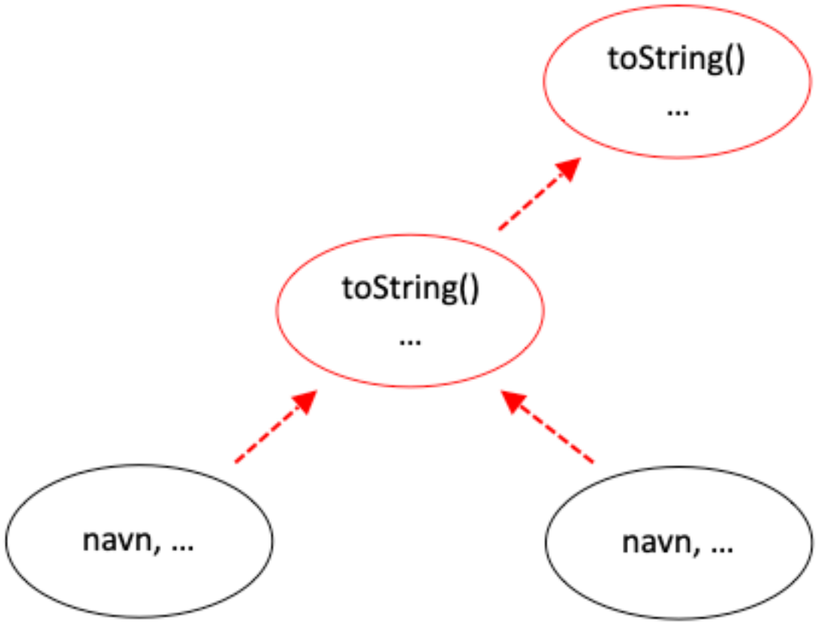
Prototyper, Constructor og class

Arv i JavaScript er baseret på prototyper

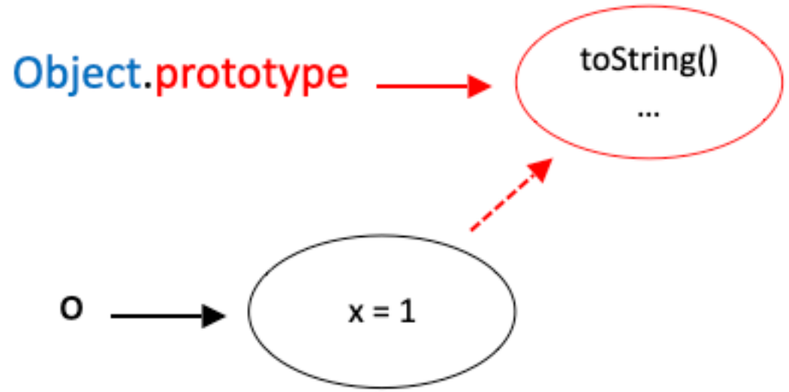
- JavaScript har ingen klasser eller interfaces
- Arv er i stedet baseret på prototyper
- Objekter har en prototype (-->)
- Prototyper er også objekter
- Prototyper kan selv have en prototype
  og danner dermed et prototype hierarki
- Et objekt arver properties og metoder
  fra sin(e) prototype(r)


toString()
...
toString()
...
navn, ...
navn, ...

Objekt literal

- En objekt literal har `Object.prototype` som sin prototype
- `Object.prototype` er roden i prototype hierarkiet

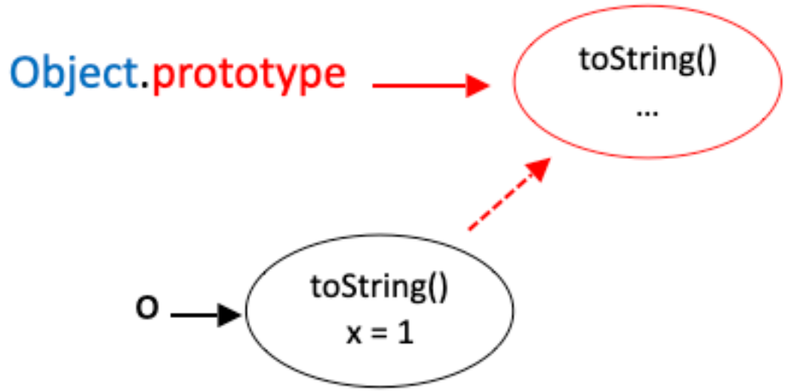
```
// objekt-literal.js
let o = {x: 1};
console.log(o.toString()); // => [object Object]
```



Egne og arvede properties

- Assignment til en arvet property tilføjer en propery med samme navn til objektet selv – den arvede property forbliver uændret
- Et objekts egne properties skjuler arvede properties med samme navn

```
// objekt-literal.js
let o = { x: 1 };
console.log(o.toString()); // => [object Object]
o.toString = function () { return 'x: ' + this.x }
console.log(o.toString()); // => x: 1
```



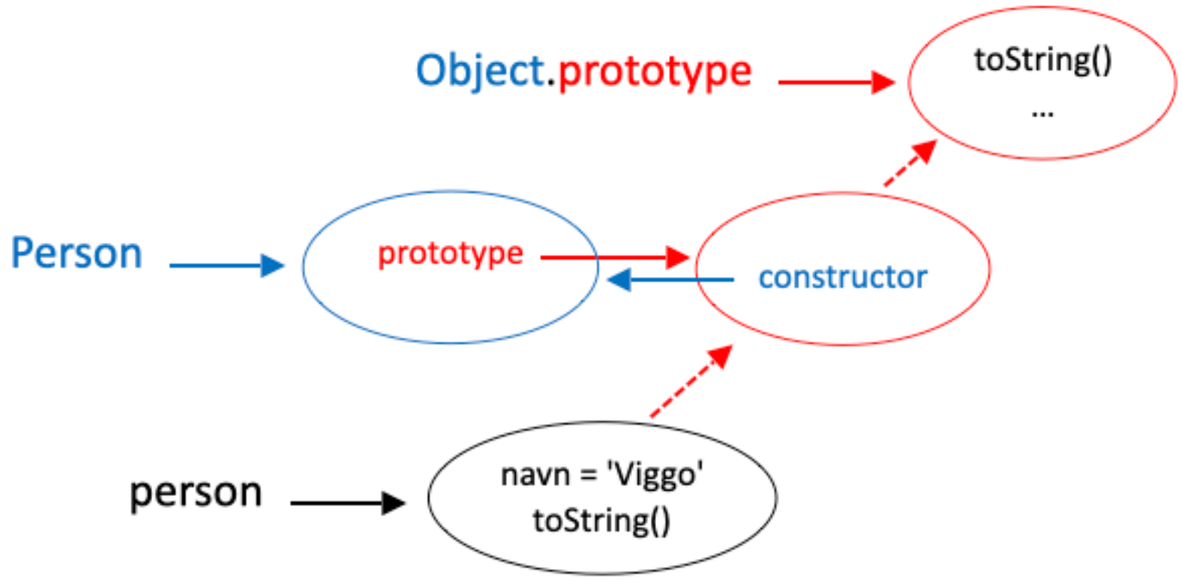
Constructor funktioner

- En constructor funktion er en speciel funktion, der opretter et nyt objekt, når den kaldes med `new` operatoren
- Det nyt objekt returneres implicit – så return skal ikke anvendes
- `this` refererer nu til det nye objekt og bruges til at initialisere det
- En constructor funktions navn skal pr. konvention starte med stort

```
// constructor1.js
function Person(navn) {
    this.navn = navn;
    this.toString = function () { return 'Person: ' + this.navn; }
}
let person = new Person('Viggo');
console.log(person.navn); // Viggo
console.log(person.toString()); // Person: Viggo
```

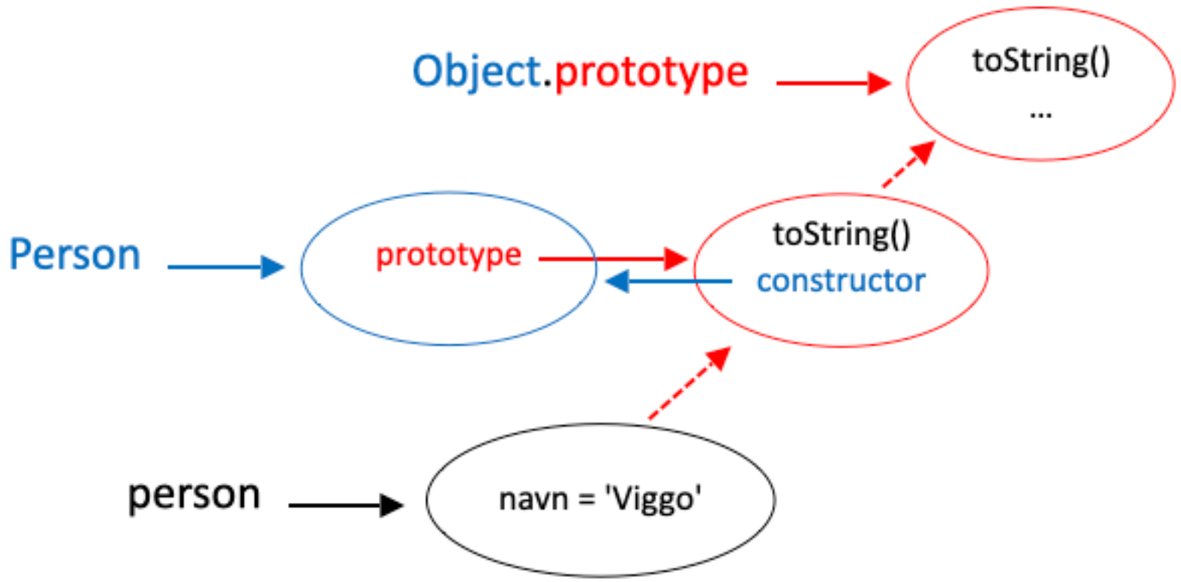
prototype og constructor

- En constructor funktion refererer til sit tilknyttet objekt med prototype
- Det tilknyttede objekt bliver prototypen for de oprettede objekter – og det refererer tilbage sin constructor funktion med constructor



Instance metoder

✤ Instance metoder skal nu sættes på prototype:



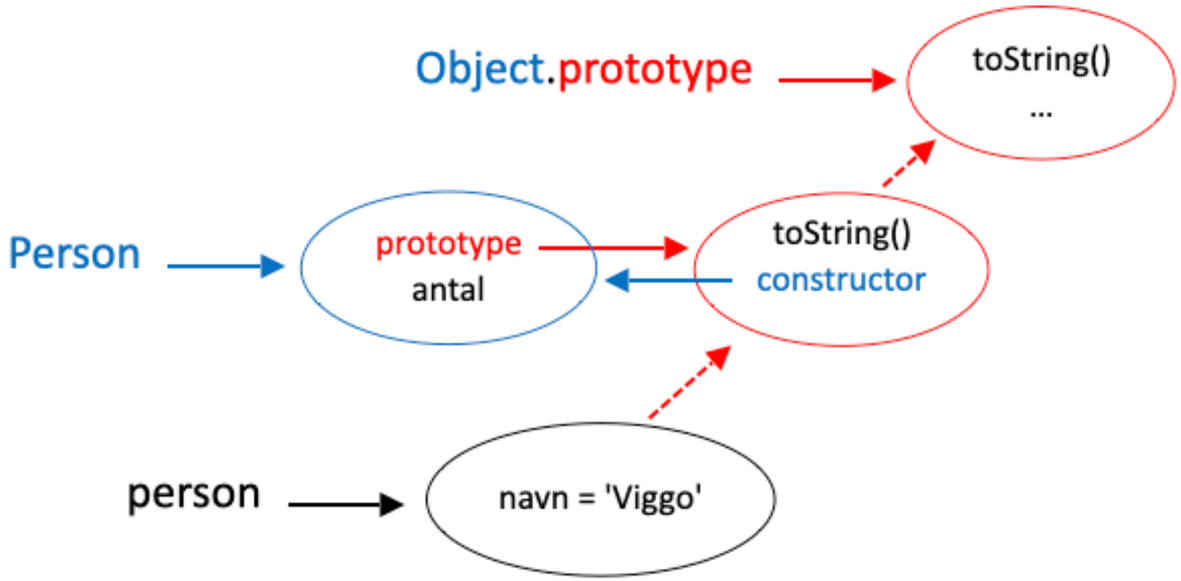
Klasser af objekter

- En constructor funktion definerer en klasse af objekter og constructor refererer dermed til objektets "klasse"
- Men husk, at objekter og prototyper kan ændres efterfølgende

```
// constructor2.js
function Person(navn) {
    this.navn = navn;
}
Person.prototype.toString = function () { return 'Person: ' + this.navn; }
let person = new Person('Viggo');
console.log(person.navn); // => Viggo
console.log(person.toString()); // => Person: Viggo
console.log(person.constructor.name); // => Person
console.log(person instanceof Person); // => true
console.log(typeof person.constructor); // => function
console.log(typeof Person.prototype); // => object
```

Static metoder og properties

✤Static metoder og properties skal sættes på constructor funktionen:



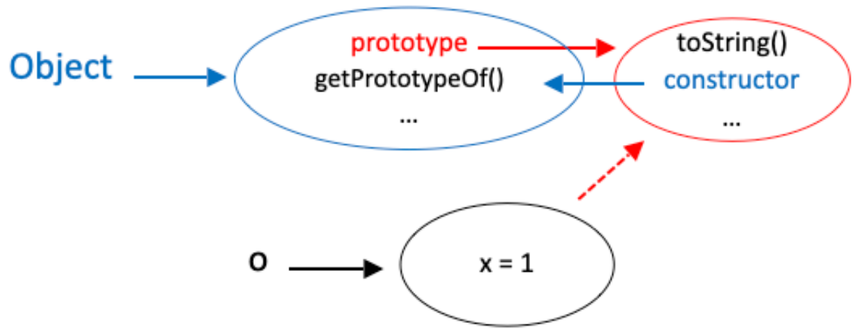
Constructor funktion: Person

```
// constructor3.js
function Person(navn) {
    this.navn = navn;
    Person.antal++
}
Person.antal = 0;
Person.prototype.toString = function () { return 'Person: ' + this.navn; }

let person = new Person('Viggo');
console.log(person.navn); // Viggo
console.log(Person.antal); // => 1
console.log(person.toString()); // => Person: Viggo
```

Object er en constructor funktion

```
// Object.js
let o = new Object();
o.x = 1;
console.log(o.toString()) // => [object Object]
console.log(o.constructor); // => [Function: Object]
console.log(Object.prototype === Object.getPrototypeOf(o)); // => true
```



Object.prototype

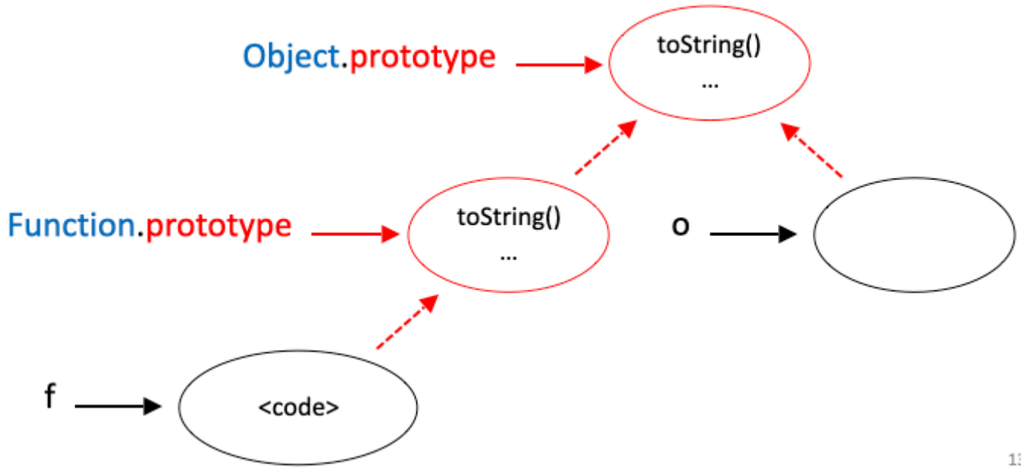
- CRUD på `Object.prototype` har virkning for alle objekter

```
// Object.js
let o = new Object();
o.x = 1;
console.log(o.toString()) // => [object Object]
console.log(o.constructor); // => [Function: Object]
console.log(Object.prototype === Object.getPrototypeOf(o)); // => true

Object.prototype.toString = () => 'overskrevet';
console.log(o.toString()); // 'overskrevet
delete Object.prototype.toString;
console.log(o.toString()); // => TypeError: o.toString is not a function
a(o.toString); // => undefined
Object.prototype.prototype = function () { return Object.getPrototypeOf(this)};
console.log(o.prototype() === Object.prototype); // => true
```

Function er også en constructor funktion

```
// function.js
let f = function () {};
console.log(f.toString()); // => function () {}
let o = {};
console.log(o.toString()); // => [object Object]
```



MDN web docs

- Søg på mdn – fx mdn array
- Array er også en constructor funktion
- Bemærk brugen af prototype i web docs:

```
// Array.js
let array = new Array(1, 2, 3);
// let array = [1, 2, 3];
console.log(array.toString()); // => 1,2,3
console.log(array.length); // => 3
console.log(Array.isArray(array)); // => true
console.log(typeof array); // => object
```

Class notation

✤Class notation er en ny måde at erklære en constructor funktion

```
// class1.js
class Person {
    constructor(navn) {
        this.navn = navn;
    }
    toString() { return 'Person: ' + this.navn; };
}

let person = new Person('Viggo');
console.log(person.navn); // => Viggo
console.log(person.toString()); // => Person: Viggo
console.log(person.constructor.name); // => Person
console.log(person instanceof Person); // => true
console.log(typeof person.constructor); // => function
console.log(typeof Person.prototype); // => object
```

Static

- Metoder og properties på constructor funktionen angives med `static`

```
// class2.js
class Person {
    constructor(navn) {
        this.navn = navn;
        Person.antal++;
    }
    static antal = 0;
    toString() { return 'Person: ' + this.navn; };
}

let person = new Person('Viggo');
console.log(person.navn); // => Viggo
console.log(Person.antal); // => 1
console.log(person.toString()); // => Person: Viggo
```

Indkapsling

- Navne på private metoder og properties skal starte med #

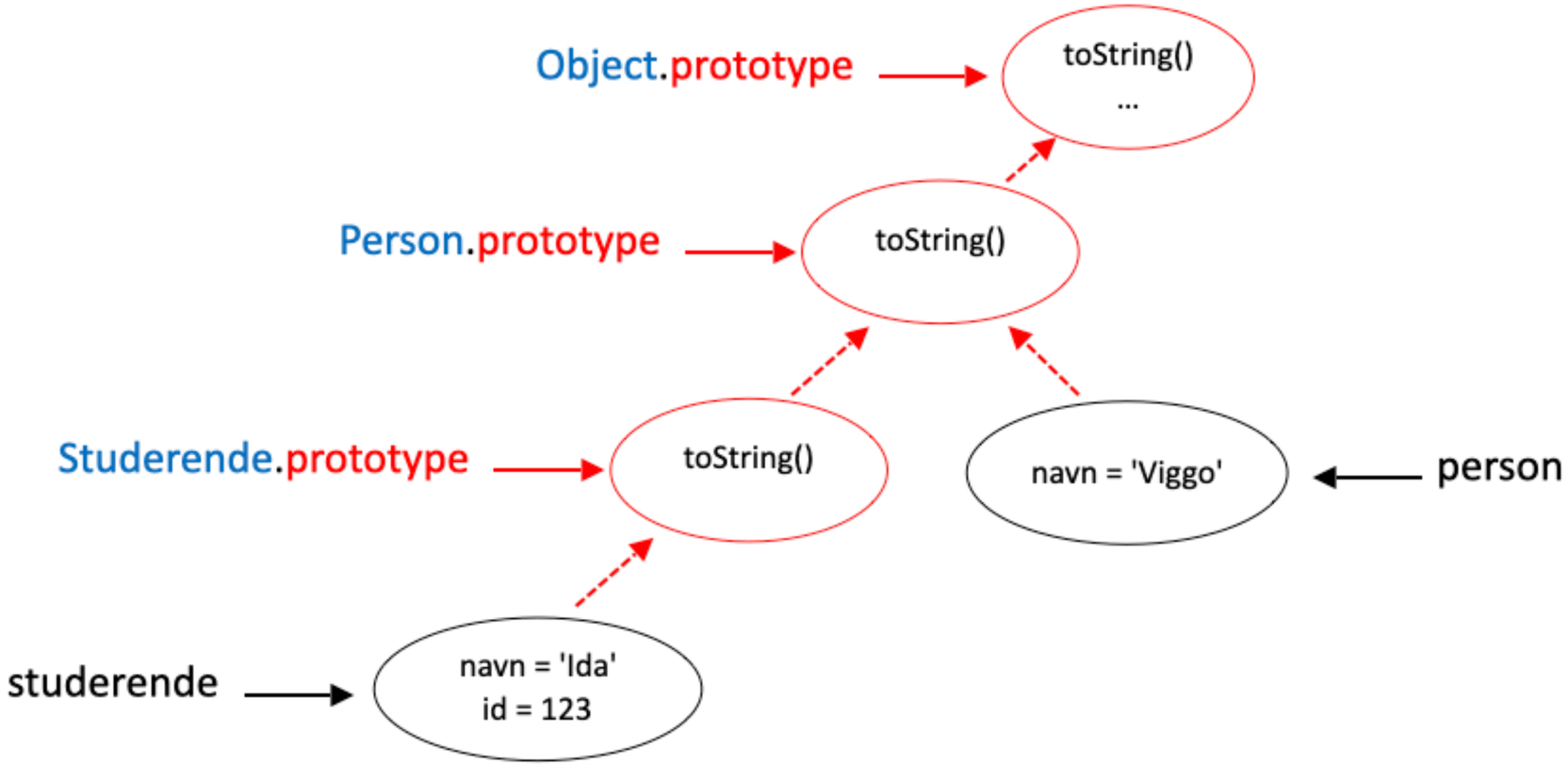
```
// class3.js
class Person {
    constructor(navn) { this.#navn = navn; Person.#antal++; }
    #navn;
    static #antal = 0;
    toString() { return 'Person: ' + this.#navn; };
    getNavn() { return this.#navn; }
    static getAntal() { return Person.#antal }
}

let person = new Person('Viggo');
console.log(person.getNavn()); // => Viggo
console.log(Person.getAntal()); // => 1
console.log(person.toString()); // => Person: Viggo
```

Specialisering

```
// specialisering.js
class Person {
    constructor(navn) { this.navn = navn; }
    toString() { return this.navn; }
}
class Studerende extends Person {
    constructor(navn, id) {
        super(navn);
        this.id = id;
    }
    toString() { return super.toString() + ": " + this.id; };
}
let person = new Person("Viggo");
let studerende = new Studerende("Ida", 123);
console.log(person instanceof Person); // => true
console.log(person instanceof Studerende); // => false
console.log(studerende instanceof Person); // => true
console.log(studerende instanceof Studerende); // => true
```

Specialisering



Polymorfi

```
// polymorfi.js
class Person {
    constructor(navn) { this.navn = navn; }
    toString() { return 'Person: ' + this.navn; };
}
class Kat {
    constructor(navn) { this.navn = navn; }
    toString() { return 'Kat: ' + this.navn; };
}

let kat = new Kat('Garfield');
let person = new Person('Viggo');

for (o of [kat, person])
    console.log(o.toString());
// => Kat: Garfield
// => Person: Viggo
```

Selektiv catching

- Lav specifikke fejltyper ved specialisering af Error
- Anvend selektiv catching, så andre fejl sendes videre

```
// selektiv-catching.js
class PersonError extends Error { } // arver constructor
// ... throw new PersoError("Person fejl: " + ...); ...

try {
    // her er der kode, der kan give en fejl
} catch (e) {
    if (e instanceof PersonError) {
        // ...
    } else {
        throw e;
    }
}
```