Lecture - Basics of Object Oriented Programming

Introduction to Object Oriented Programming

We have learned about Classes in Week 3 and 4 - classes are the fundamental building block that is used to model real word objects. Here we revisit classes again in context of Object Oriented Programming.

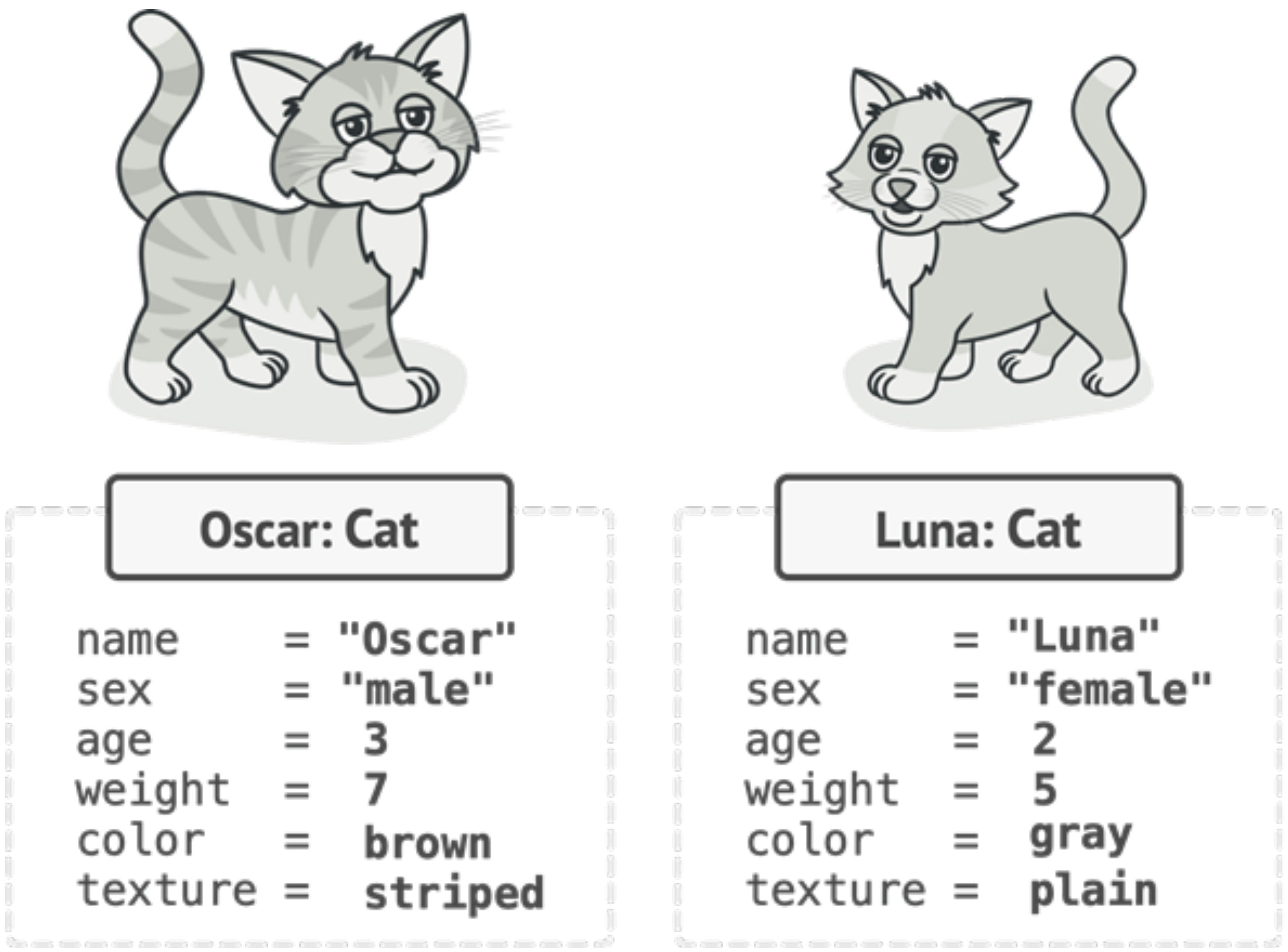
What is an Object Oriented Programming?

The Object-oriented programming is a **paradigm** based on the concept of **wrapping pieces of data**, and **behaviour related to that data**, into special bundles called **objects**, which are constructed from a set of **"blueprints"**, defined by a programmer, called **classes**.

Paradigm is a specific approach or way of thinking about how to solve problems and design computer programs. It focuses on objects that **encapsulate** data (**attributes**) and the code that manipulates that data (**methods**). Objects interact with each other through messages, and programs are built by creating and manipulating these objects.

What are Objects and Classes?

I quickly recap the concepts of Objects and Classes by using a simple **CAT** example. Say you have a cat named **Oscar**. Oscar is an object, an ***instance*** of the class. Every cat has a lot of standard attributes: name, sex, age, weight, colour, favorite food, etc. These are the class's ***fields***.



```
name    = "Oscar"
sex     = "male"
age     =  3
weight  =  7
color   =  brown
texture =  striped
```

```
name    = "Luna"
sex     = "female"
age     =  2
weight  =  5
color   =  gray
texture =  plain
```

All cats also behave: they **breathe, eat, run, sleep and meow**. These are the class's methods. Collectively, **fields** and **methods** can be referenced as the members of their class.

Data stored inside the object's fields is often referenced as state, and all the object's methods define its behaviour.

Luna, your friend's cat, is also an instance of the class. It has the same set of attributes as Oscar. The difference is in values of these attributes: its sex is female, it has a different colour, and weighs less. So a class is like a ***blueprint*** that defines the structure for objects, which are concrete instances of that class.

Class Hierarchies

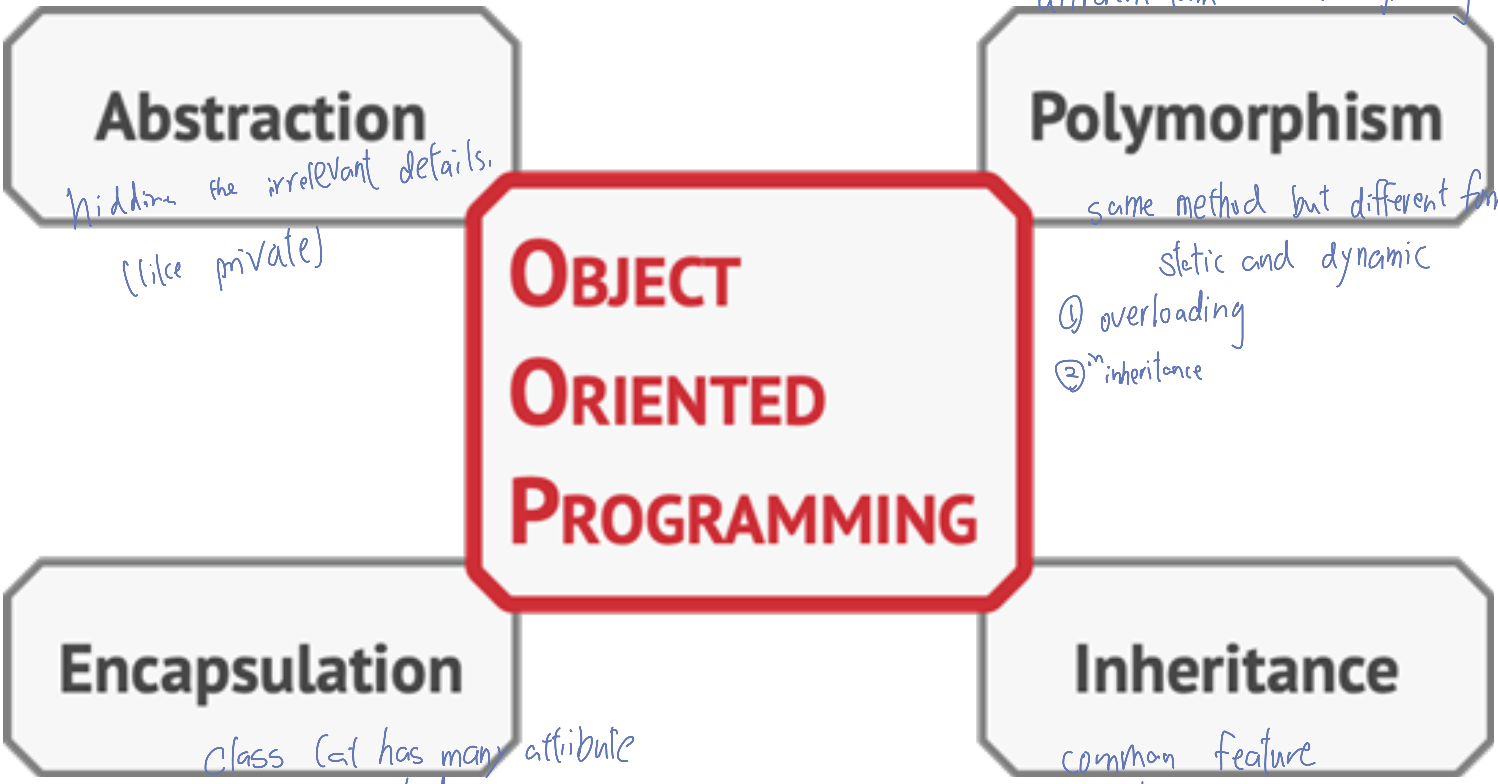
Everything's fine when we talk about one class. Naturally, a real program contains more than a single class. Some of these classes might be organised into class hierarchies. Let's find out what that means.

Say your neighbour has a dog called Fido. It turns out, dogs and cats have a lot in common: name, sex, age, and color are attributes of both dogs and cats. Dogs can breathe, sleep and run the same way cats do. So it seems that we can define the base Animal class that would list the common attributes and behaviours.

A parent class, like the one we’ve just defined, is called a superclass. Its children are subclasses. Subclasses inherit state and behaviour from their parent, defining only attributes or behaviorus that differ. Thus, the Cat class would have the meow method, and the class Dog the bark method. This is called inheritance and we will cover more about it coming slides.

Pillars of Object Oriented Programming

Object-oriented programming is based on four pillars (sometimes referred to as Principles), concepts that differentiate it from other programming paradigms. These are **Abstraction**, **Polymorphism**, **Encapsulation** and **Inheritance**.



different form of one single thing

hidding the irrelevant details.
(like private)

same method but different form
static and dynamic
① overloading
② in inheritance

class Cat has many attribute
and methods.
→ we encapsulate them into the class

common feature
want reuse.
(reduce errors)

Abstraction

Most of the time when you are creating a program with OOP, you shape objects of the program based on real-world objects. However, objects of the program do not represent the originals with 100% accuracy (and it's rarely required that they do). Instead, your objects only model attributes and behaviours of real objects in a specific context, ignoring the rest.

For example, an Aeroplane class could probably exist in both a flight simulator and a flight booking application. But in the former case, it would hold details related to the actual flight, whereas in the latter class you would care only about the seat map and which seats are available. For Example:

```
public class Airplane{

    private float speed;
    private float altitude;
    // more attributes

    public void fly()...
}
```

```
public class Airplane{
    private int seats;
    public boolean reserveSeat(int seatNumber)...
}
```

Abstraction is a model of a real-world object or phenomenon, limited to a specific context, which represents all details relevant to this context with high accuracy and omits all the rest.

Encapsulation

The word encapsulation is a noun that means the action of enclosing something in or as if in a capsule. With respect to Object Oriented Programming, encapsulation means enclosing information of objects into classes as we do in capsules - hiding the medicine powder inside the capsule.

Attributes are the pink side
Methods are the blue side

To start a car engine, you only need to turn a key or press a button. You don't need to connect wires under the hood, rotate the crankshaft and cylinders, and initiate the power cycle of the engine. These details are hidden under the hood of the car. You have only a simple interface: a start switch, a steering wheel and some pedals. This illustrates how each object has an interface - a public part of an object, open to interactions with other objects.

Encapsulation is the ability of an object to hide parts of its state and behaviors from other objects, exposing only a limited interface to the rest of the program.

To encapsulate something means to make it private, and thus accessible only from within the methods of its own class. There's a little bit less restrictive mode called *protected* that makes a member of a class available to subclasses as well.

Interfaces and abstract classes/methods of most programming languages are based on the concepts of abstraction and encapsulation. In modern object-oriented programming languages, the interface mechanism (usually declared with the interface or protocol keyword) lets you define contracts of interaction between objects. That's one of the reasons why the interfaces only care about behaviors of objects, and why you can't declare a field in an interface.

Imagine that you have a `FlyingTransport` interface with a method `fly(origin, destination, passengers)` . When designing an air transportation simulator, you could restrict the `Airport` class to work only with objects that implement the `FlyingTransport` interface. After this, you can be sure that any object passed to an airport object, whether it's an `Airplane`, a `Helicopter` or a freaking `DomesticatedGryphon` to arrive or depart from this type of airport.

```
public class Airport{
// some attributes here
    public void accept(vehicle flyingTransport){}
}

public interface FlyingTransport(){
    public fly(String origin, String destination, List[] passengers)
}

public class Helicopter{
// some attributes here
    @override
    public fly(String origin, String destination, List[] passengers){
    // helicopter's own implementation of the fly method
    }
}

public class Airplane{
    // some attributes here
    @override
    public fly(String origin, String destination, List[] passengers){
        // Airplane's own implementation of the fly method
    }
}
```

```
public class DomesticatedGryphon{
    // some attributes here
    @override
    public fly(String origin, String destination, List[] passengers){
    // DG's own implementation of the fly method
    }
}
```

You could change the implementation of the fly method in these classes in any way you want. As long as the signature of the method remains the same as declared in the interface, all instances Airport class can work with your of the flying objects just fine.

In Java, access modifiers like private, public, and protected play a crucial role in achieving encapsulation by controlling the visibility and accessibility of class members (fields and methods).

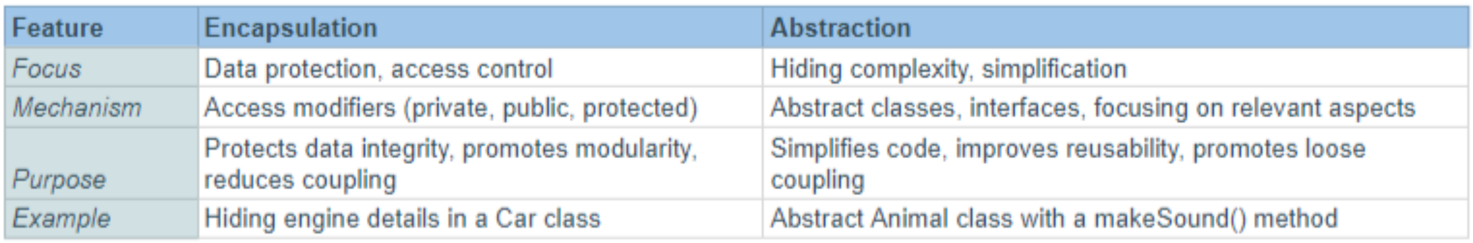
Are encapsulation and abstraction same??

A lot of students often confuse between encapsulation and abstraction. Though both look quite similar, they achieve different purpose. Here I quickly summaries the difference.

Abstraction is the process of hiding the complex implementation details and showing only the essential features of an object. It focuses on "what" an object does rather than "how" it does it. In other words, abstraction allows you to create a simplified model of a real world object or system by focusing on its relevant characteristics. **For example, when you use a car, you don't need to know how the engine works in detail; you just need to know how to drive it.**

Encapsulation is the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit, i.e., the class. It hides the internal state of an object from the outside world and only exposes a controlled interface for interacting with the object. Encapsulation helps in achieving data hiding, abstraction, and information hiding, which are essential for building robust and maintainable software systems.

I have summarised the purpose in a table below.



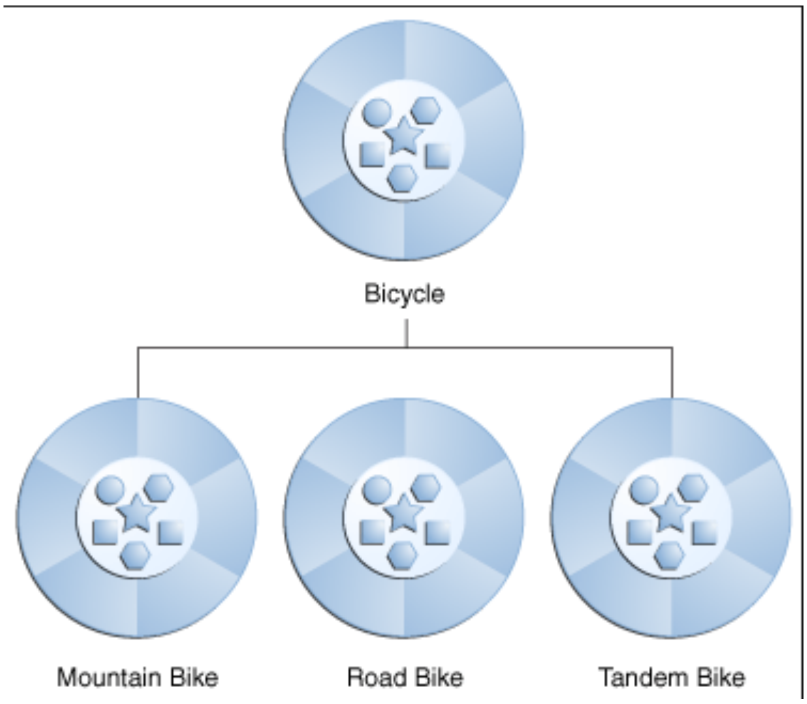
Feature	Encapsulation	Abstraction
Focus	Data protection, access control	Hiding complexity, simplification
Mechanism	Access modifiers (private, public, protected)	Abstract classes, interfaces, focusing on relevant aspects
Purpose	Protects data integrity, promotes modularity, reduces coupling	Simplifies code, improves reusability, promotes loose coupling
Example	Hiding engine details in a Car class	Abstract Animal class with a makeSound() method

In a nutshell, Encapsulation helps achieve abstraction and abstraction leverages encapsulation.

Inheritance

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In this example, Bicycle now becomes the superclass of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses:



The main benefit of inheritance is code reuse. If you want to create a class that's slightly different from an existing one, there's no need to duplicate code. Instead, you extend the existing class and put the extra functionality into a resulting subclass, which inherits fields and methods of the superclass.

Polymorphism

The word polymorphism is derived from Greek and it is a combination of two words - Poly means many and morph mean form i.e. having many forms. Apart from computer programming, the idea of polymorphism occurs in other real-world areas, including biology, chemistry and drug development. In context of Object Oriented Programming, this means that an object makes different forms i.e. it responds differently depending on the actual type provided. Confused? Let me explain it.

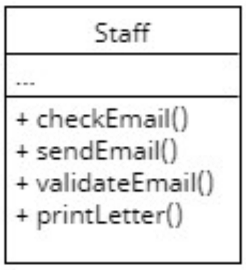
Imagine a shopping cart holding various items (products). Each product might be a subclass of a Product class, inheriting common attributes like price and name. However, each product could have its own calculateDiscount() method overridden to apply specific discount rules (e.g., percentage discount for electronics, buy-one-get-one-free for clothing). This demonstrates how polymorphism allows for diverse behavior based on the specific product type while maintaining a consistent framework for handling products in the shopping cart.

Inheritance helps us in achieving polymorphism and we are going to discuss these concepts in this and next weeks contents with code examples.

we want high cohesion, but low coupling.

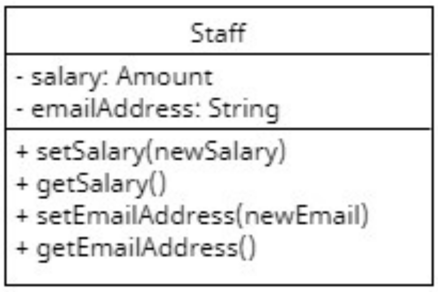
Cohesion and Coupling

Cohesion refers to what the class (or module) can do. Low cohesion would mean that the class does a great variety of actions - it is broad, unfocused on what it should do. High cohesion means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.



Here the staff class doing multiple things such as email and letter processing. This is a Low cohesion class.

As for coupling, it refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and maintain your code; since classes are closely knit together, making a change could require an entire system revamp.



Here Staff is a Highly cohesive file where it is only doing everything related to the staff.

Good software design has high cohesion and low coupling.

Relevant Reading Resources

Relevant Reading Resources

1. WALTER, S. Absolute Java, Global Edition. [Harlow]: Pearson, 2016. (Chapter 7 and 8)
2. SCHILDT, H. Java: The Complete Reference, 12th Edition: McGraw-Hill, 2022 (Chapter 8 and 9)
3. Object Oriented Programming Concepts (accessible on 14-02-2024) The Java Tutorials. Available at: https://docs.oracle.com/javase/tutorial/java/concepts/index.html
4. Shvets, A. Dive Into DESIGN PATTERNS.