JavaScript
ECMA

Bjarte Kileng

HVL

12. oktober, 2021

ECMA

- De grunnleggende strukturene i Javascript, som variabler, setningsbygning og deklarasjoner.

ECMA

JavaScript, et språk nokså ulikt Java

- Javascript er type-svak.
  - Type bestemmes ved tilordning.
- Operator *typeof* returnerer type som variabel har nå.
- Operator *instanceof* sjekker om et objekt er opprettet ved eller arver en gitt konstruktør (eller JavaScript klasse).
- Størrelse *this* avhenger av kontekst.
  - Referer ikke alltid forekomst av klasse selv i medlemsmetode.
- Formelle parametre trenger ikke matche benyttede parametre.
- JavaScript sin klasse-konstruksjon er dynamisk, og metoder og egenskaper kan bli lagt til, fjernet og endret mens program kjører.

ECMA og *strict* mode

- Bruk alltid *strict* mode.
- Skrivefeil i variabelnavn vil gi JavaScript feil.
  - Uten *strict* blir istedet en ny variabel opprettet.
- Uten *strict* blir mange feil ignorert uten noen melding, f.eks.:
  - Forsøk på tilordning til ikke skrivbar egenskap.
  - Forsøk på å slette en ikke slettbar egenskap.
- Reserverte ord som «eval» og «arguments» kan ikke brukes som variabler.
- Reserverte ord for kommende Javascript-versjoner blir ulovlige som variabelnavn, f.eks. «public», «private», «protected», «static», «interface».
- Bruk av *with* konstruksjonen er ulovlig.
- Hvis metode kjøres utenfor kontekst av et objekt er «this» udefinert.

Global og lokal *strict* mode

- Global strict mode ved å legge «"use strict";» i starten av koden.

```
"use strict";

number=7; // Error
function aFunck () {
    name="ole"; // Også dette gir Error
}
```

- Lokal strict mode ved å legge «"use strict";» i starten av funksjon:

```
number=7; // OK da vi ikke har global strict mode

function aFunck () {
    "use strict";

    name="ole"; // Error da vi er i lokal strict mode
}
```

Variabel deklarasjoner – *let*

- Oppretter variabel som er lokal til blokk, f.eks. løkke, if-setning, eller «{...}».

```
function aFunc() {
    {
        let number = 77;
        console.log(number); // "number" er definert
    }
    console.log(number); // "number" er ikke definert
}

aFunc();

console.log(number); // "number" er ikke definert
```

Variabel deklarasjoner – *const*

- Oppretter en konstant.

```
const PI=3.14;
PI=4; // Error

const a = new Array();
a[0] = 77; // OK, da det er referansen som er konstant
a = new String(); // Error
```

- Samme synlighet som *let*.

```
function aFunc() {
    {
        const number = 77;
        console.log(number); // "number" er definert

    }
    console.log(number); // "number" er ikke definert
}

aFunc();

console.log(number); // "number" er ikke definert
```

Variabel deklarasjoner – *var*

- Oppretter variabel som er synlig i funksjon eller globalt.

```
function aFunc() {
    {
        var number = 77;
        console.log(number); // number er definert
    }
    console.log(number); // number er definert
}

aFunc();
console.log(number); // number er ikke definert
```

Unngå bruk av *var*

Mange JavaScript eksempler på nettet bruker *var* for å deklarere variabler.

Variabler opprettet med *let* og *const* har mer begrenset synlighet. Ikke gi variabler større rekkevidde enn nødvendig.

Kjedet tilordning til variabler

- Kjedet tilordning:

```
a = b = 23; // Både a og b settes lik 23
```

Utpakking av Array til variabler

- Utpakking av Array:

```
[a,b] = [2,5]; // Variabel "a" settes lik 2, og "b" til 5
```

- Ombytting av to variabler:

```
[a,b] = [b,a]; // Variabler "a" og "b" bytter verdier og typer
```

- Utpakking av Array til variabler, og samtidig redefinere Array:

```
[a,b,...tabell] = tabell;
```

- Ignorere noen verdiene fra Array:

```
[a,,c] = [1,2,3,4,5]; // Varibel "a" settes lik 1 og "c" til 3
```

- Fjerne de to første verdier i Array:

```
[,,...tabell] = tabell;
```

Initsialisere objekt

- Direkte tilordning:

```
const person = {id: 121, navn: "Ole"};
```

- Via variabelverdier:

```
const number = 121;
const name = "Ole";
const person = { id: number, navn: name };
```

- Via variabeler:

```
const id = 121;
const navn = "Ole";
const person = { id, navn };
```

- Ved uttrykk for egenskap:

```
const p1 = "id";
const p2 = "navnet";
const person = { [p1]:121, [p2.substring(0,4)]: "Ole" };
console.log(person.navn);
```

Objekt med funksjon

- Som objektegenskap:

```
const person = {
    born: new Date(1997,0,5), // 5. januar 1997
    name: "Ole",
    numdays: function() {
        const milliseconds = Date.now() - this.born.getTime();
        const days = Math.floor(milliseconds / 86400000);
        return days;
    }
};
```

- Kortform:

```
const person = {
    born = new Date(1997,0,5), // 5. januar 1997
    name: "Ole",
    numdays() {
        const milliseconds = Date.now() - this.born.getTime();
        const days = Math.floor(milliseconds / 86400000);
        return days;
    }
};
```

Utpakking av Object til variabler

- Utpakking av Object:

```
const person = { id: 121, navn: "Ole" };
const {id,navn} = person;
```

- Kan endre variabelnavn:

```
// Tilordne variabel a fra person.id, og b fra person.navn
let {id:a,navn:b} = person;
```

Utpakking av funksjonsparametre

- Utpakking av Array til formelle parametre:

```
function summer([a,b]) {return a+b}
const result = summer([2,5]);
```

- Utpakking av Array med default verdier:

```
function produkt([a,b=1,c=2]) {return a*b*c}
const result = produkt([12,3]); // c vil få default verdi 2
```

- Utpakking av Object til formelle parametre:

```
function showPerson({id,name}) {
    console.log(`Person har id ${id} og navn ${name}`);
}
showPerson({id: 12, name: "Ole Olsen"});
```

- Utpakking av returverdi fra funksjon:

```
function numbers() {return [1,7,4]}
const [a,b,c] = numbers(); // a lik 1, b lik 7 og c lik 4
```

Anonyme funksjoner

- Eksempel med anonyme funksjoner.

```
let y = function (x) {return 2*x};

aFunc("A",32,y);

bFunc("A",32,function (x) {return 2*x});
```

- Pil-notasjon for anonym funksjon:

```
let y = (x) => {return 2*x};
```

- Observer at de to formene ikke er helt identiske:
  - Pil-formen binder ikke verdier som *this* og *arguments* (detaljer senere).

Pil notasjon for funksjoner

- Vi kan utelate parenteser hvis kun ett argument:

```
(x) => {return 2*x};

// Ekvivalent form

x => {return 2*x};
```

- Hvis funksjonskroppen kun returnerer en verdi:

```
x => {return 2*x};

// Ekvivalent form

x => 2*x;
```

- Les mer om Arrow functions i Mozilla sin dokumentasjon.

Noen begrensinger ved bruk av pil-notasjon for funksjoner

- Har ingen binding av **arguments**.
- Lager ikke sin egen binding av **this**.
- Kan ikke brukes for klasse-metoder eller for metoder til objekter.

ECMA primitive datatyper

- **Undefined**
- **Null**
- **Boolean**
- **Number**
- **BigInt**
- **String**
- **Symbol**
- **Boolean**, **Number** og **String** har en tilsvarende wrapper-klasse.

Litt om **Symbol**

- Funksjonen *Symbol* generer ny verdi hver gang.
- Verdiene er unike.
- Brukes for nøkler og ID-er.

```
const student = {
    id: Symbol(),
    navn: "Ole Olsen"
};
```

- Verdien i seg selv er uinteressant.
- Svarer til bruken av primærnøkkel med auto_increment i databasesystem.

Noen ECMA objekter

- Det globale objektet – Globale metoder og variabler er egenskaper til det globale objektet.
  - I et nettleservindu er *window* det globale objektet.
- **Math** – Matematiske metoder og konstanter.
- **JSON** – Metoder for å håndtere JSON.
- **Intl** – API for internasjonalisering.

JSON

- Objektet har metoder for å omforme mellom tekst i JSON syntaks og JavaScript objekter.
- Omforme et JavaScript objekt til en JSON-tekst:

```
const tabell = [1,5,7,-23];
const JSONString = JSON.stringify(tabell);
```

- Omforme en JSON-tekst til JavaScript objekt:

```
const JSONString = "[1,5,7,-23]";
const tabell = JSON.parse(JSONString);
```

- Observer at JSON objektet er nøye på syntaksen.
  - Ingen metoder er tillatt i JSON-teksten.
  - Kun vanlig hermetegn er lovlig for å angi nøkler og verdier, dvs, ".
  - Nøkler må være omsluttet av hermetegn.

Intl

- Internasjonaliserings API (egen standard, ikke ECMA).

```
const options = {
    "style": "currency",
    "currency" : "NOK",
    "currencyDisplay":"name"
};
const l10nNONumber = new Intl.NumberFormat("nb-NO", options);
const price = 234.77;
const message = `Prisen er ${l10nNONumber.format(price)}`;
```

- Utvider **Number**, **String** og **Date** med internasjonaliseringsmetoder.

```
const options = {
    "style": "currency",
    "currency" : "NOK",
    "currencyDisplay":"name"
};
const price = 234.77;
const priceString = price.toLocaleString("nb-NO",options);
const message = `Prisen er ${priceString}`;
```

Noen ECMA klasser

- **Object** – De fleste objekter arver **Object**.
- **Array** – Tabeller og lister.
  - ECMA inkluderer også en mengde TypedArray klasser.
- **Map**.
- **Set**.
- **String** – Tekststrenger.
- **Number** – Tall.
- **Boolean**.
- **Date**.
- **RegExp** – Regulære uttrykk.
  - Finne mønstre i tekst.
- **Error**.
  - Mange klasser for feil. Alle arver **Error**.
- **Promise**.

Opprette forekomst av **Object**

- JSON syntaks:

```
const person = {};
```

- Med bruk av *new*:

```
const person = new Object();
```

Arbeide med **Object** egenskap

- JSON:

```
person = {"firstname": "Ole"};
```

- JSON også uten hermetegn på egenskap:

```
person = {firstname: "Ole"};
```

- Notasjon med punktum:

```
person.firstname = "Ole";
```

- Bruk av [...]:

```
person["firstname"] = "Ole";
```

- Kan også gjøres med statiske metoder til *Object*.
  - **Object** sine statiske metoder er ikke pensum.

Array

- Nokså ulikt tabeller i Java.
  - Liste-type objekt.
- Antall elementer kan endres under kjøring.
- Kan blande elementer av forskjellige typer i samme tabell.
- Opprette tabell ved bruk av konstruktørfunksjonen **Array**:

```
const farger = new Array("rød","grønn","blå","gul");
```

- Opprette tabell ved bruk av JSON syntaks:

```
const farger = ["rød","grønn","blå","gul"];
```

Map

- Klassen brukes for å lagre nøkkel-verdi par.
- Nøkkel kan være en hvilken som helst type.
- Opprette:

```
const studenter = new Map();
studenter.set(123,{fornavn: "Ole",etternavn: "Olsen"});
studenter.set(521,{fornavn: "Anne",etternavn: "Annesen"});
```

Set

- Klassen brukes for å lagre unike verdier av hvilken som helst type.

```
const elbiler = new Set();
elbiler.add("EK12345");
elbiler.add("EK54321");
```

- Hvis verdi allerede eksisterer i **Set** skjer ingenting.

RegExp

- Representerer mønstre i tekst, *regulære uttrykk*.
- Inkluderer metoder for å søke i og arbeide med tekst.
- Opprette tekstmønster:

```
const elbilRegExp = /\bE[A-Z]\d{5}\b/;
```

- Kan også bruke **RegExp** klassen:

```
const elbilRegExp = new RegExp("\\bE[A-Z]\\d{5}\\b");
```

- Flere **String** metoder tar regulært uttrykk som parameter.
  - Vanligvis bruk disse, heller enn **RegExp** sine metoder
- HTML input-elementer kan valideres mot regulære uttrykk uten å bruke JavaScript.
  - JavaScript kan gi mer forståelige feilmeldinger.

String

- Opprette primitiv tekststreng:

```
const farge = "Rød";
```

- Tekststrenger kan opprettes også med **String** klassen.

```
const farge = new String("Rød");
```

- Normalt bruker vi kun primitive strenger.
  - Alle **String** klassen sine metoder kan brukes også på primitive strenger.
  - Primitiv streng og streng ved klassen **String** har ulik type.
- String templates lar oss kombinere tekst med variabler:

```
const student = {
    navn: "Ole Olsen",
    id: 134
};

const studentInfo = `${student.navn} har id ${student.id}`;
```

- Sjekk f.eks. MDN for primitiv streng kontra streng som objekt.

Date

- Metoder og egenskaper for å arbeide med datoer.

```
const now = new Date();
```

- Måned er 0-indeksert.

```
// 12. oktober, 2021
const dato = new Date(Date.UTC(2021,9,12));
```

Egendefinerte klasser

- JavaScript har et *class* nøkkelord for å opprettet klasser.
  - Noen vil hevde at JavaScript ikke har klasser, men JavaScript dokumentasjonen omtaler dette som klasser.
- Nøkkelord som *private*, *public* og *protected* vil gi feil.
- Nøkkelord *static* for statiske metoder og felt.
- Felt og metode er privat hvis navn starter med tegnet #.
  - Kun i de nyeste nettleserne.
  - Firefox fra sommeren 2021.

Demo med egendefinert klasser

```
class Dice {
    // Public felt
    maxnumber;
    value = null;

    constructor(maxnumber = 6) {
        this.maxnumber = maxnumber;
    }

    throwDice() {
        this.value = 1 + Math.trunc(this.maxnumber*Math.random());
    }
}

const dice = new Dice(6);
dice.throwDice();

console.log(dice.value);
```

- Observer, ingen Java-type metode *getValue* for egenskapen *value*.
- Hvis sjekk på data før tilordning, bruker istedet en *setter* (neste slide).

Getters og setters

- Getters og setters er klassemetoder som aksesseres som egenskaper.
- Binder attributt til en metode.
- Getter-metoden kjøres for å returnere verdi fra egenskapen.

```
const p = new Person("Ole", "Persen");

console.log(p.age); // Kjører Person sin get metode age()
```

- Setter-metoden kjøres for å gi verdi til egenskapen.

```
const p = new Person("Ole", "Persen");

// Kjører age(22)
p.age = 22; // Kjører Person sin set metode age(22);
```

Kodeeksempel med getters og setters

```
class Person {
    firstname;
    lastname;
    borndate;

    constructor(firstname,lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.borndate = new Date();
    }

    set age(age) {
        let bornyear = this.borndate.getFullYear() - age;
        this.borndate.setYear(bornyear);
    }

    get age() {
        const now = new Date();
        return now.getFullYear() - this.borndate.getFullYear();
    }
}

const p = new Person("Ole", "Persen");
p.age = 22;          // Kjører Person sin set metode age(22)
console.log(p.age); // Kjører Person sin get metode age()
```

Bruk av getters og setters

- Data-egenskap kan erstattes med en *getter*/*setter* uten å modifisere ekstern kode.
- *Getter*/*setter* som kun aksessere data-egenskap gir ingenting ekstra i JavaScript.
  - Men, ofte bør verdi kontrolleres ved tilodning, av en *setter*.

Getters/setters og objektorientert programmering

- Klasse instanser har en tilstand og en oppførsel.
  - Oppførsel er gitt ved klassemetodene.
  - Tilstanden er gitt av egenskapene, dvs. data attributtene, klassen sine felt variabler.
- *setters* bryter med god skikk for objektorientert programmering.
  - Både *setters* og *getters* er mye brukt i JavaScript.
- Java sine *setXyz* og *getXyz* metoder kan lage problemer i JavaScript.
  - F.eks. serialiseres *getters*, men ikke *getXyz* type metoder.

JavaScript og objektorientet programmering

En *setter* kan endre tilstand, men aksesseres som en egenskap.