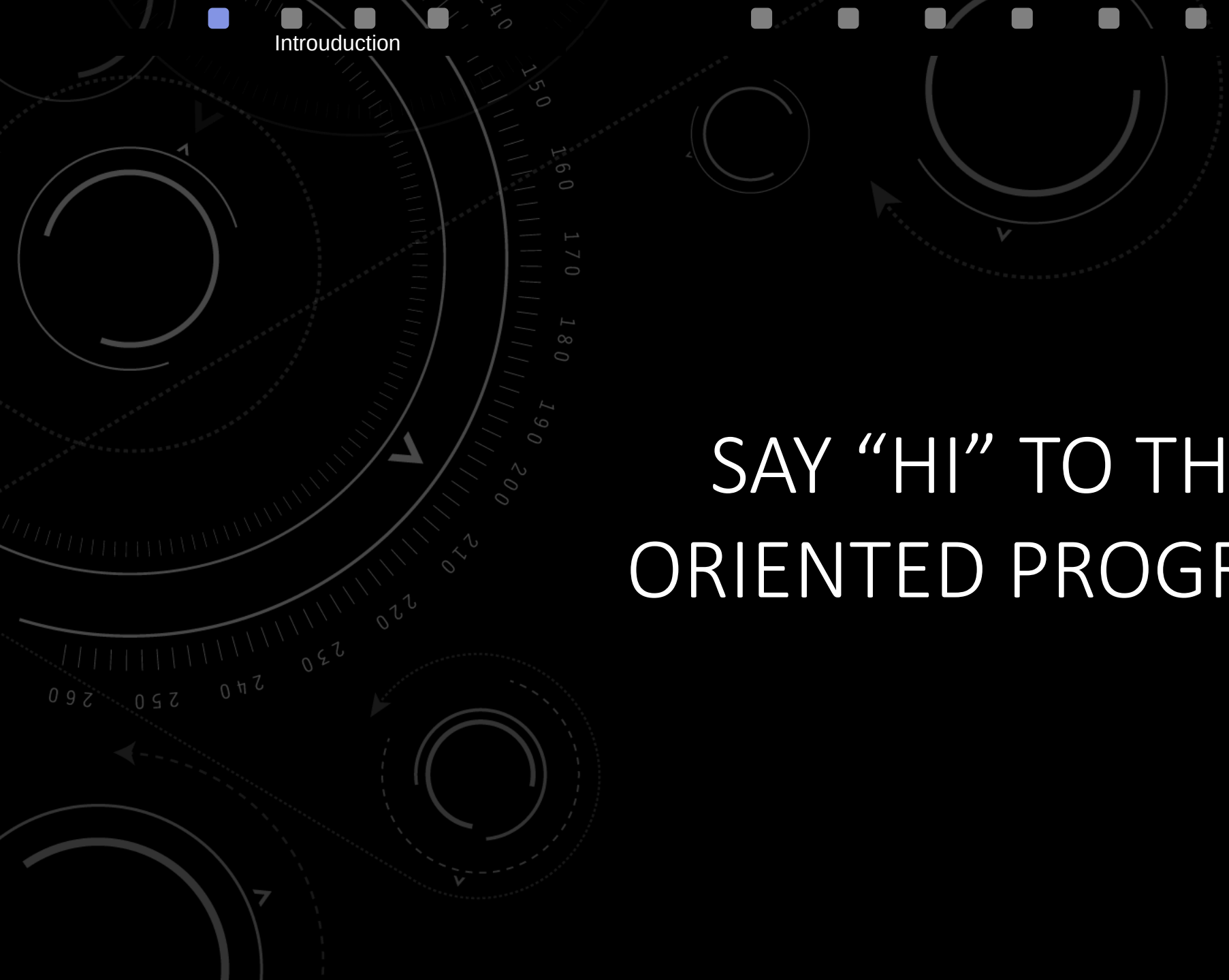

SAY “HI” TO THE OBJECT-ORIENTED PROGRAMMING

ILIA KOSENKOV

2018-09-18

WHAT ARE THE MAIN PROGRAMMING PARADIGMS?

- Imperative
  - Procedural
  - Object-oriented
- Declarative
  - Functional
  - Mathematical

Based on instructions to the machine
<- Groups instructions into procedures
<- Groups instructions and states

The properties of the result are declared
<- Result is a series of function applications
<- Result is a solution of optimizing problem

PROCEDURAL PROGRAMMING

- The core feature is a **procedure -** a set of instructions that can be called at any point of the program, including from other procedures, or from this same procedure (recursion case)

```
1   def Greet(name):
2       print( f"Hello, {name}!")
3
4   def GenerateNames():
5       yield "Alex"
6       yield "Bob"
7       yield "Eve"
8
9   def GreetAll():
10      for nm in GenerateNames():
11          Greet(nm)
12
13  if __name__ == "__main__":
14      GreetAll()
```

OBJECT-ORIENTED PROGRAMMING

```python
from numpy import array as arr
class Particle:
    def __init__(self, pos, vel, mass):
        self._position = pos.copy()
        self._velocity = vel.copy()
        self._mass = mass

    @property
    def position(self):
        return self._position
    @position.setter
    def position(self, val):
        self._position = val.copy()
    @property
    def velocity(self):
        return self._velocity
    @velocity.setter
    def velocity(self, val):
        self._velocity = val.copy()

    @property
    def mass(self):
        return self._mass

    def advanceTime(self, dt):
        self._position += self._velocity * dt

    def __repr__(self):
        return f"{{r = {self._position}, " + \
            f"v = {self._velocity}, " + \
            f"m = {self._mass}}}"

if __name__ == '__main__':
    prtcls = [
        Particle(arr([ 1.0, 0.0, 0.0]), arr([-0.5, 0.0, 0.0]), 1.0),
        Particle(arr([ 0.0, 1.0, 0.0]), arr([ 0.0,-0.5, 0.0]), 1.0),
        Particle(arr([-1.0, 0.0, 0.0]), arr([ 0.5, 0.0, 0.0]), 1.0),
        Particle(arr([ 0.0,-1.0, 0.0]), arr([ 0.0, 0.5, 0.0]), 1.0)]

    for p in prtcls:
        print(p)

    for p in prtcls:
        p.advanceTime(0.1)
    print("\r\n")
    for p in prtcls:
        print(p)
```

- Concept is centered around the **object** - which may contain data in form of fields, properties, attributes, and associated procedures (called **methods**). Methods can access objects they are associated with (a notion of **self** or **this**).
- The program is essentially an interaction of various objects

OBJECT-ORIENTED APPROACH

- Encapsulation
- Inheritance
- Polymorphism

LETS PRETEND WE ARE BUILDING A PLANE

- This explanation is borrowed from https://habr.com/post/345658/
- We are ordered to build three kinds of planes
  - Military
  - Civil
  - Cargo

ABSTRACT CLASS

- We start by designing a blueprint where we point out basic properties shared by all the planes
- For instance, what are the size of the wings, where to place the engines, how the payload is carried.
- However, we do not go much into detail - we describe only those features that are shared by all three plane types
- These preliminary blueprint is an **abstract class**

INTERFACE

- Now we need to define a set of requirements that a plane (any of three types) should meet
  - The ability to take off, land, fly -> these are **instance methods** (**virtual**)
  - Means of getting information about plane status - like speed, altitude, etc -> these are **properties**
- Unfortunately, *python* does not support interfaces directly. Interfaces are usually part of statically typed languages. *Python* replaces it by multiple inheritance and **duck typing** - “If it walks like a duck and quacks like a duck - it is a duck”. This means to implement an agreement (interface) in *python*, one needs to implement all the required methods, no special constructs or inheritance required.

INHERITANCE

- Now it is time to think of each of the ordered plane models (civil, military and cargo)
  - Lets take our preliminary blueprint of the fuselage and use it to create three new blueprints - one per each plane type. The connection between the base blueprint and derived one is **inheritance**.
  - Each new blueprint can redefine base class' methods to better suite the case.
  - It can also define new, unique methods and properties, like **getNumberOfPassengers** or **reloadWeapons**
  - All three derived classes share common features through the base class and can *do* same things via the **interface**
- We can further exercise inheritance by creating a new blueprint of the passenger plane with larger number of seats, by deriving not from the base class, but from the civil plane class.
  - This new blueprint - **derived class** - will feature all the properties of the initial preliminary blueprint, and all the properties of the civil plane. It can also implement new, unique to this particular model, features.

INSTANCE OF A CLASS

- Its time to deliver the order - let us use the blueprints to produce three different planes. Each plane is an **instance** of the **class** (blueprint). The plane - instance - can fly according to the rules defined by the **interface**, and share common features (like fuselage) with planes of other types.

POLYMORPHISM

- We designed each of our planes using the preliminary blueprint - the **abstract class**. This means that our planes of any derived class can operate on any runway that is designed to handle planes of the preliminary blueprint type. This is **polymorphism** - instances of a derived classes can act like instances of the **base class**.

COMPOSITION

- Remember we designed a mounting points for the engines in the initial blueprint? Well, engines can be constructed the same way as planes - starting from a general blueprint - **class** - all the way down to a specific blueprint of a model produced and sold to clients. Our planes can utilize different engines mounted on them.
- We do not produce the engine ourselves, we just say that each mounting point should have a manufactured engine - an instance of some engine **class**. We can then order different engines for different planes. However, in order to operate those engines, these engines should provide an **interface** - a common set of rules how to use them.
- The ability to embed instances of other classes into our own without designing them ourselves is **composition**.

LAST, BUT NOT LEAST - CLASS METHODS

- Class methods or static methods are methods invoked per class. This methods do not require an instance of a class (a plane).
- These methods usually refer to the properties of the *model* of the plane. For instance, *getNumberOfPlanesManufactured()* gives the number of planes of this type manufactured so far. Even if there are no instances of this class - no planes built with this blueprint - we can still call this method and get a result of 0.