickier" than the parent (e.g. if Parent accepts any number, Child cannot strictly require positive integers).
- **Postconditions:** The subclass cannot deliver "less" than the parent promised.

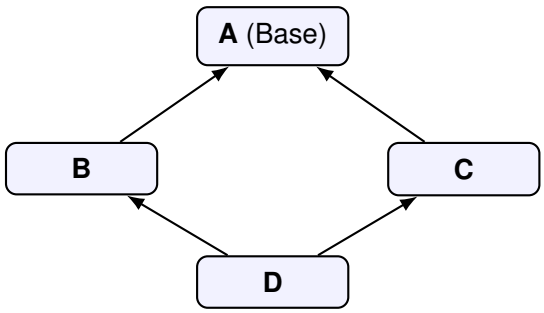
12.2.5 Multiple Inheritance and the MRO

Unlike Java, which restricts classes to a single parent, Python supports **multiple inheritance**. A class can inherit from multiple parents simultaneously:

```
class Child(Parent1, Parent2): ...
```

This feature introduces a complex structural problem: The inheritance hierarchy is no longer a simple tree; it is a **directed acyclic graph (DAG)**. To determine which method to run when `child.method()` is called, Python must flatten this graph into a specific linear sequence. This sequence is the **Method Resolution Order (MRO)**.

(The Diamond Problem). The most famous ambiguity in multiple inheritance is the "Diamond Problem," occurring when two parent classes inherit from the same grandparent.



The Conflicts:
1. If D calls a method defined in both B and C, which runs first?
2. If D initializes, does A's `__init__` run twice (once via B, once via C)?

Figure 12.2: The Diamond Structure. Python needs a path that visits every node exactly once.

C3 Linearization

To solve this, Python uses the **C3 Linearization Algorithm**. It constructs the MRO list by strictly adhering to three rules:

1. **Children precede Parents:** Class D must appear before B and C. B appears before A.
2. **Left-to-Right Ordering:** If defined as `class D(B, C)`, then B must be checked before C.
3. **Monotonicity:** The relative order of classes inherited from parents must be preserved. If B comes before A in B's MRO, it must also come before A in D's MRO.

The C3 Solution: Python effectively performs a search that respects the hierarchy constraints, resulting in:

$$\textbf{MRO: } [D, B, C, A, \text{object}]$$

`super()` delegates to the **next class in the MRO chain**, determined at **runtime** based on the instance's type.

This leads to "sideways" calls: In the diamond above, `super()` inside **B** will call **C**, not A!

```
class A:
    def method(self):
        print("A execution")

class B(A):
    def method(self):
        print("B execution")
        super().method()  # Wait for it...

class C(A):
    def method(self):
        print("C execution")
        super().method()

class D(B, C):  # B is first
    def method(self):
        print("D execution")
        super().method()

d = D()
d.method()
print(D.mro())
```

Execution Trace:

1. `d.method()` calls `D.method`.
2. `D` calls `super()`. The MRO is $[D, \mathbf{B}, C, A]$. Next is **B**.
3. `B` executes. It calls `super()`.
4. **CRITICAL STEP:** Even though we are inside class B, the instance `self` is still of type D. Python looks at D's MRO: $[D, B, \mathbf{C}, A]$. The class following B is **C**.
5. `C` executes (Sideways jump!). It calls `super()`. Next is **A**.
6. `A` executes.

Output:

```
D execution
B execution
C execution  <-- The "magic" of C3 Linearization
A execution
```

This mechanism ensures that the common base class **A** is executed **exactly once**, and only after all its subclasses have run. This is essential for cooperative multiple inheritance (e.g. initializing resources correctly without duplication).

12.2.6 Polymorphism

Polymorphism allows code to be written for a general type (e.g. `Shape`) but work correctly for specific runtime types (e.g. `Circle`, `Square`).

Dynamic Dispatch. Python resolves which method to call at **runtime**.

```
shapes = [Circle(5), Square(4)]

for s in shapes:
    # Polymorphism: strict type doesn't matter,
    # as long as .area() exists.
    print(s.area())
```

This makes code **future-proof**: we can add a `Triangle` class later, and the loop above will work without modification.

There are two related concepts, overriding and overloading:

- **Overriding:** (Supported) Subclass replaces parent method. Lookup follows the class hierarchy.
- **Overloading:** (Not natively supported) Defining multiple functions with the same name but different argument lists.

In Python, defining a function twice simply overwrites the first one. We can use variable arguments to simulate overloading.

```
def area(shape, *args):
    if shape == "circle":
        r = args[0]
        return 3.14 * r * r
    elif shape == "rect":
        w, h = args
        return w * h
```

In the example below, we combine inheritance, overriding, and `super()` to modify behavior slightly rather than replacing it entirely.

```
class Speaker(object):
    def say(self, stuff):
        print(stuff)
class Lecturer(Speaker):
    def lecture(self, stuff):
        self.say(stuff)
        self.say("You should be taking notes.")
class ArrogantLecturer(Lecturer):
    def __init__(self, favourite_phrase):
        self.favourite_phrase = favourite_phrase
    def say(self, stuff): # Extend the parent's behavior
        super().say(stuff + " " + self.favourite_phrase)
ben = ArrogantLecturer("...how cool is that?")
ben.say("Python") # Output: "Python ...how cool is that?"
```

12.2.7 `isinstance` vs `type`

When checking types in OOP, we almost always prefer checking for *behavior compatibility* (is-a) rather than *exact identity*.

1. **`isinstance(obj, Class)`:** Checks for **inheritance**. It asks, "Is this object a `Class` *OR any subclass* of `Class`?"
2. **`type(obj) == Class`:** Checks for **identity**. It asks, "Is this object *exactly* an instance of `Class` (and not a subclass)?"

```
class Vehicle(object):
    pass
class Truck(Vehicle):
    pass
t = Truck()

# PREFERRED: Checks inheritance chain
isinstance(t, Vehicle)  # True (A Truck is a Vehicle)

# DISCOURAGED: Checks exact memory class
type(t) == Vehicle      # False (A Truck is not exactly a Vehicle)
```

12.3 Summary: Benefits and Costs of OOP

In summary, we have seen a lot of benefits of working in an object-oriented style. However, there are costs associated with working in an object-oriented style.

- **Benefits:**
  - **Modularity:** Organizes code around data.
  - **Reuse:** Inheritance prevents code duplication.
  - **Extensibility:** New subclasses can be added without modifying existing code.
- **Costs:**
  - **Complexity:** Deep inheritance hierarchies are hard to understand (the "Yo-Yo Problem").
  - **Ambiguity:** Multiple inheritance requires careful management of MRO.
  - **Overhead:** Dynamic dispatch is slightly slower than direct function calls.

The "Yo-Yo Problem" (Deep Inheritance Costs)

One of the most significant hidden costs of deep inheritance hierarchies is the **Yo-Yo Problem**. This term describes the mental fatigue a developer experiences when trying to understand code logic that is fragmented across a long chain of inheritance.
To trace a method call like `self.update()`, the reader travels up and down the hierarchy:

1. **UP:** Subclass calls `super().update()`. Reader jumps to Parent.
2. **UP:** Parent calls `super().update()`. Reader jumps to Grandparent.
3. **DOWN:** Logic returns to Parent to execute remaining lines.
4. **DOWN:** Logic returns to Subclass to execute remaining lines.