Composition

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 9A's technical content. See python-composition[2] for the full reference on has-a relationships.

Today's outline

- The DRY principle, and two ways to reuse classes: composition and inheritance
- Composition ("has-a" relationships)
- Worked example: `DroneFlight` composed of `Battery`, `Camera`, and `Motor`
- Hot-swapping composed objects

Recap

A quick recap table of last lecture's dunder methods:

Category	Key methods	What they enable
Lifecycle	`__init__`	Control object initialisation

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-composition.pdf

Category	Key methods	What they enable
String rep	`__repr__`, `__str__`	Unambiguous & user-friendly text
Numeric ops	`__add__`, `__sub__`, `__mul__`, …	Custom arithmetic (`1/2 + 1/3`, `v1 * v2`)
Comparisons	`__eq__`, `__lt__`, `__gt__`, …	Sorting, `==`, `<`

And representation invariants:

What?	Why?	How?
Conditions that must hold for an object's private state at all times (e.g. $0 \leq$ altitude ≤ 120, ID is 6 digits).	Prevents invalid states; simplifies reasoning & testing (assume invariants true); catches bugs early with clear, localised checks.	Set valid values in `__init__`; centralise assertions in a private helper like `_check_invariants()`.

How to reuse code?

DRY (don't repeat yourself) principle: in general, duplication and copy/paste are bad —

- Increased risk of bugs.
- Maintenance overhead.
- Poor readability.
- Code bloat.
- Signals that a common abstraction/refactoring is needed.

Today: how to reuse *classes* in a new class? Via **composition** or **inheritance**.

Composition

We can use objects built from other classes (implemented by us or others) as attributes of our new class/object. This isn't new — we've already used integers, strings, lists, etc. as attributes for our classes; we can do this with other (more complex) classes as well. For example, if we are implementing a `Robot` class, we can use an object from a `Battery` class (already implemented by us or others) as one of its attributes.

Modularity:

- A `Student` **has a** `String` (e.g. to store the student's name).
- A `Robot` **has a** `Sensor`, `Motor`, etc.

Has-a relationship

When an object (say `Car`) includes another object inside itself as an attribute (say `Engine`), this forms a **has-a** relationship — we call this **class composition**:

```
>>> class Car():
...     def __init__(self, engine: Engine) -> None:
...         self._engine = engine   # Car has-a engine

engine = Engine()
car = Car(engine)
car.engine.start()
car.engine.stop()
print(car.engine.power)
```

Which is clearer: "is-a" or "has-a"?

Recall the `DroneFlight` representation-invariants exercise from last lecture (Drone ID, altitude, battery). Which one sounds clearer: *"A Drone is a Camera"* or *"A Drone has a Camera"*? A drone "has a" camera, "has a" motor, and "has a" battery — composition is the natural fit here, not inheritance.

Worked example: `DroneFlight`

```
class Battery:
    def __init__(self, capacity: int = 100) -> None:
        if capacity < 0:
            raise ValueError("Capacity must be non-negative.")
        self._level = capacity   # percentage

    def use_power(self, amount: int) -> None:
        if amount < 0:
            raise ValueError("Power usage must be non-negative.")
        self._level -= amount
        if self._level < 0:
            self._level = 0
        print(f"Power used: {amount}%. Remaining: {self._level}%.")

    def get_level(self) -> int:
        """Return remaining battery charge (percentage)."""
        return self._level
```

```
class Camera:
    def __init__(self, resolution: str = "12MP") -> None:
        self._resolution = resolution

    def take_photo(self) -> None:
        print(f"Photo taken at {self._resolution} resolution.")


class Motor:
    def __init__(self, power_rating: float = 100.0) -> None:
        self._power_rating = power_rating   # Watts
        self._is_running = False

    def start(self) -> None:
        self._is_running = True
        print("Motor started.")

    def stop(self) -> None:
        self._is_running = False
        print("Motor stopped.")


class DroneFlight:
    _ID_LEN = 6
    _MAX_ALT = 120.0   # metres (CASA limit)

    def __init__(self, flight_id: str, battery: Battery,
                 camera: Camera, motor: Motor) -> None:
        self._id = flight_id
        self._altitude = 0.0
        self._battery = battery   # Drone has-a Battery
        self._camera = camera     # Drone has-a Camera
        self._motor = motor       # Drone has-a Motor
        self._check_invariants()

    def get_flight_id(self) -> str:
        """Return the immutable flight identifier."""
        return self._id

    def get_altitude(self) -> float:
        """Return current altitude in metres."""
```

```
        return self._altitude

    def get_battery(self) -> int:
        """Return remaining battery charge (percentage)."""
        return self._battery.get_level()   # don't call use_power(0) here! just show the leve

    def __str__(self) -> str:
        return (f"DroneFlight({self._id}, alt={self._altitude:.1f} m, "
                f"bat={self.get_battery()}%)")

    def ascend(self, metres: float) -> None:
        """Climb *metres* metres (cannot exceed the legal ceiling)."""
        if metres < 0:
            raise ValueError("ascend() expects a positive distance.")
        self.set_altitude(min(self._altitude + metres, self._MAX_ALT))
        self._check_invariants()

    def land(self) -> None:
        """Land the drone and consume 5% battery."""
        self.set_altitude(0.0)
        self._battery.use_power(5)
        if self._motor.is_running():
            self._motor.stop()
        self._check_invariants()

    def take_photo(self) -> None:
        if self._battery.get_level() < 5:
            print("Not enough battery to take photo.")
            return
        self._camera.take_photo()
        self._battery.use_power(5)

>>> b = Battery(); c = Camera(); m = Motor()
>>> d = DroneFlight("SAD564", b, c, m)
>>> d.ascend(80); d.take_photo()
Photo taken at 12 MP resolution.
Power used: 5%. Remaining: 95%
>>> d.land()
Power used: 5%. Remaining: 90%
>>> print(d)
DroneFlight(SAD564, alt=0.0 m, bat=90 %)
```

`get_battery()` delegates straight to `self._battery.get_level()` rather

than `self._battery.use_power(0)` — the comment on the slide ("Don't call use_power(0) here! Just show the level.") is a reminder that even a "zero-amount" power use would still print a spurious "Power used: 0%…" message and is conceptually the wrong operation for a getter to perform. A getter should just report state, not have side effects.

Hot-swapping

Because a composed attribute is just an object reference, it can be swapped out for another compatible object at any time:

```
class ThermalCamera:
    def __init__(self, resolution: str = "12MP"):
        self.resolution = resolution

    def take_photo(self):
        print("Thermal image saved.")
```

```
>>> b = Battery(); c = Camera(); m = Motor()
>>> d = DroneFlight("SAD564", b, c, m)
>>> d.take_photo()
Photo taken at 12 MP resolution.
Power used: 5%. Remaining: 95%
>>> d.camera = ThermalCamera()
>>> d.take_photo()
```

`ThermalCamera` doesn't inherit from `Camera` at all — it just happens to provide a `take_photo()` method with a compatible signature, so it works as a drop-in replacement. This is an example of *duck typing*: Python doesn't check that the new object is "officially" a `Camera`, only that it supports the operations actually used on it.

The companion `composition.py` file demonstrates a more defensive hot-swap via a dedicated `swap_engine()` method on its `Car`/`Engine` example, which stops the currently-running engine *before* swapping it out — a small improvement over directly reassigning an attribute mid-flight, since a direct reassignment (like `d.camera = ThermalCamera()` above) doesn't get a chance to clean up the object it's replacing. See python-composition[3] for the full listing.

[3] courses/csse1001/python-composition.pdf

Summary

Build by pieces: snap self-contained objects together to create a richer whole.

- **Modular**: each component handles one clear task.
- **Reusable**: same parts work in new contexts; no code rewrite.
- **Readable**: data flow is explicit and easy to trace.
- **Few side effects**: isolated state keeps surprises and bugs low.

Next: 2025-09-23-inheritance[4] (Lecture 9B).

[4] courses/csse1001/2025-09-23-inheritance.pdf