Lecture 19 | OOP: exercises

Controlled methods, state, validation and exceptions

Six problems, increasing autonomy

Guided 01 — Water tanks: improve the class built in Lecture 18

Lab 01 — Animals: methods, controlled energy and sound

Lab 02 — Game characters: health, attacks and interactions between objects

Independent 01 — Person records: attributes, methods and input validation

Independent 02 — Course registrations: lists of objects and summaries

Independent 03 — Warehouse dispatch: CSV input, class methods and report

Focus: validate state changes, communicate failures and reuse classes from modules

Guided 01 | Return to the water tanks

- Continue the `WaterTank` class from Lecture 18 without changing the constructor
- Each tank stores `owner`, `liters` and `capacity`
- Previously, methods could produce 95 L in an 80 L tank or −20 L after excessive removal
- Now make `.add_water()` and `.take_water()` preserve valid quantities
- Keep the class in `water_tank.py` and use it from main.py

Rules to implement

- Reject non-positive amounts with raise `ValueError`
- For additions, accept only the amount that fits, never exceeding capacity
- For withdrawals, remove at most the water available, never going below zero
- Return the quantity actually added or removed
- Add `.get_level()` returning the current liters

Guided 01 | Method structure

- Start from the two simple methods in the previous exercise
- First reject amount `<= 0` using raise `ValueError`
- Compute the space available as capacity – liters
- Use `min(amount, available_space)` to determine how much can actually fit
- For removal, use `min(amount, liters)`
- Update `self.liters` and return the actual change

water_tank.py — partial guide

```
def add_water(self, amount):
    if amount <= 0:
        raise ValueError("Positive amount required")
    # TODO: calculate accepted liters
    # TODO: update and return the result


def take_water(self, amount):
    # TODO: reject non-positive amount
    # TODO: remove at most available liters
    # TODO: update and return the result
```

Guided 01 | Tests in the main program

- The starter `main.py` imports `WaterTank` and creates `wt_alice` and `wt_bob`
- Use the returned values to report the effective additions and removals
- Wrap a method call with a user-provided amount in try-except
- For this guided exercise assume the user enters a numeric value
- The exception is raised by the method, not by the conversion

Prepared test sequence

```
wt_alice = WaterTank("Alice", 40, 80)
wt_bob = WaterTank("Bob", 70, 80)
print(wt_alice.add_water(25))   # 25
print(wt_alice.add_water(30))   # 15
print(wt_bob.take_water(20))   # 20
print(wt_bob.take_water(70))   # 50
```

Expected final levels: Alice 80 L, Bob 0 L

Guided 01 | Understand the improvement

Before the checks, the tank could enter an impossible state

Now the methods enforce the limits every time they are called

If a request exceeds the physical limit, only the feasible part is carried out

A non-positive request is invalid: raise signals this error to the caller

The caller can catch the error and keep the program running

Discussion: why is calling a controlled method safer than modifying self.liters directly?

Lab 01 | Animals and everyday behavior

- A small animal-care center records its animals in a simple program
- Define class `Animal` in a single Python script (a separate module is not required)
- The initializer receives `name`, `species` and `energy`, an integer between 0 and 100
- Implement `.describe()`, `.make_sound()` and `.feed(amount)`
- The first method returns a descriptive string, and `.make_sound()` returns a generic sound such as "..."
- Use two separate instances, `animal_dog` and `animal_cat`, to check independent state

Lab 01 | Implement controlled methods

Method requirements

- `.describe()` returns a string containing name, species and current energy
- `.make_sound()` returns the string "..." for any Animal
- `.feed(amount)` adds energy but never exceeds 100

For amount <= 0, `.feed()` raises `ValueError`

- For valid amounts, return the units of energy actually added
- Use an if statement and `min()` to keep the code short

Prepared test calls

```
animal_dog = Animal("Buddy", "dog", 80)
animal_cat = Animal("Milo", "cat", 35)

print(animal_dog.make_sound())  # ...
print(animal_dog.feed(30))    # 20
print(animal_cat.feed(15))    # 15
print(animal_dog.energy)       # 100
```

Check: `animal_cat.energy` is still 50

Lab 01 | Check each object independently

Print `.describe()` for both animals after feeding them

Try `.feed(0)` in a try-except `ValueError` block and display the error message

Verify that the invalid operation leaves the energy unchanged

Every Animal currently produces the same generic sound

An animal care center might later need different sounds and behaviors for different species

For now, implement only `Animal`: do not define `Dog` or `Cat` classes

Lab 02 | Game characters and health

- A simple fantasy game has characters who can attack one another
- Define class Character in character.py and use it from `main.py`
- Instance attributes: `name`, `health` and `attack_power`
- Initialize each character with positive `health` and `attack` power
- Implement .take_damage(amount), `.attack(target)` and `.is_alive()`
- For now, every character uses exactly the same attack rule

Required behavior

- `.take_damage(amount)` rejects non-positive amounts with raise `ValueError`
- For valid damage, health cannot fall below zero; return actual damage suffered
- `.attack(target)` calls `target.take_damage(self.attack_power)` and returns the result
- `.is_alive()` returns True exactly when `health > 0`
- An attack on a defeated target inflicts 0 additional damage

Lab 02 | Attacks between objects

In `main.py`, import `Character` from `character.py`

- Create `character_arthur` with 40 health and attack power 12
- Create `character_merlin` with 25 health and attack power 8
- Make the objects attack one another by calling `.attack(other_character)`
- Print the health of each character and test `.is_alive()`
- Use try-except to catch a `ValueError` raised by `.take_damage(0)`

Example calls

```
character_arthur = Character("Arthur",
  40, 12)
character_merlin = Character("Merlin",
  25, 8)


print(character_arthur.attack(character_
  merlin)) # 12
print(character_merlin.health) # 13
print(character_merlin.attack(character_
  arthur)) # 8
print(character_arthur.health) # 32


Check: attacks update only the target
```

Lab 02 | Verify the game rules

Test an attack that would take a character below zero: the final health must be 0

An already defeated character has health 0 and cannot lose additional health

Check that `.is_alive()` changes from `True` to `False` after a decisive attack

Invalid damage amounts must raise `ValueError` without changing health

The `Character` class provides a shared set of rules for all characters

Do not define separate Warrior or Mage classes yet; the basic system comes first

Independent 01 | Person records

A small registry stores information about people and can update their records

Create class Person in person.py and a separate `main.py` to test it

The initializer accepts `name`, `age` and `city`

Store these values in instance attributes, with age initially at least 0

Implement `.introduce()` returning a short sentence containing name, age and city

Implement `.birthday()` to increase age by one and return the new age

Implement `.move_to(new_city)` to update the city

Reject an empty `new_city` with `raise ValueError`; catch this error in `main.py`

Independent 01 | Test the Person class

Create `person_alice = Person("Alice", 20, "Lisbon")`

Create `person_bob = Person("Bob", 32, "Porto")`

Print their introductions before and after the operations

Call `person_alice.birthday()`: her age should become 21

Move `person_bob` to `"Coimbra"`: only his city should change

Try `person_bob.move_to("")` and handle `ValueError` in `main.py`

Verify that invalid input leaves the original `city` unchanged

Independent 02 | Course registrations

A course coordinator maintains separate records of student registrations

Define `CourseSection` in `course_section.py` and use it in `main.py`

Instance attributes: `code`, `capacity` and `enrolled` (starting at 0)

Implement `.enroll()`, `.cancel()`, and `.get_available_places()`

An enrollment is accepted only when a seat remains; a cancellation only when at least one student is enrolled

Return `True` or `False` to indicate whether each operation succeeds

Create at least three `CourseSection` objects in a list

Independent 02 | Process a list of sections

- Start with sections with capacities 2, 1 and 3
- Make enrollment attempts so that at least one section becomes full
- Try one extra enrollment in the full section, then cancel one registration
- Use a `for` loop to print for each section: code, enrolled and available places
- Calculate the total enrolled across the list using a loop and accumulator

Expected checks

An extra enrollment in a full section returns False

A cancellation from an empty section returns False

No section may have enrolled < 0 or enrolled > capacity

The final total equals the sum of the enrolled attributes

Independent 03 | Warehouse dispatch report

A warehouse receives dispatch requests from a CSV file

Each request contains `Product, Requested` and `Available`

Define `DispatchRequest` in `dispatch_request.py` with instance attributes `product`, `requested` and `available`

Add `.process()` returning the number of units actually dispatched, no more than available

For requested <= 0, `.process()` must `raise ValueError`

For a valid request, update `available` and return dispatched units

Write `main.py` to read requests, create objects, process them and export a report

Independent 03 | Input and outputs

File: requests.csv

```
Product,Requested,Available
Pencils,8,10
Notebooks,7,4
Folders,2,2
Erasers,-3,5
Markers,5,0
```

All fields are present and numeric quantities are integers

Output file: dispatch_report.csv

- Columns: `Product, Requested, Dispatched, Remaining`
- For a rejected request, write Dispatched = 0 and Remaining equal to initial Available
- Use try-except around `.process()` to handle `ValueError`
- Compute total dispatched units from the resulting objects
- Do not retry rejected requests

Independent 03 | Verification

Pencils: dispatch 8, leaving 2

Notebooks: dispatch 4, leaving 0

Folders: dispatch 2, leaving 0

Erasers: invalid request, dispatch 0, leaving 5

Markers: dispatch 0, leaving 0

Expected total dispatched units: 14

Use CSV reading and writing techniques already studied; do not change the original input file

Learning check | Controlled objects

What makes a method a safer interface than direct modification of an attribute?

What is the difference between returning `False` and raising `ValueError`?

When should the caller use try-except rather than the method?

Which attributes belong to each instance and which can be shared at class level?

How do we keep a class in its own module and import it into `main.py`?