Python: Day 03

Object-Oriented Programming

Previous Agenda

01

Lists & Tuple

Ordered Group

02

Dictionary & Set

Unordered Group

03

String

Handling Text

04

File Handling

Data outside code

05

Comprehension

Iteration Shortcut

06

Lab Session

Culminating Exercise

Agenda

01

Definition

Data-Centric Approach

02

Hierarchy

Organizing Data

03

Polymorphism

Handling data types

04

Encapsulation

Data Hiding

05

GUI

Introduction to Tkinter

06

Lab Session

Culminating Exercise

01

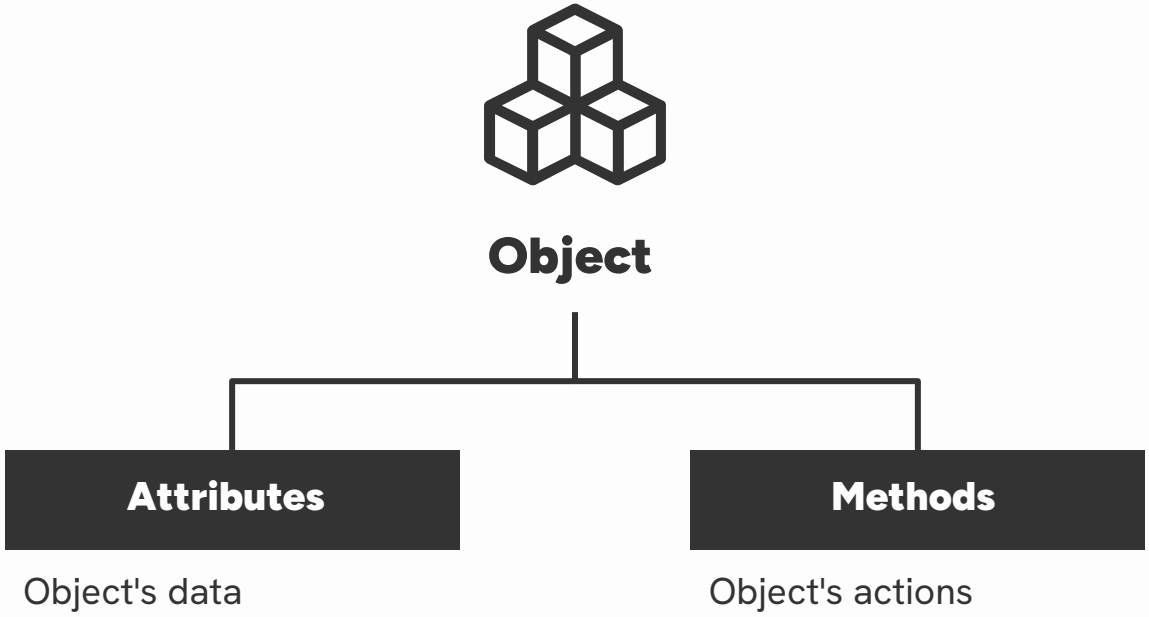
Definition

Programming with a focus on concepts

What makes something something ?


PARKER
PARKER

What is Object Oriented Programming?



Has → Is

Functional Identity

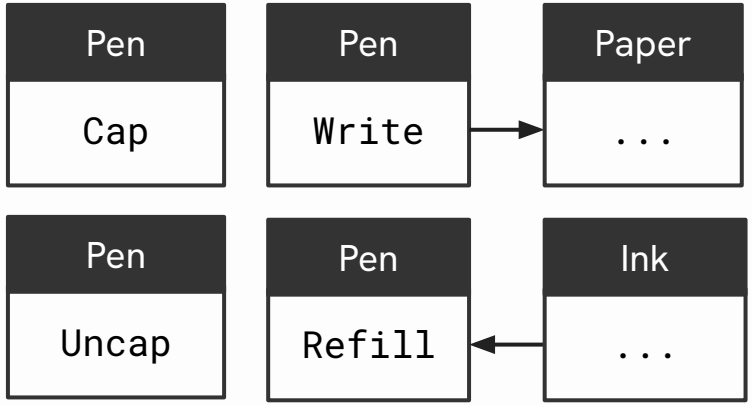
Attributes

- Attributes are unique to one object

Pen	
Brand	Pilot
Color	Black
Capped	False

Methods

- Methods can change itself or others



Object Similarities

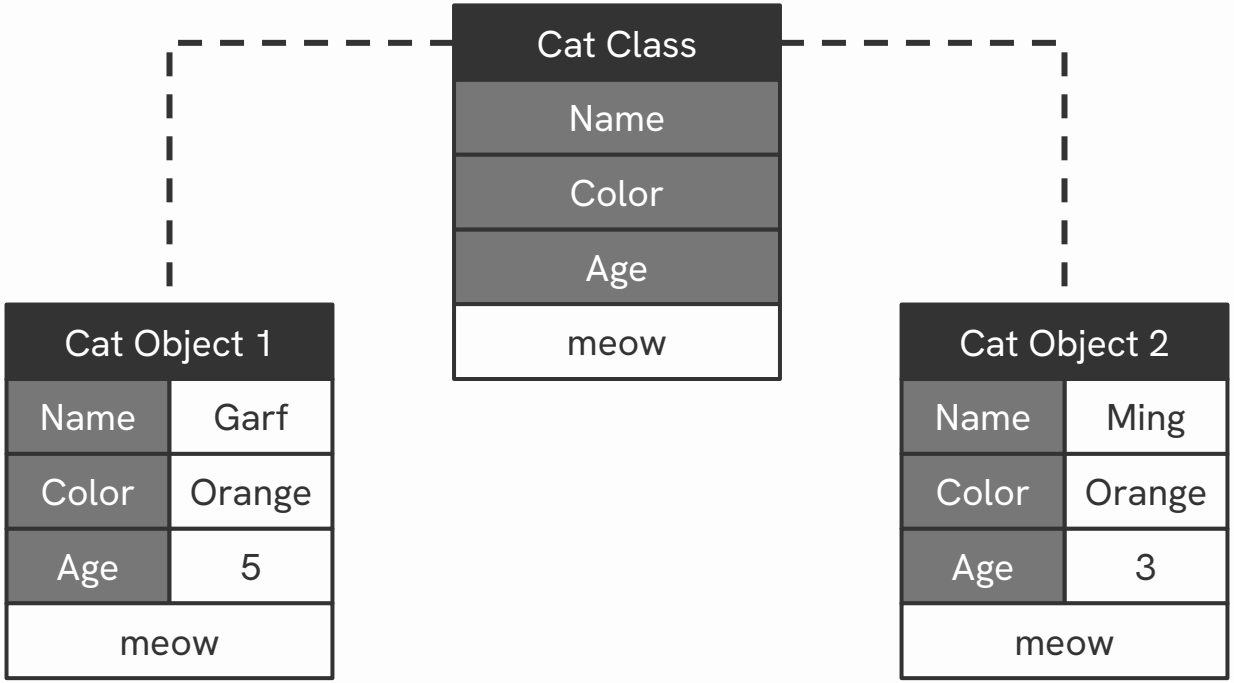
cat1	
Name	Garf
Color	Orange
Age	5
meow	

cat2	
Name	Ming
Color	Orange
Age	3
meow	

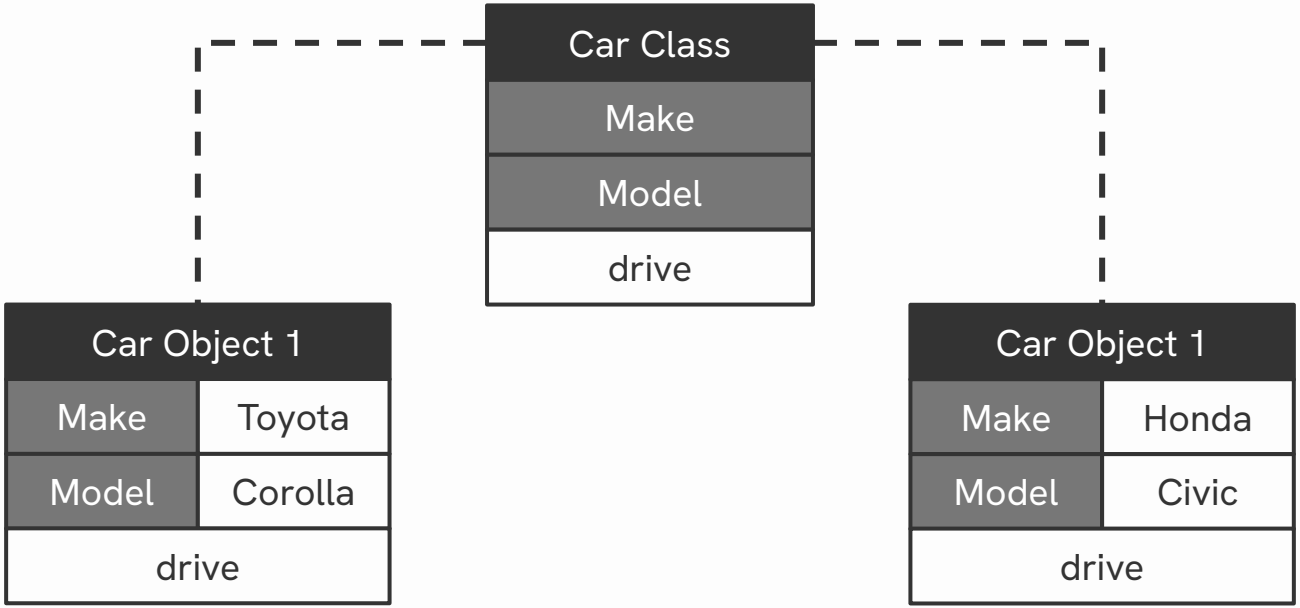
cat3	
Name	Mona
Color	Black
Age	2
meow	

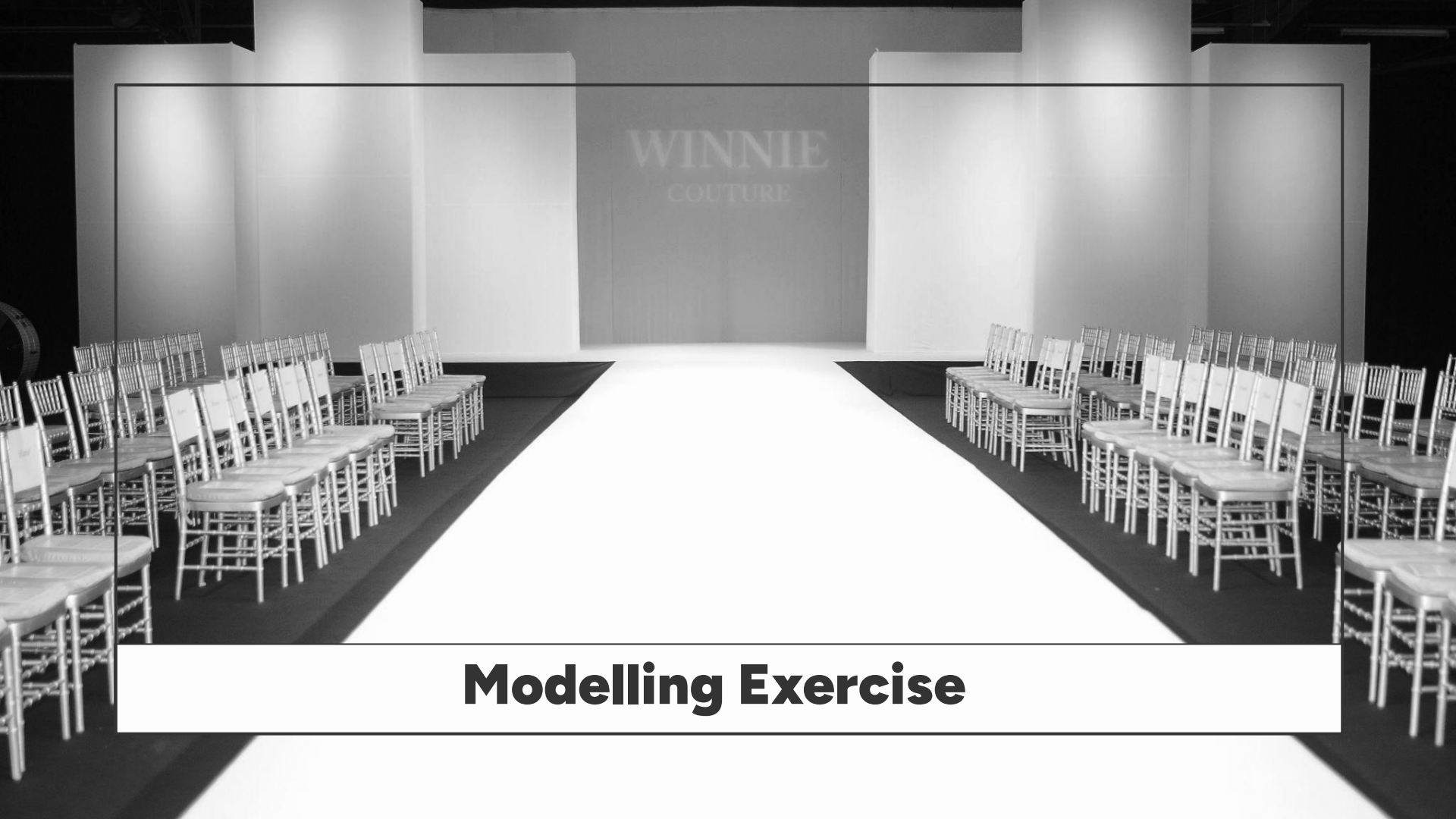
What makes them different/same?

Classes to Objects

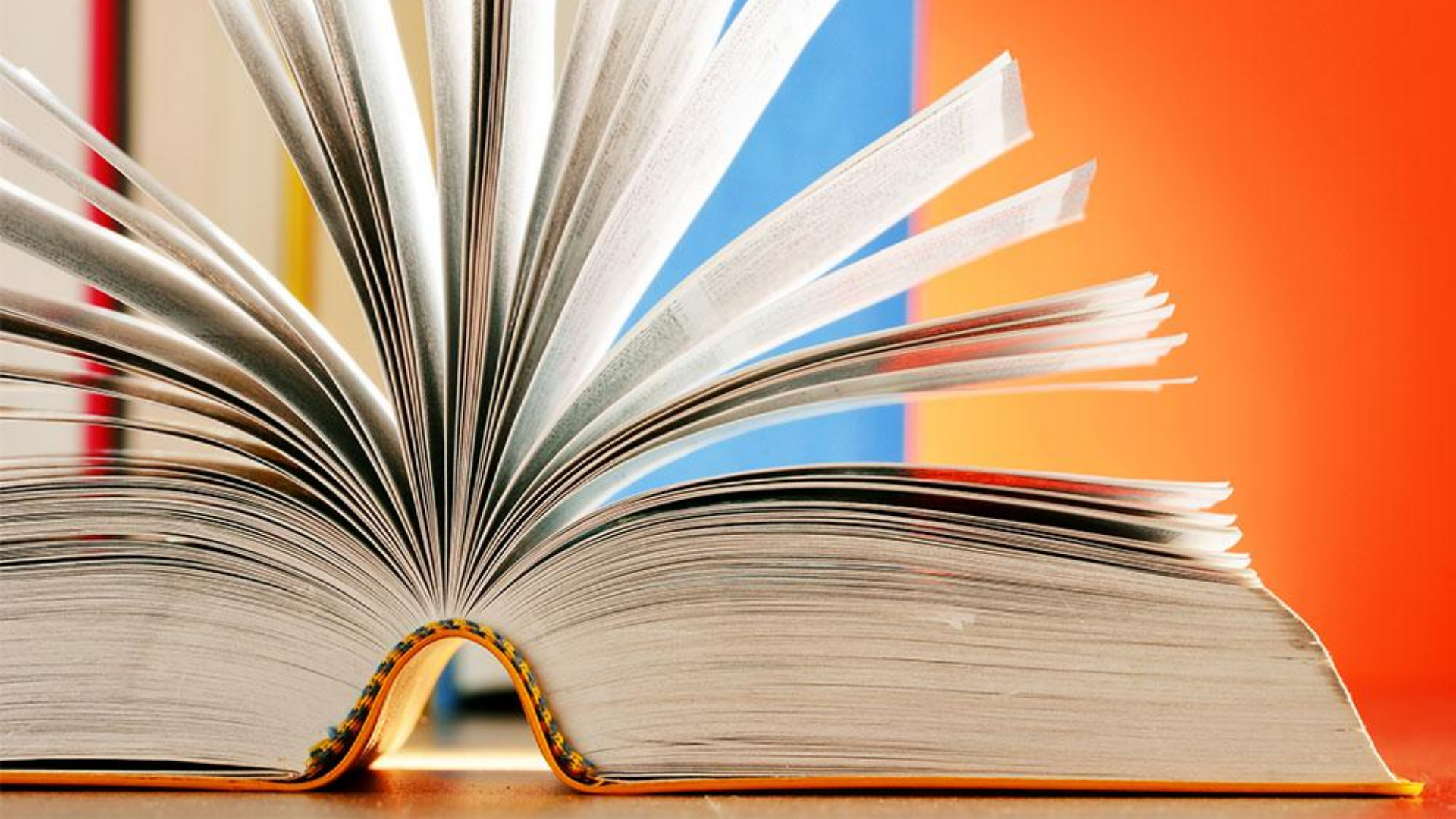


Classes to Objects



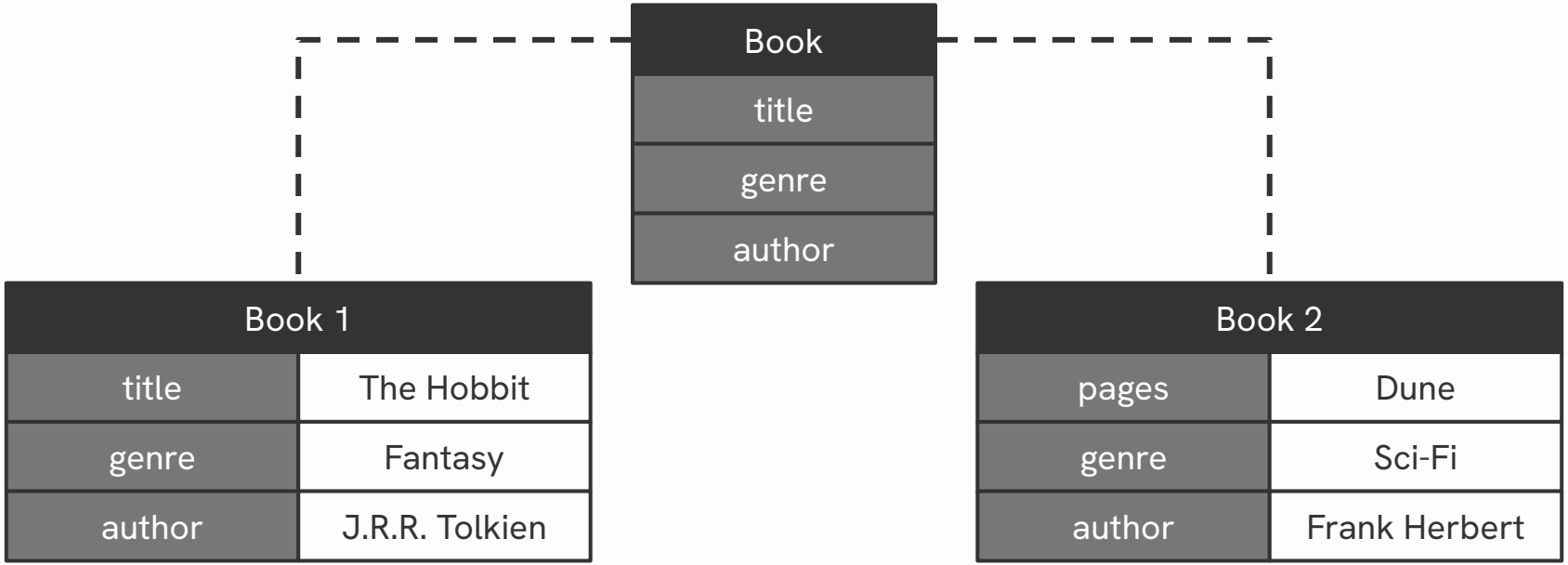


Modelling Exercise

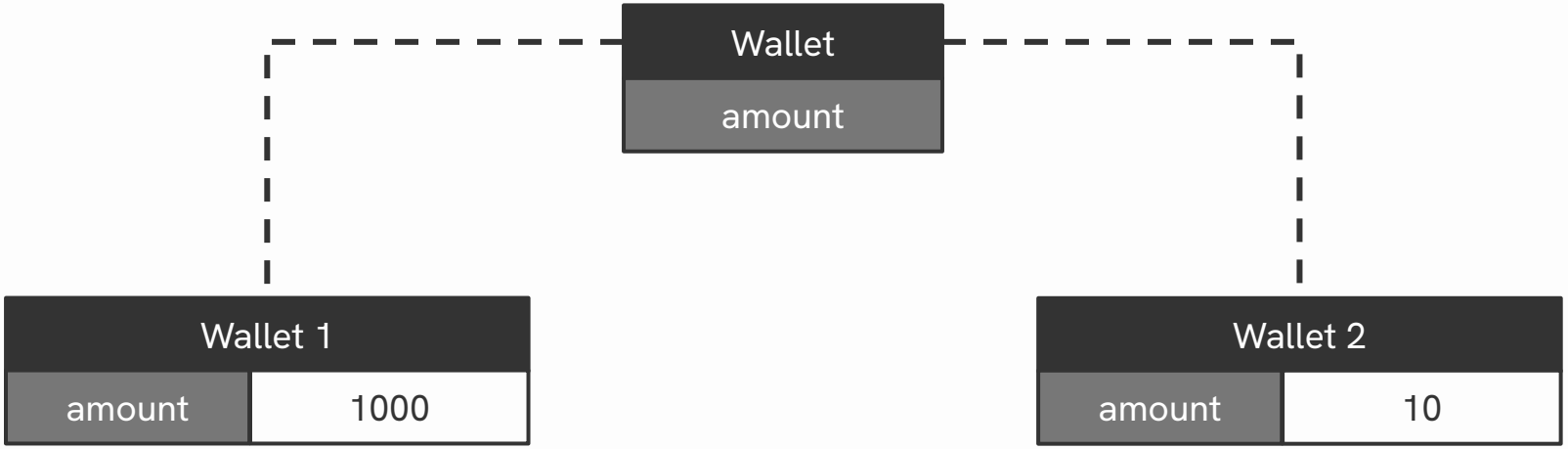



BPI
BPI
Makati Main
OPEN
WELCOME

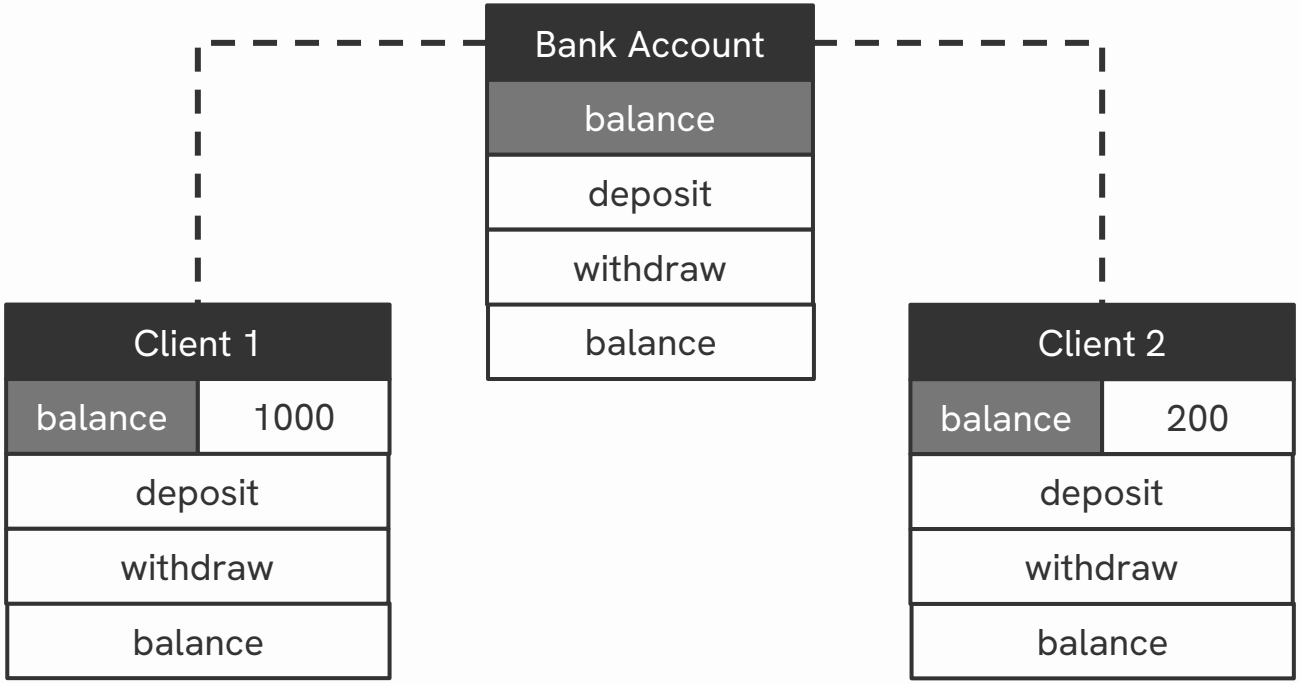
Book


Book
title
genre
author
Book 1
title
The Hobbit
genre
Fantasy
author
J.R.R. Tolkien
Book 2
pages
Dune
genre
Sci-Fi
author
Frank Herbert

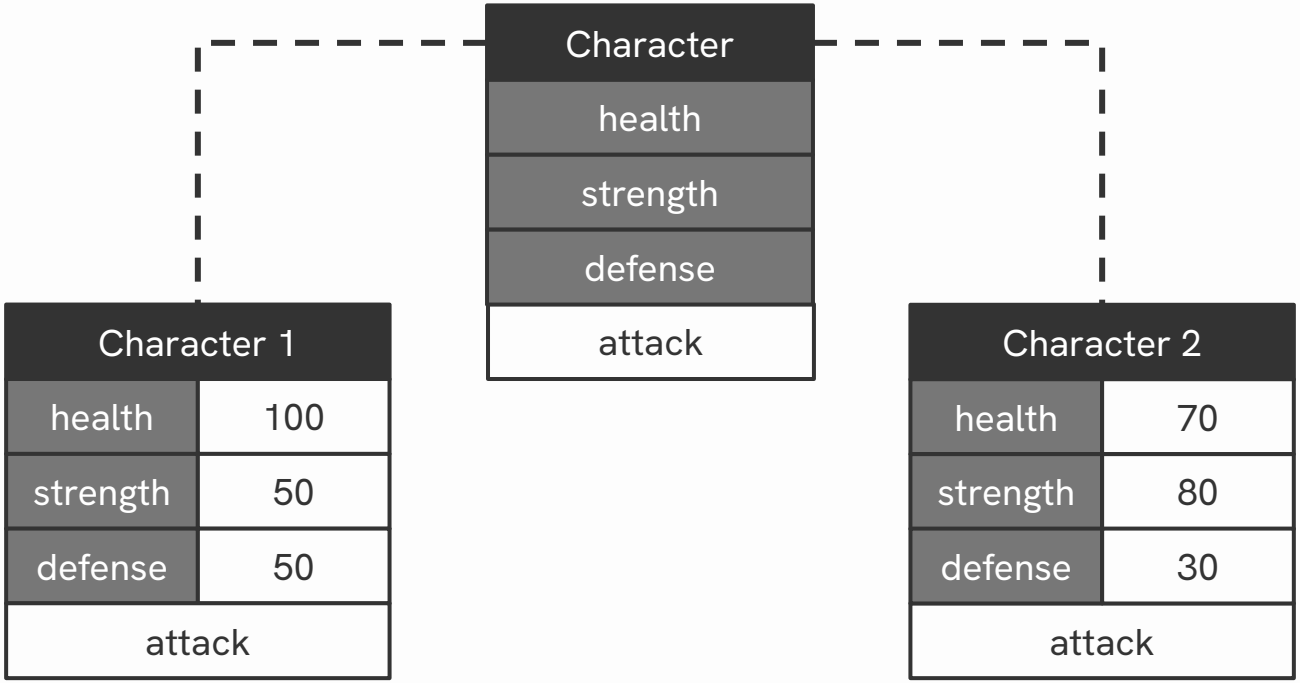
Wallet

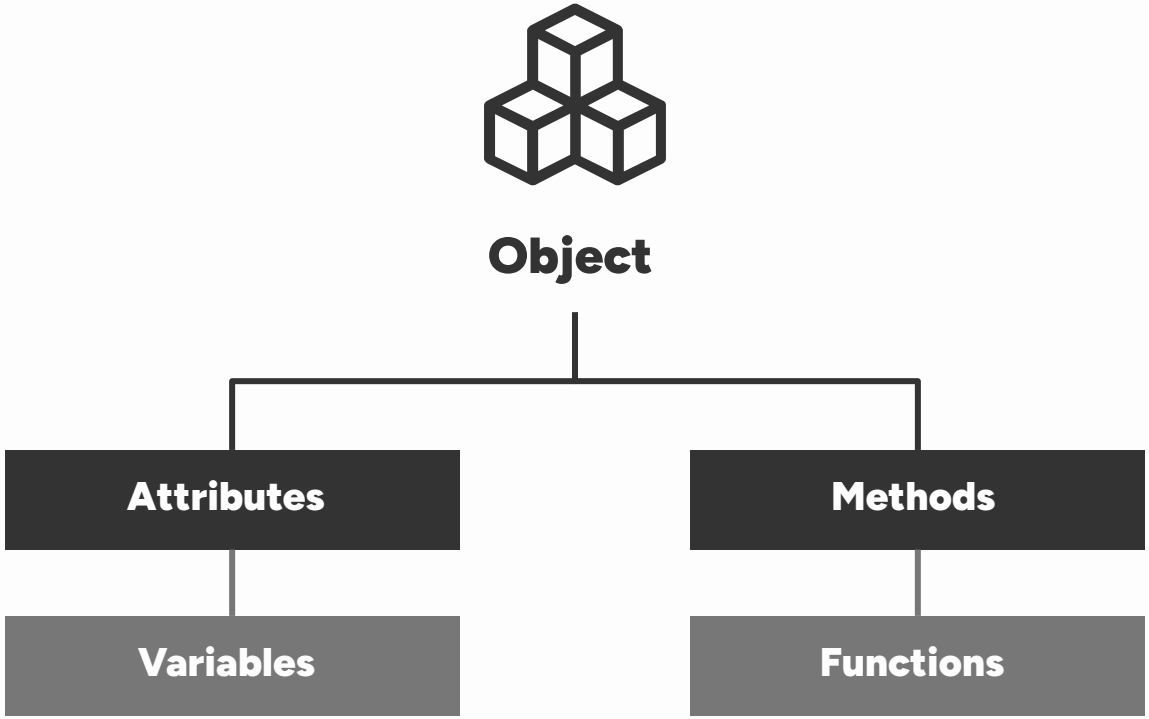


Bank Account



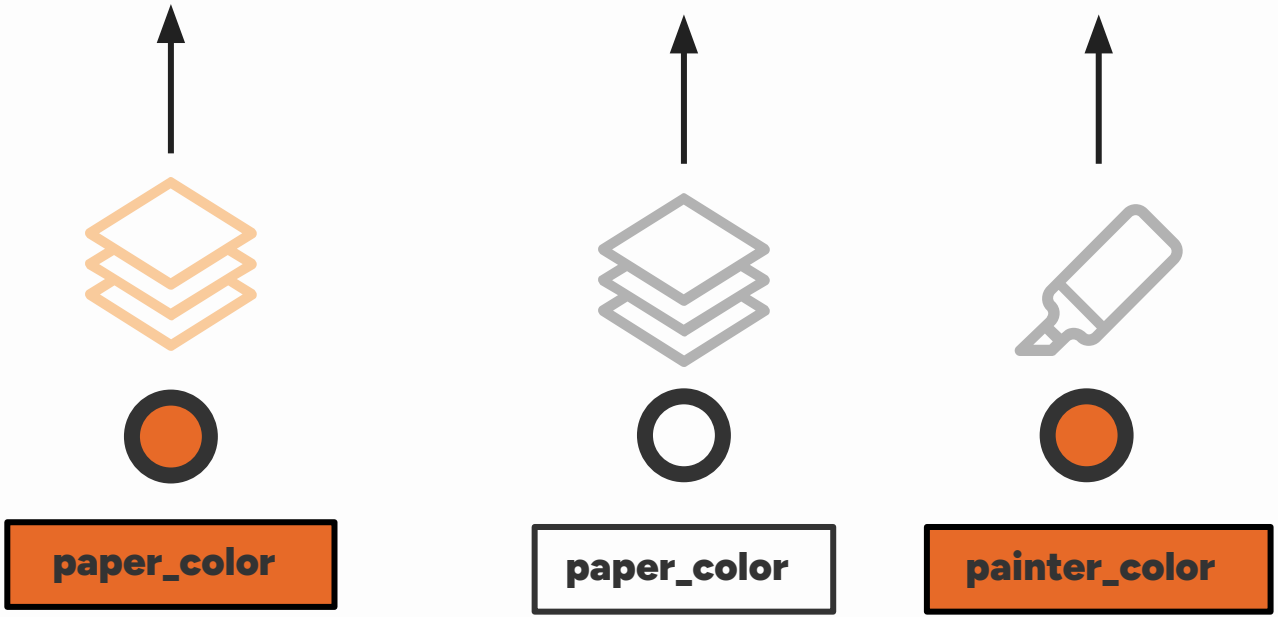
Game Character




Object
Attributes
Methods
Variables
Functions

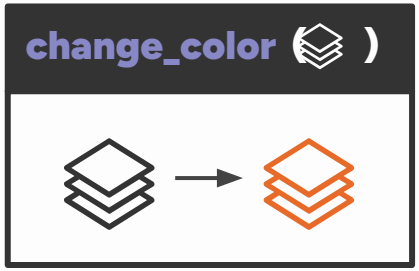
Functional Approach

```
paper_color = change_color(paper_color, marker_color)
```



Object Oriented Approach

```
marker.change_color(paper)
```



Building Exercise

Example Class

```
class Employee:
    """Class representation for employee data"""
```

Object Creation

```
1  class Employee:
2      """Class representation for employee data"""
3
4  employee1 = Employee()
5
6
7
8
9
10
11
12
13
14
15
```

Multiple Object Creation

```
1  class Employee:
2      """Class representation for employee data"""
3
4  employee1 = Employee()
5  employee2 = Employee()
6
7
8
9
10
11
12
13
14
15
```

Class Constructor/Initialization

```
1  class Employee:
2      """Class representation for employee data"""
3
4      def __init__(self, name, id):
5          print(f"Employee {name} assigned ID {id}")
6
7  employee1 = Employee("Richard", "1234")
8  employee2 = Employee("Jelly", "9876")
9
10
11
12
13
14
15
```

Object Attributes

```
1  class Employee:
2      """Class representation for employee data"""
3
4      def __init__(self, name, id):
5          self.name = name
6          self.id = id
7
8          print(f"Employee {self.name} assigned ID {self.id}")
9
10 employee1 = Employee("Richard", "1234")
11 employee2 = Employee("Jelly", "9876")
12
13 print("Employee 1 Name:", employee1.name)
14 print("Employee 2 Name:", employee2.name)
15
```

Object Attributes

self .name

employee1 .name

Object Methods

```
1  class Employee:
2        """Class representation for employee data"""
3
4        def __init__(self, name, id):
5              self.name = name
6              self.id = id
7              self.tasks = []
8
9              print(f"Employee {self.name} assigned ID {self.id}")
10
11       def add_work(self, task):
12             return self.tasks.append(task)
13
14 employee = Employee("Richard", "1234")
15 employee2 = Employee("Jelly", "9876")
16
17 print("Employee 1 Name:", employee1.name)
18 print("Employee 2 Name:", employee2.name)
19
20 employee.add_work("create charts")
21 print(employee.tasks)
```

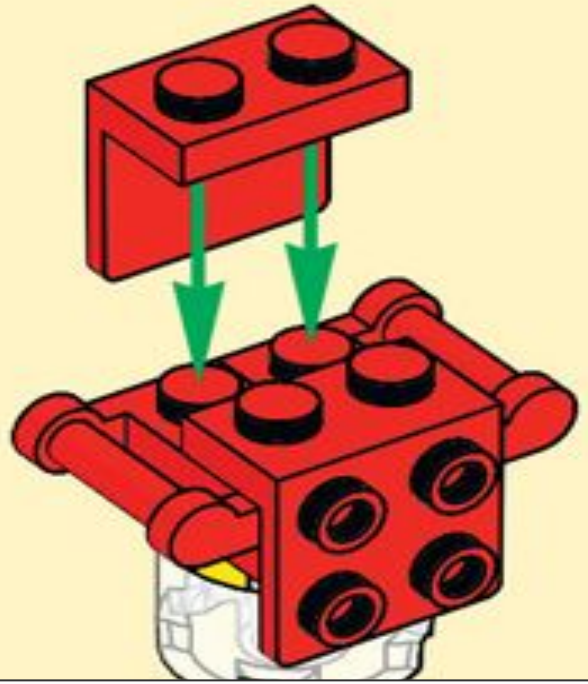
Object Methods

```
employee.add_work(task)
add_work(employee, task)
```

1

H1

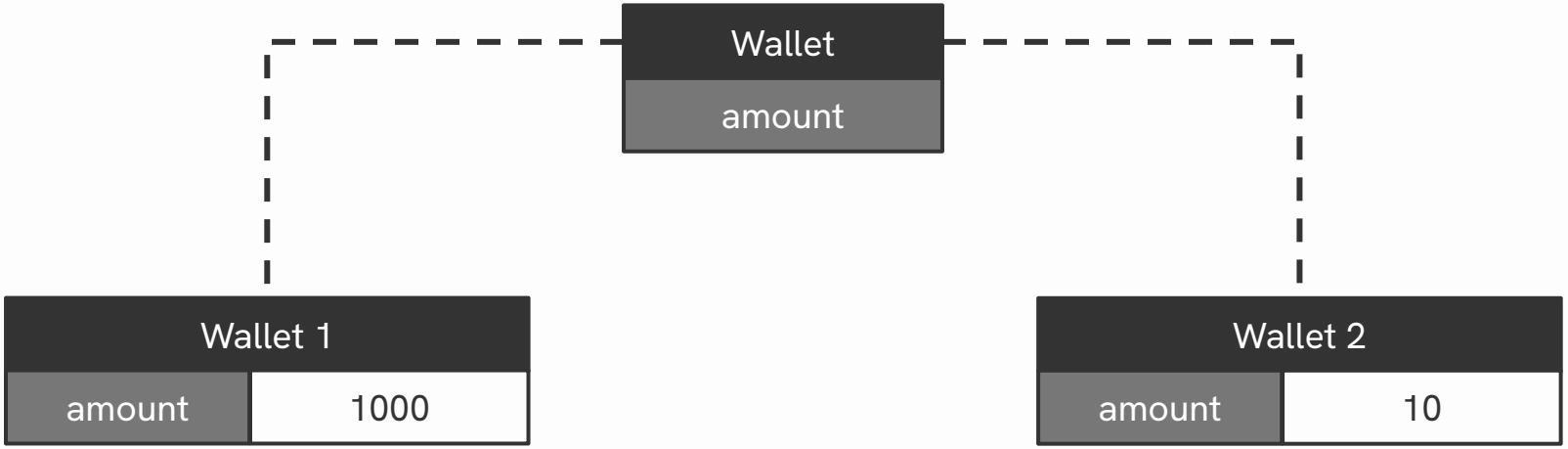
4



Hands-On Building

2

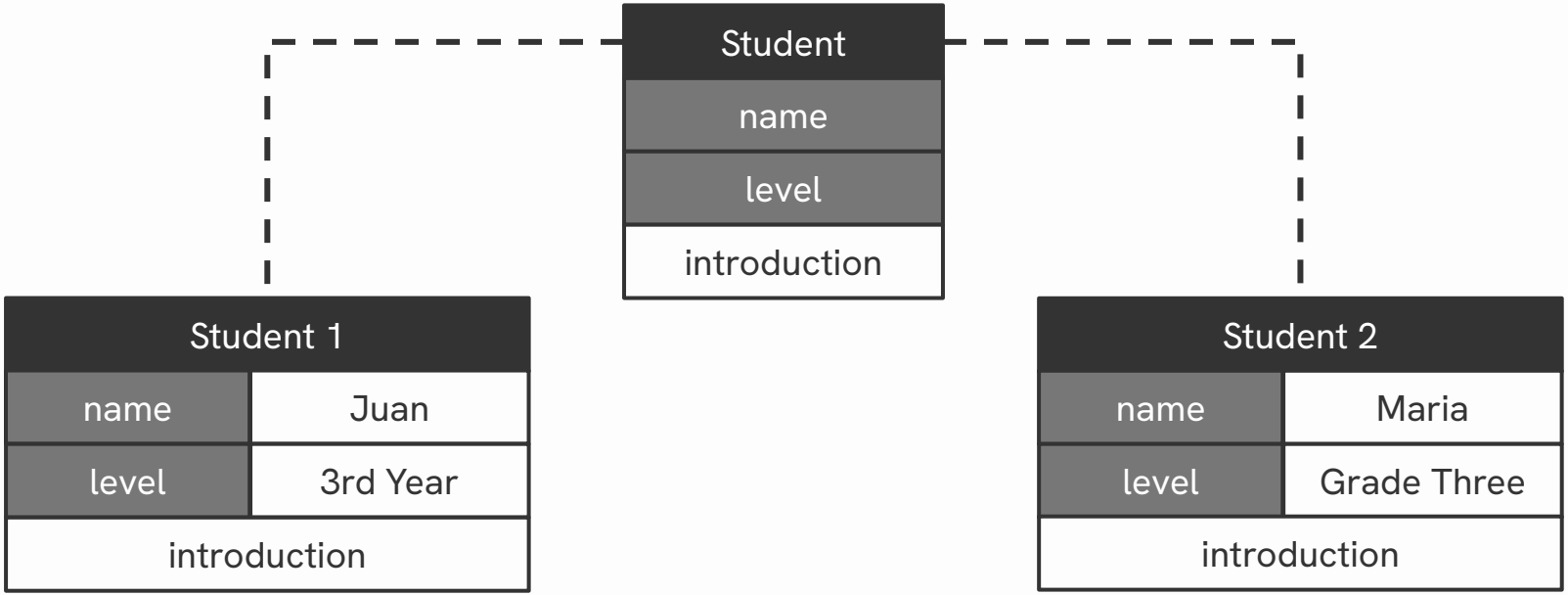
Wallet



Implement : Wallet

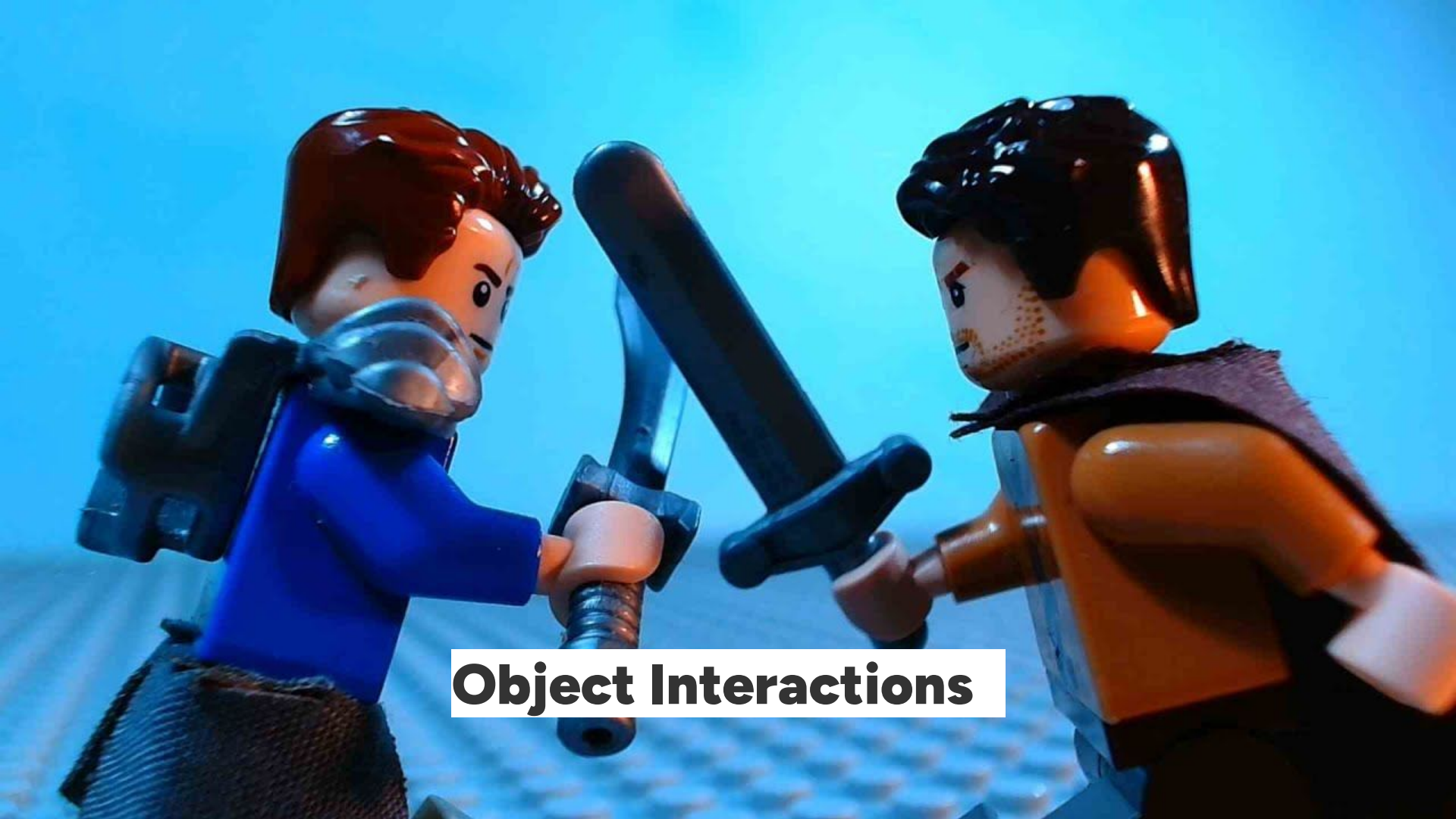
```
1  class Wallet:
2      def __init__(self, initial_amount=0):
3          self.amount = initial_amount
4
5  food_wallet = Wallet(250)
6  transport_wallet = Wallet(1000)
7
8  print("Food Budget:", food_wallet.amount)
9  print("Transport Budget:", transport_wallet.amount)
```

Student

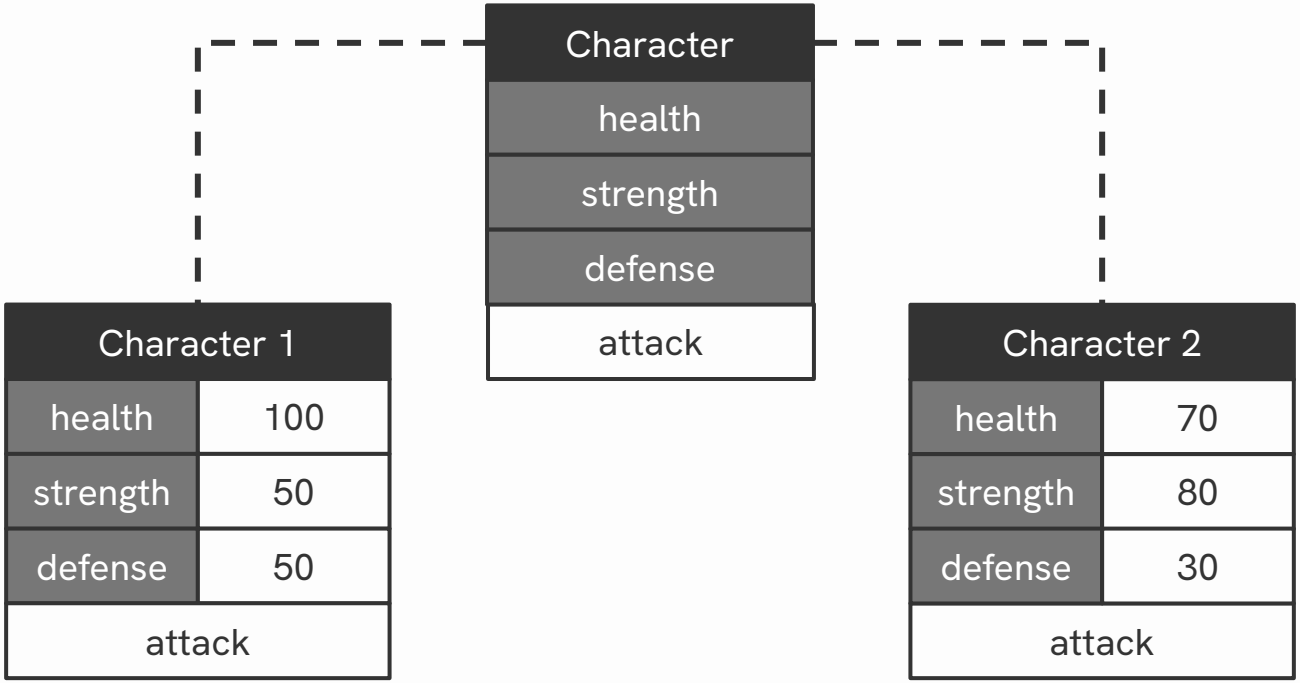

Student
name
level
introduction
Student 1
name
Juan
level
3rd Year
introduction
Student 2
name
Maria
level
Grade Three
introduction

Implement : Student

```
1  class Student:
2      def __init__(self, name, level):
3          self.name = name
4          self.level = level
5
6      def introduction(self):
7          return f"I'm {self.name}! I'm a {self.level} student."
8
9  student1 = Student("Juan", '3rd Year College')
10 print(student1.introduction())
11
12 student2 = Student("Maria", 'Grade Three')
13 print(student2.introduction())
```

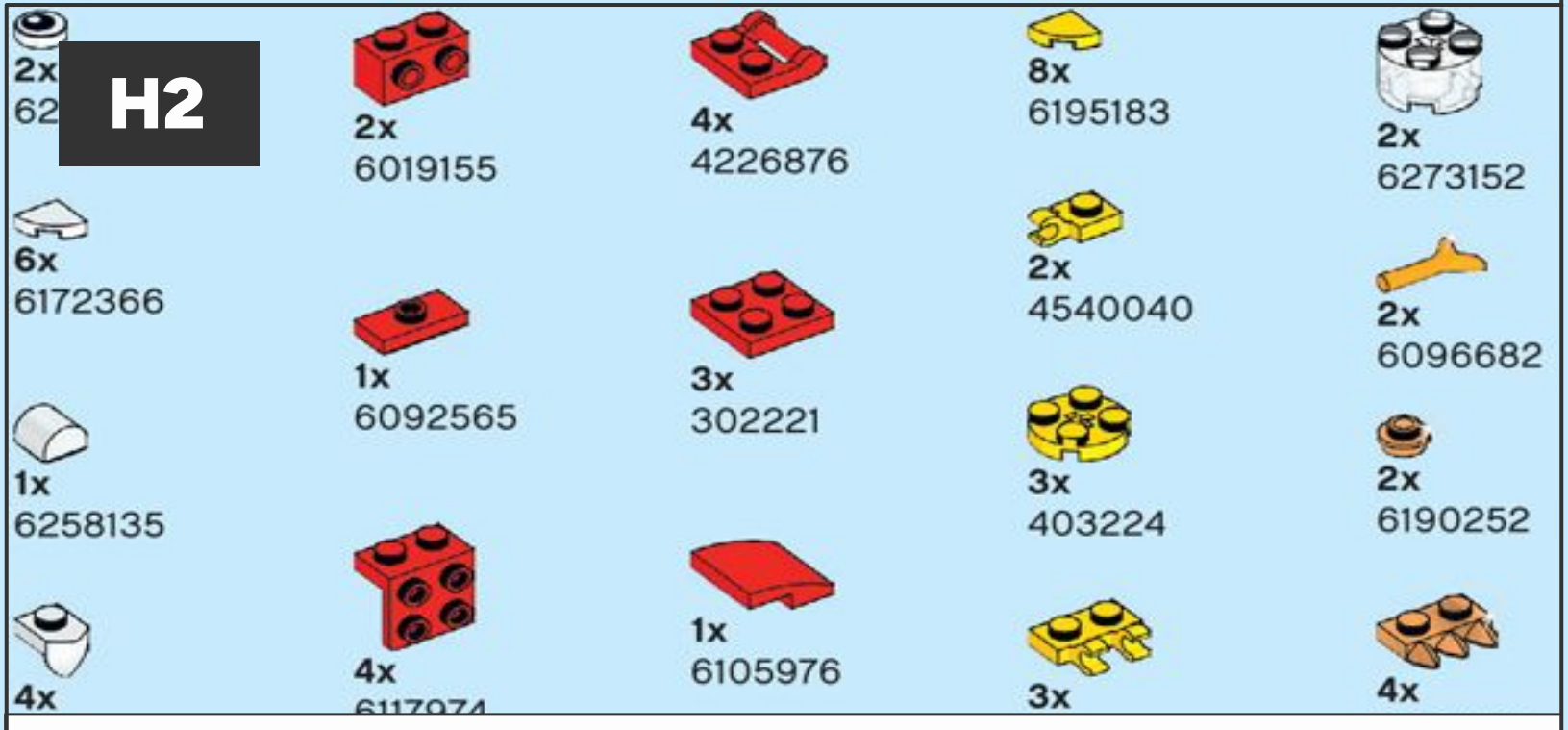

Object Interactions

Game Character



Implement: Character

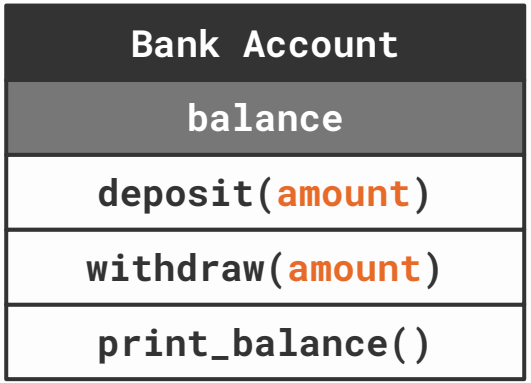
```
1  class Character:
2      def __init__(self, health=10, strength=10, defense=10):
3          self.health = health
4          self.strength = strength
5          self.defense = defense
6
7      def attack(self, other):
8          damage = self.strength - other.defense
9          other.health -= damage
10
11 player = Character(strength=100)
12 enemy = Character()
13
14 player.attack(enemy)
15 print(enemy.health)
```



H2

Hands-Off Building

Implement: Bank Account



Implement: Pen and Paper

Pen
brand
color
capped
ink_level
cap()
uncap()
write(paper, text)
refill(amount)

Paper
contents

02

Hierarchy

Reducing redundancy in classes

Code Redundancy

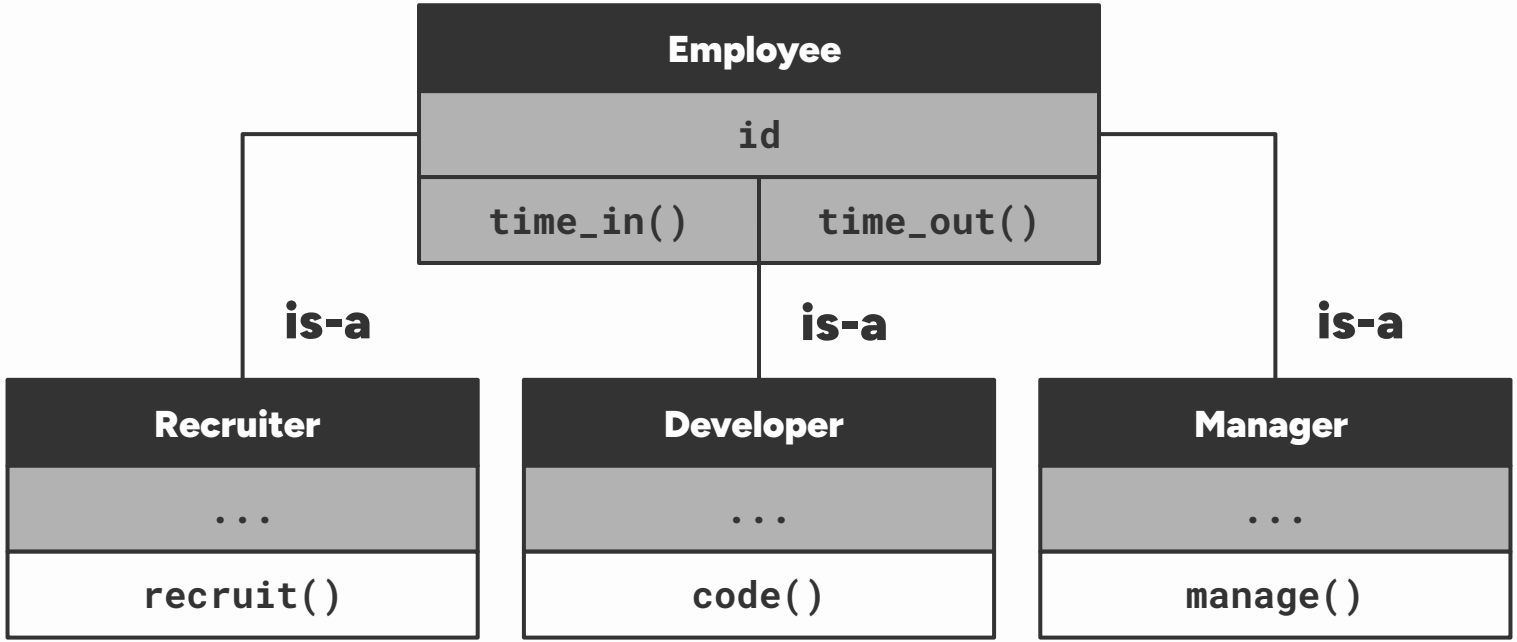
```
class Recruiter:
    def __init__(self, id)
    def time_in(self)
    def time_out(self)

    def recruit(self)
```

```
class Designer:
    def __init__(self, id)
    def time_in(self)
    def time_out(self)

    def design(self)
```

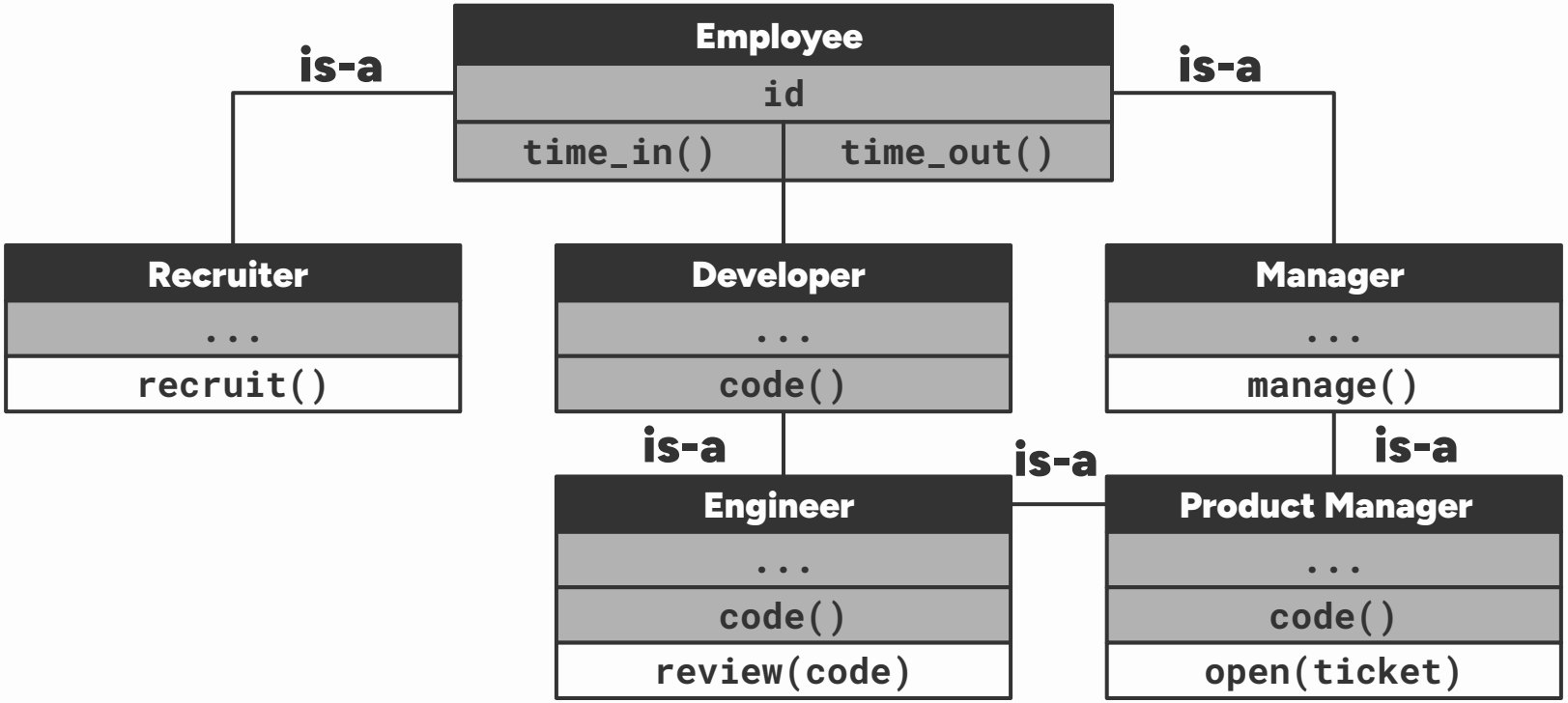
```
class Developer:
    def __init__(self, id)
    def time_in(self)
    def time_out(self)

    def code(self)
```

```
class Manager:
    def __init__(self, id)
    def time_in(self)
    def time_out(self)

    def manage(self)
```

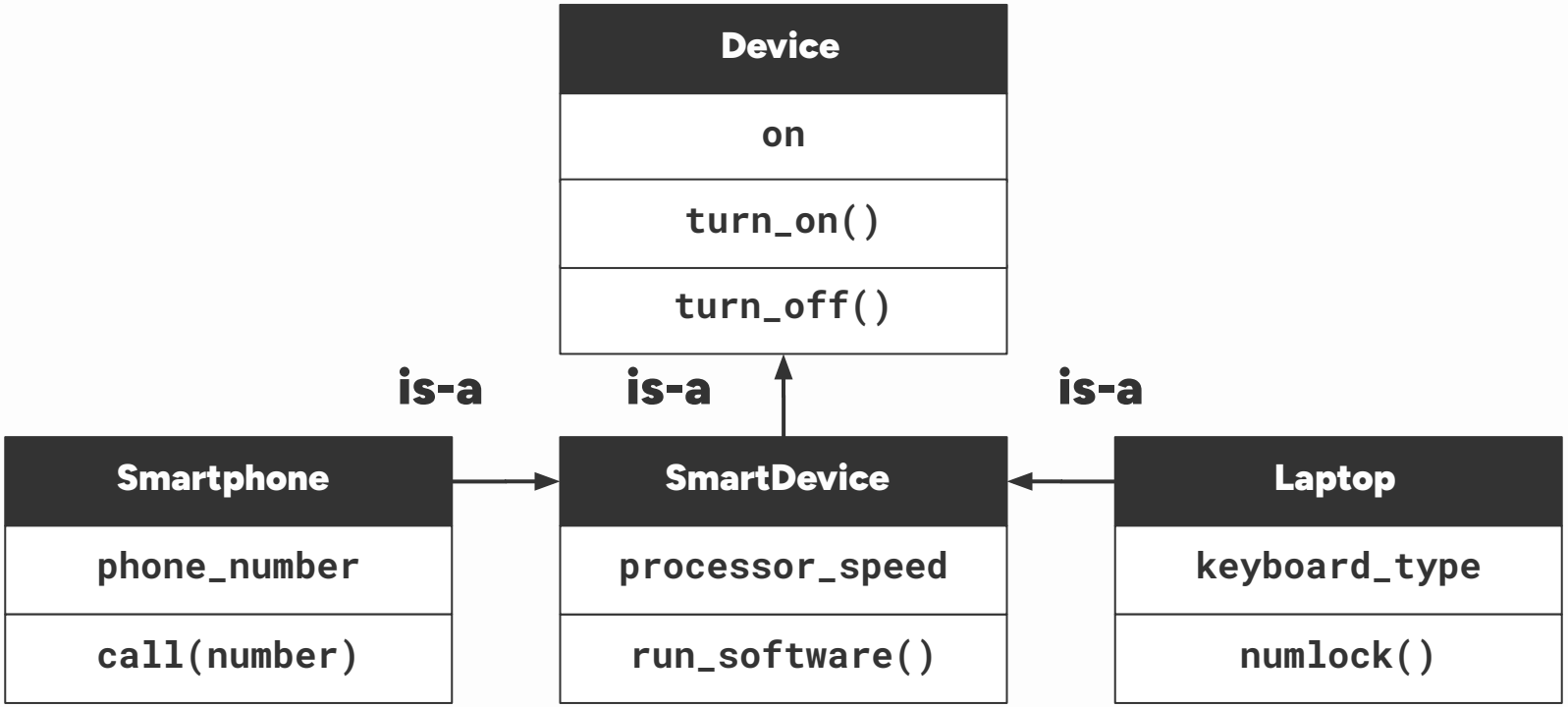
Hierarchy


Employee
id
time_in()
time_out()
is-a
is-a
is-a
Recruiter
...
recruit()
Developer
...
code()
Manager
...
manage()

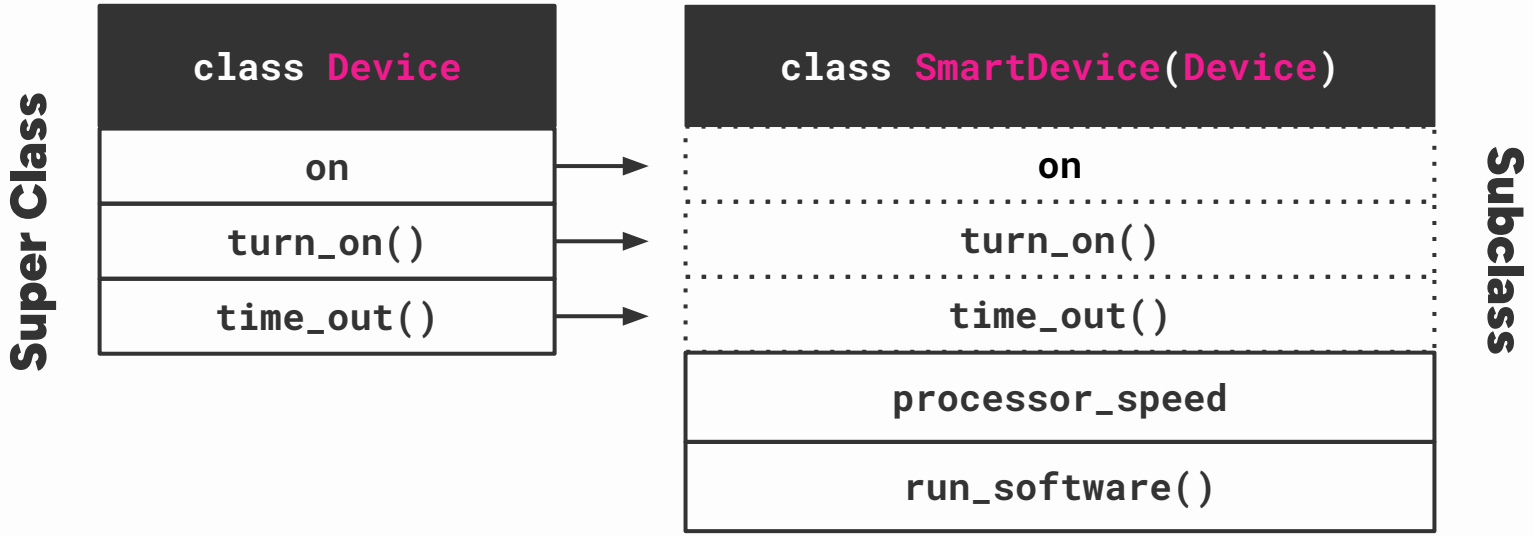
Hierarchy (Complex)



Hierarchy


Device
on
turn_on()
turn_off()
is-a
is-a
is-a
Smartphone
phone_number
call(number)
SmartDevice
processor_speed
run_software()
Laptop
keyboard_type
numlock()

Class Inheritance

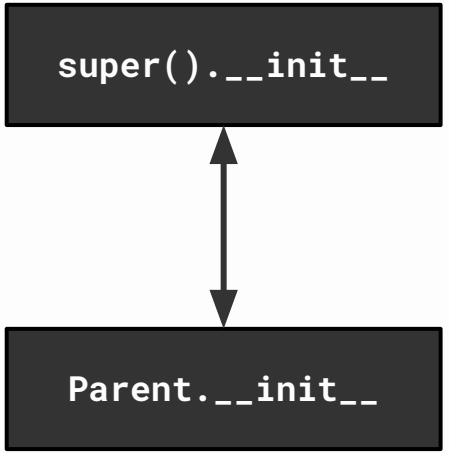


Inheritance Format

```
class Person:
     def __init__(self, first_name, last_name):
          self.first_name = first_name
          self.last_name = last_name
```

```
class Student(Person):
     def __init__(self, first_name, last_name, level):
          super().__init__(first_name, last_name)
          self.level = level
```

```
class Student(Parent):
     def __init__(self, likes, toys):
          self.first_name = first_name
          self.last_name = last_name
          self.level = level
```



Implement: Person-Student Class

```
class Person:
     def __init__(self, first_name, last_name):
          self.first_name = first_name
          self.last_name = last_name

     def introduction(self):
          return f"I'm {self.first_name} {self.last_name}!"

     def sleep(self):
          print("I will sleep for eight hours")

class Student(Person):
     def __init__(self, first_name, last_name, level):
          super().__init__(first_name, last_name)
          self.level = level

     def introduction(self):
          return super().introduction() + f" I'm a {self.level} student."
```

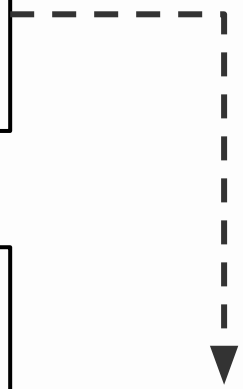
Silent Inheritance

```
class Person:
     def sleep(self):
          print("I will sleep for eight hours")
```

```
class Student(Parent):
     def sleep(self):
          print("I will sleep for eight hours")
```

Overriding

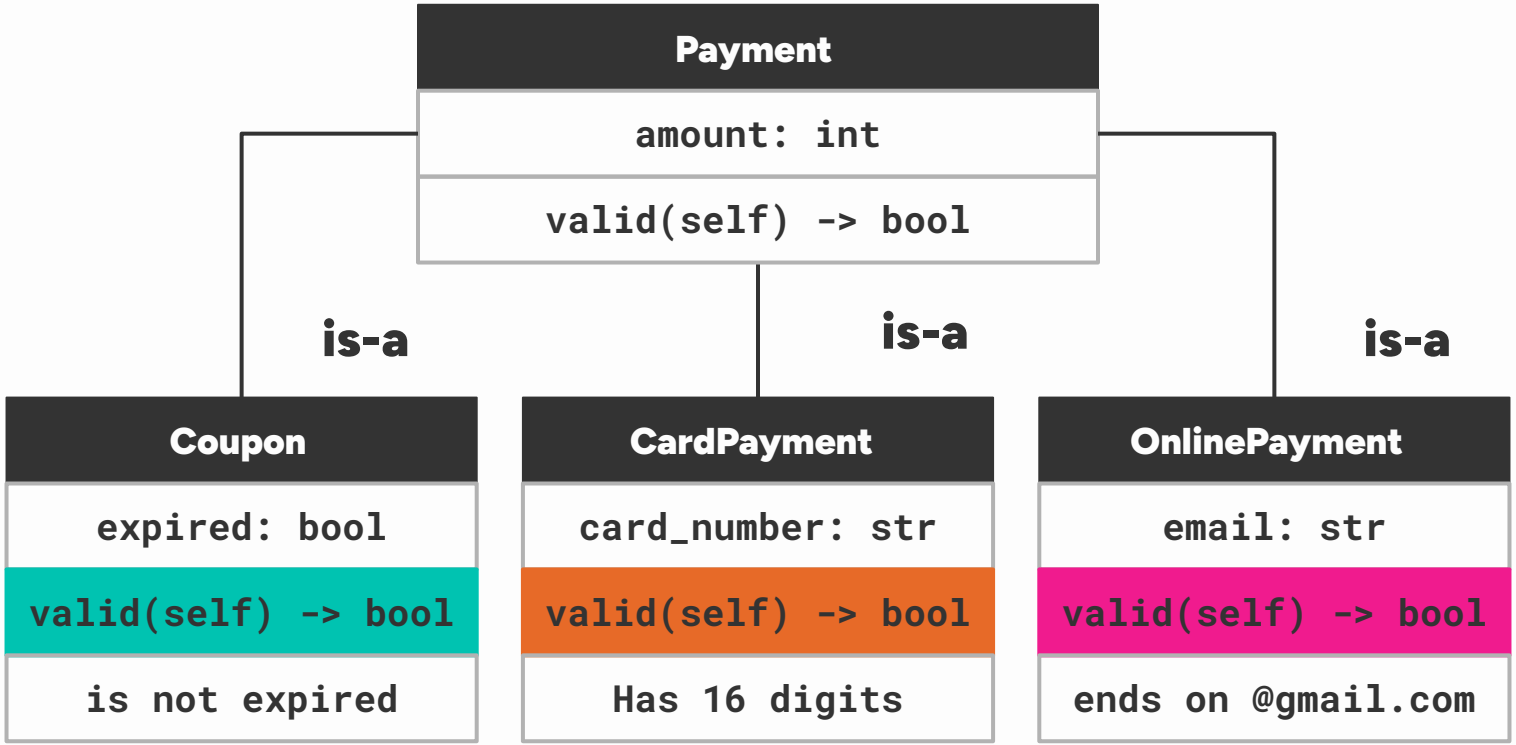
```
class Person:
     def sleep(self):
          print("I will sleep for eight hours")
```



```
class Child(Parent):
     def sleep(self):
          print("I will sleep for six hours")
```


H3
Payment Time
Yes, it’s time to pay

Define the Following System



03

Polymorphism

Using different classes in the same context

Type Checking

You can directly check the type of a class using the `isinstance` function

```
1  class Parent:
2      pass
3
4  class Child(Parent):
5      pass
6
7  child = Child()
8  print(isinstance(child, Parent))
```

```
True
```

Multiple Type Checking

The `isinstance` function can be used to check for multiple types at once

```
1  class Unrelated:
2      pass
3
4  class Parent:
5      pass
6
7  class Child(Parent):
8      pass
9
10 child = Child()
11 type_check = isinstance(child,(Unrelated, Parent))
12 print(type_check)
```

Use Case: Input Checking

Type checking with `isinstance` is often done to apply different logic depending on the input

```
1  def add(x, y):
2      if isinstance(x,(list, int)) and isinstance(y,(list, int)):
3          return x + y
4      elif isinstance(x,(set, dict)) and isinstance(y,(set, dict)):
5          return x | y
6      else:
7          raise NotImplementedError()
```

Quick Exercise: General Add

Add a case for the following conditions

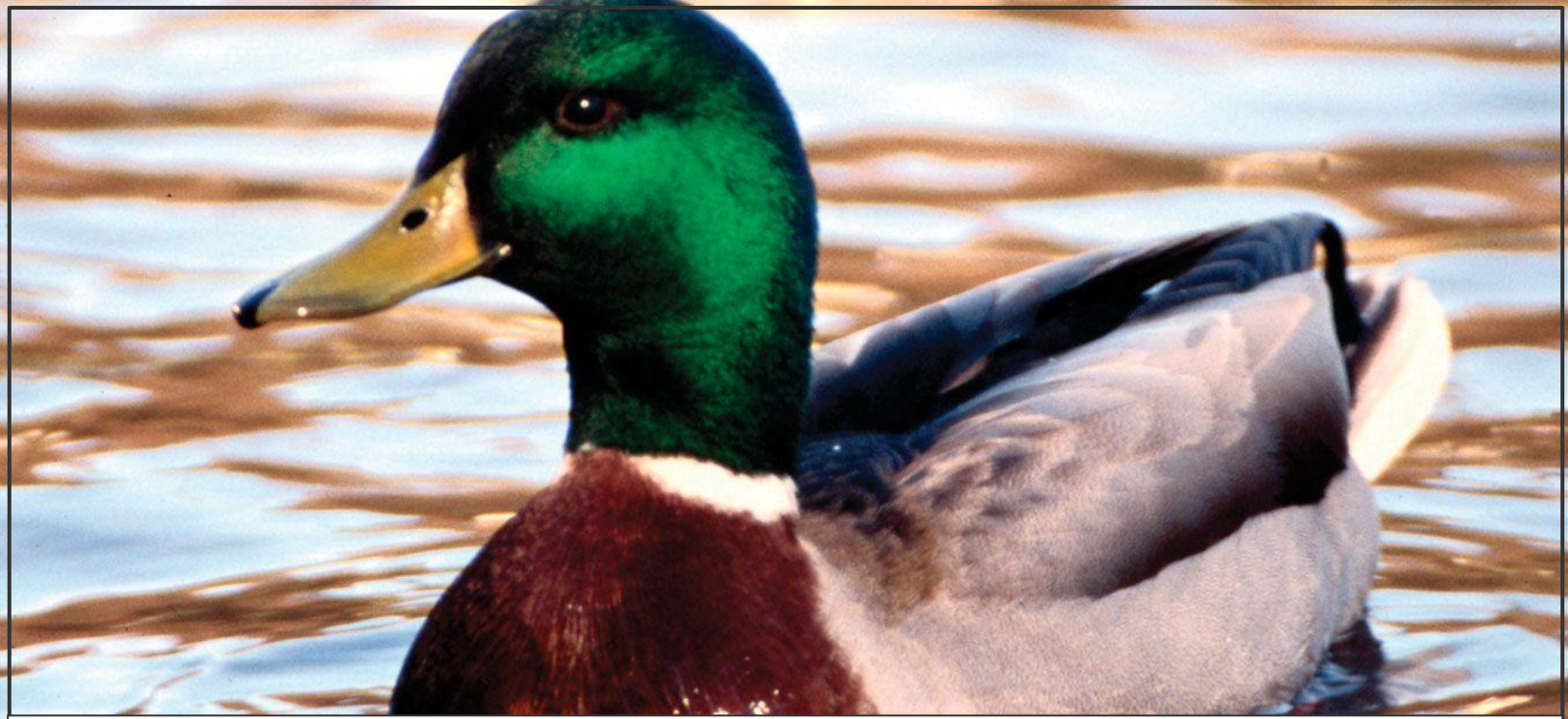
- List and non-list (and vice versa)
- Dict and non-dict (and vice versa)
- Set and non-set (and vice versa)

```
1  def add(x, y):
2      if isinstance(x,(list, int)) and isinstance(y,(list, int)):
3          return first + second
4      elif isinstance(x,(set, dict)) and isinstance(y,(set, dict)):
5          return first | second
6      else:
7          raise NotImplementedError()
```

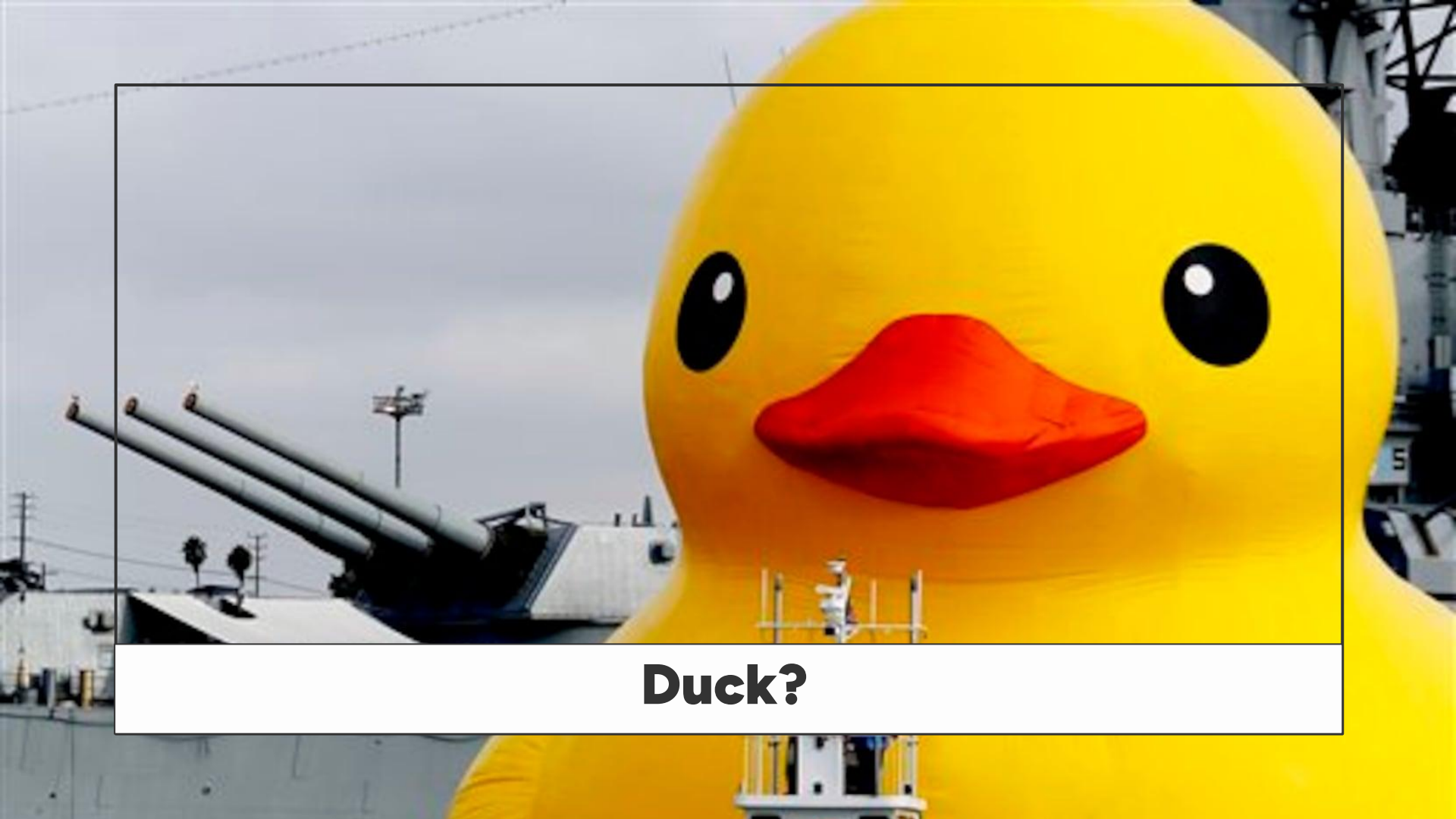
""If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.""

—Duck Typing

Has → Is



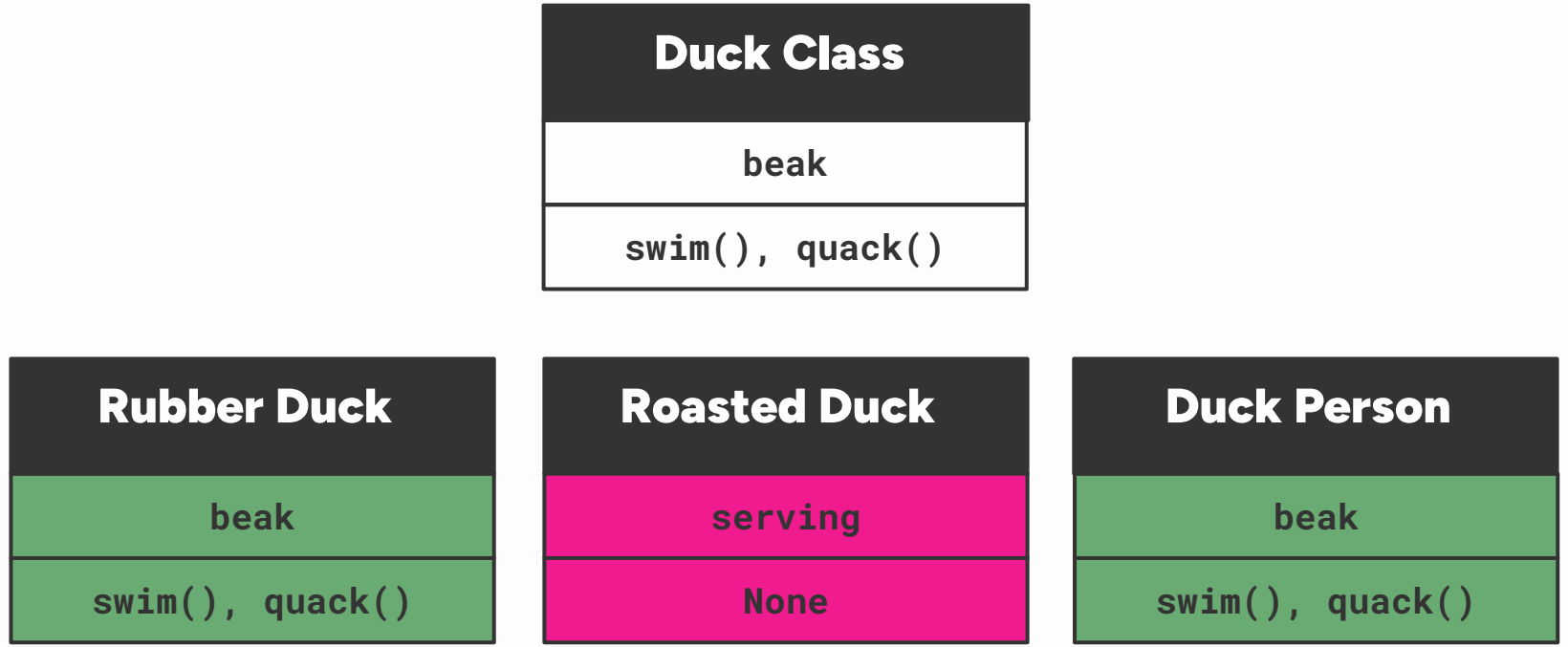
Duck?


Duck?

Duck?

Duck?

Duck Typing


Duck Class
beak
swim(), quack()
Rubber Duck
beak
swim(), quack()
Roasted Duck
serving
None
Duck Person
beak
swim(), quack()

Implement: "Duck" Classes

```
class Duck:
    def __init__(self, beak):
        self.beak = beak
    def swim(self):
        print("Swimming")
    def quack(self):
        print("Quack")
```

```
class RubberDuck:
    def __init__(self, beak):
        self.beak = beak
    def swim(self):
        print("Splish Splosh")
    def quack(self):
        print("Squeak Quack")
```

```
class DuckPerson:
    def __init__(self, beak):
        self.beak = beak
    def swim(self):
        print("Swim hehe!")
    def quack(self):
        print("Quack hehe")
```

```
class RoastedDuck:
    def __init__(self, serving):
        self.serving = serving
```

Attribute Checking

Similar to `isinstance` the function `hasattr` can be used to check if an instance has an **attribute** or **method** with the **same name**

```
1  dish = RoastedDuck(1)
2  if hasattr(dish, 'beak') and hasattr(dish, 'quack'):
3      dish.quack()
4  else:
5      print("Dinner is served")
```

```
Dinner is served
```

Attribute Checking

Similar to `isinstance` the function `hasattr` can be used to check if an instance has an **attribute** or **method** with the **same name**

```
1  duck = DuckPerson("plastic")
2  if hasattr(dish, 'beak') and hasattr(dish, 'quack'):
3      dish.quack()
4  else:
5      print("Dinner is served")
```

```
Quack hehe
```

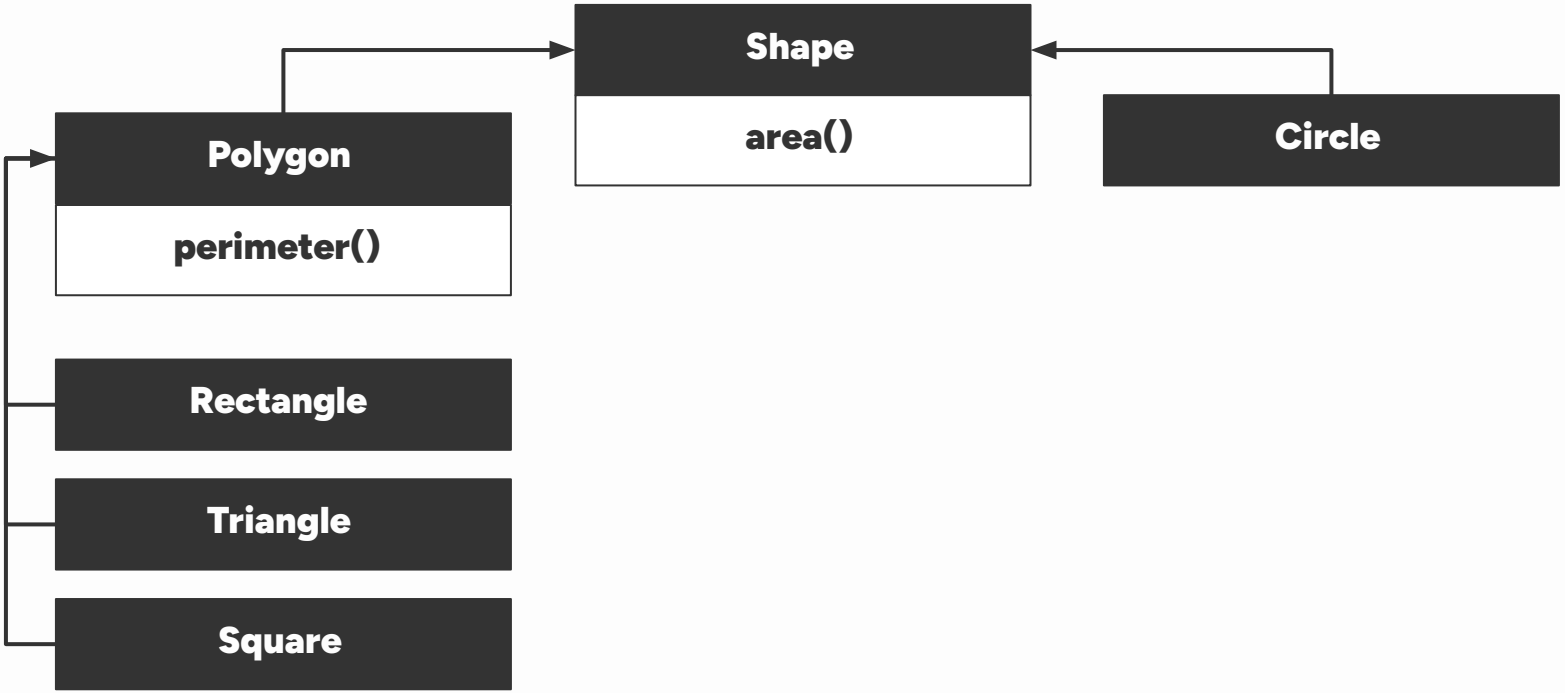
Implement: Knight

```
1  class Knight:
2      def __init__(self, health=10, defense=10):
3          self.health = 10
4          self.defense = defense
5
6      def attack(self, other):
7          damage = self.defense - other.defense
8          other.health -= damage
9
10 player = Knight(defense=20)
11 enemy = Character()
12 if hasattr(player,"attack"):
13     player.attack(enemy)
14     print(enemy.health)
```

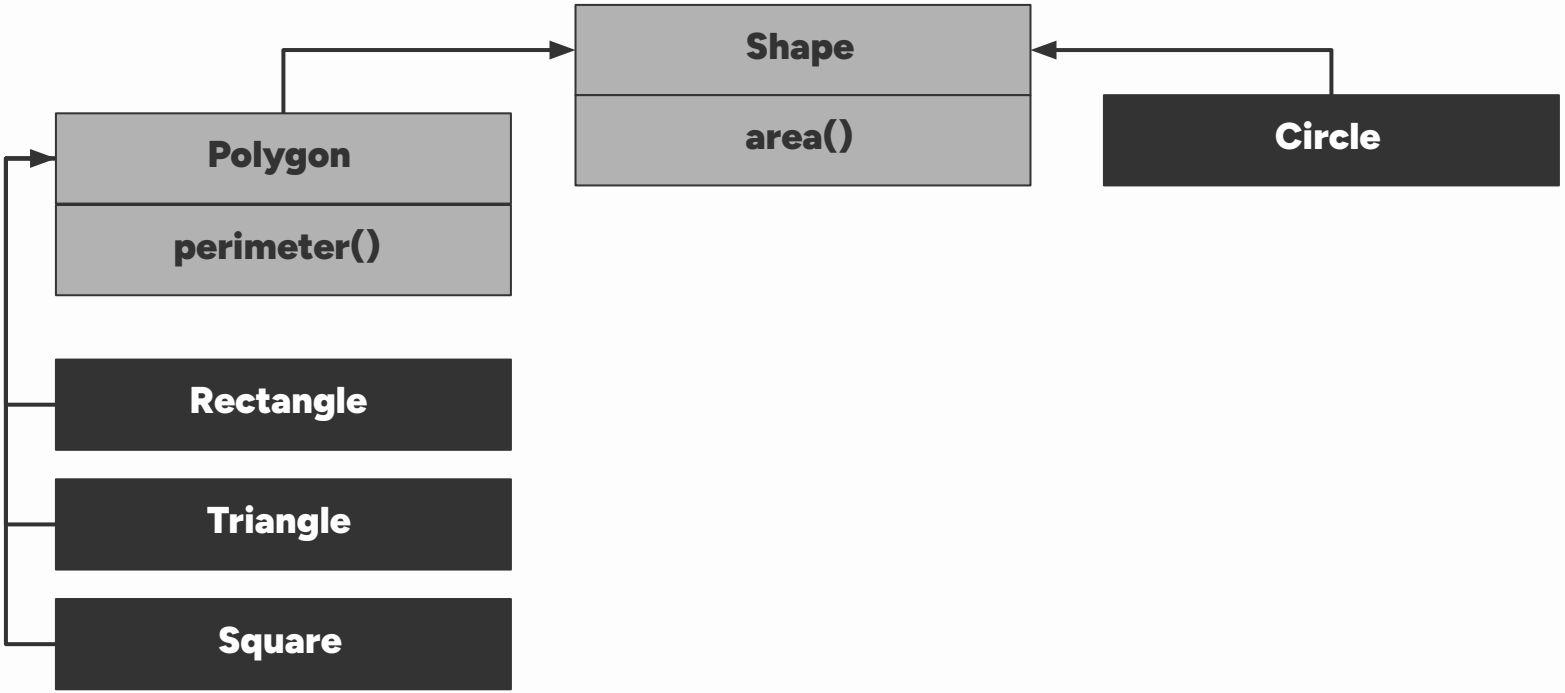
Abstraction

Contractual Implementation

Shape Hierarchy



Shape Hierarchy



Method Implementation

```
1  class Shape:
2      def area(self):
3          pass
```

How do we define the area of a "Shape?"

Method Implementation

```
1  class Circle:
2      def area(self):
3          return 3.14 * self.radius
```

```
1  class Square:
2      def area(self):
3          return self.side_length * self.side_length
```

The area of a "Shape" is defined by its more specific implementation

Abstract Classes

```
1  from abc import ABC, abstractmethod
2
3  class Shape(ABC):
4
5      @abstractmethod
6      def area(self):
7          pass
```

```
8  class Circle(Shape):
9      pass
```

We are making a list of requirements or contract

Extended Abstract

```
1  from abc import ABC, abstractmethod
2
3  class Shape(ABC):
4
5      @abstractmethod
6      def area(self):
7          pass
```

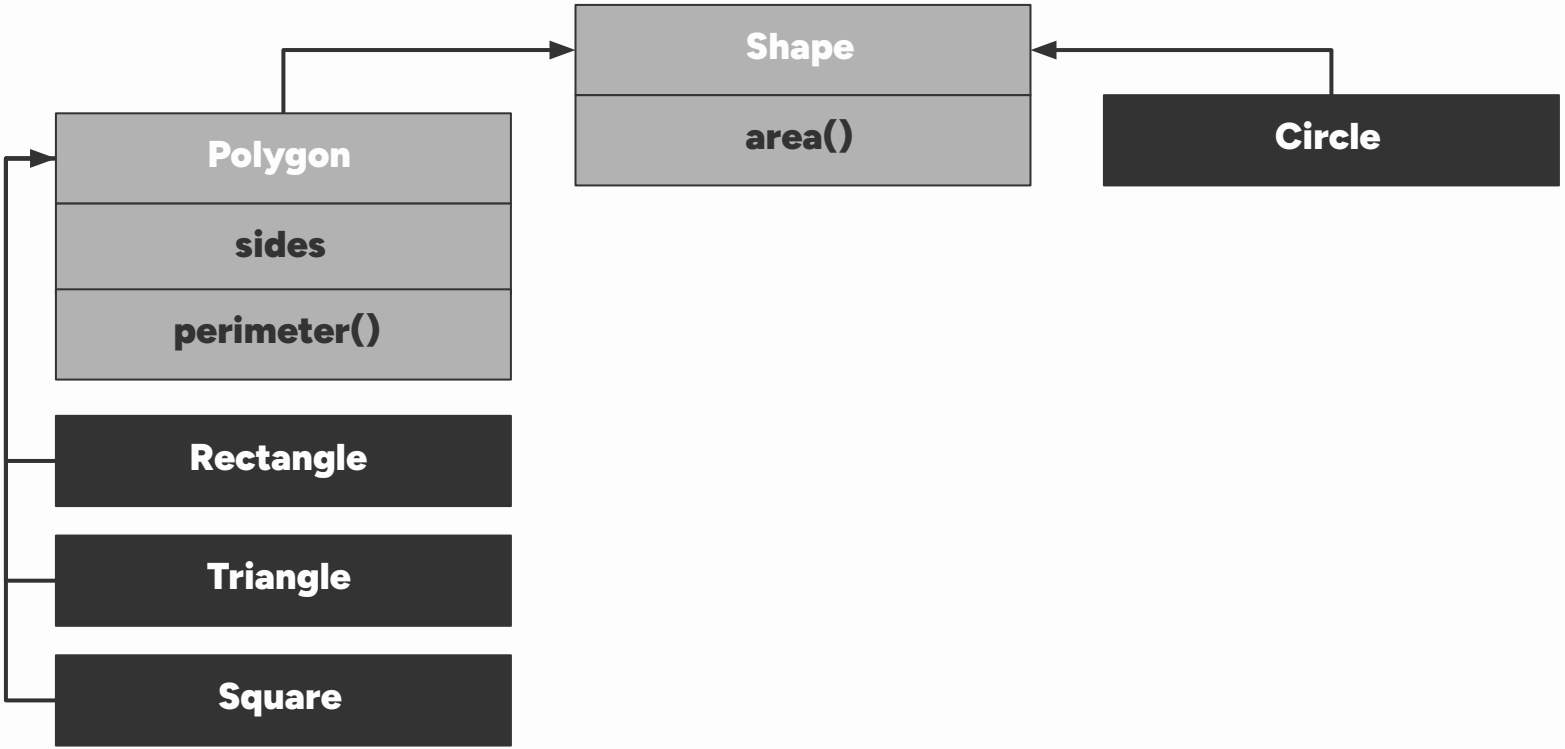
```
9  class Polygon(Shape):
10
11     @abstractmethod
12     def perimeter(self):
13         pass
```

Concrete Child

```
17 class Rectangle(Polygon):
18     def __init__(self, width, height):
19         super().__init__(4)
20         self.width = width
21         self.height = height
22 
23     def perimeter(self):
24         return (self.width * 2) + (self.height * 2)
25 
26     def area(self):
27         return self.width * self.height
```


H5
Shape of You

Implement: Shape Hierarchy



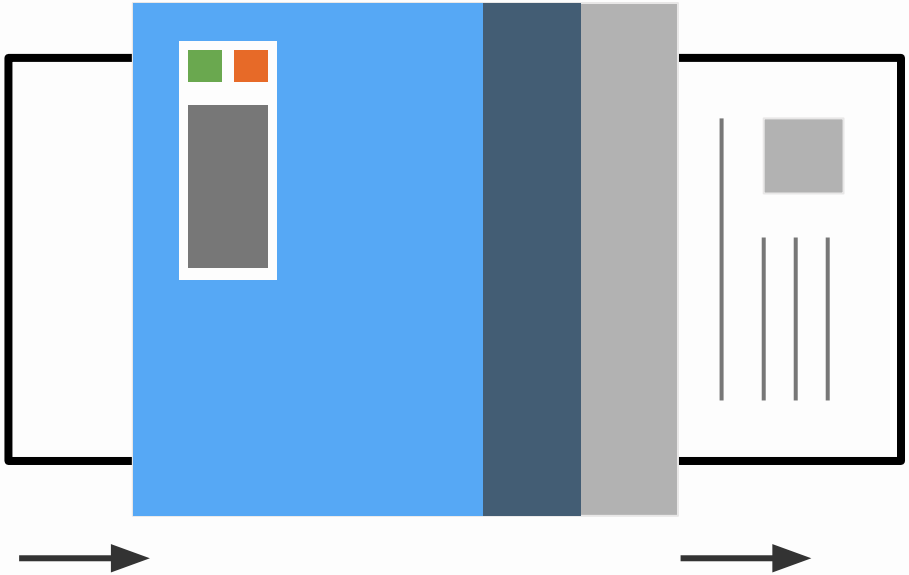
04

Encapsulation

Manage which parts are accessible to the public

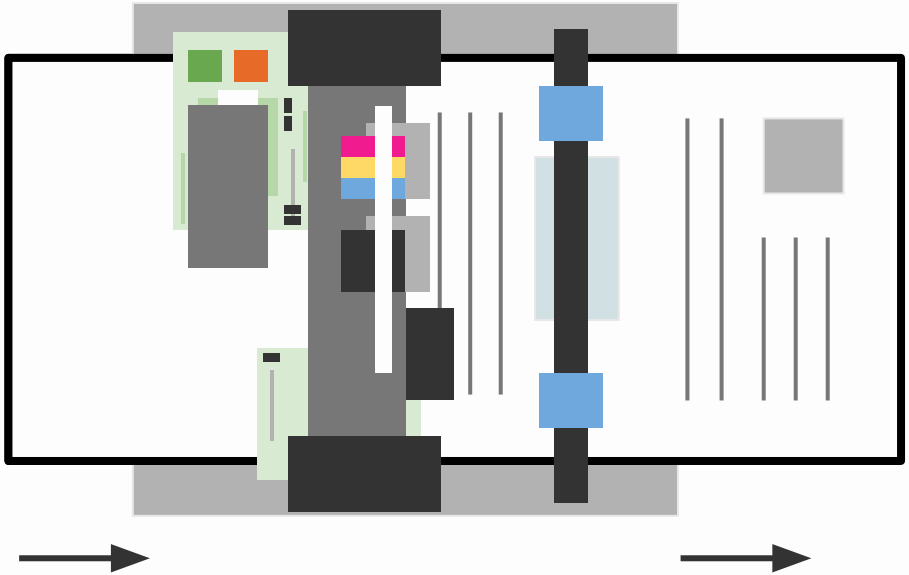
Strategic Data Hiding

Manage which variables are accessible to the public



Strategic Data Hiding

Manage which variables are accessible to the public



Why not show the parts of a printer?

Reasons to Encapsulate

Code Security

Prevent unauthorized read or write operations to sensitive data and processes within the code

Simplification

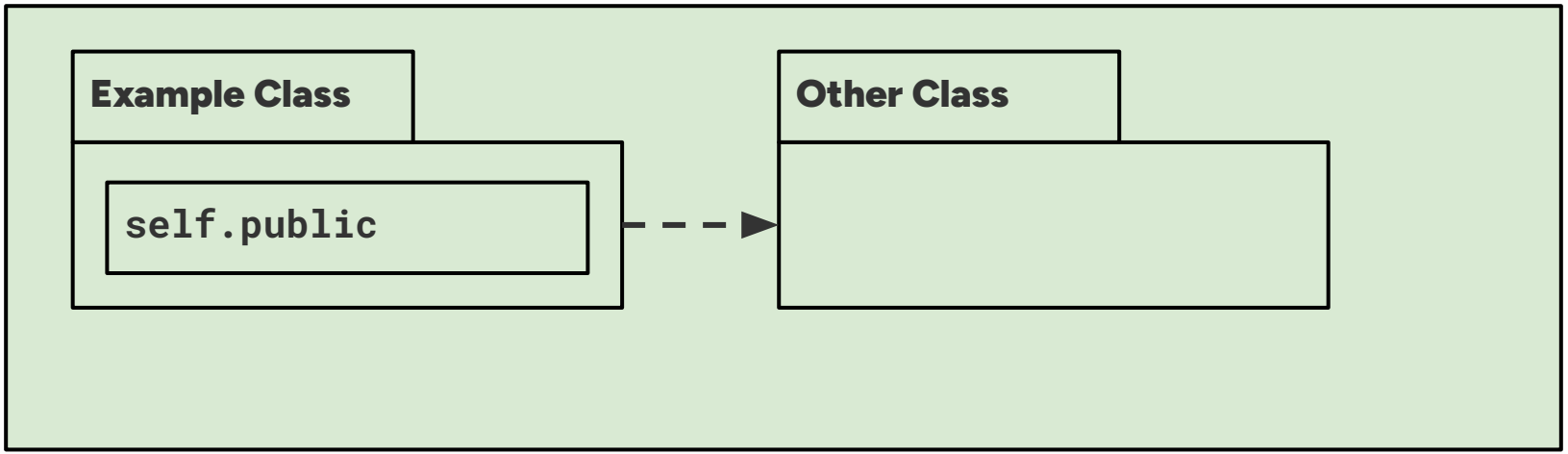
Not every detail of a process needs to be known. Classes can set up their own logic to handle changes

Maintainability

Less access to data means less suspects when debugging problems or issues when developing

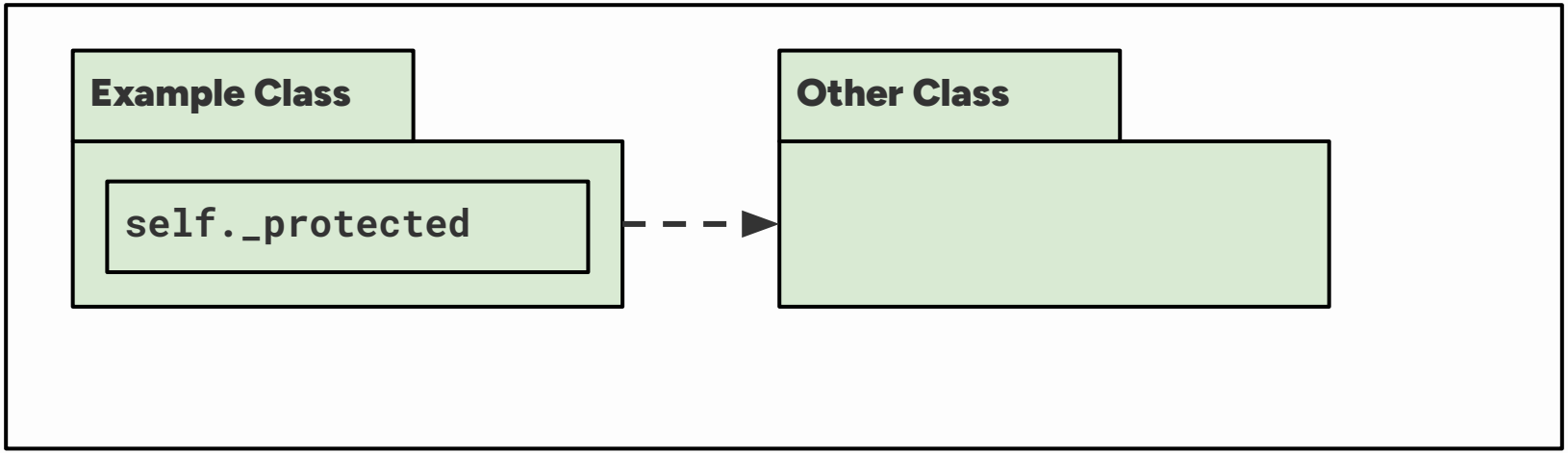
Public Attribute

```
1  class Example:
2      def __init__(self):
3          self.public = 1
```



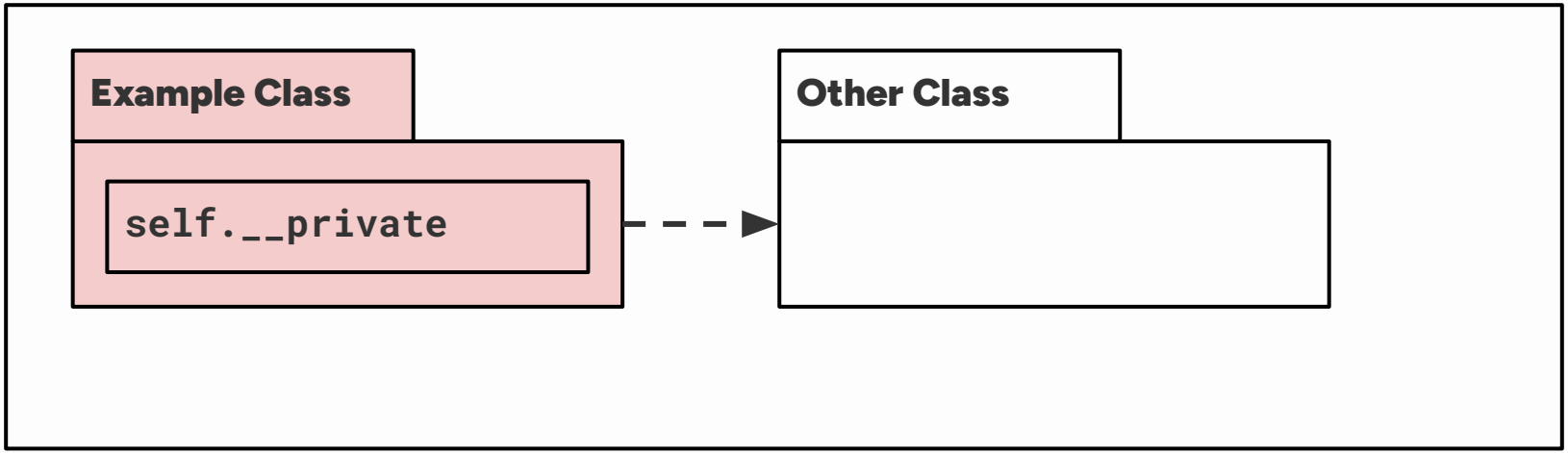
Protected Attribute

```
1 class Example:
2     def __init__(self):
3         self._protected = 2
```



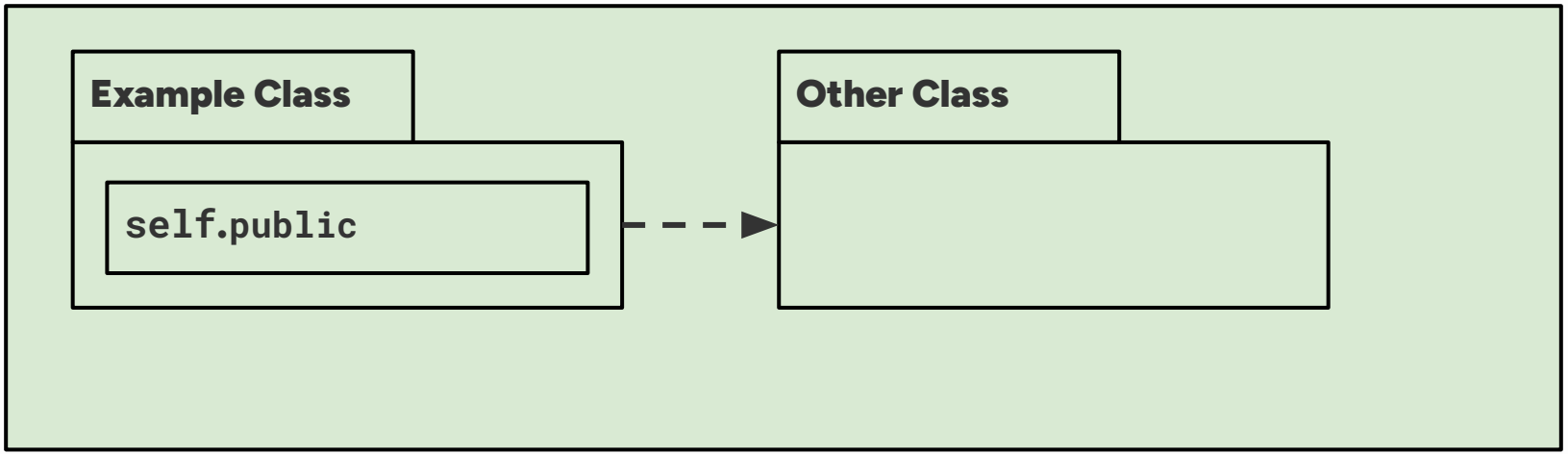
Forced Private Attribute

```
1  class Example:
2      def __init__(self):
3          self.__private = 3
```



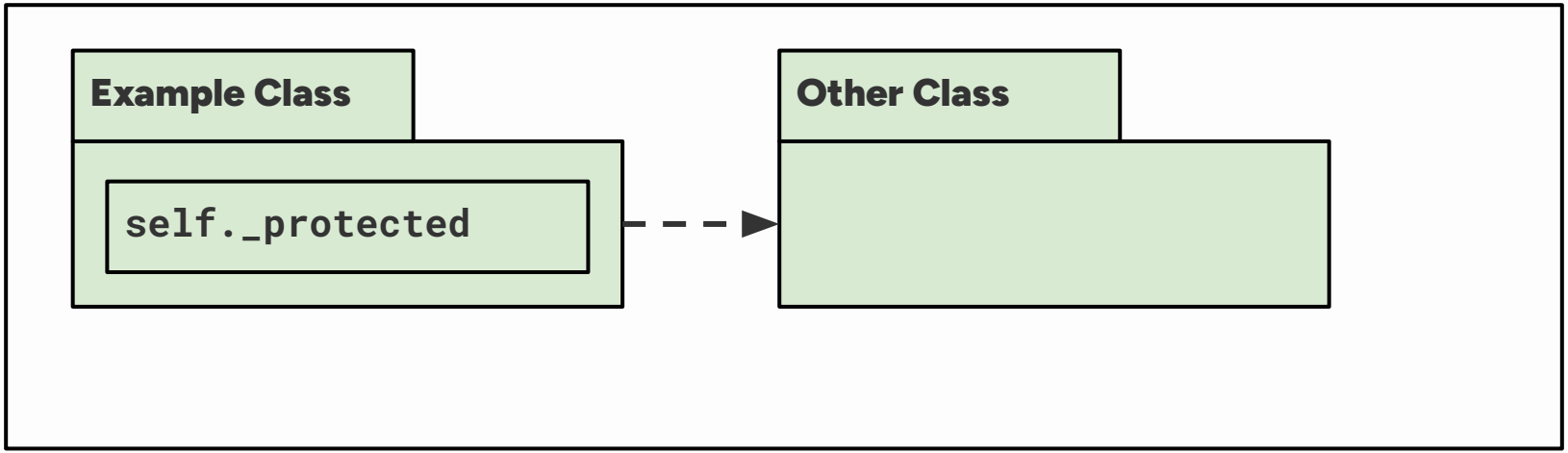
Public Method

```
1 class Example:
2     def public(self):
3         print(1)
```



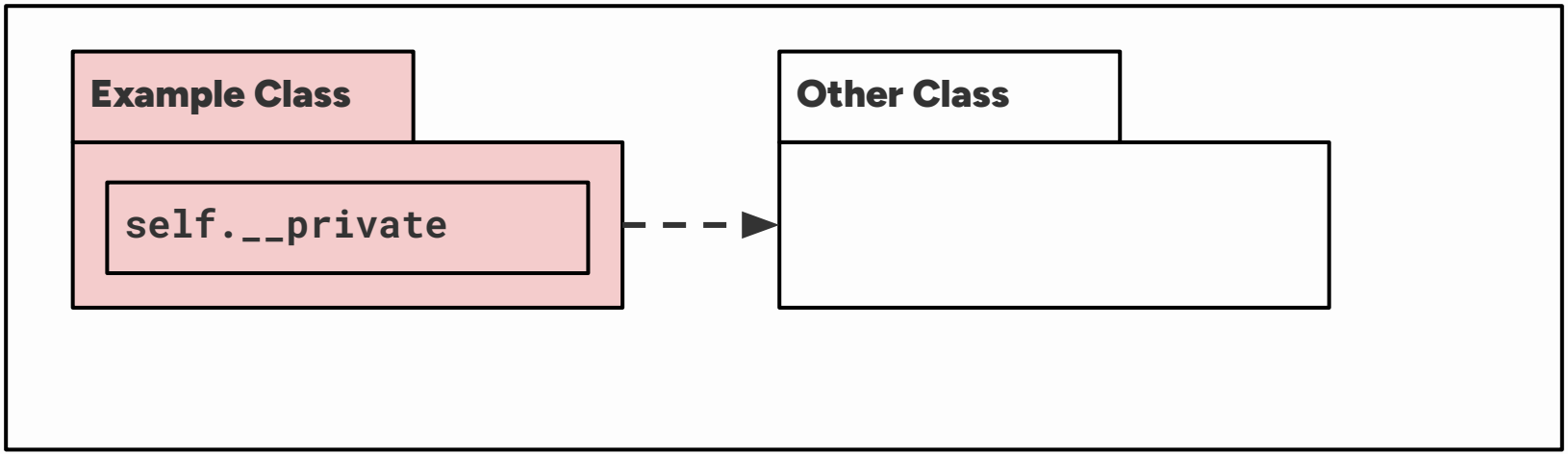
Protected Method

```
1 class Example:
2     def _protected(self):
3         print(2)
```

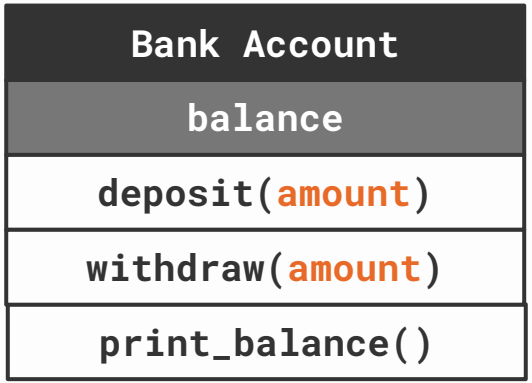


Forced Private Attribute

```
1  class Example:
2      def __private(self):
3          print(3)
```



Refactor: Prevent unnecessary changes



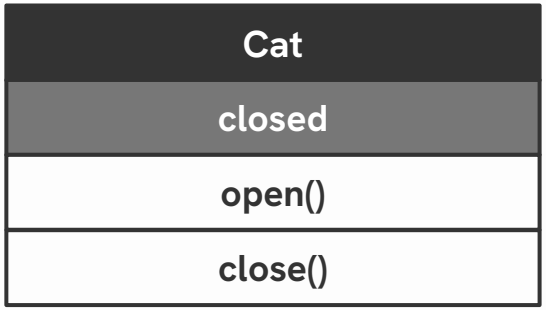
Attribute Getter and Setter

```
class Example:

    def __init__(self, value):
        self._value = value
        self.read = True
        self.write = True

    def get_value(self):
        if self.read:
            return self._value

    def set_value(self, new):
        if self.write:
            self._value = new
```

H6

Close the Door

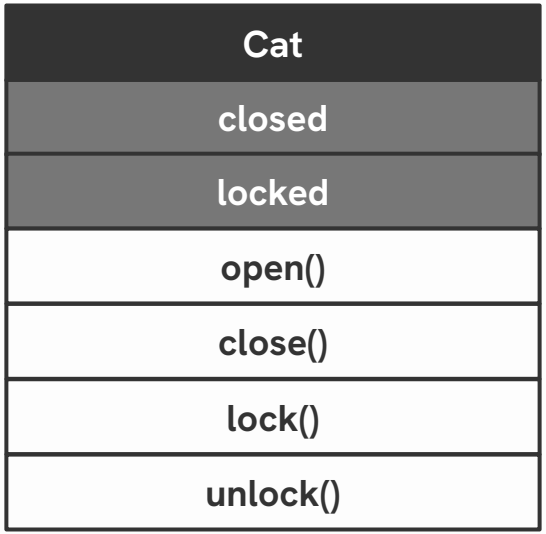
Implement: Door



H7

Lock the Door

Implement: Locked Door



Custom Exception

Create your own errors

Custom Error

You can make your own errors by inheriting from the **Exception** class

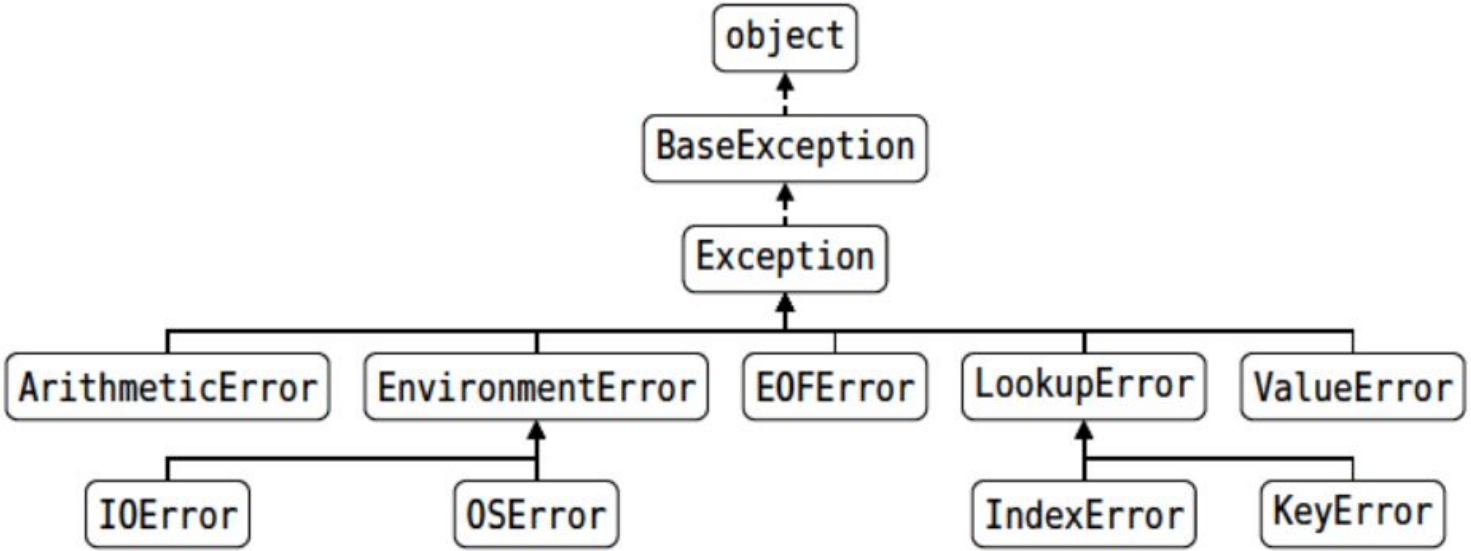
```
1  class CustomError(Exception):
2      pass
3
4  raise CustomError("yikes")
```

Custom Error (Complex)

Inheritance is the same as before so the class can be as complex as necessary.

```
1  class CustomError(Exception):
2
3      count = 0
4
5      def __init__(self, message, id):
6          CustomError.count += 1
7
8          message = f"Custom Error {CustomError.count} {id}"
9          super().__init__(message)
10
11 raise CustomError("yikes", 'a')
```

Exception Hierarchy Excerpt



Custom Error (Specific)

It is best practice to inherit from the closest existing error class

```
1  class InvalidChoiceError(ValueError):
2      pass
3
4  options = ["rock","paper", "scissors"]
5  user_choice = input("Pick move (rock/paper/scissors): ")
6
7  if user_choice not in options:
8      raise InvalidChoiceError()
```

05

GUI

Graphical User Interface

Benefits of Graphical User Interfaces

User Experience

Easier to understand and provides appeal and interactivity

Separation

Clearly Separate Frontend (Design, UI/UX) and Backend (Logic)

Limitations

Limit the possible edge cases and directly get needed data type

Python GUI Libraries

Tkinter

Standard GUI toolkit available in (almost) all Python distributions immediately. Easy to understand and great for building simple applications quickly.

PyQt

Python bindings or implementations for the Qt application framework. It has a lot of flexible components and great for building complex applications.

Kivy

Library built specifically for multi-touch platforms (mobile) but can be used in Desktops as well. Good for complex, cross-platform applications.

Window

Starting with the background

Window

```
1  import tkinter
2
3  root = tkinter.Tk()
4  root.mainloop()
5
6
7
```

Try this in a new file

Window (with Title)

```
1  import tkinter
2
3  root = tkinter.Tk()
4  root.title("Sample GUI Application")
5
6  root.mainloop()
7
```

Edit the last file

Window (with Size)

```
1  import tkinter
2
3  root = tkinter.Tk()
4  root.title("Sample GUI Application")
5  root.geometry("1200x400")
6
7  root.mainloop()
```

Edit the last file

Label

Adding text to the window

Label

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  label = tkinter.Label(root, text="Hello")
6  label.pack()
7
8  root.mainloop()
9
10
11
12
13
```

Try this in a new file

Multiple Labels

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  label = tkinter.Label(root, text="Hello")
6  label.pack()
7
8  next_label = tkinter.Label(root, text="World")
9  next_label.pack()
10
11 root.mainloop()
12
13
```

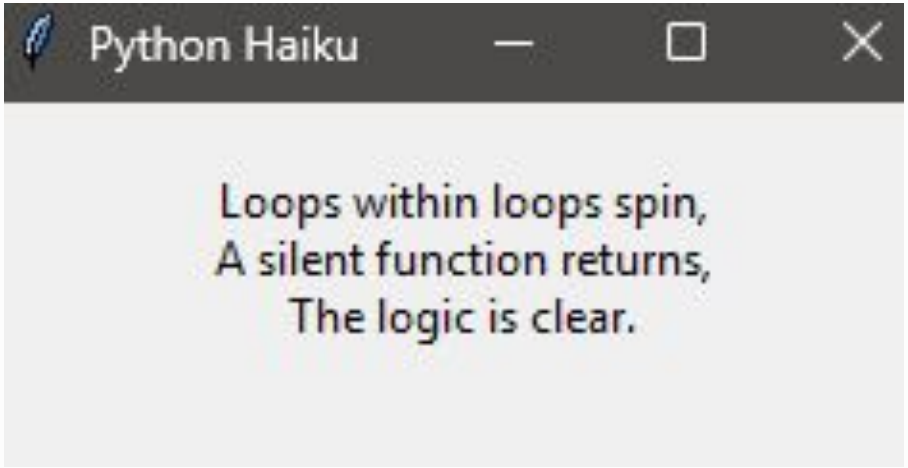
Edit the last file

Multiline Label

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  message = """
6      Hello World
7      In Multiple Lines
8  """
9
10 label = tkinter.Label(root, text=message)
11 label.pack()
12
13 root.mainloop()
```

Quick Exercise: Haiku

Recreate the following window using root properties and label(s)



Properties

Adding styling and layout to components

Component Font Style

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  label = tkinter.Label(
6      root,
7      text="Hello",
8      font=("Arial", 14, "bold italic")
9  )
10 label.pack()
11
12 root.mainloop()
13
```

Try this in a new file

Find Other Fonts Available

```
1  import tkinter
2  from tkinter import font
3
4  root = tkinter.Tk()
5
6  all_fonts = font.families()
7  print(all_fonts)
```

Try this in a new file

Component Color

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  label = tkinter.Label(
6      root,
7      text="Hello",
8      fg="red",
9      bg="yellow"
10 )
11 label.pack()
12
13 root.mainloop()
```

Try this in a new file

Component Size

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  label = tkinter.Label(
6      root,
7      text="Hello",
8      width=100,
9      height=20
10 )
11 label.pack()
12
13 root.mainloop()
```

Try this in a new file

Component Pad

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  label = tkinter.Label(
6      root,
7      text="Hello",
8      padx=10,
9      pady=200
10 )
11 label.pack()
12
13 root.mainloop()
```

Try this in a new file

Component Pack Side

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  label1 = tkinter.Label(root, text="Left")
6  label1.pack(side="left")
7
8  label2 = tkinter.Label(root, text="Right")
9  label2.pack(side="right")
10
11 root.mainloop()
```

Try this in a new file

Entry

Asking the user for text input

Blank Entry

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  entry = tkinter.Entry(root)
6  entry.pack()
7
8  root.mainloop()
9
10
11
12
13
14
15
```

Key Bind Entry

```
1  import tkinter
2
3  def print_left(event):
4      print("Left pressed")
5
6  root = tkinter.Tk()
7
8  entry = tkinter.Entry(root)
9  entry.pack()
10
11 root.bind("<Left>", print_left)
12 root.mainloop()
13
14
15
```

Entry Echo

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  entry = tkinter.Entry(root)
6  entry.pack()
7
8  def print_input(event):
9      user_input= entry.get()
10     print(user_input)
11
12 root.bind("<Return>", print_input)
13 root.mainloop()
14
15
```

Multiple Keybindings

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  entry = tkinter.Entry(root)
6  entry.pack()
7
8  def print_input(event):
9      user_input= entry.get()
10     print(user_input)
11
12 root.bind("<Return>", print_input)
13 root.bind("<space>", print_input)
14 root.mainloop()
15
```

Available Bindings

Type of Key	Behavior
Numbers	`<0>, <1>, <2>, <3>, <4>, <5>, <6>, <7>, <8>, <9>`
Lowercase Letters	`<a>, <b>, <c>, ...`
Uppercase Letters	`<A>, <B>, <C>, ...`
Space	`<space>`
Special Keys	`<Return>, <Tab>, <Shift>, <Alt_L>, <Escape>, ...`
Function Keys	`<F1>, <F2>, <F3>, ...`
Navigation Keys	`<Left>, <Right>, <Up>, <Down>`
Multiple Keys	`<Control-Shift-s>`

Entry Marker

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  entry = tkinter.Entry(root)
6  entry.pack()
7
8  def mark_input(event):
9      label = tkinter.Label(root, text=entry.get())
10     label.pack()
11
12 root.bind("<Return>", mark_input)
13 root.mainloop()
14
15
```

String Variable

Dynamic text for components

String Variable

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  text = tkinter.StringVar(root, value="Hello")
6  label = tkinter.Label(root, textvariable=text)
7  label.pack()
8
9  root.mainloop()
10
11
12
13
14
15
```

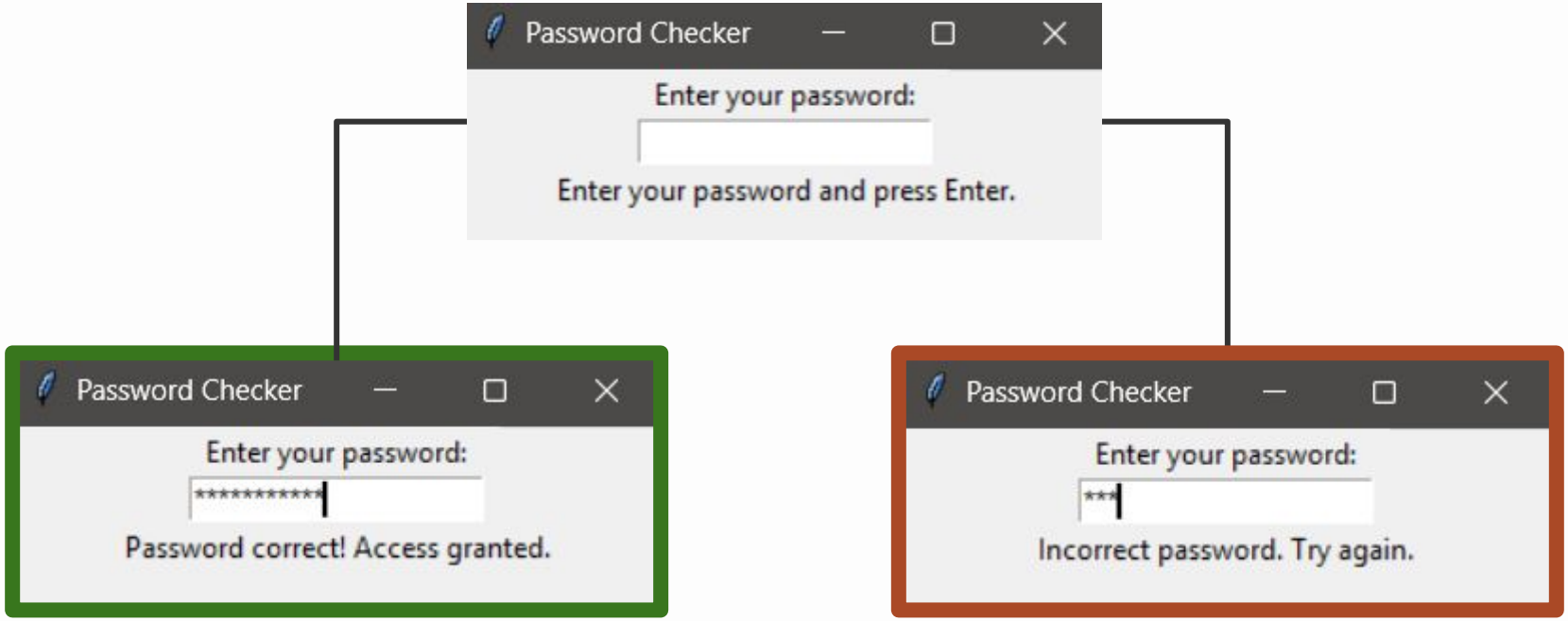
Dynamic Label

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  user_input = tkinter.StringVar(root, value="Enter any text")
6  label = tkinter.Label(root, textvariable=user_input )
7  label.pack()
8
9  entry = tkinter.Entry(root)
10 entry.pack()
11
12 def display(event):
13     user_input.set(entry.get())
14
15 root.bind("<Return>", display)
16 root.mainloop()
```

Variable Read and Write

```
var.get()
var.set(value)
```

Quick Exercise: Password Checker



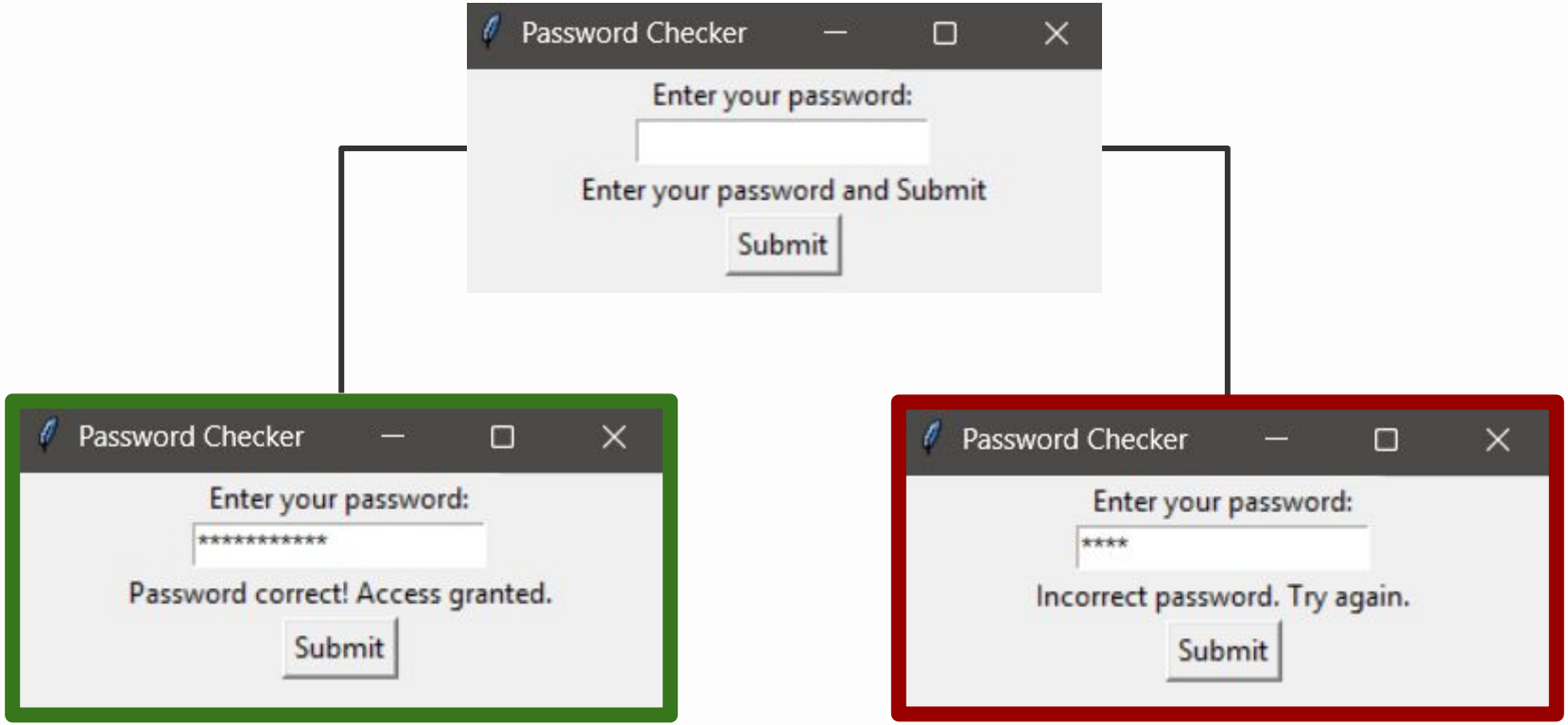
Buttons

Trigger functions on command

Dynamic Label

```
1  import tkinter
2  root = tkinter.Tk()
3
4  user_input = tkinter.StringVar(root, value="Enter any text")
5  label = tkinter.Label(root, textvariable=user_input )
6  label.pack()
7
8  entry = tkinter.Entry(root)
9  entry.pack()
10
11 def display(event):
12     user_input.set(entry.get())
13
14 button = tkinter.Button(root, text="Submit", command=display)
15 button.pack()
16 root.mainloop()
```

Quick Exercise: Password Checker



Int Variable

Dynamic number for components

Counter

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  count = tkinter.IntVar(root, value=0)
6  label = tkinter.Label(root, textvariable=count)
7  label.pack()
8
9  def increment():
10     count.set(count.get() + 1)
11
12 button = tkinter.Button(root, text=" + ", command=increment)
13 button.pack()
14
15 root.mainloop()
```

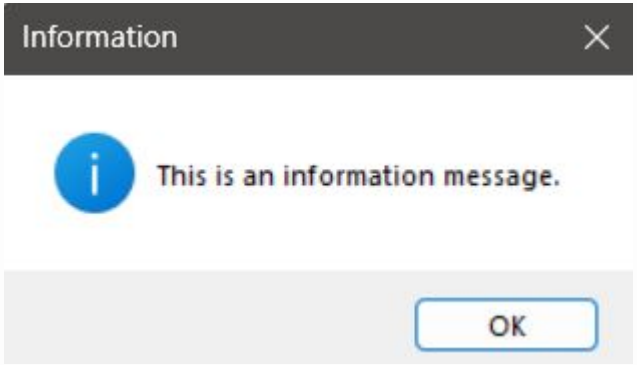
Quick Exercise: Full Counter

Message Boxes

Sudden message displays for the user

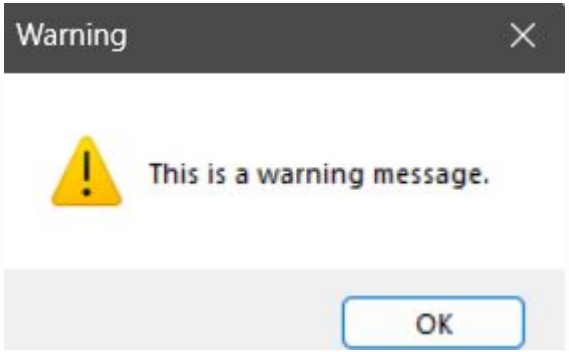
Information Box

```
1  import tkinter
2  from tkinter import messagebox
3
4  root = tkinter.Tk()
5
6  messagebox.showinfo(
7      "Information",
8      "This is an information message."
9  )
10
11 root.mainloop()
12
13
14
15
```



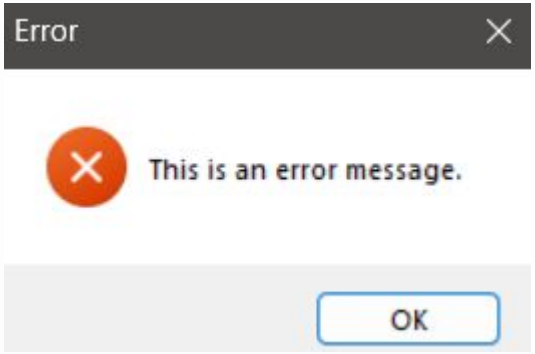
Warning Box

```
1  import tkinter
2  from tkinter import messagebox
3
4  root = tkinter.Tk()
5
6  messagebox.showwarning(
7      "Warning",
8      "This is a warning message."
9  )
10
11 root.mainloop()
12
13
14
15
```



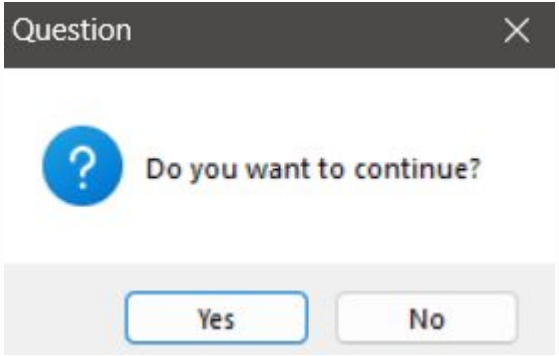
Error Message Box

```
1  import tkinter
2  from tkinter import messagebox
3
4  root = tkinter.Tk()
5
6  messagebox.showerror(
7      "Error",
8      "This is an error message."
9  )
10
11 root.mainloop()
12
13
14
15
```



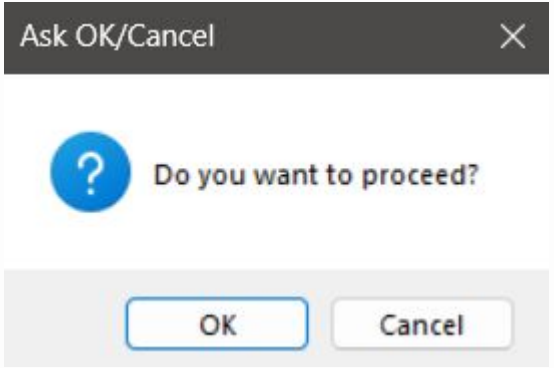
Question Message Box

```
1  import tkinter
2  from tkinter import messagebox
3
4  root = tkinter.Tk()
5
6  response = messagebox.askquestion(
7      "Question",
8      "Do you want to continue?"
9  )
10
11 root.mainloop()
12
13
14
15
```



Ask OK Message Box

```
1  import tkinter
2  from tkinter import messagebox
3
4  root = tkinter.Tk()
5
6  response = messagebox.askokcancel(
7      "Ask OK/Cancel",
8      "Do you want to proceed?"
9  )
10
11 root.mainloop()
12
13
14
15
```



Boolean Variable

Dynamic boolean for components

Checkbox

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  check_value = tkinter.BooleanVar()
6  checkbox = tkinter.Checkbutton(
7      root,
8      text="Enable",
9      variable=check_value
10 )
11 checkbox.pack()
12
13 root.mainloop()
14
15
```

Input Components

Other basic components for getting user data

Radio Buttons

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  radio_var = tkinter.StringVar(value="Option A")
6  radio1 = tkinter.Radiobutton(
7      root, text="Option A", variable=radio_var, value="Option A")
8  radio2 = tkinter.Radiobutton(
9      root, text="Option B", variable=radio_var, value="Option B")
10 radio1.pack()
11 radio2.pack()
12
13 root.mainloop()
14
15
```

Dropdown

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  dropdown_var = tkinter.StringVar(value="Choice 1")
6  dropdown_menu = tkinter.OptionMenu(
7      root, dropdown_var,
8      "Choice 1",
9      "Choice 2",
10     "Choice 3"
11 )
12 dropdown_menu.pack()
13
14 root.mainloop()
15
```

Slider

```
1  import tkinter
2
3  root = tkinter.Tk()
4
5  slider_value = tkinter.IntVar(value=0)
6  slider = tkinter.Scale(
7      root,
8      from=0,
9      to=100,
10     orient="horizontal",
11     variable=slider_value
12 )
13 slider.pack()
14
15 root.mainloop()
```

Layout

Setup the layouting for all of the components by group

Frames

```
1  import tkinter
2  root = tkinter.Tk()
3
4  top_frame = tkinter.Frame(root, bg="lightblue", height=100)
5  top_frame.pack(fill="x")
6
7  bottom_frame = tkinter.Frame(root, bg="lightgreen", height=200)
8  bottom_frame.pack(fill="both", expand=True)
9
10 label1 = tkinter.Label(top_frame, text="Top Frame", bg="blue")
11 label1.pack(pady=10)
12
13 label2 = tkinter.Label(top_frame, text="Bot Frame", bg="green")
14 label2.pack(pady=20)
15
16 button = tkinter.Button(top_frame, text="Click Me")
17 button.pack(pady=10)
18
19 root.mainloop()
```

Grids

```
1  import tkinter
2  root = tkinter.Tk()
3
4  label1 = tkinter.Label(root, text="Label 1")
5  label1.grid(row=0, column=0, padx=10, pady=10)
6
7  label2 = tkinter.Label(root, text="Label 2")
8  label2.grid(row=0, column=1, padx=10, pady=10)
9
10 label3 = tkinter.Label(root, text="Label 3")
11 label3.grid(row=1, column=0, padx=10, pady=10)
12
13 label4= tkinter.Label(root, text="Label 4")
14 label4.grid(row=1, column=1, padx=10, pady=10)
15
16 button = tkinter.Button(root, text="Click Me")
17 button.grid(row=2, column=0, columnspan=2, pady=10)
18
19 root.mainloop()
```

Tkinter Class

Code Organization in GUI Programming

Common Structure

```
1  import tkinter
2
3  class Application(tkinter.Tk):
4       def __init__(self):
5            super().__init__()
6            self.title("Tkinter Class Structure")
7            self.geometry("300x200")
8            self.create_widgets()
9
10      def create_widgets(self):
11           label = tkinter.Button(self, text="Hello", command=self.hello)
12           label.pack()
13
14      def hello(self):
15           print("Hello")

app = Application()
app.mainloop()
```

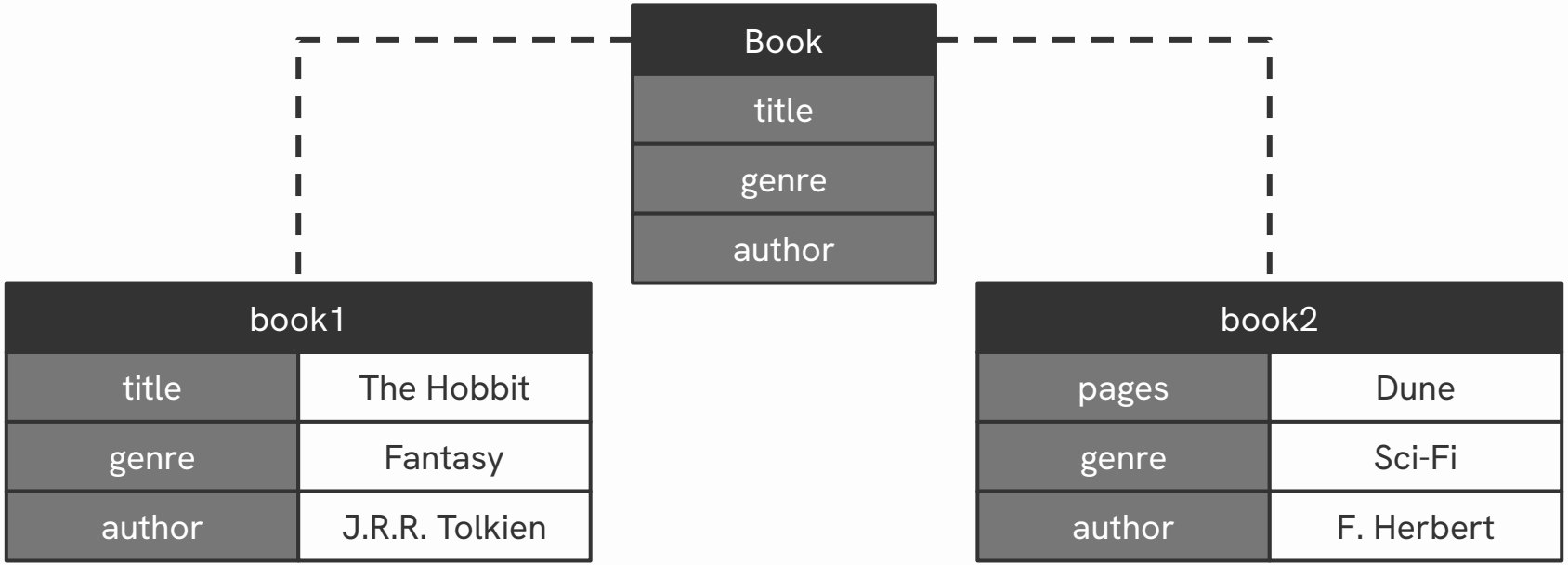

+
Magic Methods

Magic/Dunder Methods

Dunder methods are special, built-in methods that start and end with dunders (double underscores). Using these methods change or add custom behaviors to classes.

Method Name	Input(s)	Output(s)	Note
__init__	*	None	Sets behavior when creating objects
__str__	None	String	Used in **`str()`** and **`print()`**
__eq__	Any	Boolean	Sets behavior for **==** operations
__add__	Any	Any	Sets behavior for **+** operations
__len__	None	Integer	Sets behavior when used in **`len()`**

Book


Book
title
genre
author
book1
title
The Hobbit
genre
Fantasy
author
J.R.R. Tolkien
book2
pages
Dune
genre
Sci-Fi
author
F. Herbert

Implement: Book

```
1  class Book:
2      def __init__(self, title=None, genre=None, author=None):
3          self.title = title
4          self.genre = genre
5          self.author = author
6
7  book1 = Book("The Hobbit", "Fantasy", "Tolkien")
8  book2 = Book("Dune", "Sci-Fi", "Herbert")
9  print(book1)
```

```
<__main__.Book object at 0x0000019FE4F27BC0>
```

Magic Method __repr__

The **__str__** dunder method defines what is used if the object is printed or used in **print()**

```
1  class Book:
2      def __init__(self, title=None, genre=None, author=None):
3          self.title = title
4          self.genre = genre
5          self.author = author
6
7      def __repr__(self):
8          return f"{self.title} [{self.genre}] - {self.author}"
9
   book1 = Book("The Hobbit", "Fantasy", "Tolkien")
   book2 = Book("Dune", "Sci-Fi", "Herbert")
   print(book1)
```

```
The Hobbit [Fantasy] - Tolkien
```

Magic Method __add__

The **__add__** dunder method defines the result when an **+** operation is used with the object

```
1  class Wallet:
2      def __init__(self, initial_amount=0):
3          self.amount = initial_amount
4
5      def __add__(self, other):
6          new_amount = self.amount + other.amount
7          return Wallet(new_amount )
8
9  food_wallet = Wallet(250)
10 transport_wallet = Wallet(1000)
11 total_wallet = food_wallet + transport_wallet
12
13 print("Food Budget: ", food_wallet.amount)
14 print("Transport Budget: ", transport_wallet.amount)
15 print("Total Budget: ", total_wallet.amount)
```

Object Identity

Python uses the memory location of an object to check for equality

```
1  class Candy:
2      def __init__(self, flavor):
3          self.flavor = flavor
4
5  choco1 = Candy("chocolate")
6  choco2 = Candy("chocolate")
7  milk = Candy("milk")
8
9  print(choco1 == milk)
10 print(choco1 == choco2)
```

Magic Method __eq__

The **__eq__** dunder method defines whether two objects are equal (or not)

```
1  class Candy:
2      def __init__(self, flavor):
3          self.flavor = flavor
4
5      def __eq__(self, other):
6          return self.flavor == other.flavor
7
8  choco1 = Candy("chocolate")
9  choco2 = Candy("chocolate")
10 milk = Candy("milk")
11
12 print(choco1 == milk)
13 print(choco1 == choco2)
```

Implement: Ticket System

Ticket
name
`__str__(self)`

TicketManager
queue
`issue(self, ticket)`
`cancel(self, name)`
`process(self)`
`__str__(self)`

```
def ticket_app():
    queue = TicketManager()
    user_choice = input("command: ")

    while user_choice != "exit":

        if user_choice == "add":
            ticket_name = input("Enter name: ")
            ticket = Ticket(ticket_name)
            queue.issue(ticket)

        ...
        user_choice = input("command: ")

ticket_app()
```

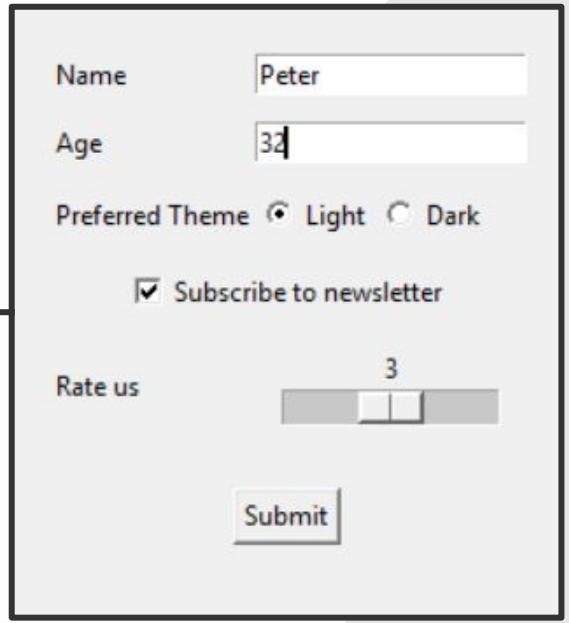
06

Lab Session

All the Major Features Covered

user.json

```
{
    "Name": "Peter"
    "Age": 32
    "Theme": "Light"
    "Subscribe": True
    "Rating": 3
}
```



FORM


Deck of Cards

Review: Deck of Cards

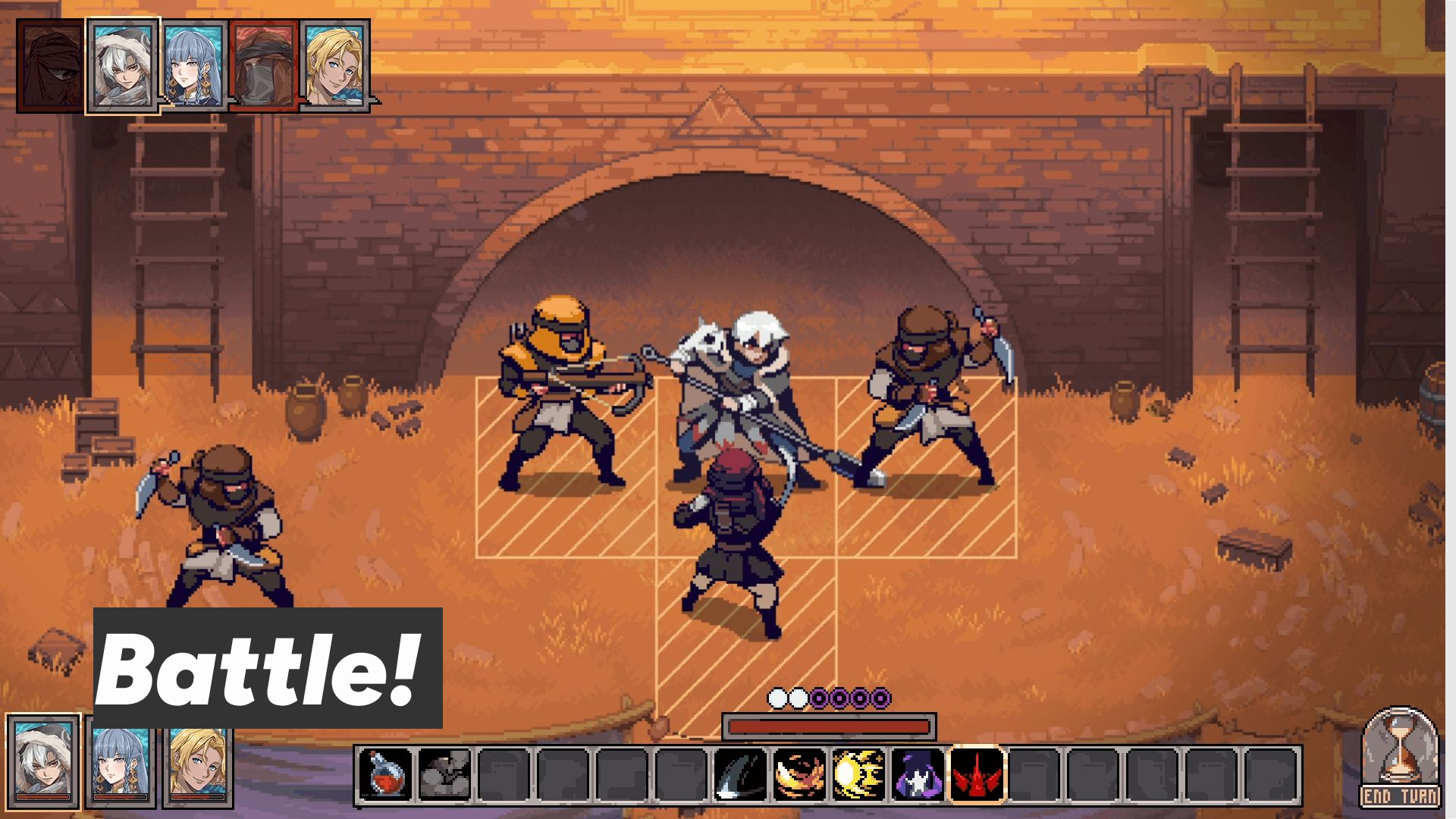
```
1  def create_deck() -> list[str]:
2      # Return a list of 52 strings containing a standard deck
3
4  def draw_top(deck: list[str], count: int =1)-> list[str]:
5      # Remove count return count cards from the start from deck
3
4  def draw_bottom(deck: list[str], count: int =1) -> list[str]:
5      # Remove and return count cards from the end of the deck
6
7  def draw_random(deck: list[str], count: int =1) -> list[str]:
8      # Remove and return count random cards from the deck
9
10 def show(deck):
11     # Print all cards in deck
```

Review: Dynamic Adding

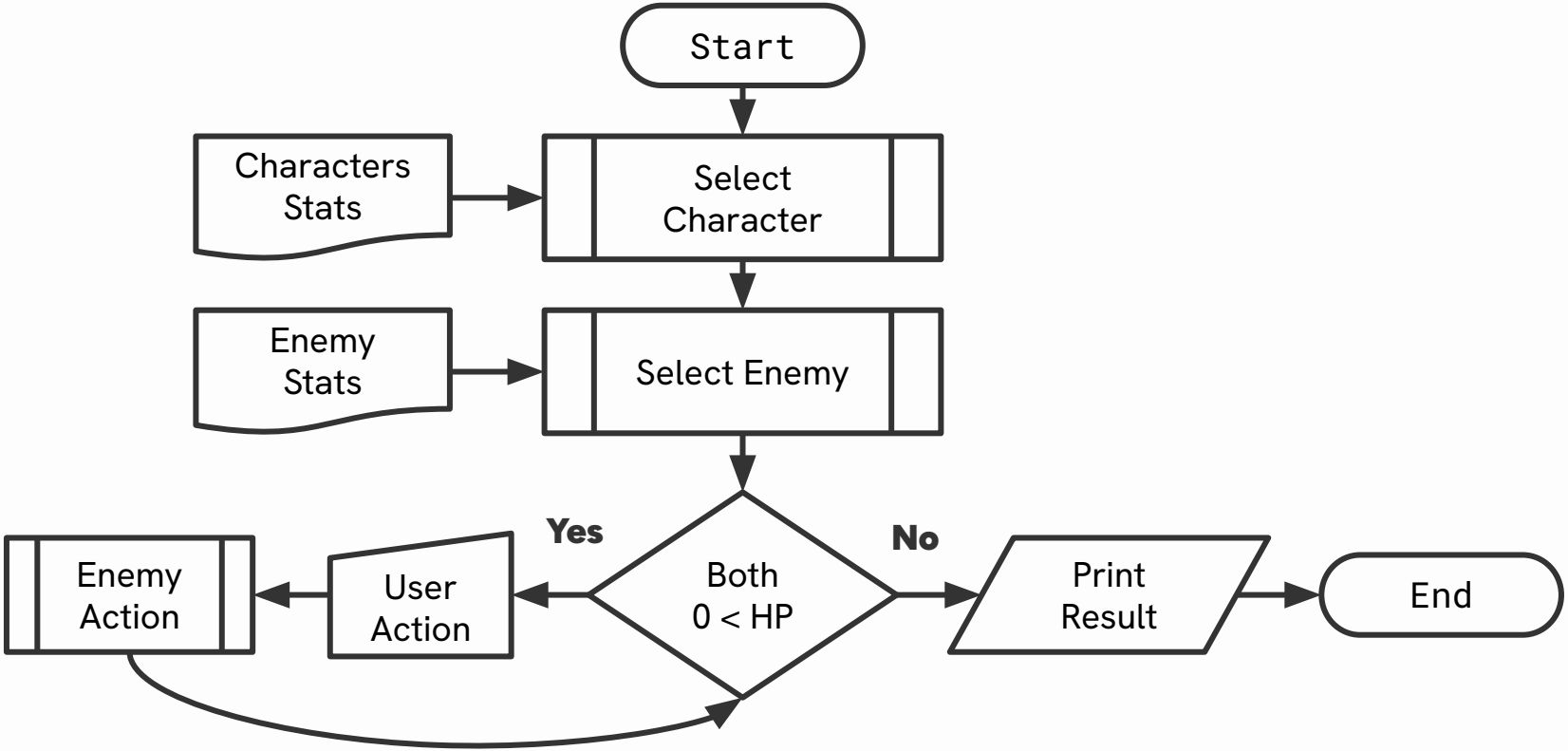
```
12  def add_top(deck: list[str], other: list[str])-> list[str]:
13      # Add cards in other to the first parts of deck
14
15  def add_bottom(deck: list[str], other: list[str])-> list[str]:
16      # Add cards in other to the last parts of deck
17
18  def add_random(deck: list[str], other) -> list[str]:
19      # Add cards in other randomly to deck
```

Template: Deck of Cards

```
class Deck:
    def __init__(self)
        self.cards = [...]

    def draw_top(self, count: int = 1)-> list[str]:
    def draw_bottom(self, count: int = 1) -> list[str]:
    def draw_random(self, count: int = 1) -> list[str]:
    def show(self):
    def add_top(self, other: list[str]):
    def add_bottom(self, other: list[str]):
    def add_random(self, other: list[str]):
    def load(self, filename: str):
    def save(self, filename: str):
```


Battle!
END TURN

Battle! Game Flow



Sneak Peak

01

Packaging

Handling Python Files

02

Concurrency

Multitasking techniques

03

Best Practices

Writing Pythonic code

04

Testing

Code correctness

05

Web Dev

Introduction to Flask

06

Lab Session

Culminating Exercise

Python: Day 03

Object-Oriented Programming