Lesson: Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

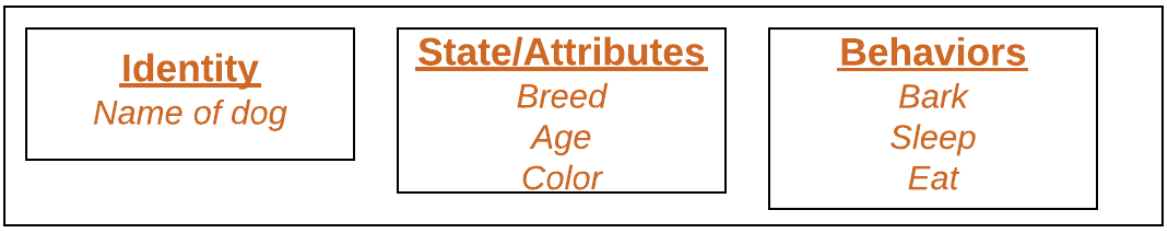
What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State** : It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior** : It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

Example of an object : dog

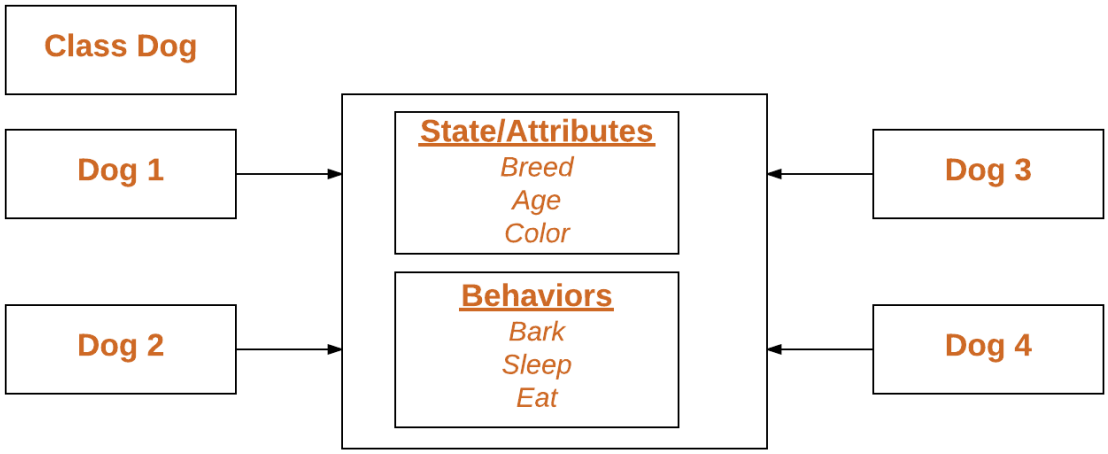


Objects correspond to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behaviour of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example :



As we declare variables like (type name;). This notifies the compiler that we will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variable, type must be strictly a concrete class name. In general,we **can't** create objects of an abstract class or an interface.

```
Dog tuffy;
```

If we declare reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

Initializing an object

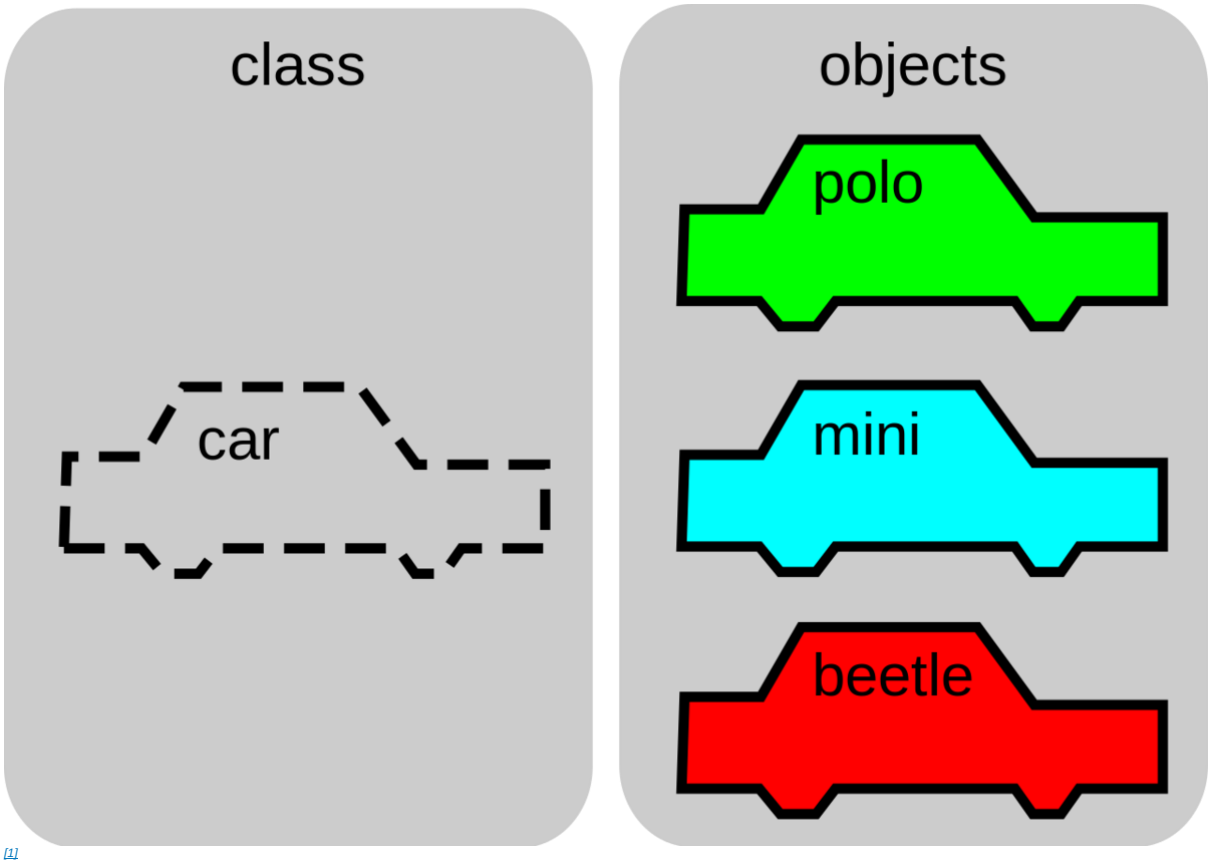
The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

What Is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

In object-oriented programming, a **class** is a blueprint for creating **objects** (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

The user-defined objects are created using the `class` keyword. The class is a blueprint that defines a nature of a future object. An **instance** is a specific object created from a particular class. Classes are used to create and manage new objects and support **inheritance**—a key ingredient in object-oriented programming and a mechanism of reusing code.



[1]

The image above shows how a `Car` object can be the template for many other `Car` instances. In the image, there are three instances: `polo`, `mini`, and `beetle`. Here, we will make a new class called `Car`, that will structure a `Car` object to contain information about the car's model, the color, how many passengers it can hold, its speed, etc. A class can define types of operations, or methods, that can be performed on a `Car` object. For example, the `Car` class might specify an `accelerate` method, which would update the `speed` attribute of the car object.

```
class Car(object):
    def __init__(self, model, passengers, color, speed):
        self.model = model
        self.passengers = passengers
        self.color = color
        self.speed = speed

    def accelerate(self):
        self.speed = self.speed + 2
        print (self.speed)
```

```
11
12 bmw = Car("BMW", 4, "red", 5)
13 ferrari = Car("Ferrari", 2, "black", 10)
14 ford = Car("Ford", 6, "blue", 6)
15
16 bmw.accelerate()
17 print (bmw.color)
18
19 ferrari.accelerate()
20 print (ferrari.color)
21 ferrari.accelerate() #note that the speed has been updated from the previous accelerat
22
23 print (ford.passengers)
24 ford.accelerate()
```

A class is a way of organizing information about a type of data so a programmer can reuse elements when making multiple instances of that data type—for example, if a programmer wanted to make three instances of `Car`, maybe a BMW, a Ferrari, and a Ford instance. The `Car` class would allow the programmer to store similar information that is unique to each car (they are different models, and maybe different colors, etc.) and associate the appropriate information with each car.

The four principles of object-oriented programming are **encapsulation**, **abstraction**, **inheritance**, and **polymorp hism**.
These words may sound scary for a junior developer. And the complex, excessively long explanations in Wikipedia sometimes double the confusion.

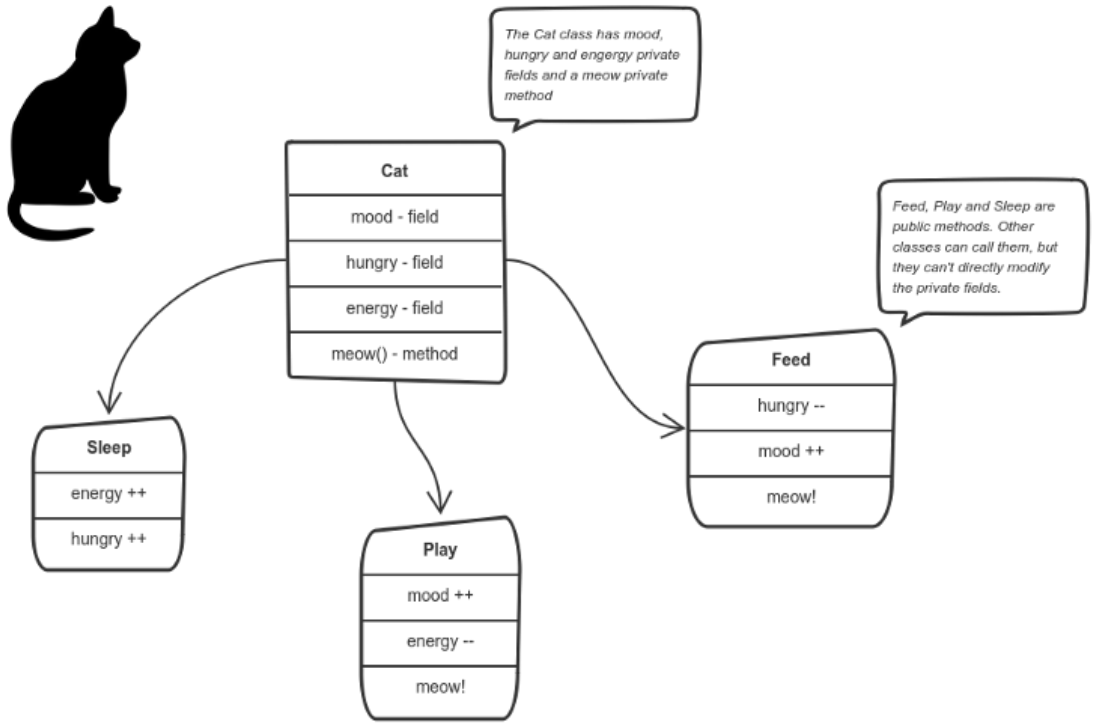
That's why I want to give a simple, short, and clear explanation for each of these concepts. It may sound like something you explain to a child, but I would actually love to hear these answers when I conduct an interview.

Encapsulation

Say we have a program. It has a few logically different objects which communicate with each other — according to the rules defined in the program.

Encapsulation is achieved when each object keeps its state **private**, inside a class. Other objects don't have direct access to this state. Instead, they can only call a list of public functions — called methods.
So, the object manages its own state via methods — and no other class can touch it unless explicitly allowed. If you want to communicate with the object, you should use the methods provided. But (by default), you can't change the state.

Let's say we're building a tiny Sims game. There are people and there is a cat. They communicate with each other. We want to apply encapsulation, so we encapsulate all "cat" logic into a `Cat` class. It may look like this:



You can feed the cat. But you can't directly change how hungry the cat is.

Here the “state” of the cat is the **private variables** `mood`, `hungry` and `energy`. It also has a private method `meow()`. It can call it whenever it wants, the other classes can’t tell the cat when to meow.
What they can do is defined in the **public methods** `sleep()`, `play()` and `feed()`. Each of them modifies the internal state somehow and may invoke `meow()`. Thus, the binding between the private state and public methods is made.
This is encapsulation.

Abstraction

Abstraction can be thought of as a natural extension of encapsulation.

In object-oriented design, programs are often extremely large. And separate objects communicate with each other a lot. So maintaining a large codebase like this for years — with changes along the way — is difficult.

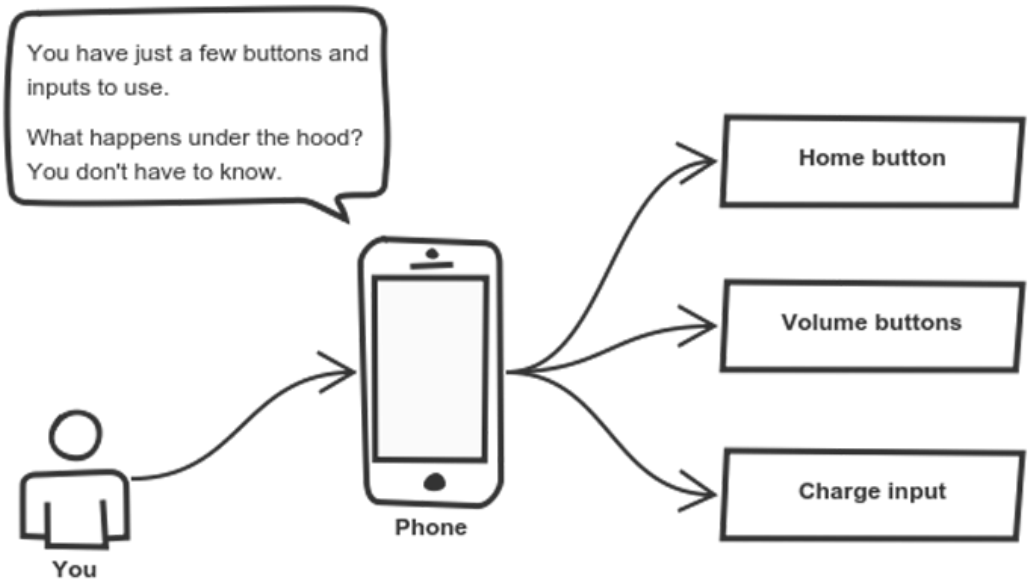
Abstraction is a concept aiming to ease this problem.

Applying abstraction means that each object should **only** expose a high-level mechanism for using it. This mechanism should hide internal implementation details. It should only reveal operations relevant for the other objects.

Think — a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button.

Preferably, this mechanism should be easy to use and should rarely change over time. Think of it as a small set of public methods which any other class can call without “knowing” how they work.

Another real-life example of abstraction?
Think about how you use your phone:



Cell phones are complex. But using them is simple.

You interact with your phone by using only a few buttons. What's going on under the hood? You don't have to know — implementation details are hidden. You only need to know a short set of actions.

Implementation changes — for example, a software update — rarely affect the abstraction you use.

Inheritance

OK, we saw how encapsulation and abstraction can help us develop and maintain a big codebase.

But do you know what is another common problem in OOP design?

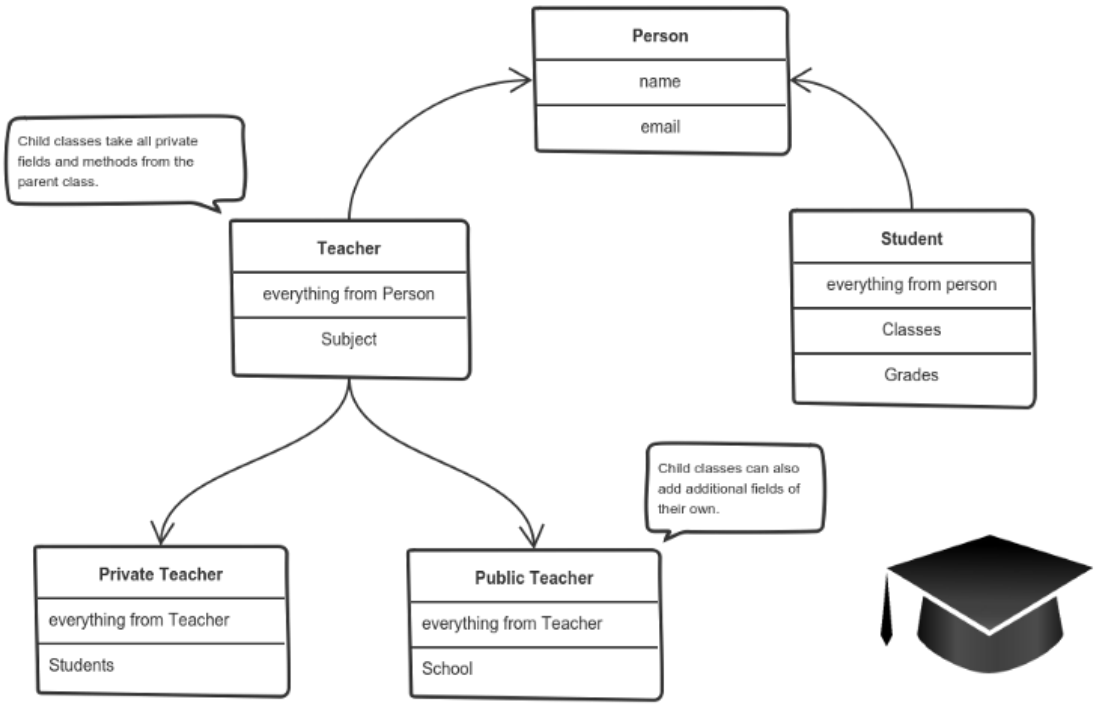
Objects are often very similar. They share common logic. But they're not **entirely** the same. Ugh…

So how do we reuse the common logic and extract the unique logic into a separate class? One way to achieve this is inheritance.

It means that you create a (child) class by deriving from another (parent) class. This way, we form a hierarchy.

The child class reuses all fields and methods of the parent class (common part) and can implement its own (unique part).

For example:



A private teacher is a type of teacher. And any teacher is a type of Person.

If our program needs to manage public and private teachers, but also other types of people like students, we can implement this class hierarchy.

This way, each class adds only what is necessary for it while reusing common logic with the parent classes.

Polymorphism

We’re down to the most complex word! Polymorphism means “many shapes” in Greek.

So we already know the power of inheritance and happily use it. But there comes this problem.

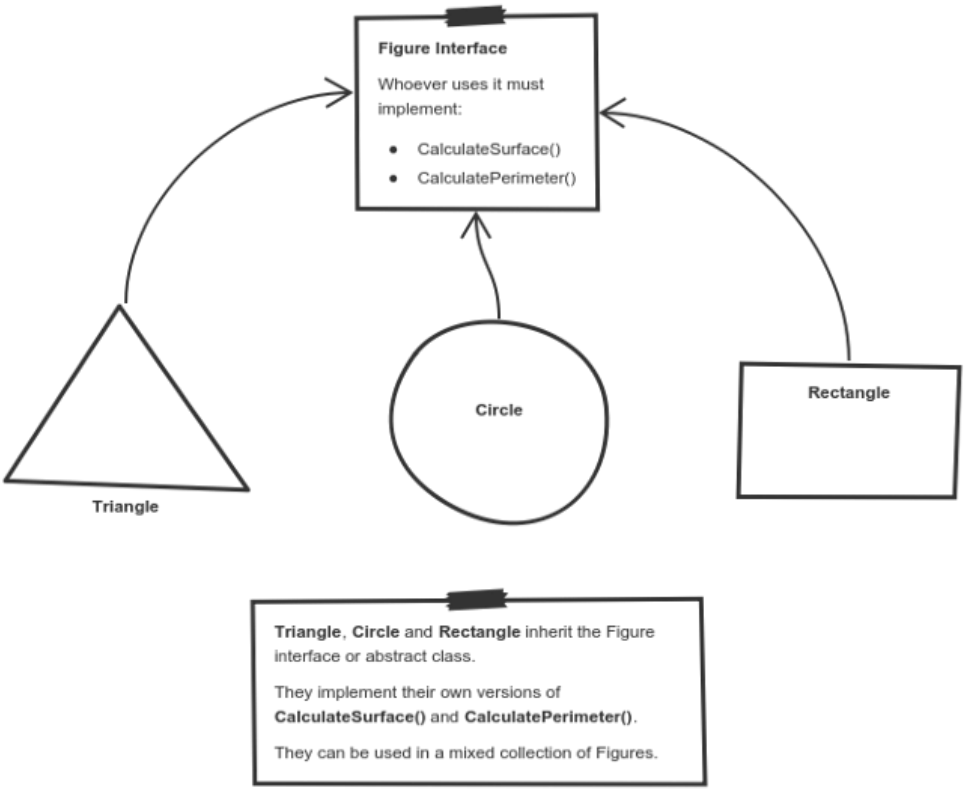
Say we have a parent class and a few child classes which inherit from it. Sometimes we want to use a collection — for example a list — which contains a mix of all these classes. Or we have a method implemented for the parent class — but we’d like to use it for the children, too.

This can be solved by using polymorphism.

Simply put, polymorphism gives a way to use a class exactly like its parent so there’s no confusion with mixing types. But each child class keeps its own methods as they are.
This typically happens by defining a (parent) interface to be reused. It outlines a bunch of common methods. Then, each child class implements its own version of these methods.

Any time a collection (such as a list) or a method expects an instance of the parent (where common methods are outlined), the language takes care of evaluating the right implementation of the common method — regardless of which child is passed.

Take a look at a sketch of geometric figures implementation. They reuse a common interface for calculating surface area and perimeter:



Triangle, Circle, and Rectangle now can be used in the same collection

Having these three figures inheriting the parent `Figure Interface` lets you create a list of mixed `triangles`, `circles`, and `rectangles`. And treat them like the same type of object. Then, if this list attempts to calculate the surface for an element, the correct method is found and executed. If the element is a triangle, triangle's `CalculateSurface()` is called. If it's a circle — then cirlce's `CalculateSurface()` is called. And so on.
If you have a function which operates with a figure by using its parameter, you don't have to define it three times — once for a triangle, a circle, and a rectangle.

You can define it once and accept a `Figure` as an argument. Whether you pass a triangle, circle or a rectangle — as long as they implement `CalculateParamter()`, their type doesn't matter.

What Is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

What Is a Package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.