

(Better)
**Object
Oriented
Programming**

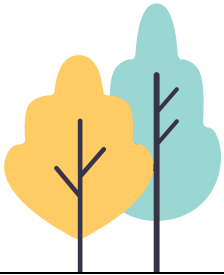
The Royal Edition.

Based on the OOP lecture of Harvard's CS50 course [1]

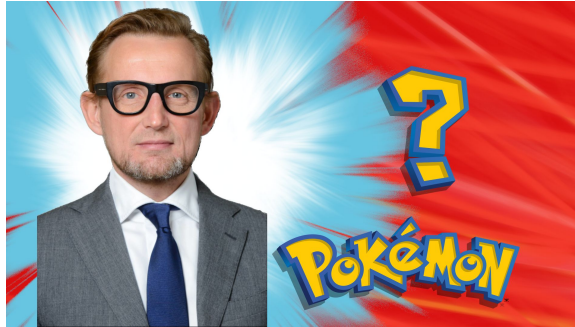


Better OOP → Want wij doen het beter dan CS50
Alle basic features van OOP.
Royal edition

Who this?



Who this?



Het is Prins Bernhard Jr.!

- Pandjesbaas met 600 huizen in deelbezit, waarvan 100 in Amsterdam [1]
- Zit niet in de top-300 met maar 27 privé panden [2,3]
- Heeft recent 15 leuke nieuwe vertrekjes in Maastricht gekocht [4]

Bernhard heeft je hulp nodig! Hoe kan hij zijn administratie bijhouden?


600 huizen deelbezit, waarvan 100 in Amsterdam

27 huizen privé → Geen top 300!

Dus 15 erbij gekocht in Maastricht, nu administratie nodig

Nieuwe pandjes, en wij gaan een administratie voor em bouwen.

Prins Bernhards wensen



Locatie

Stad van het pand

Prijs

Huurprijs van het pand

Niet te duur

Wij mogen niet te veel kosten

Mooie code!

Het liefst OOP..

4

In de administratie nodig:

- Locatie
- Prijs

Wij mogen niet te duur zijn als developers

De code moet overzichtelijk en netjes → OOP

01

Wat is OOP?

Alles is een object.

02

Waarom OOP?

Overzichtelijk!

03

Classes & Instances

Aanmaken van objecten.

04

Getters & Setters

Hanteren van variabelen.

05

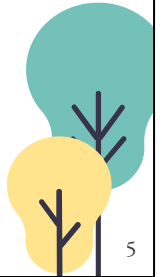
Class Methods

Functies die een instantie
aangepassen.

06

Inheritance

Class 2 is een uitbreiding van
Class 1.



1. Wat is OOP?
 - a. Object georiënteerd → Je schrijft een **Class** definitie, waarvan je meerdere **instanties** kan maken (Objecten)
2. Waarom OOP?
 - a. Overzichtelijk en gecompartmenteerd.
3. Classes & Instances
 - a. Voorbeelden van class definities + aanmaken van instances
4. Getters & Setters
 - a. Het verkrijgen en aanpassen van variabelen van instances
5. Class Methods
 - a. Functies die behoren tot de **Class** definitie, maar werken op 1 instantie
6. Inheritance
 - a. Classes kunnen elkaar uitbreiden → Handig voor vergelijkbare functionaliteit (zoals bijv. huizen & appartementen)

Object Oriented Programming

Een overzichtelijke manier voor het verwerken van gelijksoortige variabelen.

Standaard:

- **Dictionaries** met vars → (locatie, prijs, oppervlak, eigenaar)
- Functies die dictionaries aanpassen

OOP:

- Een algemene **Class** met de **properties**.
- **Instanties** met verschillende waarde voor properties
- Funties die op instantie niveau werken



Standaard:

- Een manier om classes na te bootsen is dictionaries voor ieder huis.

OOP:

- Voor OOP hebben we 1 algemene definitie en meerdere instanties.
- Een variabele wordt een **property** (eigenschap) genoemd
- **Functies** kunnen tot een class behoren

Waarom OOP?

STANDAARD

OOP

```
7 def get_house():
8     location = input("City: ")
9     price = int(input("Price: "))
10    return (location, price)
11
12
13 def main():
14     house = get_house()
15     if house[0] == "Amsterdam" and house[1] < 500:
16         house[1] = 1000
17
18     print(f"{house[0]} for {house[1]}")
19
20
21 if __name__ == "__main__":
22     main()
```

Raises error!

7

Algemeen:

- We willen huizen bijhouden
- Eerst de locatie dan de prijs, die we van de user opvragen
- Minimale prijs is 1000
- Dan printen we de locatie en prijs

Tuple:

- Wat gaat er fout? → Het aanpassen van **house[1] = 1000** op regel **14** → Tuples zijn immutable, dus error

Dict:

- Met dictionaries kan dit wel, die zijn mutable.
- Één dictionary per huis

OOP:

- We hebben een **Class** House, en vervolgens maken we een **Instantie** van die class.
- Properties kan je bij door simpelweg op het object **house.location** te doen

Laat op bord zien hoe meerdere instanties werken.

Waarom OOP?

STANDAARD

```
6 def get_house():
7     location = input("City: ")
8     price = int(input("Price: "))
9     return {"location": location, "price": price}
10
11
12 def main():
13     house = get_house()
14     if house["location"] == "Amsterdam" and house["price"] < 500:
15         house["price"] = 1000
16
17     print(f"{house['location']} for {house['price']}")
18
19
20 if __name__ == "__main__":
21     main()
```

OOP

```
6 class House:
7     ...
8
9
10 def get_house():
11     house = House()
12     house.location = input("City: ")
13     house.price = int(input("Price: "))
14     return house
15
16
17 def main():
18     house = get_house()
19     if house.location == "Amsterdam" and house.price < 500:
20         house.price = 1000
21
22     print(f"{house.location} for {house.price}")
23
24
25 if __name__ == "__main__":
26     main()
```

8

Algemeen:

- We willen huizen bijhouden
- Eerst de locatie dan de prijs, die we van de user opvragen
- Minimale prijs is 1000
- Dan printen we de locatie en prijs

Tuple:

- Wat gaat er fout? → Het aanpassen van **house[1] = 1000** op regel 14 → Tuples zijn immutable, dus error

Dict:

- Met dictionaries kan dit wel, die zijn mutable.
- Één dictionary per huis

OOP:

- We hebben een **Class** House, en vervolgens maken we een **Instantie** van die class.
- Properties kan je bij door simpelweg op het object **house.location** te doen

Laat op bord zien hoe meerdere instanties werken.

Classes zijn een definitie, een **Object** is een instantie van deze definitie.
De `__init__` functie heet ook wel de **constructor**.

```
6 class House:
7     def __init__(self, location, price):
8         self.location = location
9         self.price = price
10
11
12 def get_house():
13     location = input("City: ")
14     price = input("Price: ")
15     return House(location, price)
16
17
18 def main():
19     house = get_house()
20     print(f"{house.location} for {house.price}")
21
22
23 if __name__ == "__main__":
24     main()
```

Hoe werkt het aanmaken van zo'n instantie?

Je schrijft een `__init__` functie, ook wel de **constructor** genoemd.
De argumenten zijn altijd (**self**, **arg1**, **arg2**, ..)

Als je `House("Amsterdam", 500)` uitvoert maak je dus 1 instantie.

Error handling kan al tijdens het aanmaken van een object via **raise**.

```
6 class House:
7     def __init__(self, location, price):
8         if location not in ["Amsterdam", "Utrecht", "Maastricht"]:
9             raise ValueError("Given city is not supported.")
10
11         self.location = location
12         self.price = price
13
14
15 def get_house():
16     location = input("City: ")
17     price = int(input("Price: "))
18     return House(location, price)
19
20
21 def main():
22     house = get_house()
23     print(f'{house.location} for {house.price}')
24
25
26 if __name__ == "__main__":
27     main()
```

Je kan in deze **__init__** ook foutieve invoer afvangen.

Gebruik **raise** voor het maken van errors. → Als je een error krijgt vanuit je programma gebeurt dit ook met **raise**!

Er zijn verschillende errors, **ValueError** is daar een voorbeeld van.

String representatie kan aangepast worden via `__str__`.

```
6 class House:
7     def __init__(self, location, price):
8         if location not in ["Amsterdam", "Utrecht", "Maastricht"]:
9             raise ValueError("Given city is not supported.")
10
11         self.location = location
12         self.price = price
13
14     def __str__(self):
15         return f"{self.location} for {self.price}"
16
17
18 def get_house():
19     location = input("City: ")
20     price = int(input("Price: "))
21     return House(location, price)
22
23
24 def main():
25     house = get_house()
26     print(house)
27
28
29 if __name__ == "__main__":
30     main()
```

Je kan met een `__str__` functie aanpassen hoe het object geprint wordt.
Standaard is dit het geheugen adres en de functie waarbinnen hij geprint wordt →
<__main__.House object at 0x7f9b5738c550>

Getters & Setters

Functies binnen een Class die de properties verwerken.
Handig voor extra functionaliteit en checks!



Getters & Setters zijn ervoor om met properties te werken.

Direct oproepen en aanpassen van properties wordt afgeraden, omdat je dan niet op één plek de logica kan wijzigen van een correct object.

Denk bijvoorbeeld aan de check op locatie!

Getters & Setters

STANDAARD

```
6 class Houses:
7     def __init__(self, location, price):
8         if location not in ["Amsterdam", "Utrecht", "Maastricht"]:
9             raise ValueError("Given city is not supported.")
10
11         self.location = location
12         self.price = price
13
14
15 def get_house():
16     location = input("City: ") # Enter "Amsterdam"
17     price = int(input("Price: "))
18     return House(location, price)
19
20
21 def main():
22     house = get_house()
23     print(house.location) # Prints "Amsterdam"
24     house.location = "Eindhoven"
25     print(house.location) # Prints "Eindhoven"
26
27
28 if __name__ == "__main__":
29     main()
```

GETTERS & SETTERS

```
6 class House:
7     valid_cities = ["Amsterdam", "Utrecht", "Maastricht"]
8
9     def __init__(self, location, price):
10         if location not in self.valid_cities:
11             raise ValueError("Given city is not supported.")
12
13         self._location = location
14         self._price = price
15
16     @property
17     def location(self):
18         return self._location
19
20     @location.setter
21     def location(self, location):
22         if location not in self.valid_cities:
23             raise ValueError("Given city is not supported.")
24         self._location = location
25
26
27 def get_house():
28     location = input("City: ") # Enter "Amsterdam"
29     price = int(input("Price: "))
30     return House(location, price)
31
32
33 def main():
34     house = get_house()
35     print(house.location) # Prints "Amsterdam"
36     house.location = "Eindhoven" # Raises ValueError
37
38
39 if __name__ == "__main__":
40     main()
```

13

Links zie je hoe je properties kan aanpassen zonder getters en setters. Bij het aanmaken zijn alleen de steden **Amsterdam**, **Utrecht**, en **Maastricht** toegestaan. Maar in de main kunnen we de locatie aanpassen naar **Eindhoven**.

Om dit af te vangen kunnen we binnen de functie een Setter definiëren. Hierin kan je dezelfde logica als voor de `__init__` verwerken.

Let op de **decorators** boven de functies. Dit laat Python weten wat de functionaliteit van de onderstaande functies zijn.

@property

@location.setter

Ze werken als **wrappers** voor degene die er meer over willen weten.

Het is dan ook netjes om een getter te maken die deze variabele opvraagt.

Let op het gebruik van **self._location**, de underscore wordt gebruikt om aan te geven dat variabelen (en functies!) bedoelt zijn voor **intern** gebruik.



Class Methods

Functies binnen een Class die aangeroepen kunnen worden door een **Instantie**.



Class methods zijn functies binnen classes.
Deze hoeven **niet** met een instance te werken, kan ook losstaand zijn!

Class Methods

Class methods zijn functies die uitgevoerd kunnen worden zonder een instantie te maken.

```
6 class House:
7     valid_cities = ["Amsterdam", "Utrecht", "Maastricht"]
8
9     @classmethod
10    def is_valid(cls, city):
11        if city in cls.valid_cities:
12            return True
13        else:
14            return False
15
16
17 def main():
18     city = "Amsterdam"
19     city_valid = House.is_valid(city)
20     print(city_valid) # Prints "True"
21
22     city2 = "Eindhoven"
23     city2_valid = House.is_valid(city2)
24     print(city2_valid) # Prints "False"
25
26
27 if __name__ == "__main__":
28     main()
```

15

Hier zie je een voorbeeld van een **class method** die zonder instantie werkt.

Je gebruikt de **@classmethod** decorator.

Self is hier vervangen voor **cls** omdat het niet via een instantie wordt aangeroepen, maar via de class.

De validiteit van een stad kan zo via een functie bepaald worden.

Je kan deze functies aanroepen via de **Class**, zonder een **instantie** aan te maken.

Class methods kunnen wel de **constructor** van de Class aanroepen.

```
4 class House:
5     def __init__(self, location, price):
6         if location not in ["Amsterdam", "Utrecht", "Maastricht"]:
7             raise ValueError("Given city is not supported.")
8
9         self.location = location
10        self.price = price
11
12        @classmethod
13        def get(cls):
14            location = input("City: ")
15            price = input("Price: ")
16            return House(location, price)
17
18
19 def main():
20     house = House.get()
21     print(f"House is in {house.location}")
22
23
24 if __name__ == "__main__":
25     main()
26
```

Het is mogelijk om in deze class method de **constructor** aan te roepen.

Inheritance

Een Class kan op een andere Class voortbouwen door de **properties** en **methods** te "erven".



Inheritance kan gebruikt worden om een class **uit te breiden** met een andere class. Class 2 is dan een **extensie** van class 1.

Een Class (**Apartment**) kan de **properties** en **functies** van de Class **House** overnemen, en daarnaast zijn eigen definiëren.

```
6 class House:
7     def __init__(self, location, price):
8         if location not in ["Amsterdam", "Utrecht", "Maastricht"]:
9             raise ValueError("Given city is not supported.")
10
11         self.location = location
12         self.price = price
13
14
15 class Apartment(House):
16     def __init__(self, location, price, kind):
17         super().__init__(location, price)
18
19         if kind not in ["studio", "loft", "duplex"]:
20             raise ValueError("Apartment type is invalid.")
21
22         self.kind = kind
23
24
25 def main():
26     house = House("Amsterdam", 750) # Creates House
27     apartment1 = Apartment("Amsterdam", 750, "studio") # Creates Apartment
28     apartment2 = Apartment("Amsterdam", 750, "micro") # ValueError Apartment
29     apartment3 = Apartment("Eindhoven", 350, "studio") # ValueError House
30
31
32 if __name__ == "__main__":
33     main()
```

Hier zie je een nieuwe class **Apartment** die de functionaliteit van **House** overerft, en vervolgens extra functionaliteit toevoegt.

Regel 15 met **super().__init__(..)** is nodig om de functionaliteit van **House** uit te voeren.

RESOURCES

- [1] <https://hollywoodhuizen.nl/pandjeskoning-bernhard/>
- [2] <https://www.volkskrant.nl/mensen/waarom-prins-bernhard-niet-tot-de-grootste-verhuurders-van-nederland-behoort-b384729d/>
- [3] <https://www.ad.nl/wonen/prins-bernhard-hoort-niet-bij-grootste-huizenbezitters-aed4a62d/>
- [4] <https://nieuwspaal.nl/prins-bernhard-koopt-15-pandjes-tijdens-bezoek-aan-maastricht/>
- [i] <https://cs50.harvard.edu/python/2022/notes/8/>



Dat was em!

Prins Bernhard Jr. is blij!
Nog vragen?

CREDITS: This presentation template was created by **Slidesgo**,
including icons by **Flaticon**, and infographics & images by **Freepik**.
Please keep this slide for attribution.

