JS Advanced & OOP concepts

FullStack VDO

Object Oriented Programming: de praktische theorie

In de applicaties die je in het verleden hebt ontwikkelt heb je vast geprobeerd structuur aan te brengen in je code. Je hebt je bestanden waarschijnlijk gegroepeerd in in een bepaalde structuur, bijvoorbeeld een map images voor je afbeeldingen en een map includes voor je scripts die je include. Waarschijnlijk gebruik je ook functies voor acties die je meerdere keren moest doen.

Groeperen en verzamelen

Onthoud een voorbeeld van een Volkswagen Polo, een Ferrari en een Ford Focus. Je ziet in één opslag dat het allemaal auto's zijn. Ongetwijfeld heb je er niet aan gedacht aan een verzameling van schroeven, ramen en wielen zijn. De eigenschap van de hersenen om voorwerpen te groeperen en te verzamelen als een voorwerp, gebruiken we ook bij het OOP programmeren.

Eigenschappen en methodes

Stel, we moeten een routeplanner programmeren. We gebruiken verschillende vervoersmiddelen, waar we de route vanaf laten hangen. We introduceren het object auto in onze applicatie. Deze auto heeft eigenschappen die van belang zijn in onze applicatie: snelheid, actieradius enzovoorts. Onze routeplanner hoeft niet te weten hoe de motor is gemaakt

Het gewicht van de auto is ook is het niet van. Daarentegen, wanneer we een programma moesten schrijven voor een automonteur, had dat wel van belang geweest. Bij OOP werken we objecten alleen uit tot het detailniveau dat we nodig hebben.

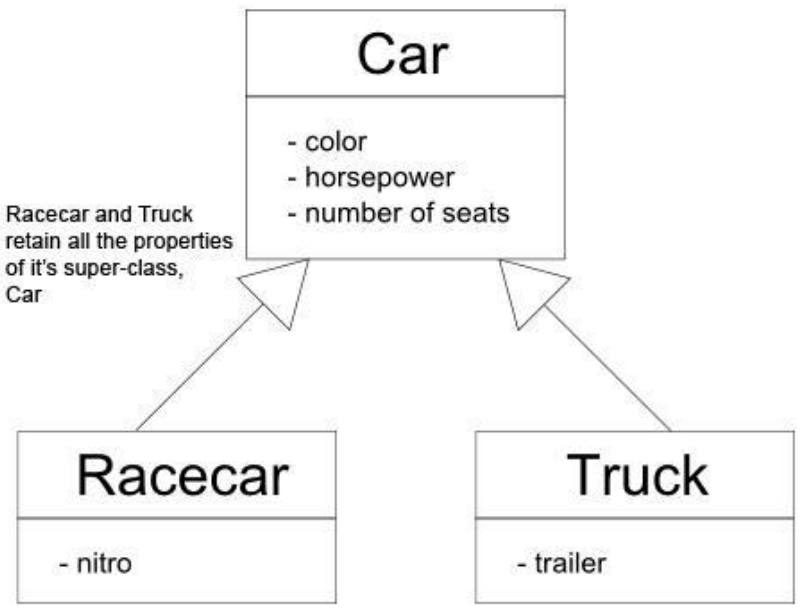
Dit zorgt voor extra overzicht en besparen we ons de last van erg veel ballast.
Een ander belangrijk kenmerk van OOP is de herbruikbaarheid van de code. Wanneer je een class hebt geschreven voor een applicatie, kan je deze moeiteloos hergebruiken in een andere applicatie. Het voordeel hiervan benodigd geen toelichting. Alsook het immense voordeel van de reeds ontwikkelde classes.

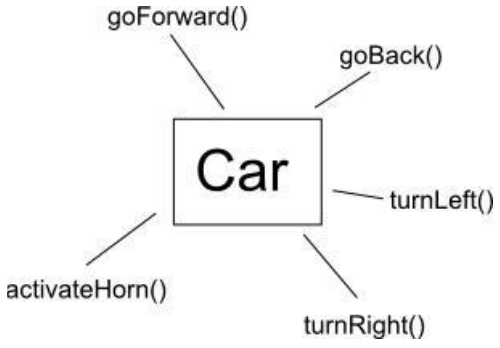
OOP Voordelen

Leesbaarheid
Door de modulaire opbouw is het eenvoudig het overzicht te bewaren.

Onderhoudbaarheid
Vanwege de duidelijke structuur kan je gemakkelijker het systeem aanpassen. Vanwege de modulaire opbouw hoef je niet telkens het hele systeem om te gooien.

Herbruikbaarheid
Door de modulaire opzet kan je gemakkelijk code hergebruiken in een volgend project. Na een tijdje zul je merken dat je een paar modules hebt die je in veel applicaties gebruikt.





Object Oriented Programming: de technische theorie

De 2 belangrijkste begrippen van OOP zijn **klasse en object**. Een object is een **instantie** van een van een class. In ons auto voorbeeld kan je een class zien als het ontwerp van een auto en een object als de bouw ervan. Er kunnen dus meerdere objecten van één klasse worden gemaakt. Immers kunnen er ook meerdere auto's van één ontwerp worden gemaakt.

Een class bestaat uit eigenschappen. Dit is een verzameling voor methodes, attributen en constanten.

Een methode kan je vergelijken met een functie. Een auto kan bijvoorbeeld rijden of remmen.

Een attribuut is een variabele van een class. Deze kan niet van buiten de class worden benaderd, dit moet via de class zelf. Bij een auto is dit bijvoorbeeld de kleur...

Een constante is een constante van een class. Deze is, net als een attribuut niet van buiten de class benaderbaar...

inheritance, polymorphism, encapsulation, abstraction

Een klasse kan een andere class uitbreiden, met vaak een specialisatie. De nieuwe klasse erft dan sommige eigenschappen van de SuperClass, de class die hij uitbreidt. De simpelste vorm van overerving is Single **Inheritance**, oftewel Enkele Overerving.

De class, bijvoorbeeld Ferrari, die een uitbreiding is op onze Auto class, is een voorbeeld van Single Inheritance. Wanneer we onze Ferrari class nog verder willen uitbreiden naar verschillende types, spreken we van Multiple Inheritance. We kunnen onze class Ferrari bijvoorbeeld uitbreiden in een class FerrariF430 en een class FerrariEnzo. Deze twee klassen zullen dan nog steeds eigenschappen hebben van de SuperClass auto, maar ook de specialisatie van de Ferrari class.

Wanneer we een class laten overerven van een andere class, heb je al gauw nodig dat een bepaalde methode zich iets anders gedraagt. Je kan nieuwe methodes en attributen toevoegen, maar wat wanneer je een bepaalde methode iets wilt wijzigen?

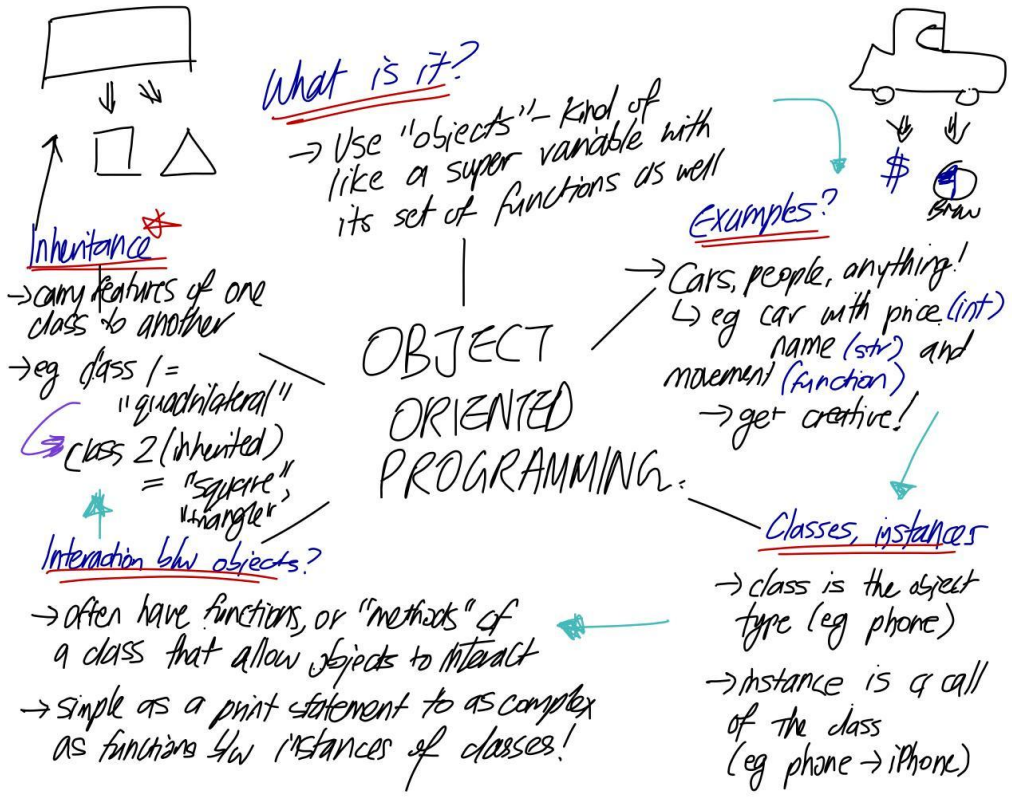
Het is verkeerd om de SuperClass te veranderen, omdat hier immers meerdere klassen van kunnen erven en hij al naar behoren werkt. Daar is in OOP een begrip voor: polymorfisme. Met **polymorfism** kan je bepaalde methodes opheffen en ze voor de nieuwe class specifiek aanpassen.

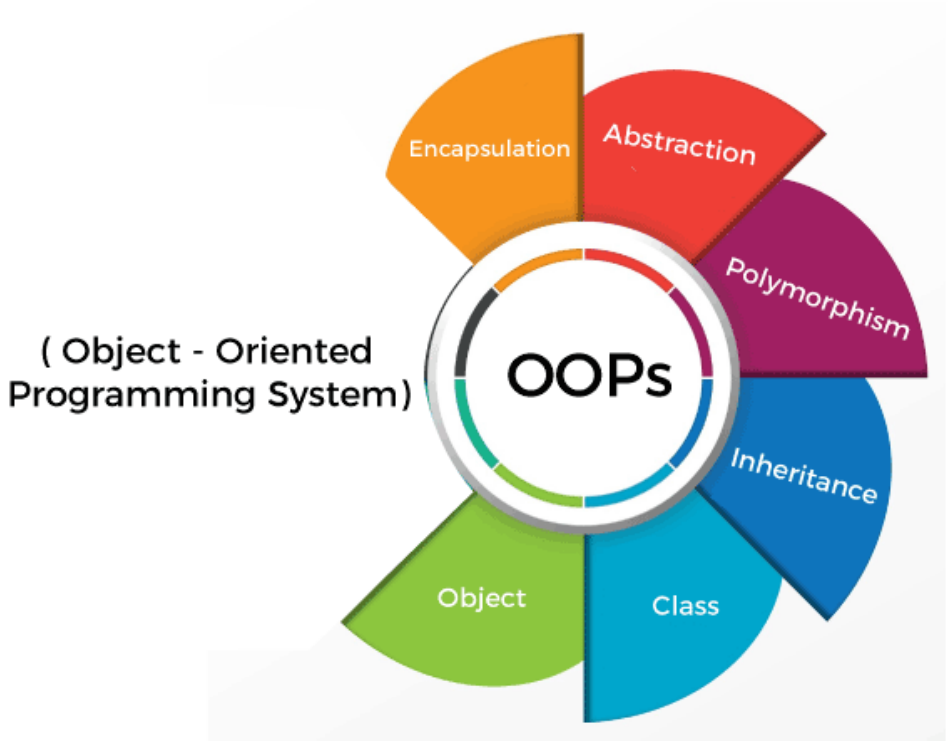
Gerelateerd aan overerving hebben we het begrip encapsulation, oftewel inkapseling.

Een ander woord voor inkapseling is Information Hiding. Een auto kan bijvoorbeeld remmen en gas geven. De bestuurder van de auto kan dit allemaal doen zonder te weten hoe de auto het precies doet.

Encapsulation/abstraction kan je definiëren als het afscheiden en het verbergen van hoe iets gebeurt. Een goed ontworpen object kan doen wat je wilt zonder dat je ooit weet hoe het gebeurt. Inkapseling wordt bereikt door het zichtbaarheid of toegangsbeheer. Hiermee specificeer je hoe toegankelijk eigenschappen van klassen zijn (public/private)

https://www.freecodecamp.org/news/object-oriented-javascript-for-beginners/#the-four-principles-of-oop


What is it?
→ Use "objects" - kind of like a super variable with its set of functions as well
OBJECT ORIENTED PROGRAMMING:
Inheritance
→ carry features of one class to another
→ eg class 1 = "quadrilateral"
class 2 (inherited) = "square", "triangle"
Interaction btw objects?
→ often have functions, or "methods" of a class that allow objects to interact
→ simple as a print statement to as complex as functions btw instances of classes!
Examples?
→ Cars, people, anything!
↳ eg car with price (int) name (str) and movement (function)
→ get creative!
Classes, instances
→ class is the object type (eg phone)
→ instance is a call of the class (eg phone → iPhone)


(Object - Oriented Programming System)
OOPs
Encapsulation
Abstraction
Polymorphism
Inheritance
Class
Object

Objects

Bijna alles is een object in JavaScript. Het is dus belangrijk om objecten door en door te begrijpen.

```
const person = {
   firstName: "John",
   lastName: "Doe",
   fullName: function () {
     return this.firstName + " " + this.lastName;
   },
 };
  const person2 = {
   firstName: "Massimo",
   lastName: "De Nittis",
   fullName: () => {
     return `${this.firstName} ${this.lastName}`;
   },
 };
  console.log(person.fullName()); // This prints "John Doe"
  console.log(person2.fullName()); // This prints "undefined undefined" -
IMPORTANT!
```

Er is geen verschil in de return instructie van beide functies, ze retourneren allebei een samengevoegde (samengevoegde) string.

Er is een verschil tussen het toewijzen van een anonieme functie function () {} en het toewijzen van een anonieme pijl functie () => {}.
De pijl functie weet niet wat this is . Er moet dus een normale anonieme functie worden gebruikt om ervoor te zorgen dat this gekend is

Class

Een klasse kan worden gezien als een sjabloon om een object te maken.

Neem het onderstaande voorbeeld, waar drie objecten worden gemaakt. Elk object vertegenwoordigt een persoon. Elk object heeft twee eigenschappen, namelijk firstName en lastName. Elk object heeft één methode, namelijk fullName.

Voor elke nieuwe persoon moet elke eigenschap opnieuw worden getypt en moet elke methode opnieuw worden getypt fullName.

Stel dat elke persoon een nieuwe eigenschap krijgt, bijnaam genaamd. Vervolgens moet in elk object de nieuwe eigenschap afzonderlijk worden toegevoegd.

Als er tien objecten worden gemaakt die een persoon voorstellen, betekent dit dat er tien keer een eigenschap moet worden toegevoegd.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  fullName: function () {
    return `${this.firstName} ${this.lastName}`;
  },
};
 const person2 = {
  firstName: "Massimo",
  lastName: "De Nittis",
  fullName: function () {
    return `${this.firstName} ${this.lastName}`;
  },
};
 const person3 = {
  firstName: "Donald",
  lastName: "Duck",
  fullName: function () {
    return `${this.firstName} ${this.lastName}`;
  },
};
```

Om ervoor te zorgen dat het niet nodig is voor elke persoon een nieuw object met alle eigenschappen te maken, kan een klasse worden gemaakt.

```
class Person {
  firstName = "Anonymous";
  lastName;
```

```
    fullName() {
     return `${this.firstName} ${this.lastName}`;
   }
}
const person = new Person();
person.firstName = "John";
person.lastName = "Doe";

const person2 = new Person();
person2.firstName = "Donald";
person2.lastName = "Duck";

console.log(person.fullName());
console.log(person2.fullName());
```

Opgelet:
Properties gedeclareerd in de class block mogen **niet** worden voorafgegaan met het **let** sleutelwoord. Methodes of functies in een class mogen **niet** beginnen met het sleutelwoord **function**.

het **new** sleutelwoord instantieert EN retourneert een nieuw object met een default property firstName (Anonymous)

Nadat een instantie van een klasse aan een variabele is toegewezen, kunnen de eigenschappen en methoden worden gebruikt alsof het een object is (omdat het een object is) met behulp van de **puntnotatie**. Dit wordt bijvoorbeeld gedaan om waarden toe te wijzen aan firstName en/of lastName

constructor

Om ervoor te zorgen dat we niet altijd na de instantiëring één voor één een waarde aan de eigenschappen moet toekennen, gebruiken we een speciale methode: de constructor. Dit is een methode die altijd wordt aangeroepen wanneer een instantie van een object wordt aangemaakt.

```
class Person {
   firstName;
   lastName;
    constructor() {
```

```
    /*
     * Code in this code block is executed at the moment
     * that 'new Person()' is called.
     */
    console.log("I am being executed");
  }
   fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}
 const person = new Person(); // The constructor method is executed
// 'I am being executed' is printed
 const person2 = new Person(); // The constructor method is executed
// 'I am being executed' is printed
```

Op onderstaande manier kunnen we ook parameters doorgeven aan de constructor. **Let op de this notatie!**

```
class Person {
  // attributes defined without let keyword
  firstName;
  lastName;
   constructor(parameter1, parameter2) {
    // attributes need to the assigned to this if declared outside of the
constructor or methods
    this.firstName = parameter1;
    this.lastName = parameter2;
  }
   fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}
 const person = new Person("Massimo", "De Nittis");
console.log(person.fullName()); // Massimo De Nittis
```

Soms is het interessant om geen instantie van een klasse te hoeven maken en toch de methoden te kunnen aanroepen. Hiervoor gebruiken we een voorbeeld met een klasse genaamd 'Utils'. (bv DB login credentials of config settings)

```
// Use a little imagination, Utils is in a separate JavaScript file.
class Utils {
  // New keyword: static
  // This allows the function to be called via: Utils.largestNumber(a, b);
```

```
  static largestNumber(a, b) {
    if (a > b) {
      return a;
    }
    return b;
  }
}
 // Again use imagination, this is the second JavaScript file.
const a = 3,
  b = 5;
console.log(`The largest number is: ${Utils.largestNumber(a, b)}`);
 // Again use imagination, this is the third JavaScript file.
const x = 4,
  y = 6;
console.log(`The largest number is: ${Utils.largestNumber(x, y)}`);
```

extends & super

Stel je voor dat er code is waarin verschillende klassen worden aangeboden om verschillende dieren te creëren…

```
class Lion {
  name;
  age;
   constructor(name, age) {
    this.name = name;
    this.age = age;
  }
   eat() {
    return `${this.name} is eating.`;
  }
   sleep() {
    return `${this.name} is sleeping.`;
  }
   hunt() {
    return `${this.name} is hunting.`;
  }
}
 class Dog {
  name;
  age;
```

```
  petName;
   constructor(name, age, petName) {
    this.name = name;
    this.age = age;
    this.petName = petName;
  }
   eat() {
    return `${this.name} is eating.`;
  }
   sleep() {
    return `${this.name} is sleeping.`;
  }
}
 class Elephant {
  name;
  age;
   constructor(name, age) {
    this.name = name;
    this.age = age;
  }
   eat() {
    return `${this.name} is eating.`;
  }
   sleep() {
    return `${this.name} is sleeping.`;
  }
   eat() {
    return `${this.eat()} And continues to eat.`;
  }
}
const lion = new Lion("Lion", 13);
const dog = new Dog("Samson", 8);
const elephant = new Elephant("Dumbo", 9);
console.log(lion.eat()); // Lion is eating.
console.log(lion.hunt()); // Lion cub is hunting.
console.log(lion.sleep()); // Lion cub is sleeping.
console.log(dog.eat()); // Samson is eating.
console.log(dog.sleep()); // Samson is sleeping.
console.log(elephant.eat()); // Dumbo is eating.
console.log(elephant.sleep()); // Dumbo is sleeping.
```

Zoek de overeenkomsten:

Elke class heeft 2 dezelfde attributen of properties (name & age)

Elke class heeft 2 dezelfde methodes (eat() & sleep())

Warning: DRY is in the house! (Don't Repeat Yourself)

Oplossing: **Extends & Super**

Een class dat "extends" van een andere class heet ook een subclass of child class. De klasse waarvan ge-extend wordt heet dan een parent class of een super class… de **super() methode** wordt aangeroepen om te voorkomen dat enkel de **constructordelen** worden gedupliceerd die gemeenschappelijk zijn tussen klassen.

In English, **extends** can be phrased by **inheritance.**

Onderstaande code doet identiek hetzelfde:

```
class Animal {
   name;
   age;
    constructor(name, age) {
     this.name = name;
     this.age = age;
   }
    eat() {
     return `${this.name} is eating.`;
   }
    sleep() {
     return `${this.name} is sleeping.`;
   }
 }
  class Lion extends Animal {
   constructor(name, age) {
     super(name, age);
   }
    hunt() {
     return `${this.name} is hunting.`;
   }
 }
  class Dog extends Animal {
   petName;
    constructor(name, age, petName) {
```

```
    super(name, age);
    this.petName = petName;
  }
}
 class Elephant extends Animal {
  constructor(name, age) {
    super(name, age);
  }
   eatLots() {
    return `${super.eat()} And continues to eat.`;
  }
}
 const lion = new Lion("Lion", 13);
 const dog = new Dog("Samson", 8);
 const elephant = new Elephant("Dumbo", 9);
 console.log(lion.eat()); // Lion is eating.
 console.log(lion.hunt()); // Lion cub is hunting.
 console.log(lion.sleep()); // Lion cub is sleeping.
 console.log(dog.eat()); // Samson is eating.
 console.log(dog.sleep()); // Samson is sleeping.
 console.log(elephant.eat()); // Dumbo is eating.
 console.log(elephant.eatLots()); // Dumbo is eating. And keeps eating.
 console.log(elephant.sleep()); // Dumbo is sleeping.
```

Access modifiers

In andere OOP programmeertalen is het mogelijk om gebruik te maken van access modifiers. Met andere woorden, het is mogelijk om de toegang tot bepaalde eigenschappen en methoden te beperken.

Meestal is er een trefwoord **public** en een trefwoord **private** die worden gebruikt om te bepalen of een eigenschap/methode overal beschikbaar is (public) of alleen binnen een instantie van de klasse (private).

Stel dat deze termen in JavaScript bestaan, dan ziet het er als volgt uit:

```
// NO VALID JAVASCRIPT CODE
class Person {
```

```
  private pinCode;
  private password;
  public name;
  public age;
   constructor(pin, password, name, age) {
    this.pinCode = pin;
    this.password = password;
    this.name = name;
    this.age = age;
  }
   private getPin() {
    return this.pinCode;
  }
   public getPinCode() {
    return this.pinCode;
  }
}
 const person = new Person(1111, 1234, 'John Doe', 29);
console.log(person.name); // "John Doe"
console.log(person.age); // 29
console.log(person.pinCode); // This would not print anything, pinCode is private
console.log(person.password); // This would not print anything, password is
private
 console.log(person.getPin()); // This would not print anything, getPin() is
private
console.log(person.getPinCode()); // 1111
/*
* The property pinCode is private, and thus only available within the code block
of the class Person.
* The method getPinCode() is public, so it can be called on the instance.
* Because getPinCode() is inside the code block of the class, it can access the
private properties/methods.
*/
```

In JavaScript bestaan de keywords public and private niet: (Duh! ^^ JS!)

```
class Person {
  pinCode;
  password;
  name;
  age;
   constructor(pin, password, name, age) {
    this.pinCode = pin;
    this.password = password;
    this.name = name;
```

```
    this.age = age;
  }
   getPin() {
    return this.pinCode;
  }
   getPinCode() {
    return this.pinCode;
  }
}
 const person = new Person(1111, 1234, "John Duck", 29);
 console.log(person.name); // "John Duck"
 console.log(person.age); // 29
 console.log(person.pinCode); // 1111
 console.log(person.password); // 1234
 console.log(person.getPin()); // 1111
 console.log(person.getPinCode()); // 1111
```

Alles is standaard openbaar/public, alle eigenschappen en methoden kunnen worden aangeroepen op de instantie van een klasse.

Er is echter een conventie die onder ontwikkelaars wordt gebruikt om nog steeds aan te geven dat iets eigenlijk privé/private is. Door een _ (underscore) voor de naam van de eigenschap/methode toe te voegen, wordt aangegeven dat een eigenschap/methode eigenlijk privé is.

```
class Person {
  _pinCode;
  _password;
  name;
  age;
   constructor(pin, password, name, age) {
    this._pinCode = pin;
    this._password = password;
    this.name = name;
    this.age = age;
  }
   _getPin() {
    return this._pinCode;
  }
   getPinCode() {
    return this._pinCode;
  }
```

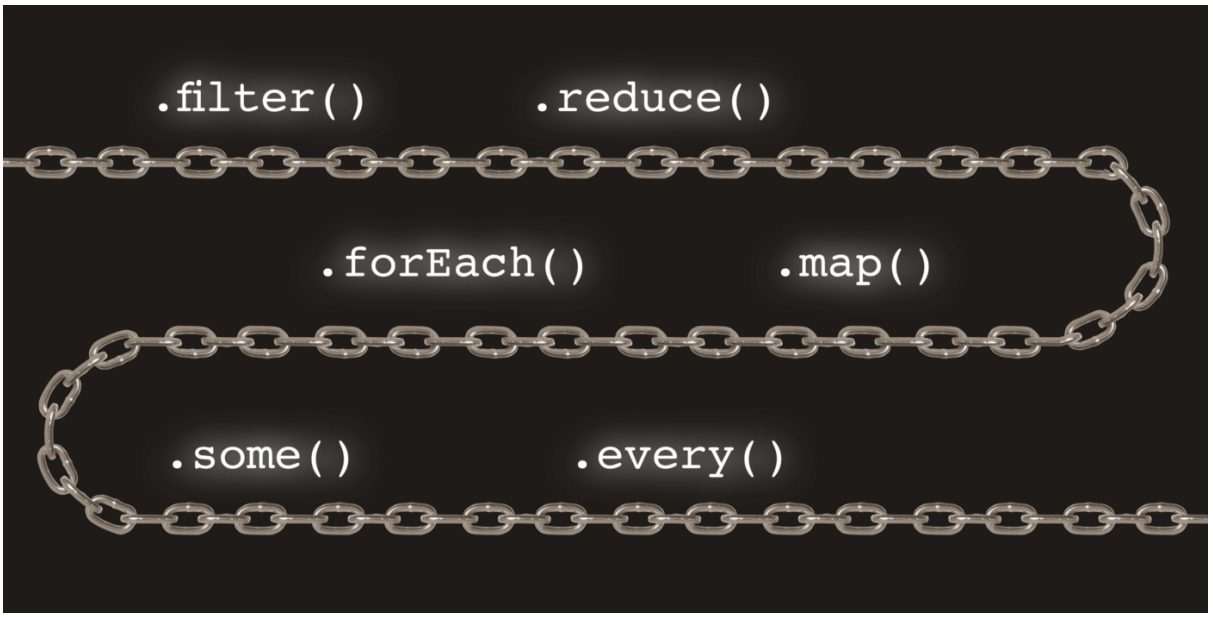
```
}
const person = new Person(1111, 1234, "John Duck", 29);
console.log(person.name); // "John Duck"
console.log(person.age); // 29
console.log(person._pinCode); // 1111
console.log(person._password); // 1234
console.log(person._getPin()); // 1111
console.log(person.getPinCode()); // 1111
```

Built-in objects

Er zijn veel verschillende ingebouwde objecten. Deze objecten maken gebruik van ook een ingebouwde klasse.

Het is niet nodig om letterlijk elke functie en elke eigenschap uit het hoofd te kennen. Om te weten wat er mogelijk is met een object dat wereldwijd aanwezig is binnen JavaScript kan documentatie worden geraadpleegd.

Een voorbeeld is de klasse Array. De documentatie is te vinden op een website die wordt onderhouden door Mozilla (de makers van Firefox en Thunderbird).
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array



[Array vs Set vs Map vs Object — Real-time use cases in Javascript (ES6/ES7) | by Rajesh Babu | codeburst]

[JavaScript ES6]

[Top 10 Features of ES6 | Board Infinity]