Javascript OOP

Pengenalan OOP (Object Oriented Programming)

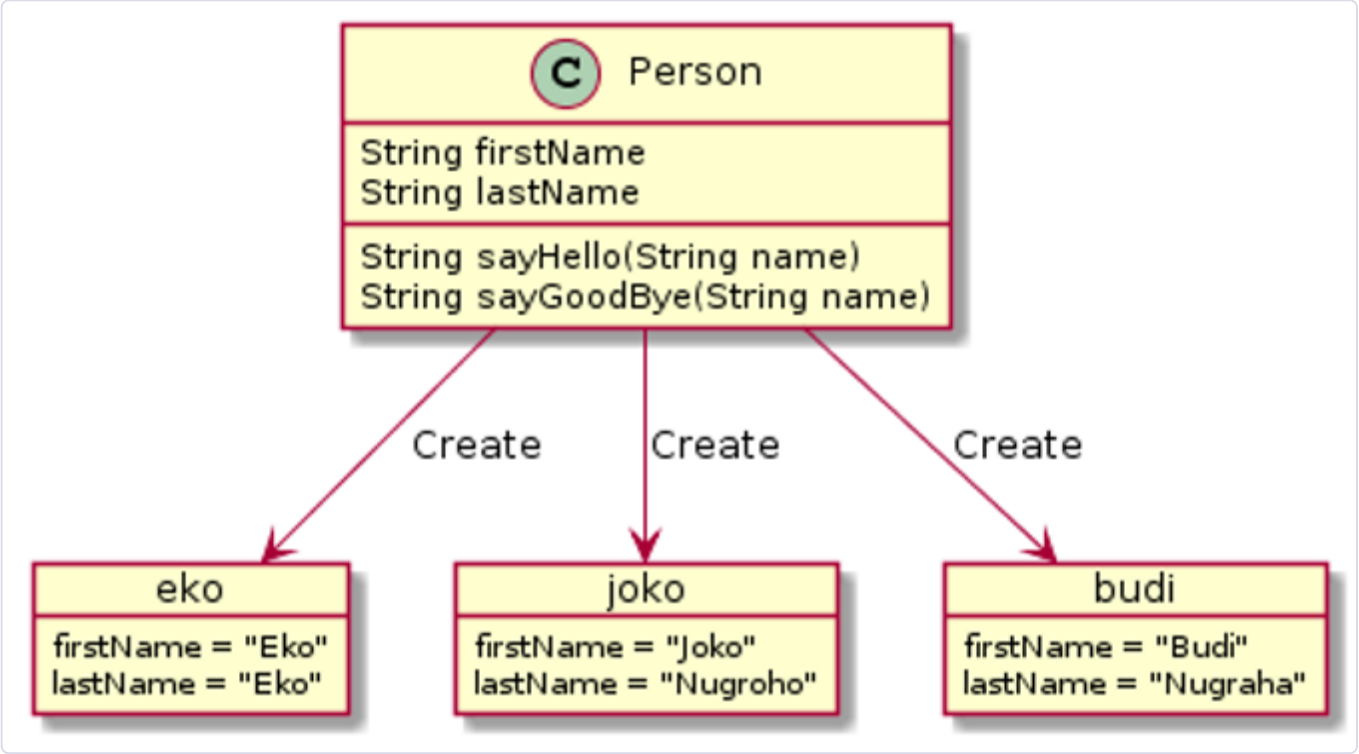
Object Oriented Programming adalah sudut pandang bahasa pemrograman yang berkonsep “objek”. Ada banyak sudut pandang bahasa pemrograman, namun OOP adalah yang sangat populer saat ini. Ada beberapa istilah yang perlu dimengerti dalam OOP, yaitu: Object dan Class

Apa itu Object?

Object adalah data yang berisi field / properties / attributes dan method / function / behavior

Apa itu Class?

Class adalah blueprint, prototype atau cetakan untuk membuat Object.
Class berisikan deklarasi semua properties dan functions yang dimiliki oleh Object
Setiap Object selalu dibuat dari Class
Dan sebuah Class bisa membuat Object tanpa batas



OOP di Javascript

JavaScript sendiri sebenarnya sejak awal dibuat sebagai bahasa prosedural, bukan bahasa pemrograman berorientasi objek.

Oleh karena, implementasi OOP di JavaScript memang tidak sedetail bahasa pemrograman lain yang memang dari awal merupakan bahasa pemrograman OOP

seperti Java atau C++.

Membuat Constructor Function

Sebenarnya kita sudah belajar tipe data object, dengan cara membuat variable dengan tipe data object.

Namun pembuatan object menggunakan tipe data object, akan membuat object yang selalu unik, sedangkan dalam OOP, biasanya kita akan membuat class sebagai cetakan, sehingga bisa membuat object dengan karakteristik yang sama berkali, kali, tanpa harus mendeklarasikan object berkali-kali seperti menggunakan tipe data object.

```javascript
const danang = {

    firstName: "Danang",

    lastName: "Setiawan"

};
```

Sebelum EcmaScript versi 6, pembuatan class, biasanya menggunakan function. Hal ini dikarenakan sebenarnya JavaScript bukanlah bahasa pemrograman yang fokus ke OOP.

Untuk membuat class di JavaScript lama, kita bisa membuat function.

Function ini kita sebut dengan Constructor Function

```javascript
function Person() {

}
```

Membuat Object dari Constructor Function

Setelah kita membuat class, jika kita ingin membuat object dari class tersebut, kita bisa menggunakan kata kunci new, lalu diikuti dengan nama constructor function nya.

```javascript
function Person() {

}

const obj1 = new Person()
const obj2 = new Person()
```

Property di Constructor Function

Sebenarnya setelah kita membuat object, kita bisa dengan mudah menambahkan property ke dalam object tersebut hanya dengan menggunakan nama variable nya, diikuti tanda titik dan nama property.

Namun jika seperti itu, alhasil, constructor function yang sudah kita buat tidak terlalu berguna, karena property nya hanya ada di object yang kita tambahkan property.

Untuk menambahkan property di dalam semua object yang dibuat dari constructor function, kita bisa menggunakan kata kunci `this` lalu diikuti dengan nama property nya.

```JavaScript
function Person() {
        this.firstName = ""
        this.lastName = ""
}
```

Method di Constructor Function

Sama seperti pada tipe data object biasanya, kita juga bisa menambahkan method di dalam constructor function.

Jika kita tambahkan method di constructor function, secara otomatis object yang dibuat akan memiliki method tersebut.

```JavaScript
function Person() {
        this.firstName = ""
        this.lastName = ""
        this.message = function (name) {
                console.log(`Hello ${name}, my name is ${this.firstName}`)
        }
}
```

Parameter di Constructor Function

Karena dalam JavaScript, class adalah berbentuk function, jadi secara default, function tersebut bisa memiliki parameter.

Constructor function sama seperti function biasanya, bisa memiliki parameter, hal ini membuat ketika kita membuat object, kita bisa mengirim langsung data lewat parameter di constructor function tersebut.

```JavaScript
function Person(firstName, lastName) {
        this.firstName = firstName
        this.lastName = lastName
        this.message = function (name) {
                console.log(`Hello ${name}, my name is ${this.firstName}`)
        }
}
```

Constructor Inheritance

Dalam constructor kita biasanya membuat property baik itu berisi value ataupun function.

Di dalam constructor, kita bisa memanggil constructor lain, dengan begitu kita bisa mewarisi semua property yang dibuat di constructor lain tersebut
Untuk memanggil constructor lain, kita bisa menggunakan `NamaConstructor.call(this, parameter)`.

```JavaScript
function Employee(firtstName) {
        this.firstName = firstName
        this.message = function (name) {
                console.log(`Hello ${name}, my name is ${this.firstName}`)
        }
}

function Manager(firstName, lastName) {
        Employee.call(this, firstName)
        this.lastName = lastName
}
```

Prototype

JavaScript sebelumnya dikenal dengan pemrograman berbasis prototype.

Memang agak sedikit membingungkan, dan tidak dipungkiri, banyak sekali yang bingung dengan konsep prototype di JavaScript.

Pada chapter ini, kita akan bahas tentang konsep prototype.

Prototype Inherintance

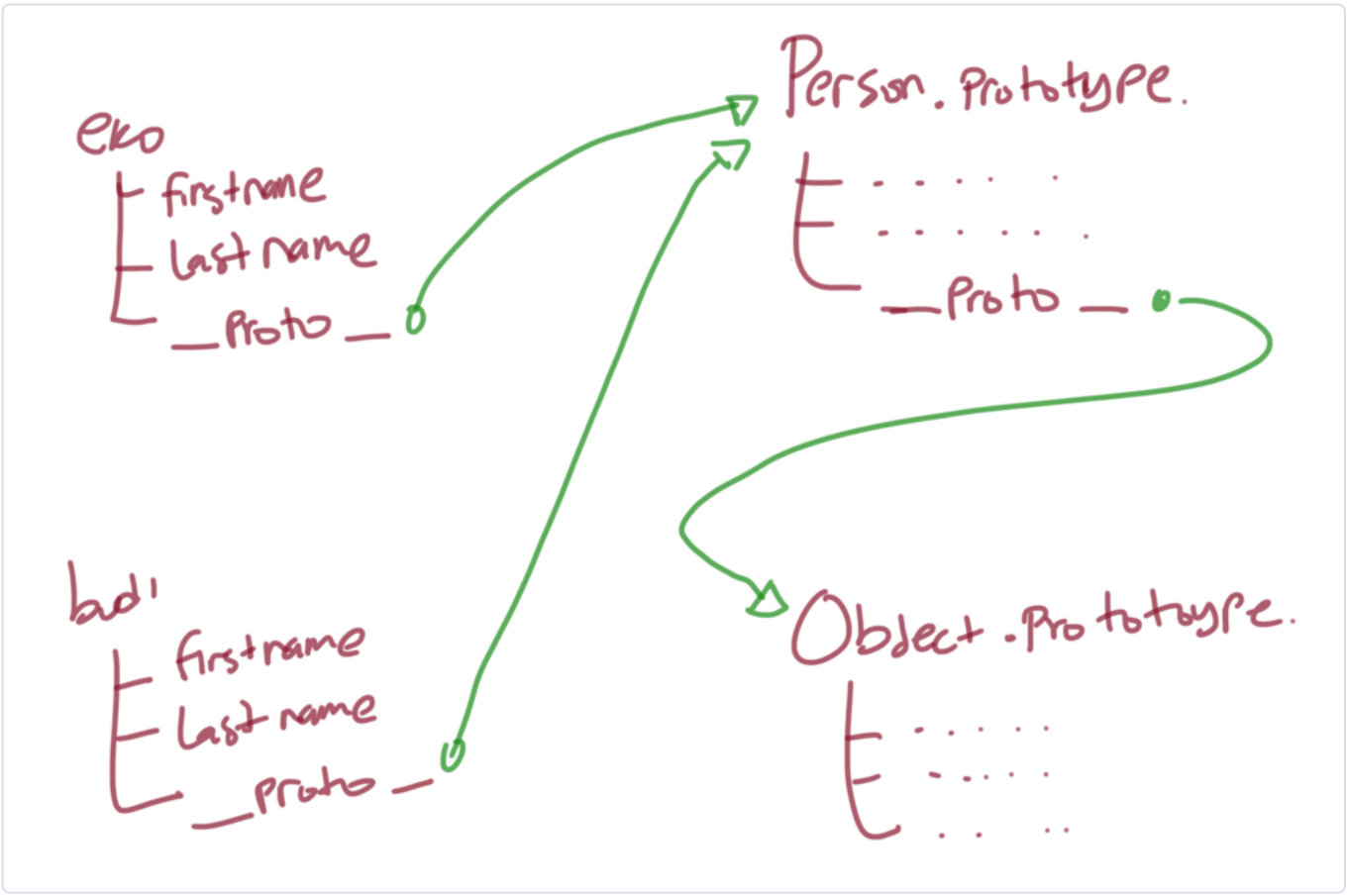
Saat kita membuat object dari constructor function, object tersebut disebut instance, semua property (baik itu value atau method), akan berada di dalam instance object nya.

Setiap kita membuat sebuah constructor function, maka secara otomatis akan dibuatkan prototype nya, misal ketika kita membuat constructor function Person, maka akan ada Person.prototype.

Saat kita membuat sebuah object instance, secara otomatis object tersebut adalah turunan dari Constructor.prototype nya. Untuk mengakses prototype milih sebuah instance, kita bisa menggunakan `__proto__`

```javascript
const obj1 = new Person("Ucup", "Otong")
const obj2 = new Person("Otong", "Surotong")

console.log(obj1)
console.log(obj2)
```

Console: Object Instance



Console: Object Instance Inheritance

```
                                                    index.html?_ijt=163u…fhcju26uf2oq6u
▼Person {firstName: "Eko", lastName: "Khannedy", sayHello: f} i
    firstName: "Eko"
    lastName: "Khannedy"
  ▶ sayHello: f (name)
  ▼ __proto__:
    ▶ constructor: f Person(firstName, lastName)
    ▼ __proto__:
      ▶ constructor: f Object()
      ▶ hasOwnProperty: f hasOwnProperty()
      ▶ isPrototypeOf: f isPrototypeOf()
      ▶ propertyIsEnumerable: f propertyIsEnumerable()
      ▶ toLocaleString: f toLocaleString()
      ▶ toString: f toString()
      ▶ valueOf: f valueOf()
      ▶ __defineGetter__: f __defineGetter__()
      ▶ __defineSetter__: f __defineSetter__()
      ▶ __lookupGetter__: f __lookupGetter__()
      ▶ __lookupSetter__: f __lookupSetter__()
      ▶ get __proto__: f __proto__()
      ▶ set __proto__: f __proto__()
```

Menambahkan Property ke Prototype

Property mirip object, dimana kita bisa menambah property baik itu value ataupun method.

Saat kita menambah sebuah property ke Prototype, secara otomatis, semua object instance yang turunan dari prototype tersebut akan memiliki property tersebut.

Menambahkan Property ke Instance Object

```JavaScript
const obj1 = new Person("Ucup", "Otong")

// hanya untuk instance object obj1
obj1.message = function () {
        console.info("Good Bye")
}
```

Menambahkan Property ke Prototype

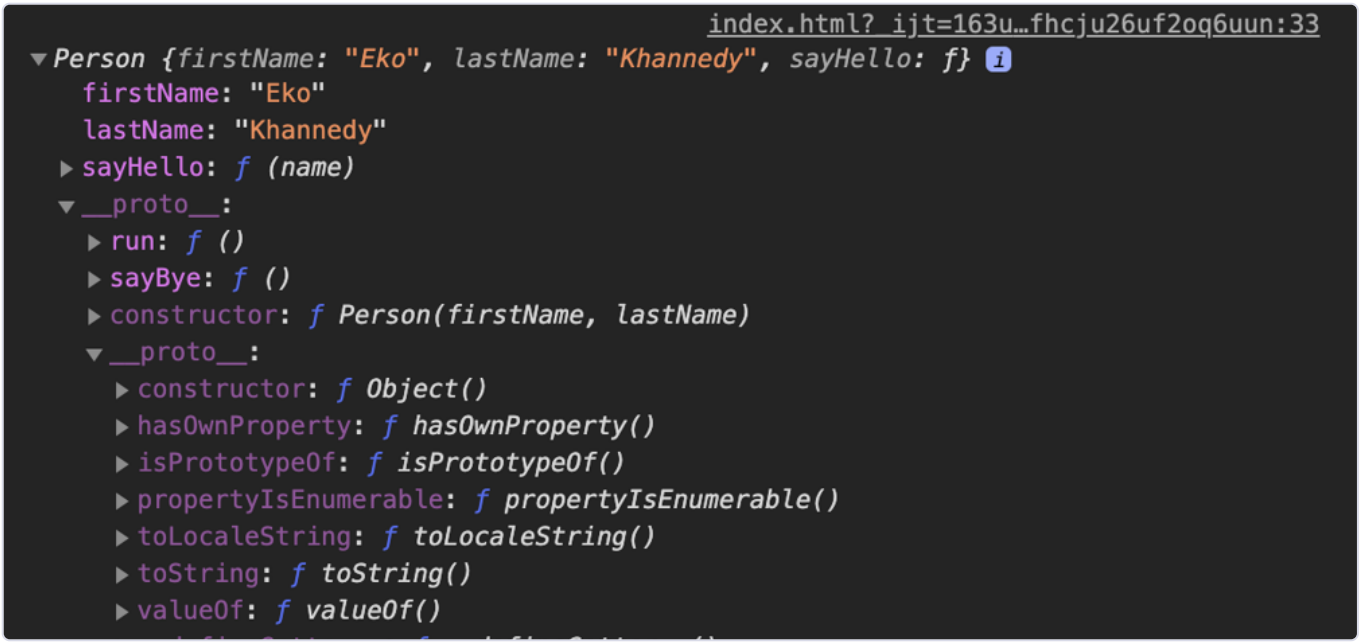
```JavaScript
Person.prototype.message = function () {
        console.info("Good bye")
}

Person.prototype.run = function () {
        console.info(`${this.firstName} is running`)
}
```

Cara kerja Prototype Inheritance

Bagaimana bisa property di prototype diakses dari object instance?

Ketika kita mengakses property di object instance, pertama akan di cek apakah di object tersebut terdapat property tersebut atau tidak, jika tidak, maka akan di cek di **proto** (prototype) nya, jika masih tidak ada, akan di cek lagi di **proto** (prototype) yang lebih tinggi, begitu seterusnya, sampai berakhir di Object Prototype.

Console: Prototype Inheritance



Prototype Inheritance

Sekarang kita sudah tahu, bahwa prototype selalu memiliki parent, artinya dia adalah turunan, parent tertinggi adalah Object prototype.

Pertanyaannya bagaimana jika kita ingin melakukan inheritance ke Prototype lain?

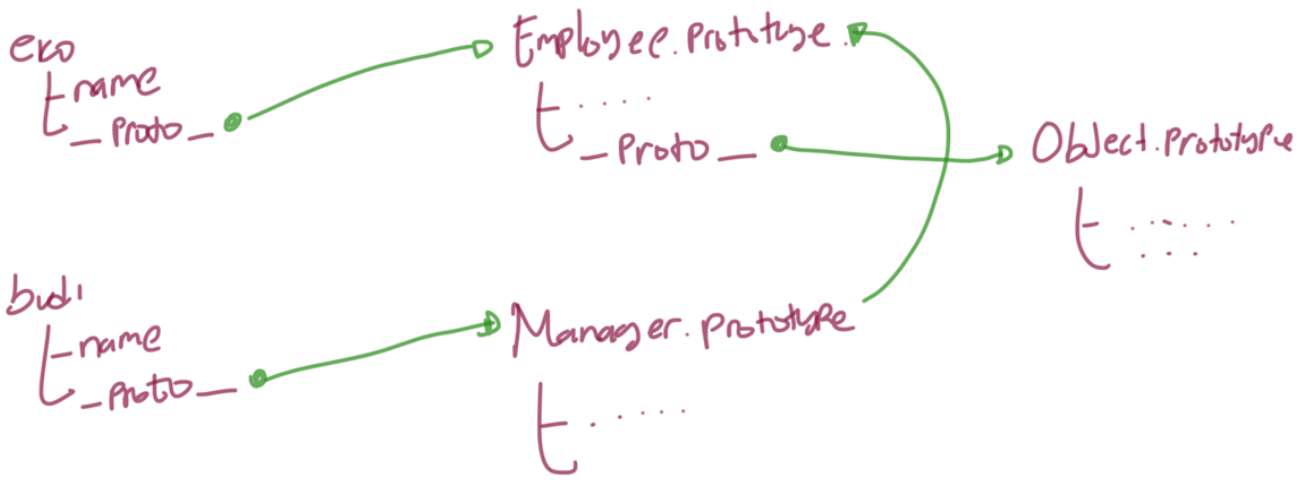
Hal ini juga bisa dilakukan, namun agak sedikit tricky, karena hal ini, sebenarnya untuk JavaScript modern, tidak direkomendasikan lagi praktek OOP menggunakan Prototype, karena di ES6 sudah dikenalkan kata kunci class yang akan nanti dibahas di chapter tersendiri.

Prototype Inheritance SALAH

```javascript
function Employee(name) {
        this.name = name;
}

function Manager(name) {
        this.name = name;
}

Manager.prototype = Employee.prototype;

Manager.prototype.message = function (name) {
        console.log(`Hello ${name}, my name is ${this.name}`)
}

Employee.prototype.message = function (name) {
```

```
        console.log(`Hello ${name}, my name is ${this.name}`)
}

const employee = new Employee("Otong")
employee.message("Ucup")

const manager = new Manager("Ucup")
manager.message("Otong")
```

Diagram Prototype SALAH



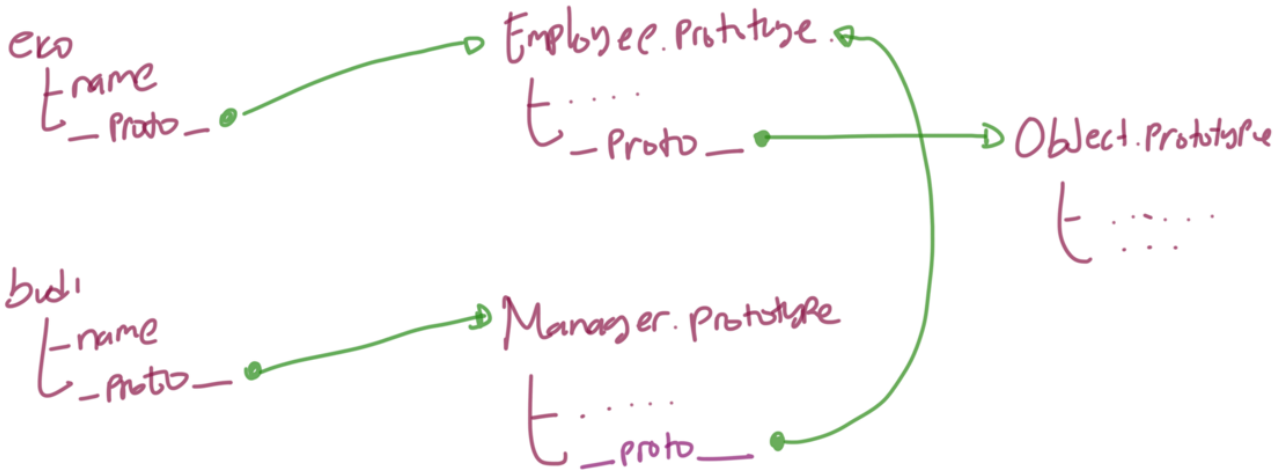
Prototype Inheritance BENAR

JavaScript

```
function Employee(name) {
        this.name = name;
}

function Manager(name) {
        this.name = name;
}

Manager.prototype = Object.create(Employee.prototype);

Manager.prototype.message = function (name) {
        console.log(`Hello ${name}, my manager name is ${this.name}`)
}

Employee.prototype.message = function (name) {
        console.log(`Hello ${name}, my employee name is ${this.name}`)
}

const employee = new Employee("Otong")
employee.message("Ucup")
```

```
const manager = new Manager("Ucup")
manager.message("Otong")
```

Diagram: Prototype Inheritance Benar



Kata Kunci Class

Membuat Class

Sejak EcmaScript versi 6, diperkenalkan kata kunci baru, yaitu class, ini merupakan kata kunci yang digunakan untuk membuat class di JavaScript.

Dengan kata kunci class, kita tidak perlu lagi menggunakan constructor function untuk membuat class.

```
class Person {                                                    JavaScript

}

const obj1 = new Person()
console.log(obj1)
```

Constructor di Class

Karena bentuk constructor function mirip dengan function, jadi kita bisa menambah parameter pada constructor function, lantas bagaimana dengan class?

Di class juga kita bisa menambah constructor, dimana dengan menggunakan constructor, kita juga bisa menambah parameter saat pertama kali membuat object nya.

Untuk membuat constructor di class, kita bisa menggunakan kata kunci constructor.

```JavaScript
class Person {
        constructor(name) {
        }
}

const obj1 = new Person("Ucup")
```

Property di Class

Sama seperti pada constructor function, dalam class pun kita bisa menambahkan property.

Karena hasil akhirnya adalah sebuah object, jadi menambahkan property di class bisa juga dilakukan di instance object nya.

```JavaScript
class Person {
        constructor(name) {
                this.name = name
        }
}
```

Method di Class

Membuat method di class sebenarnya bisa dilakukan dengan cara seperti menambahkan method di constructor function.

Namun, hal tersebut sebenarnya menambahkan method ke dalam instance object.

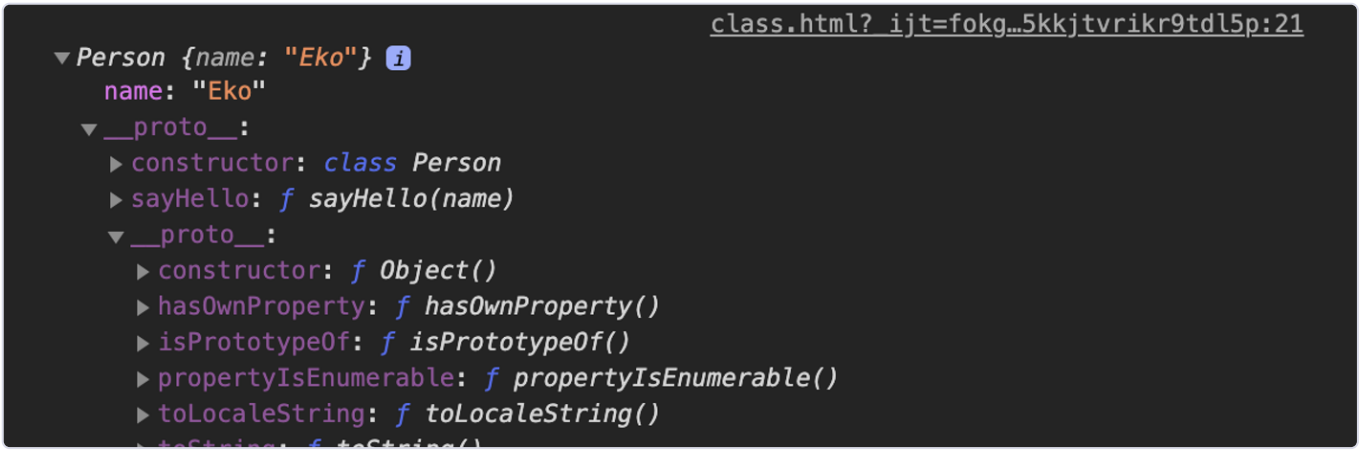
Khusus untuk method sebaiknya kita menambahkan ke prototype, bukan ke instance object.

Untung nya di class, ada cara mudah menambahkan method dan secara otomatis ditambahkan ke prototype.

Method di Class

```JavaScript
class Person {
        constructor(name) {
                this.name = name
        }

        message(name) {
                console.log(`Hi ${name}, my name is ${this.name}`)
        }
}
```

Console: Method di prototype



Class Inheritance

Sebelumnya kita sudah tahu bahwa prototype mendukung pewarisan, walaupun agak sedikit tricky cara pembuatannya.

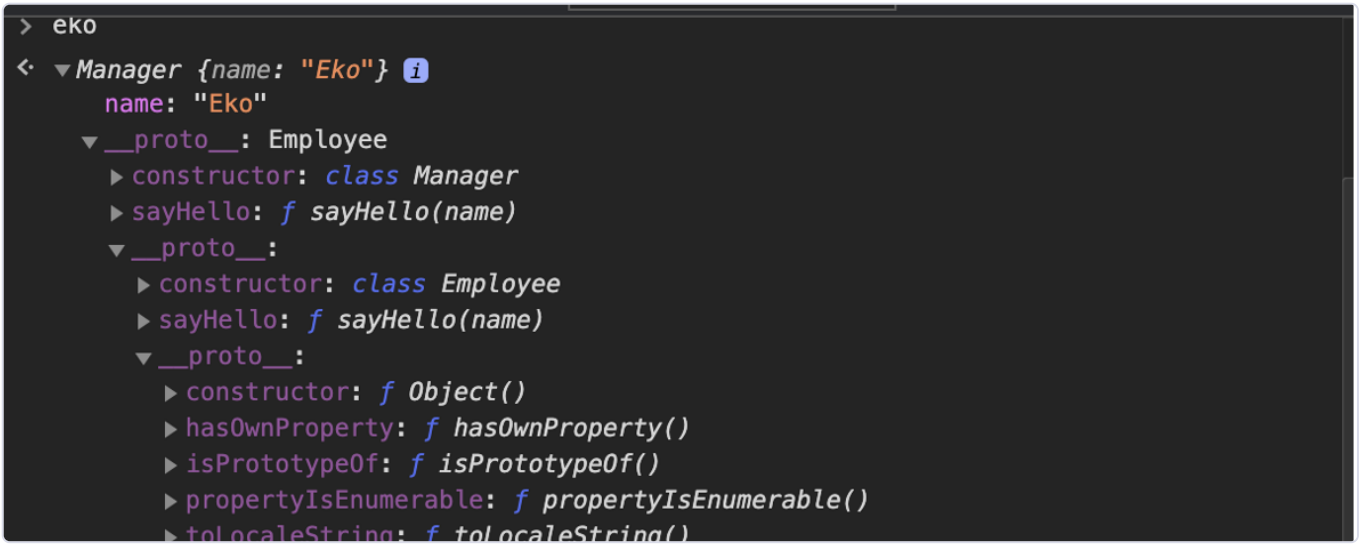
Untungnya itu diperbaiki di ES6 dengan fitur class nya.

Sebuah class bisa melakukan pewarisan dari class lainnya dengan menggunakan kata kunci extends.

Di JavaScript, class inheritance sama seperti prototype inheritance, hanya bisa memiliki satu parent class.

```JavaScript
class Employee {
        message(name) {
                console.log(`Hi ${name}, my name is employe ${this.name}`)
        }
}

class Manager extends Employee {
        message(name) {
                console.log(`Hi ${name}, my name is manager ${this.name}`)
        }
}

const empl = new Employee();
empl.name = "Otong"
empl.message("Ucup")

const manager = new Mananger();
manager.name = "Ucup"
manager.message("Otong")
```

Console: Class Inheritance



Super Constructor

Class Inheritance sifatnya seperti Prototype Inheritance.

Bagaimana dengan Constructor Inheritance? Sebenarnya Constructor Inheritance hanyalah melakukan eksekusi constructor lain dengan tujuan agar property di constructor lain bisa ditambahkan ke instance object ini.

Dalam kasus ini, jika kita ingin mencapai hasil yang sama, kita bisa menggunakan kata kunci `super` di dalam constructor.

Kata kunci `super` digunakan untuk memanggil constructor super class.

Jika di child class kita membuat constructor, maka kita wajib memanggil parent constructor, walaupun di parent tidak ada constructor.

```javascript
class Employe {
        constructor(firstName) {
                this.firstName = firstName;
        }
        message(name) {
                console.log(`Hi ${name}, my name is employe
${this.firstName}`)
        }
}

class Manager {
        constructor(firstName, lastName) {
                super(firstName);
                this.lastName = lastname;
        }
```

```
        message(name) {
        console.log(`Hi ${name}, my name is manager ${this.firstName}`)
    }
}
```

Super Method

Selain digunakan untuk memanggil constructor milih parent class, kata kunci `super` juga bisa digunakan untuk mengakses method parent class.

Caranya bisa menggunakan super titik nama function nya. Dengan kata lain, super sebenarnya adalah reference ke parent prototype, mirip seperti `__proto__`.

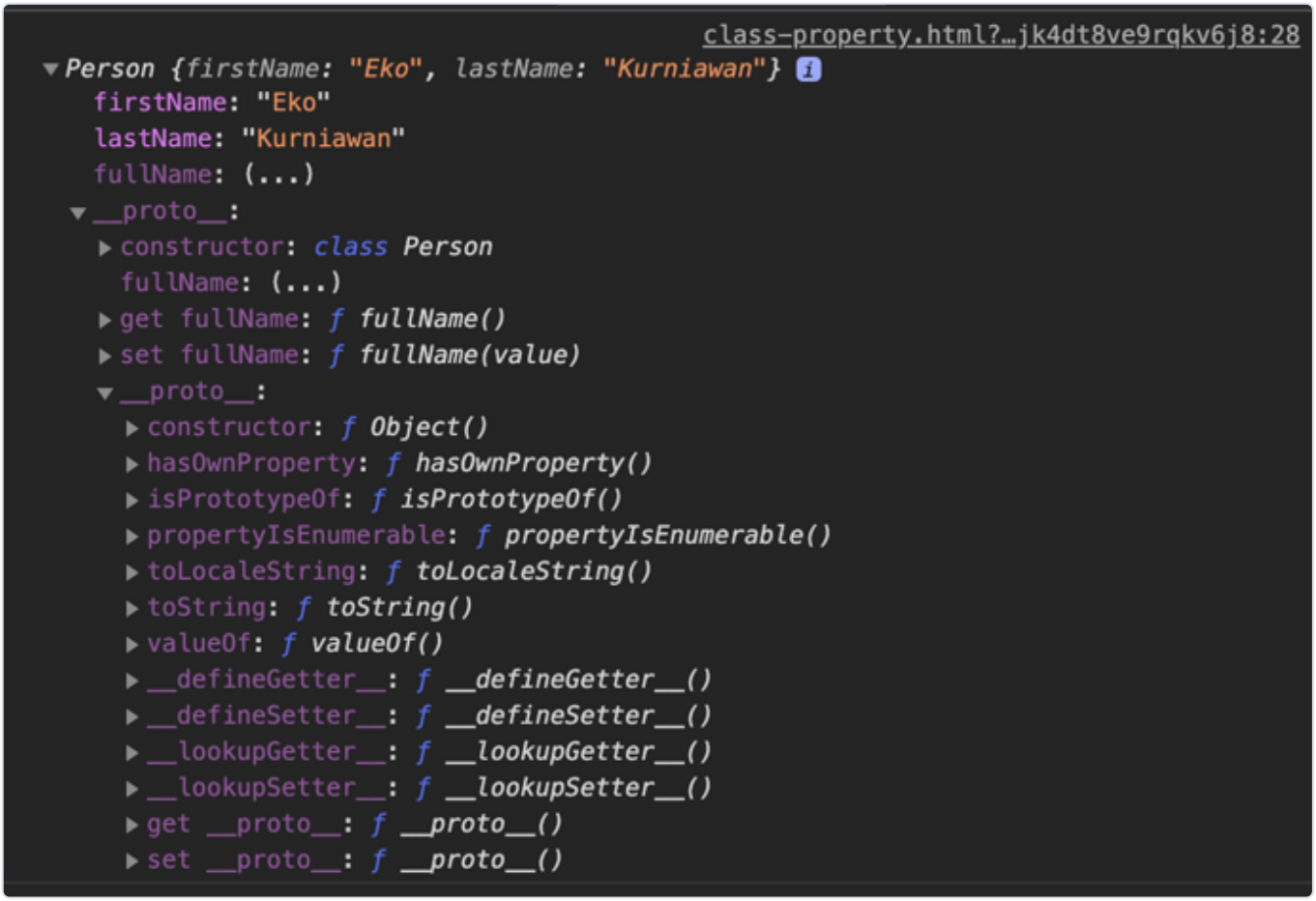
```JavaScript
class Shape {
        paint() {
                console.info("Paint Shape")
        }
}

class Circle extends Shape {
        paint(){
                super.paint(); // memanggil paint() method parent class
                console.info("Paint Circle")
        }
}
```

Getter dan Setter di Class

Class juga mendukung pembuatan getter dan setter. Perlu diingat, getter dan setter ini akan berada di prototype, bukan di instance object.

```JavaScript
class Person {
        constructor(firstName, lastName) {
                this.firstName = firstName
                this.lastName = lastName
        }

        get fullName() {
                return `${this.firstname} ${this.lastName}`
        }

        set fullName(value) {
                const result = value.split(" ")
                this.firstName = result[0]
                this.lastName = result[1]
```

```
        }
}
```



Public Class FIeld

Biasanya, saat kita ingin menambahkan field (property yang berisi value), kita biasanya tambahkan di constructor.

Namun, ada proposal di EcmaScript yang mengajukan pembuatan public class field ditempatkan diluar constructor, selevel dengan penempatan method.

Proposal ini masih belum final, namun beberapa browser sudah mendukung nya https://github.com/tc39/proposal-class-fields

Dalam proposal tersebut juga disebutkan bahwa EcmaScript akan mendukung access modifier public dan private.

Public artinya bisa diakses dari luar class, dan private hanya bisa diakses dari dalam class saja.

Private class field akan kita bahas di chapter selanjutnya.

Untuk membuat public class field, kita bisa langsung buat nama field dengan value nya selevel dengan method.

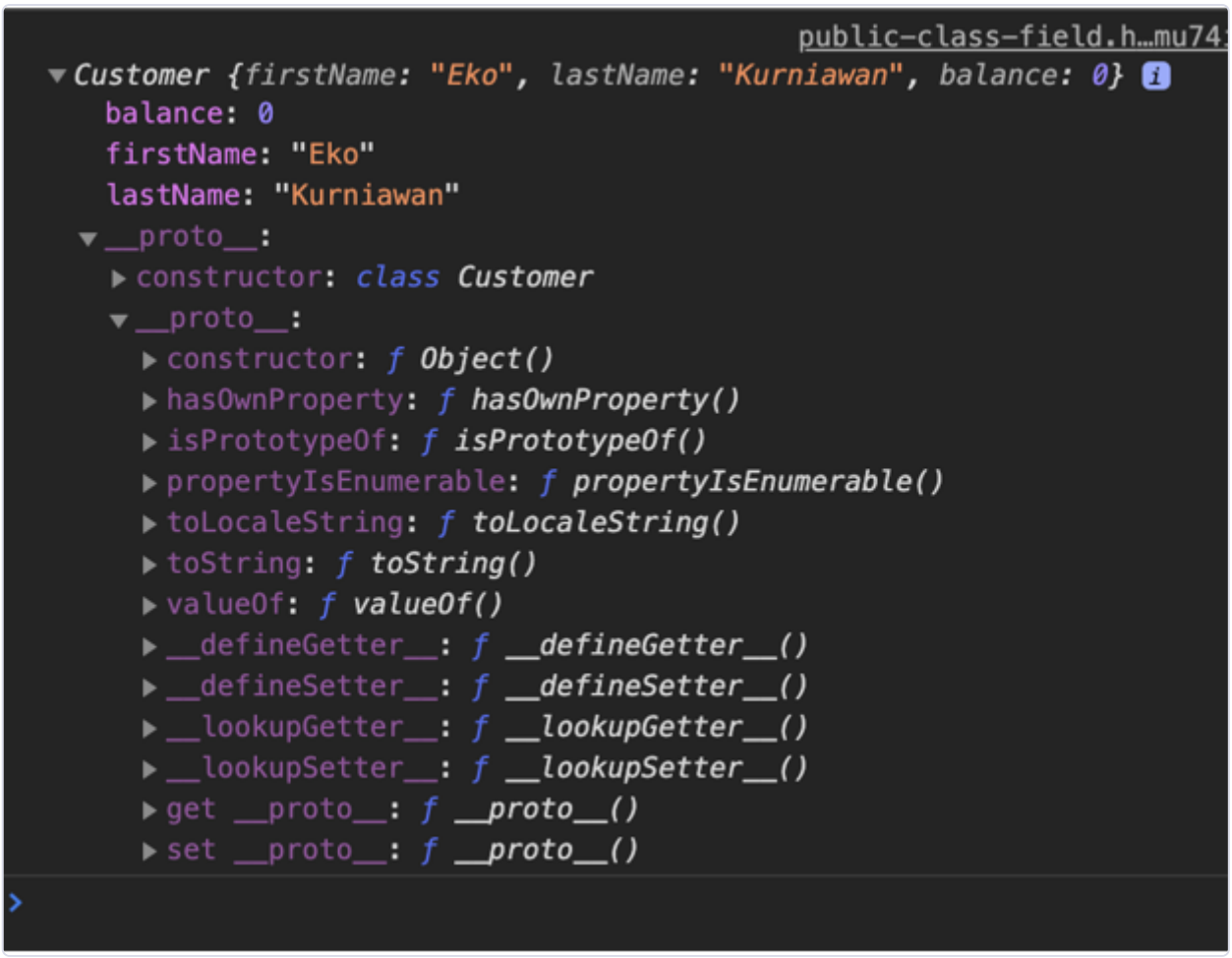
Jika kita tidak memasukkan value ke dalam field tersebut, artinya field tersebut memiliki value undefined.

```javascript
class Customer {
        firstName;
        lastName;
        balance = 0;
}

const obj1 = new Customer();
console.log(obj1)
```

Public Class Field dan Constructor

```javascript
class Customer {
        firstName;
        lastName;
        balance = 0;
```

```
        constructor(firstName, lastName) {
                this.firstName = firstName
                this.lastName = lastName
        }
}

const obj1 = new Customer("Ucup", "Otong")
console.log(obj1)
```



Private Class Field

Secara default, saat kita menambahkan field, maka field tersebut bisa diakses dari manapun.

Jika kita ingin membuat field yang bersifat private (hanya bisa diakses di dalam class), kita bisa menggunakan tanda `#` sebelum nama field nya.

Ini dinamakan private class field, dan hanya bisa diakses dari dalam class saja.

```
class Counter {                                                    JavaScript
        #counter = 0
```

```
        increment() {
                this.#counter++;
        }

        decrement() {
                this.#counter--;
        }

        get() {
                return this.#counter;
        }
}

const counter = new Counter();

counter.increment()
counter.increment()
counter.increment()
counter.increment()
counter.increment()
counter.increment()

console.log(counter.get())
```

Private Method

Sama seperti field, terdapat proposal juga untuk menambah fitur private method di EcmaScript. Dengan demikian, access modifier private juga bisa digunakan di method.

Caranya sama, dengan menambahkan tanda # diawal method, maka secara otomatis method tersebut adalah private.

Ingat fitur ini masih dalam tahapan, belum benar-benar menjadi standard EcmaScript, jadi mungkin tidak semua browser mendukung fitur ini
https://github.com/tc39/proposal-private-methods

```JavaScript
class Person {
        message(name) {
                if (name) {
                        this.#messageName(name)
                } else {
                        this.#messageNoName()
                }
        }
```

```
        #messageName(name) {
                console.log(`Hello ${name}`)
        }

        #messageNoName() {
                console.log("Hello")
        }
}

const obj1 = new Person()
obj1.message("Ucup")
```

Operator instanceof

Kadang ada kasus kita ingin mengecek apakah sebuah object merupakan instance dari class tertentu atau bukan.

Kita tidak bisa menggunakan operator typeof, karena object dari class, jika kita gunakan operator typeof, hasilnya adalah “object”.

Operator `instanceof` akan menghasilkan boolean, `true` jika benar object tersebut adalah instance object nya, atau `false` jika bukan.

```
class Employee {}
class Manager {}

const obj1 = new Employee();
const obj2 = new Manager();

console.log(obj1 instanceof Employee)
console.log(obj1 instanceof Manager)
console.log(obj2 instanceof Employee)
console.log(obj2 instanceof Manager)
```
JavaScript

Operator instanceof di Class Inheritance

Operator instanceof mendukung class inheritance, artinya instanceof juga bisa digunakan untuk mengecek, apakah sebuah object adalah instance dari class tertentu, atau turunan dari class tertentu?

```
class Employee {}
class Manager extends Employee {}

const ucup = new Employee()
const otong = new Manger()
```
JavaScript

```
console.log(ucup instanceof Employee); // true
console.log(ucup instanceof Manager); // false

console.log(otong instanceof Employee); // true
console.log(otong instanceof Manager); // true
```

Static Class Field

`static` adalah kata kunci yang bisa kita tambahkan sebelum field atau method, biasannya ketika kita membuat field atau method, maka secara otomatis field akan menjadi property di instance object, dan method akan menjadi function di prototype.

Jika kita tambahkan static, maka hal itu tidak terjadi.

Jika kita tambahkan static dalam class field, secara otomatis field tersebut bukan lagi milik instance object, melainkan milik class nya itu sendiri.

Biasanya static digunakan jika kita ingin membuat utility field atau function.

Cara mengakses static class field pun tidak lagi lewat object, melainkan lewat class nya.

Static class field bisa diartikan sifatnya global, tidak peduli diakses dimana atau siapa yang mengakses, hasilnya akan sama.

```JavaScript
class Configuration {
        static name = "Javascript"
        static version = 1.0
        static author = "Otong"
}

console.log(Configuration.name)
console.log(Configuration.version)
console.log(Configuration.author)
```

Static Method

Kata kunci static juga tidak hanya bisa ditambahkan di field, tapi juga di method
Jika kita tambahkan di method, artinya method tersebut jadi milik class nya, bukan prototype.

Dan untuk mengakses method tersebut, kita juga bisa lakukan seperti mengakses static class field.

```JavaScript
class MathUtil {
        static sum( ...numbers) {
```

```
                let total = 0
                for (const number of numbers) {
                        total += number
                }
                return total;
        }
}

const sum = MathUtil.sum(1, 2, 3, 3, 2, 1)
console.log(sum)
```

Error

Saat membuat aplikasi, sudah tentu kita tidak akan terhindar dari yang namanya error.

Di JavaScript, Error merupakan sesuatu yang sudah standar.

Banyak sekali class error di JavaScript, namun semua class error di JavaScript selalu berujung di class Error, artinya class Error adalah superclass untuk semua jenis error di JavaScript.

Contoh class error yang terdapat di JavaScript contohnya SyntaxError, TypeError, EvalError, dan lain-lain

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#error_types

Throw Error

Saat kita membuat instance object dari class Error, tidak lantas otomatis terjadi error.

Kita perlu memberitahu program kita, bahwa kita akan mentrigger sebuah error terjadi, atau istilahnya adalah melempar error (throw error).

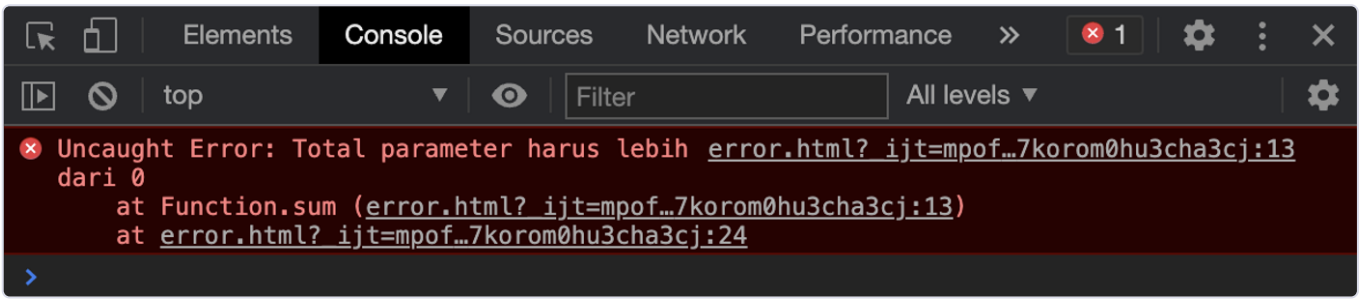
Untuk melempar error, kita bisa gunakan kata kunci throw, diikuti dengan instance object error nya.

Jika terjadi error, maka otomatis kode program kita akan terhenti, dan kita bisa melihat detail errornya di console di aplikasi browser kita.

```
class MathUtil {                                                    JavaScript
        static sum( ...numbers) {
                if (numbers.length === 0) {
                        throw new Error("Total parameter harus lebih dari
```

```
0")
                }

                let total = 0
                for (const number of numbers) {
                        total += number
                }
                return total;
        }
}

console.log(mathUtil.sum())
console.log(MathUtil.sum(1, 2, 3, 3, 2, 1))
```

Console: Throw Error



Error Handling

Saat terjadi error di kode program JavaScript, kadang kita tidak ingin program kita berhenti.

Di JavaScript, kita bisa menangkap jika terjadi error.

Kita bisa menggunakan try catch statement untuk menangkap error.

Pada block try, kita akan mencoba mengakses kode program yang bisa menyebabkan error, dan jika terjadi error, block try akan berhenti dan otomatis masuk ke block catch.

Jika tidak terjadi error, block catch tidak akan dieksekusi.

```JavaScript
try {
        console.log(MathUtil.sum());
        console.log("Kode Block Try akan berhenti")
} catch (error) {
        console.error(`Terjadi error: ${error.message}`)
}

console.log("Kode program tidak akan berhenti")
```

Kata Kunci finally

Kadang kita ingin melakukan sesuatu entah itu terjadi error ataupun tidak.

Dalam try catch, kita bisa menambahkan block finally. Block finally ini akan selalu dieksekusi setelah try catch selesai, entah terjadi error atau tidak, block finally akan selalu dieksekusi.

```JavaScript
try {
        console.log(MathUtil.sum());
        console.log("Kode block try akan berhenti")
} catch (error) {
        console.error(`Terjadi error: ${error.message}`)
} finally {
        console.log("Kode program selesai")
}
```

Try Finally

Kata kunci finally juga bisa digunakan tanpa perlu menggunakan catch
Biasanya ini digunakan dalam kasus tertentu.

```JavaScript
class Counter {
        constructor() {
                this.value = 1
        }

        next() {
                try {
                        return this.value
                } finally {
                        this.value++
                }
        }
}

const counter = new Counter()
console.log(counter.next())
console.log(counter.next())
console.log(counter.next())
console.log(counter.next())
console.log(counter.next())
console.log(counter.next())
```

Membuat Class Error Manual

Walaupun JavaScript sudah memiliki standard class Error. Namun alangkah baiknya, kita membedakan tiap jenis error.

Untuk membuat error sendiri secara manual sangatlah mudah, cukup membuat class turunan dari class Error.

Dan jangan lupa tambahkan parameter message, agar bisa dikirimkan ke parameter di constructor class Error.

Validation Error

```JavaScript
class ValidationError extends Error {
        constructor(message, field) {
                super(message);
                this.field = field;
        }
}
```

```JavaScript
class MathUtil {
        static sum( ...numbers) {
                if (numbers.length === 0) {
                        throw new ValidationError("Total parameter harus lebih dari 0", "numbers")
                }

                let total = 0
                for (const number of numbers) {
                        total += number
                }
                return total;
        }
}


try {
        console.log(MathUtil.sum())
} catch (error) {
        if (error instanceof ValidationError) {
                console.error(`Terjadi error di field ${error.field} dengan error: ${error.message}`)
        } else {
                console.error(`Terjadi error: ${error.message}`)
        }
}
```

Iterable dan Iterator

Salah satu fitur terbaru di ES6 adalah iterable.

Iterable adalah spesial object yang memiliki standarisasi. Dengan mengikuti standarisasi Iterable, secara otomatis kita bisa melakukan iterasi terhadap data tersebut dengan menggunakan perulangan `for...of`.

Contoh yang sudah mengikuti standarisasi Iterable adalah string, Array, Object, dan lain-lain.

Kontrak Iterable (dalam TypeScript)

```
interface Iterable<T> {
        [Symbol.iterator](): Iterator<T>;
}
```
JavaScript

Kontrak Iterator (dalam TypeScript)

```
interface Iterator<T, TReturn = any, TNext = undefined> {
    // NOTE: 'next' is defined using a tuple to ensure we report the correct ass
    next(...args: [] | [TNext]): IteratorResult<T, TReturn>;
    return?(value?: TReturn): IteratorResult<T, TReturn>;
    throw?(e?: any): IteratorResult<T, TReturn>;
}
```

Kontrak: IteratorResult (dalam TypeScript)

```
interface IteratorYieldResult<TYield> {
        done?: false;
        value: TYield;
}

interface IteratorReturnResult<TReturn>{
        done: true;
        value: TReturn;
}

type IteratorResult<T, TReturn = any> = IteratorYieldResult<T> |
IteratorReturnResult<TReturn>;
```
TypeScript

Cara Keraja Iterable dan Iterator

Jika kita mengikuti kontrak Iterable, maka object yang kita buat akan bisa dilakukan iterasi menggunakan `for...of`.

Setiap kita melakukan perulangan, object Iterator akan dibuat. Hal ini menjadi aman jika kita melakukan iterasi berulang-ulang, karena Iterator baru akan dibuat terus

menerus.

Membuat Counter Iterator Result

```JavaScript
class CounterIteratorResult {
	constructor(value, done) {
		this.done = done;
		this.value = value;
	}
}
```

Membuat Counter Iterator

```JavaScript
class CounterIterator {
	constructor(value, max) {
		this.value = value;
		this.max = max;
	}

	next() {
		try {
			if (this.value > this.max) {
				return new
CounterItaratorResult(this.value, true);
			} else {
				return new
CounterIteratorResult(this.value, false);
			}
		} finally {
			this.value++;
		}
	}
}
```

Membuat Counter Iterable

```JavaScript
class Counter {
	constructor(value, max) {
		this.value = value;
		this.max = max;
	}

	[Symbol.iterator]() {
		return new CounterIterator(this.value, this.max)
	}
}
```

Menggunakan Counter Iterable

```javascript
const counter = new Counter(1, 10)
for (const element of counter) {
        console.log(element)
}
```

Materi Selanjutnya

- Javascript Standard Library
- Javascript Modules
- Javascript Document Object Model
- Javascript Async
- Javascript Web API