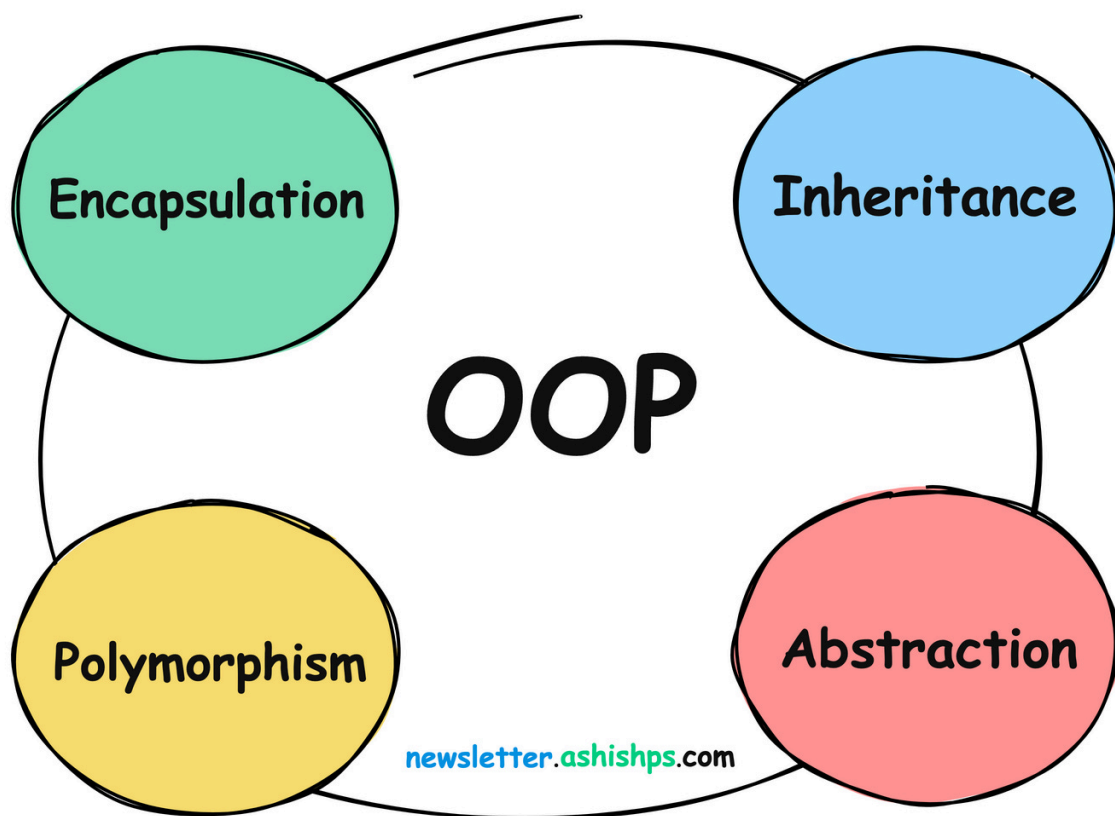


# OOP

object oriented programming

raoul heurneman  
547394



## inhoud:

<b>What is object oriented programming?</b> .....	<b>3</b>
Encapsulation:.....	4
code examples:.....	4
reflection on principle:.....	5
Abstraction:.....	6
code examples:.....	6
reflection on principle:.....	7
inheritance:.....	8
code examples:.....	8
reflection on principle:.....	9
polymorphism:.....	10
code examples:.....	10
reflection on principle:.....	11
<b>general reflection:</b> .....	<b>12</b>
<b>sources:</b> .....	<b>13</b>

## What is object oriented programming?

OOP is a way of coding where instead of making one big messy code document you help to make things more easy to read and so it's less messy. you can look at this like organizing something in a way that is more logical to read or look at.

OOP is made of 4 main subjects:

- encapsulation
- abstraction
- inheritance
- polymorphism

These are all part of OOP but they all serve a different purpose. These 4 'pillars' will help make your code go from some spaghetti mess that someone left behind to something that is readable and easy to understand for others.

We will go over all 4 of these 'pillars' not in the order above, however I have watched a ton of videos and it doesn't seem like OOP has a hierarchy meaning there is no full order to them. These 'pillars' are things you can use to improve your code, there is no guarantee you can actually use all of them in a single part of your code.

## Encapsulation:

you pair a property of something together with the method that it uses, this part we then call an object and thus the name OOP.

as an example in one of the sources they use a car, a car has different properties and different function that make it a car:

properties:

- model
- brand
- colour

functions:

- start
- stop
- move
- remove ceiling(some)

all of these things are used by a lot of different things but we take these properties and functions and label them under 1 object as a car. This is also what we will do with OOP taking code functions and properties and labeling them under 1 object or making them derive from 1 object.

## code examples:

so in my example here we have the first one is a way of making an enemy deal damage to the player.

Here we first define everything, every property from enemy health to enemy attack speed and then use all of those properties inside the function by using parameters that are in the brackets behind function attack. However the most useful part of this is to make a function with the least amount of parameters possible. To achieve this we use encapsulation

```
3   let enemyhealth = 20
4   let enemydamage = 5
5   let enemyAttackSpeed = 0.5;
6   function attack(damage,attackspeed)
7       {
8           return damage * attackspeed;
9       }
10
```

over here we have our changed version to make use of encapsulation. Instead of going first declaring all of the properties we bundle them all up under an object called enemy. We then give the enemy the properties of health, damage and attack speed and the function attack now has no parameters between the brackets as you can see. it uses all of those properties up there but they aren't inside those brackets anymore to make things look ugly.

```
11  let enemy =  
12  {  
13      health: 20,  
14      damage: 5,  
15      attackspeed: 0.5,  
16      attack: function() {  
17          return this.damage * this.attackspeed;  
18      }  
19  }
```

### reflection on principle:

This is the main principle of encapsulation to make things less complex so it's easier to understand for others or for yourself if you look back on your project after a while. It also increases reusability by making it very easy to copy this to another project. If I ever need another enemy somewhere in this project or another I can make use of this piece of code and just copy and paste then change the values to fit the new game and be done.

## Abstraction:

abstraction we can see as hiding things we don't really care about or things that are unnecessary to know. You can think of it as helping your parents or grandparents with something computer related, you will fix it for them but you won't need to explain all the details cause so long as it works that's what matters to them.

Abstraction focuses on getting things out of the way and only showing what is necessary. so if we use the same example from above with our enemy. All of that code will be in a enemy script and all we will have to do is call that function from another script

## code examples:

This is our enemy script in this case with the function attack. you would normally call this script "Enemy"

```
11  let enemy =  
12  {  
13      health: 20,  
14      damage: 5,  
15      attackspeed: 0.5,  
16      attack: function() {  
17          return this.damage * this.attackspeed;  
18      }  
19  }
```

you would then use this to call the attack method in another script for example if you want the enemy to attack at a certain point you just call this.

The capitalized enemy at the start will be the name of your script then the lowercase enemy is your object we assigned up there. we then make a new enemy in the scene and we make that enemy run the attack function that we established earlier.

```
Enemy enemy = new Enemy();  
  
enemy.attack();
```

### reflection on principle:

So this is the principle of abstraction. This is another way just like encapsulation to help reduce complexity in your code. It will also help reduce the risk of changes, if you change something in a spaghetti code (term for bad code) there is a chance that a small change will make another function not work and that makes another thing not work leading to your whole code breaking, this should help prevent that since not all your code is accessing your other code so to break it is a lot more difficult.

## inheritance:

inheritance is a way of copying code or things over. to make a real world example is if you are sitting on your couch and watching tv or doing anything else. Sometimes you need to go stand up to walk to the toilet or to walk to the kitchen to make something to drink. so to go to the toilet you do:

- stand up
- walk to toilet
- do that something
- walk back to the couch
- sit down

and for the kitchen you do:

- stand up
- walk to kitchen
- get something to drink or eat
- walk back to the couch
- sit down

Now as you can see there are a few things the same. Those 3 things would be the main class and the things that are not the same are a separate change to that specific action. to go back to our normal explanation we have our enemy but if you have ever played a game, not all enemies are the same so that's where inheritance comes from.

## code examples:

This is once again our enemy class for a standard enemy. with the same properties as the one before. Now if I want a second enemy that uses the same attack function but it's a bit less tanky so a bit less health and deals more damage but at a slower rate I could type everyone over again or I could use inheritance.

```
11  let enemy =
12  {
13      health: 20,
14      damage: 5,
15      attackspeed: 0.5,
16      attack: function() {
17          return this.damage * this.attackspeed;
18      }
19  }
```

now here you can see enemy2 is our normal enemy but after that i change the properties of enemy2 so it has less hp but deals more damage while attacking slower but it still returns the attack function only now with the properties of enemy2

```
let enemy2 = enemy
{
  enemy2.health = 10
  enemy2.damage = 10
  enemy2.attackspeed = 0.1
  return enemy2.attack();
}
```

### reflection on principle:

This is inheritance. It uses the same function and everything, however we just change a few properties or you could even do it the other way around where it has the same properties but it deals damage in a different way.

we use inheritance to eliminate over typing things so we aren't typing the same thing over and over. This manner eliminates redundant code as it is called.

you will quickly realise that this is an option you can use but isn't always needed. This method mostly starts shining in big projects where there is a lot going on and a lot being made and you don't want to waste time writing the same thing again and again. If you are making a small project you could also choose to skip this step however I would say you should always use this for making things easier to read and more navigable but don't over do it not everything needs to be a new script!



## polymorphism:

polymorphism stands for poly meaning many and morph meaning form or forms. This technique is used to get rid of long if, else statements or long switch statements because who are we kidding those aren't nice to look at. Okay so for a real world example imagine you want to light a candle, you want to find something that makes a flame. first you will look for a lighter if you don't have that then u use matches if you even don't have that well you either have to buy them or go cavemen style and use a flint.

In code polymorphism comes from inheritance. if you try to call something from the parent object or main class but it's overwritten in a child object or subclass then it will run that. That might sound confusing so let's get the code examples.

## code examples:

here we have our main enemy once again now if i run the attack function it will run this code making it go  $\text{damage} * \text{attack speed}$  so  $5 * 0.5$ . That's for our main enemy. now if i want to run this but it has been changed in the subclass we cant run this it has to run the other one.

```
11  let enemy =
12  {
13      health: 20,
14      damage: 5,
15      attackspeed: 0.5,
16      attack: function() {
17          return this.damage * this.attackspeed;
18      }
19  }
```

Now as you can see, first we make enemy number 3 the same as the main enemy. Then we change some properties and then we make the main enemy the same as enemy number 3. If we now return enemy.attack it will check our main class for any changes. Now our attack function is the same so that one is gonna run the one from our main class however because we changed some properties so those are changed. so the enemy that it runs now has no changes to the health so it's still 20, it does have changes to the damage so that's now 10 and has no changes to the attack speed so that's still 0.5 so the attack function will now return  $10 * 0.5$  instead of what it used to do.

```
let enemy3 = enemy
{
  enemy3.damage = 10
  return enemy3.attack();
}

let enemy = enemy3
{
  return enemy.attack();
}
```

### reflection on principle:

This is polymorphism so instead of manually writing out if it has changed the program does that all for you and this is called refactoring. polymorphism really starts to shine if things are for example in a list so if you list all the enemies we created in a list called enemy and then call the attack function for all the entries it will call all the enemies in the list with the correct changes and if there is or isn't something changed it will use the right values.

## general reflection:

The four methods are listed below with the benefits they all give. I have already been using some of these methods unknowingly but now knowing that they exist makes it easier to reuse next time I am making a project so I can also make my code more readable and more useful for others. Of course this document is a way of showing what I found out by watching videos, reading articles and a lot of trying for myself.

Encapsulation	Reduce complexity + increase reusability
Abstraction	Reduce complexity + isolate impact of changes
Inheritance	Eliminate redundant code
Polymorphism	Refactor ugly switch/case statements

I did change some of the ways I did this project. from my first hearing of this i thought that i would make a simple game where i show all of the principles and what and how they work however after going through some research i found out that physically showing these changes in a game is quite difficult and it's mostly just principles to make my code look better so instead of making a game to show these principles i made this almost tutorial document to show you, and probably me if i ever forget, how they work and what to use them for.

Now I did realize that the way I was making this was not gonna last me the 60 hours outside of class time to make this. so i decided that practice and practice and a bit more practice was the way i was gonna fill the rest of the time. This still aligned with my research problem or question that was approved since in there I stated that I wanted to make it my own so I would be able to use all 4 principles at will if I wanted or needed to.

For this I used two sites, one called code wars, which was a beginner's step where people make problems based on something and then you have to fix them. On this site there is a tag for OOP so all of the problems on there are needed to be fixed by using one or more of the principles. The other site I used is LeetCode which is a tutorial or training site where you also get a problem and then have to fix them. The design section on this site is fully focused on making your code look better and thus uses the principles of OOP as well. These two were a big help and completing these things made me understand more and more how the principles worked. i do have to say that the codewars one was easier to get into than the leetcode ones since they were a bit of a higher difficulty. the nice thing about codewars as well is that you can see how others fixed it after you are done and see how they did it different and you can learn from that as well(or not there are some people on there just looking to make things difficult) unfortunately

I can't share my answers here since those are locked in the website, because you first need to fix the problem to see other answers. However, it took me a good few problems and research before I started making good progress and getting things right. In the second week on Tuesday was the first one I got right in one go. and from then on it started going easier and easier and I was able to finish most of the problems handed to me in a reasonable time(1-2 hours) and in one go which makes me think that I have succeeded in my goal. In general I think I will benefit a lot from making and learning about this. hopefully making it easier later in this study to make group projects and things like that, and also just generally making my code look better.

## sources:

codewars:

*Kata Practice*. (z.d.). Codewars.

[https://www.codewars.com/kata/search/my-languages?q=&tags=Object-oriented%20Programming&beta=false&order\\_by=sort\\_date%20desc](https://www.codewars.com/kata/search/my-languages?q=&tags=Object-oriented%20Programming&beta=false&order_by=sort_date%20desc)

leetcode:

<https://leetcode.com/problem-list/design/>

programming with mosh:


Programming with Mosh. (2018b, maart 30). *Object-Oriented programming,*

*simplified* [Video]. YouTube.  Object-Oriented Programming, Simplified

keep on coding:

Keep On Coding. (2020, 27 februari). *Object Oriented Programming - The Four*


*Pillars of OOP* [Video]. YouTube.

 Object Oriented Programming - The Four Pillars of OOP

awesome Tuts - anyone can learn to make games:

Awesome Tuts - Anyone Can Learn To Make Games. (2018, 14 oktober). *How object*

*oriented programming works in Unity* [Video]. YouTube.

 How Object Oriented Programming Works In Unity

wikipedia:

Wikipedia contributors. (2025, 20 mei). *Object-oriented programming*. Wikipedia.

[https://en.wikipedia.org/wiki/Object-oriented\\_programming](https://en.wikipedia.org/wiki/Object-oriented_programming)

generation it:

It, G. (2024, 6 augustus). *Object oriented programming (OOP), wat is dat?*

Generation IT. <https://generationit.nl/object-oriented-programming-oop/>

techtarget:

Techtarget. (14 juni). *Object-Oriented Programming (OOP). what is it?*

<https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP>

tuple:

Tuple. (2025, 15 april). *Object-Oriented Programming (OOP)*. Tuple.

<https://www.tuple.nl/nl/kennisbank/object-oriented-programming>