***The Secret Life of Objects***

**Object-oriented programming** = een set technieken om objecten te gebruiken als centraal principe in de organisatie (structuur) van een programma. -🡪 in essentie is het idee om programma’s in kleine gedeeltes op te delen en elk onderdeel verantwoordelijk te maken voor zijn eigen staat. Zo kan elk onderdeel locaal blijven tot dat onderdeel en de rest niet beïnvloeden.

Object properties die bij de interface horen heten **public** en de locale properties die andere code niet mag beïnvloeden heet **private.**

**Encapsulation** = interface scheiden van implementatie 🡪 goed idee.

Wanneer een method iets moet doen met het object waarop het was gecalled, zoals bij object.method, moet je de binding **this** op de body gebruiken. Deze wijst automatisch naar het object die hem aanriep. Voorbeeld:

*function speak(line) {*

*console.log(`The ${this.type} rabbit says '${line}'`);*

*}*

*let whiteRabbit = {type: "white", speak};*

*let hungryRabbit = {type: "hungry", speak};*

*whiteRabbit.speak("Oh my ears and whiskers, " +*

*"how late it's getting!");*

*// → The white rabbit says 'Oh my ears and whiskers, how*

*// late it's getting!'*

*hungryRabbit.speak("I could use a carrot right now.");*

*// → The hungry rabbit says 'I could use a carrot right now.'*

Met **speak.call** krijg je hetzelfde effect:

*speak.call(hungryRabbit, "Burp!");*

*// → The hungry rabbit says 'Burp!'*

**Arrow functions** kunnen de this binding van de scope om hun heen zien, bij een reguliere functie heeft de this zijn eigen binding.

Bijna elk object heeft naast zijn eigen properties, ook een **prototype**. Een prototype is een ander object die als fallback wordt gebruikt voor properties. Wanneer een object bijvoorbeeld wordt gevraagd om een property die die niet heeft, wordt er gezocht in het prototype van dat object, en vervolgens de prototype van de prototype, etc.

Object.prototype heeft een aantal methods, zoals bijvoorbeeld toString. En zo kan je met object.create een object kopiëren. De protoRabbit is hier een container van de properties die gedeeld worden door alle rabbits. Een individueel rabbit object, zoals killerRabbit, heeft alleen zijn eigen properties:

let protoRabbit = {

speak(line) {

console.log(`The ${this.type} rabbit says '${line}'`);

}

};

let killerRabbit = Object.create(protoRabbit);

killerRabbit.type = "killer";

killerRabbit.speak("SKREEEE!");

// → The killer rabbit says 'SKREEEE!'

Prototype lijkt een beetje op het object-oriented concept **classes.** Een class definieert de vorm van een object – welke methods en properties het heeft 🡪 heet de **instance** van de class. Dus om dit te maken, moet je een object creëren met new. Een **constructor function** doet:

*class Rabbit {*

*constructor(type) {*

*this.type = type;*

*}*

*speak(line) {*

*console.log(`The ${this.type} rabbit says '${line}'`);*

*}*

*}*

*let killerRabbit = new Rabbit("killer");*

*let blackRabbit = new Rabbit("black");*

*^ Moet een hoofdletter zijn*

Wanneer je eerst een property aan de prototype geeft en vervolgens dezelfde prototype aan het object, wordt deze overschreven door het object.

Een **map** (niet die we eerder tegenkwamen) is een data structuur die values met andere values associeert, zoals bijv. de object names met de object properties.

**Object.create(null)** = creëert een object zonder prototype.

Een andere manier om dit te voorkomen is via **set, get en has.** Voorbeeld:

*let ages = new Map();*

*ages.set("Boris", 39);*

*ages.set("Liang", 22);*

*ages.set("Júlia", 62);*

*console.log(`Júlia is ${ages.get("Júlia")}`);*

*// → Júlia is 62*

*console.log("Is Jack's age known?", ages.has("Jack"));*

*// → Is Jack's age known? false*

*console.log(ages.has("toString"));*

*// → false*

Als je toch bij een object de eigen keys (properties) wil weten en niet die van het prototype, kan je dit gebruiken:

*console.log({x: 1}.hasOwnProperty("toString"));*

*// → false*

**Polymorfisme** = een stuk code is geschreven om te werken met een bepaalde interface. Alleen deze interfaces zullen werken met de code.

**Symbol** = een soort strings, maar kunnen niet twee keer gemaakt worden. Ze kunnen gebruikt worden als propert names en kunnen daarom vreedzaam leven naast andere properties, met wat voor naam dan ook. Voorbeeld:

*const toStringSymbol = Symbol("toString");*

*Array.prototype[toStringSymbol] = function() {*

*return `${this.length} cm of blue yarn`;*

*};*

*console.log([1, 2].toString());*

*// → 1,2*

*console.log([1, 2][toStringSymbol]());*

*// → 2 cm of blue yarn*

Dit kan ook:

*let stringObject = {*

*[toStringSymbol]() { return "a jute rope"; }*

*};*

*console.log(stringObject[toStringSymbol]());*

*// → a jute rope*

Over een symbol heen loopen doe je zo:

*let okIterator = "OK"[Symbol.iterator]();*

*console.log(okIterator.next());*

*// → {value: "O", done: false}*

*console.log(okIterator.next());*

*// → {value: "K", done: false}*

*console.log(okIterator.next());*

*// → {value: undefined, done: true}*

**Getters, setters en static:**

*class Temperature {*

*constructor(celsius) {*

*this.celsius = celsius;*

*}*

*get fahrenheit() {*

*return this.celsius \* 1.8 + 32;*

*}*

*set fahrenheit(value) {*

*this.celsius = (value - 32) / 1.8;*

*}*

*static fromFahrenheit(value) {*

*return new Temperature((value - 32) / 1.8);*

*}*

*}*

*let temp = new Temperature(22);*

*console.log(temp.fahrenheit);*

*// → 71.6*

*temp.fahrenheit = 86;*

*console.log(temp.celsius);*

*// → 30*

**Inheritance** = wanneer een nieuwe class de properties van een oude class (gedeeltelijk) overneemt). Dit doe je zo;

*class SymmetricMatrix extends Matrix* 🡪 hiermee zeg je dat de class niet gebasseerd moet zijn op het object prototype maar op de superclass. De gekopieerde class is de subclass.

**super** provides a way to call methods as they were defined in the superclass.

Inheritance zorgt ervoor dat je classes aan elkaar koppelt. Dit zorgt voor minder werk en code, maar meer gecompliceerde samenhangende code. Daarom is inheritance controversieel om te gebruiken.

Met deze code kun je checken of een object afkomstig is van een specifieke class. Dit kan met **instanceof.** Voorbeeld:

*console.log(*

*new SymmetricMatrix(2) instanceof SymmetricMatrix);*

*// → true*