Introduction to PHP's Object Orientation

Introduction

In this introduction I will try to explain basic object orientation (focused on PHP). It is by no means meant to be a complete guide. There are a lot of concepts I am avoiding for simplicity's sake. I will try to give a theoretical explanation first, and later on provide some code examples.

What is it?

Object orientation is a way of reflecting objects in the real world. If you have a problem that can be described by interacting objects, there's a good change an object oriented approach will help you. It is by no means the holy grail, not every problem requires an OO solution.

Suppose you need to build an access control system. In the real world you'd have at least the following objects available: keycards and cardlocks. So if you need to build something like it, why not use a method that will represent these objects?

Terminology

In the following paragraphs I'll try to provide a short theoretical explanation and example for some key concepts in object orientation.

Classes

A class is basically a blueprint for an object you are modeling. Like a blueprint, it defines what goes where, how you will be able to use it and how big it's going to be.

Example: The blueprint for a keycard will define it's as big as a credit card (physical dimension), that it will contain a magnetic stripe (data track) able to contain information, and a method for retrieving that information.

Properties

Properties (or fields) are used for data storage (and often behaviour control). If you read about object orientation and encounter "Encapsulation", this will be it's simplest example. A class "encapsulates" data in the form of properties.

Example: Properties of a keycard can be: a data track, the manufacturer, it's color.

Methods

Methods (functions) are actions on an object. They can request certain information from an object, or tell an object to perform a certain action.

Example: A cardlock will need to be able to read keycard data, and open and close the actual attached lock.

Visibility

Visibility is a way of access control, determining who can use a specific property or method. Some information is public (everyone can see), other information is protected (we'll get to that in the examples) or private (only the object knows).

Example: You can see a keycard's track, size and color (public). You cannot see what material it is actually made of (private).

Objects

An object is an instance of a class. Just think of it as: a keycard's blueprint is the class, but the actual constructed keycard is the object. You can see that a single class (blueprint) can be used to construct multiple objects (keycards). The objects (keycards) may have the same class (blueprint), but they don't have to be identical: two keycards with the same blueprint will most likely have different unique identifiers (production codes).

Constructor

A constructor defines initial processing when the object is instantiated.

Example: A newly constructed keycard will be imprinted with a unique identifier.

Destructor

A destructor defines final (cleanup) processing when the object is about to be destroyed.

Example: Before destroying a cardlock you will need to detach it from the actual attached lock.

Inheritance

Inheritance is the method of making a new class (child- or sub-class) based on an existing one (parent- or super-class), thus changing or extending the original behaviour.

Example: A new keycard with an added chip will still behave as it's predecessor with the magnetic stripe, but will have additional storage on the chip and will need a method of retrieving that information.

Method Overriding

Method overriding is a way of changing the functionality of a specific function, but keeping the name and parameters intact.

Example: A cardlock will validate a keycard differently in newer models, because security is improved by using a different encryption method.

What else?

Like diafol said: *"taking your procedural code, wrap a class around it and make everything public is NOT object orientation"*. If you want to change your procedural code to OO, rethink your code. See if you can write down the separate objects you are using, what data they contain and how they should interact.

The key guideline here is that **every class should have one clear purpose**. Building a class containing all you need is not the right way to go about it. You should not build an SQL wrapper class that will retrieve your users, calculate your statistics and write the actions to a text file for logging. You should build an SQL class, a user class, a statistics class and a logging class. It is more work initially, but swapping in new functionality (think inheritance and method overriding) will prove much easier. Each class will have a specific purpose, that can more easily be reused in your next OO project, without having to start over from scratch time and time again.

When I first learned OO it was considered very bad practice to use public properties for example. It was considered the right way to provide a public method to retrieve or change the private property (this is where the terms getter and setter come from, not discussed here).
In most languages today you can choose more than just a public property, you can make it read-only for example. This feature is not yet available in PHP, but there's a work-around (not discussed here).

Further Reading

- PHP Manual on objects.

Example Code