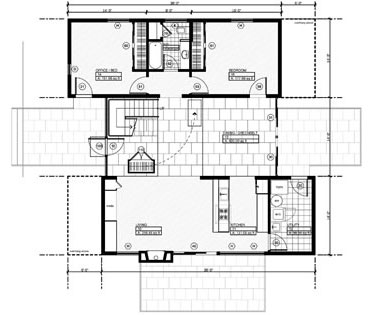
# Objekt Orientert programmering / Object Oriented Programming (OOP)

© Ingar Tødenes







## Hva er OOP

OOP står for "Objektorientert Programmering," og det er en måte å organisere og skrive dataprogrammer på. Objektorientert Programmering vil herfra bli referert til som OOP. I OOP tenker vi på programmet som en samling av "objekter," som er ting eller konsepter i den virkelige verden. Disse objektene har egenskaper (kalt egenskaper/attributter) og ting de kan gjøre (kalt metoder). For eksempel, hvis vi tenker på et smart hus som et objekt, kan egenskaper være farge, serie og størrelse, mens metodene kan være "automatisk låsing" og "oppdaget bevegelse".

OOP hjelper med å organisere og gjenbruke kode ved å gruppere relaterte data og funksjoner sammen i objekter. Dette gjør det lettere å forstå, vedlikeholde og utvide programmer.

Så, i enkle ord, OOP er en måte å skrive programmer som etterligner den virkelige verden ved å bruke objekter med egenskaper og handlinger.

## Pros vs Cons

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * Modularitet * Gjenbrukbarhet * Produktivitet * Enkelt å utvide * Kommunikasjonsgrensesnitt * Sikkerhet (innkapsling & abstraksjon) * Fleksibelt (polymorfism) | * For stort fokus på data delen * Lite fokus på beregninger og algoritmer * Komplisert å skrive * Tar lengre tid å kompilere |

(*What Is Object-Oriented Programming (OOP)?*, u.å.)

## Objekter brukes til

Objekter brukes i programmering for å modellere og representere virkelige eller abstrakte konsepter på en strukturert måte. Her er noen av hovedformålene med å bruke objekter:

1. **Organisering av data:** Objekter tillater deg å gruppere relaterte data sammen. Dette gjør det lettere å organisere og administrere informasjonen i programmet ditt. For eksempel kan du ha en "Person" klasse som inneholder attributter som navn, alder og adresse for å representere en person.
2. **Logisk komponent (Logic component):** Dette er programdeler som utfører en oppgave. Det var logiske komponenter kan opptre som objekt er med på å støtte opp under gjenbrukbarhet og modularitet punktum
3. **Modularitet (Modularity):** Objekter bidrar til å dele programmet ditt inn i mindre, gjenbrukbare deler. Dette gjør koden din mer modulær, noe som forenkler vedlikehold og utvidelse.
4. **Abstraksjon (Abstraction):** Objekter tillater deg å abstrahere bort unødvendige detaljer og fokusere på de viktige egenskapene og funksjonene til en enhet. Dette forenkler designen og gjør koden mer forståelig.
5. **Gjenbruk av kode (Reuse):** Du kan opprette flere instanser (objekter) av samme klasse, noe som gjør det enkelt å gjenbruke funksjonalitet. For eksempel kan du opprette flere "Bil" objekter med samme metoder og attributter.
6. **Polymorfisme (Polymorphism):** Objekter tillater deg å oppnå polymorfisme, der objekter av forskjellige klasser kan respondere på samme metode på forskjellige måter. Dette øker fleksibiliteten i programmet.
7. **Sikkerhet og innkapsling (Encapsulation):** Objekter tillater deg å begrense tilgangen til data ved å bruke privat og offentlig tilgangsnivå. Dette bidrar til å sikre dataene i programmet.
8. **Design og struktur:** OOP fremmer god designpraksis ved å oppmuntre til å dele ansvar og oppgaver mellom klasser og objekter på en logisk måte.
9. A**rv (inheritance)** er et grunnleggende konsept innen OOP der en ny klasse kan arve egenskaper og metoder fra en eksisterende klasse. Dette gjør det mulig å gjenbruke kode og opprette hierarkier av klasser som deler felles egenskaper og funksjonalitet.
10. **Feilhåndtering:** Objekter kan brukes til å representere og håndtere feil og unntak i programmet ditt, noe som gir en mer strukturert tilnærming til feilhåndtering.

I hovedsak gjør bruk av objekter programmering mer organisert, lesbar og vedlikeholdbar. Det lar deg også bygge mer komplekse systemer ved å kombinere objekter på en meningsfull måte.

## Programmering med OOP

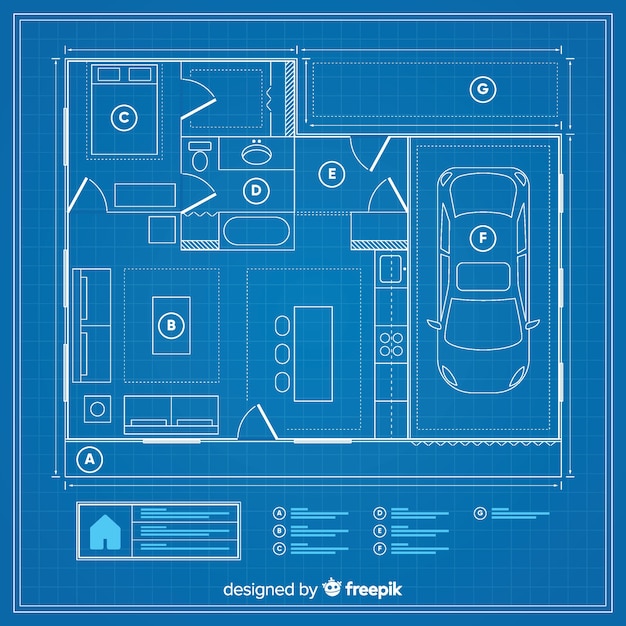
Vi kan utføre OOP i JavaScript. OOP i JS kan virke litt forvirrende, derfor vil vi her benytte ECMAScript 2015 (også kjent som ES6).

«ECMAScript (eller ES) er en programmeringsspråk-spesifikasjon som er standardisert av Ecma International i standardene ECMA-262 og ISO/IEC 16262. Spesifikasjonen var opprinnelig basert på JavaScript, men dette har nå snudd slik at JavaScript følger utviklingen i ECMAScript-standarden.

ECMAScript brukes vanligvis til klientside-skripting på World Wide Web, og i stadig større grad også til serverside-applikasjoner gjennom Node.js-plattformen. Blant andre implantasjoner av ECMAScript finnes JScript og ActionScript.» («ECMAScript», 2019)

### Class (Klasse) vs Object (Objekt)

For å programmere med OOP må vi i de aller fleste tilfellene opprette en klasse som er basisen for objektene som vi skal bruke.

* En **klasse** er en "oppskrift" eller en "mal" for å lage objekter. Den definerer egenskapene (attributtene eller properties på engelsk) og handlingene (metodene også referert til som oppførsel/behavior) som objekter av denne klassen vil ha. For eksempel, hvis vi har en klasse som heter "Hus", den kan definere attributter og metoder som nevnt over. Klassen er ikke et konkret objekt, men den beskriver hvordan objekter av denne typen skal opprettes og fungere. Classes blir ofte referert til som blueprint-en for objektet.  
  <https://www.freepik.com/free-vector/modern-house-sketch-blueprint_5657617.htm>
* Et **objekt** er en konkret forekomst av en klasse. Vi sier at vi lager instanser eller objekter av klassen. Eksempel: I et spill kan vi lage et byggefelt av hus med samme utseende ved å lage flere instanser/objekter av klassen for huset. Hver instans har varianter av farge, størrelse og posisjon. Når du oppretter et objekt av en klasse, følger det egenskapene og metodene som er definert i klassen. For eksempel kan du lage et objekt av klassen "Hus" med en spesifikk farge og modell. Dette objektet representerer et hus med de angitte egenskapene, og du kan bruke metodene for å utføre handlinger på den på huset, som å starte eller stoppe den.

Så, kort sagt, en klasse er en abstrakt beskrivelse av en type objekt, mens et objekt er en konkret forekomst av denne typen, med spesifikke egenskaper og muligheten til å utføre handlinger.

### En klasse (objekt) er oppbygget av

Enkelt forklart har en klasse (objekt) to hovedelement:

* **Data** vanligvis referert til som field, class memebers men også som propertie (egenskap/attributter), men egenskaper er mer en bare data og vi skal komme tilbake til det senere.
* **Metoder** vanligvis referert til som methods, men kan også bli kalt for en funksjon. Metoder samer kode i logiske deler og støtter opp om DRY (Do not Repeat Yourself).
  + En klasse har også noe spesielle metoder, men den aller viktigste er constructor (konstruktøren) som blir gjennomgått om litt.

### Vår første klasse

Med ES6 kan vi bruke nøkkelordet class for å skrive en klasse. Den gjør det samme som den opprinnelige måten å lage klasser/objekter i JavaScript men er ryddigere å enklere å forstå. Denne måten å skrive å skrive klasser på er mer i tråd hva vi gjør i Java and C#.

### Skjelettet til en klasse

Vi starter med nøkkelordet class etterfulgt av navnet til klassen. Navnet til klassen skrives med stor forbokstav. Dette er en uskrevet regel (konvensjon) som hjelper programmerer å legge merke til hva koden sier.

Vi må så opprette kodeblokk for klassen. Kodeblokken for klassen er det som inneholder medlemsdata og metodene (instruksjoner). Kodeblokken i de fleste programmeringsspråk markeres ved å bruke krøllparenteser, også kjent som klammeparenteser {}. Disse klammeparentesene markerer begynnelsen og slutten av en kodeblokk. Se Metoder nedenfor for mer.

Kode Kodeskjelletet

class ClassName {

*// code*

}

Kode Kodeskjellettet for en Hus klasse

class House {

*// code*

}

Denne klassen kan vi bruke til å definere et hus og lage instanser/objekter i programmet som vi kan bruke.

### Fra klasse til objekt

For å opprette et objekt fra en klasse i ES6 (ECMAScript 2015 og senere), følger du disse trinnene:

1. Vi har de**finert klassen**: Vi bruker klassen som vi laget for å bruke den som en mal for å lage objekter.

class Person {

    constructor(name, age) {

      this.name = name;

      this.age = age;

    }

}

1. **Opprett objekt**:
   1. Holderen forobjektet: Vi lager en variabel/konstant for å holde objektet som vi skal bruke:

// Her oppretter vi en konstant som skal holde et objekt fra en klassen

const person1 …;

// eller vi kan lagen en variabel med nøkkelordene let eller var

let person2 …;

* 1. Bruke tildelingsoperatoren (assignment operator) for å fortelle at vi skal lagre noe inn i variablen.

// Tildelingsoperator

const person1 = …;

let person2 = …;

* 1. Bruk `new`-nøkkelordet etterfulgt av klassenavn og eventuelle nødvendige argumenter for konstruktøren. Dette vil lage en instans av klassen (objekt) og lagre den i en variabel.

// Opprett et objekt fra Person-klassen

const person1 = new Person("Alice", 30);

let person2 = new Person("Bob", 25);

### Properties (egenskaper) / field (felt) / class members (klassemedlemmer)

Dette er data som vi oppretter i klassen og er tilgjengelig i hele klassen. Vi lager dem med å lage variabler i klassen og det er vanlig å legge dem øverst. JavaScript trenger at de ligger øverst, men i Java og C# er dette ikke et krav.

**Merk**: klassemedlemmer begrepet omfatter egenskaper og metodene for klassen.

Smart huset vårt trenger noen data. Her er noen eksempel

* Lengde
* Bredde
* Antall etasjer
* Etasje plan(er)
* Utvendig farge
* Lys
* Temperatur
* M.m.

Kode SmartHus klasse med egenskaper (medlemsvariabler)

class SmartHouse {

let length;

  let width;

  let numberOfFloors;

  let floorPlan[];

  let bodyColor;

  let isLightOn;

  let temperature;

}

I Java og C# måtte vi ha spesifisere hvilke typer data som disse variablene skulle være:

Kode SmartHus klasse med egensaper i Java/C#

class SmartHouse {

  float length;          *// float er et desimal tall*

  float width; ;         *// float er et desimal tall*

  int numberOfFloors;    *// int er et heltall*

  FloorPlan floorPlan[]; *// dette er et array (tabell) som holder*

*// alle etasje planene. FloorPlan er en annen*

*// klasse som definerer en etasjeplan*

  String bodyColor;      *// String er tekst som representer en fargekode*

  Boolean isLightOn;     *// Boolean er en verdi som holder true (sant)*

*// eller false (usant) og sier om lyset er på*

*// eller ikke.*

  float temperature;     *// float er et desimal tall for å holde*

*// temperaturen i huset*

}

### Metoder

En metode (funksjon) i programmering handler om å lage kodeblokker som utfører spesifikke oppgaver eller handlinger.

En metode består av følgende deler:

1. **Metodenavn:** Dette er navnet du gir til metoden. Det bør være beskrivende og fortelle hva metoden gjør.
2. **Parenteser:** Metodenavnet er etterfulgt av parenteser, vanligvis tomme eller med parametere inni dem. Parameter er verdier eller data som metoden trenger for å utføre sin oppgave. Hvis metoden ikke tar imot parametere, bruker du tomme parenteser.
3. **Kodeblokk:** Metoden har en kodeblokk som inneholder instruksjonene som skal utføres når den kalles. Dette er selve koden som utfører oppgaven til metoden.  
   Kodeblokken i de fleste programmeringsspråk markeres ved å bruke krøllparenteser, også kjent som klammeparenteser {}. Disse klammeparentesene markerer begynnelsen og slutten av en kodeblokk. Her er hvordan de fungerer:  
   *Åpning av kodeblokk:*  
   Kodeblokken starter med en åpningskrøllparentes {.  
   Dette indikerer begynnelsen på koden som skal utføres innenfor blokken.  
   *Avslutning av kodeblokk:*  
   Kodeblokken avsluttes med en lukkekrøllparentes }.  
   Dette indikerer slutten på koden som skal utføres innenfor blokken.
4. **Returner verdi (valgfritt):** Noen ganger returnerer metoder en verdi tilbake til det stedet der de ble kalt fra. Denne verdien er angitt ved bruk av `return`-nøkkelordet. Hvis metoden ikke returnerer noe, vil ikke metoden i JavaScript benytte ikke `return`-nøkkelordet. I andre programmeringsspråk må man bruke `void`-nøkkelodet slik som void displayResult().

Kode En kalkulator med metoder som returnerer verdi

class Calculator {

*// Spesiell metode kalt constructor (konstruktør).*

  constructor() {

    this.result = 0;

  }

*// Metode for å legge sammen to tall og returner verdien*

  add(*a*, *b*) {

    this.result = a + b;

    return this.result;

  }

*// Metode for å trekke fra to tall og returner verdien*

  subtract(*a*, *b*) {

    this.result = a - b;

    return this.result;

  }

*// Metode returner ikke data. Viser resultat i console.log()*

  displayResult() {

    console.log(`Resultatet er: ${this.result}`);

  }

*// Metode returner ikke data. Resetter result til 0 og gir melding i*

*// console.log()*

  clearResult() {

    this.result = 0;

    console.log("Resultatet er ryddet.");

  }

}

*// Opprett en instans av Kalkulator-klassen*

const minKalkulator = new Calculator();

*// Bruk metodene til å utføre beregninger*

const resultat1 = minKalkulator.add(5, 3);       *// resultat1 blir 8*

const resultat2 = minKalkulator.subtract(10, 2); *// resultat2 blir 8*

minKalkulator. displayResult(); *// Skriver resultatet av siste kalte*

*// funksjon i console.log()*

minKalkulator. clearResult();   *// Nullstiller resultatet og gir beskjed*

*// i console.log()*

I dette eksempelet er `add()` og `subtract()` metoder i `Calculator`‑klassen. De har metodenavn, parametere (a og b), kodeblokker som utfører addisjon og subtraksjon, og de returnerer resultatene ved hjelp av `return`-nøkkelordet. Når metoden kalles, utfører den handlingen og gir tilbake resultatet.

#### Spesielle metoder i JavaScript

I JavaScript-klasser er det flere spesielle metoder, også kjent som "magiske metoder" eller "spesielle metoder," som gir ekstra funksjonalitet til klassene. Her er noen av de viktigste spesielle metodene og eksempler på hvordan de brukes:

##### Constructor Metoden (`constructor()`):

`constructor()`-metoden er den første metoden som kalles når du oppretter en ny instans av en klasse. Den brukes til å initialisere egenskaper og tilstand for objektet.

##### JavaScript har også en destruktormetode kalt `destructor()`.

En destructor()-metode skal egentlig kjøres når objektet blir destruert av run-time kjøreren/tolkeren. Denne metoden er en del av ECMAScript-proposisjonen, spesifisert i ECMAScript Private Fields-proposisjonen, som ble introdusert for å hjelpe til med håndtering av ressurser som må frigjøres når et objekt blir ødelagt.

Her er en enkel illustrasjon av hvordan du kan bruke en destruktormetode:

Kode Eksempel på en destructor()

class Resource {

  constructor() {

    this.data = [1, 2, 3, 4];

  }

  destructor() {

    console.log("Ressursen frigjøres.");

*// Legg til koden for å frigjøre ressurser, som f.eks. lukking*

*// av filer eller nettverksforbindelser.*

  }

}

*// Opprett en instans av Resource-klassen*

const resource = new Resource();

*// Når du er ferdig med ressursen, kan du kalle destructor()-metoden*

*// for å frigjøre ressurser.*

resource.destructor();

Det er viktig å merke seg at støtten for destruktormetoden er begrenset til bestemte JavaScript-miljøer og implementeringer, og den er ennå ikke en offisiell del av ECMAScript-standardene. Det er også viktig å merke seg at JavaScript har en innebygd søppeltømming (garbage collection) som normalt håndterer frigjøring av ressurser, så manuell ressursfrigjøring er sjelden nødvendig i JavaScript.

Man bør likevel tenke på å bruke destructor() for klasser som jobber med data. Eksempelvis med objekter som jobber mot databaser der man må passe på å stenge ned koblingen til database når man er ferdig.

JavaScripts sin søppelsamler kjører ikke destruktørmetoder slik noen andre programmeringsspråk gjør. I stedet frigjør den automatisk minne ved å identifisere og rydde opp i objekter som ikke lenger kan bli nådd eller referert i koden. Hvis du har spesifikke oppryddingsoppgaver som skal utføres når et objekt ikke lenger er nødvendig, kan du bruke `addEventListener()`-metoden til å håndtere slike oppgaver manuelt.

I JavaScript finnes det ikke en spesifikk hendelseslytter (EventListener) som direkte oppdager når et objekt blir slettet eller plukket opp av søppelsamler-en. JavaScripts sin søppelsamler gjøre dette automatisk og generelt abstrahert fra direkte utviklerkontroll.

Du kan imidlertid bruke `beforeunload`-hendelsen i noen sammenhenger for å utføre opprydding før en side blir avlastet, noe som kan innebære å frigjøre referanser til objekter. For eksempel:

Kode EventListener som kan brukes for å fange opp når et objet blir fjernet

window.addEventListener('beforeunload', function (*event*) {

*// Utfør opprydding her, frigjør referanser til objekter hvis nødvendig*

*// Dette garanterer kanskje ikke at objektet umiddelbart søppelsamles,*

*// men det er en måte å utføre opprydding før siden avlastes."*

});

Husk at JavaScripts søppelsamling er ikke-deterministisk, så du kan ikke stole på en bestemt hendelse for å fortelle deg når et objekt umiddelbart blir slettet. Det er beste praksis å håndtere objektlivssykluser og referanser nøye for å sikre riktig minnehåndtering."

##### Getter og Setter Metoder:

Getter- og setter-metoder brukes til å lese og endre verdier på objektegenskapene. De gir kontroll over tilgang til egenskaper. get og set er to nøkkelord som angir funksjonen.

Mer om dette i emenet *Bruke egenskaper med Getter og Setter* nedenfor.

##### valueOf()**-**Metoden**:**

`valueOf()`-metoden brukes til å hente den "primitiv" verdien av et objekt, som vanligvis er den samme som `toString()`. Den brukes i sammenheng med operasjoner som sammenligner objekter.

Eksempel:

class Temperature {

  constructor(*value*) {

    this.value = value;

  }

  valueOf() {

    return this.value;

  }

}

const temp1 = new Temperature(25);

const temp2 = new Temperature(30);

console.log(temp1 < temp2); *// Sammenligner verdiene, Utskrift: true*

##### toString()-Metoden:

`toString()`-metoden er en metode som brukes til å konvertere et objekt til en tekststreng når objektet representeres som en streng.

Eksempel:

class Book {

  constructor(*title*, *author*) {

    this.title = title;

    this.author = author;

  }

  toString() {

    return `${this.title} by ${this.author}`;

  }

}

const book = new Book("The Catcher in the Rye", "J.D. Salinger");

console.log(book.toString()); *// Utskrift: The Catcher in the Rye by*

*// J.D. Salinger*

#### constructor()-Metoden

Som nevnt over så er `constructor()`-metoden er den første metoden som kalles når du oppretter en ny instans av en klasse. Initialisere betyr å sette opp objektet som vi skal bruke, ved å sette variabler/egenskaper.

I JavaScript definerer vi en metode constructor() som blir konstruktøren for denne klassen/objektet. Mens i andre programmeringsspråk vil konstruktøren defineres ved å lag en metode som heter de samme som klassen.

Konstruktøren kan også ta parametere som er data som skal settes ved et opprettelsen av objektet.

I JavaScript-klasser blir constructor()-metoden kalt automatisk når et nytt objekt blir opprettet fra klassen.

constructor()-metoden tar ofte imot parametere som skal brukes til å initialisere objektets egenskaper. Disse parametrene blir vanligvis satt som medlemsvariabler i objektet ved hjelp av this-nøkkelordet. Mer om this-nøkkelordet senere.

#### valueOf()

I JavaScript er `valueOf()` en innebygd metode som finnes i mange objekter. Denne metoden brukes til å hente den "primitive" verdien av et objekt. Primitiv verdi betyr den mest grunnleggende, enkle representasjonen av dataen, som kan være et tall, en streng, en boolsk verdi, null eller undefined, avhengig av objekttypen.

##### Primitiv verdi / Primitiv form

Primitiv verdi, også kjent som primitiv form, refererer til den mest grunnleggende typen data som kan representeres i et programmeringsspråk. Dette er data som ikke er sammensatt av flere deler, og som ikke har metoder eller egenskaper. Primitiv verdi er det enkleste nivået av datatyper, og den eksisterer som den er, uten å være en referanse til andre objekter.

I JavaScript er det flere primitiv verdityper:

1. **Number:** Dette representerer numeriske verdier, for eksempel heltall eller desimaltall.
2. **String:** Dette representerer tekststrenger, for eksempel ord eller setninger.
3. **Boolean:** Dette representerer boolske verdier, som kan være enten `true` eller `false`.
4. **Null:** Dette representerer en spesifikk verdi, nemlig "ingen verdi" eller "mangler verdi."
5. **Undefined:** Dette representerer variabler som er deklarert, men ikke tildelt noen verdi.
6. **Symbol:** Dette representerer unike og uendringsbare verdier som brukes som objektegenskaper.
7. **BigInt:** Dette representerer store heltall som ikke kan representeres med vanlige Number-verdier.

Primitiv verdier er enkle å forstå og sammenligne, da de sammenlignes etter verdi, ikke referanse. For eksempel, når du sammenligner to tall, sammenligner du de faktiske tallverdiene, ikke referansene til tallene.

I motsetning til primitive verdier er objekter sammensatte verdier som kan inneholde metoder og egenskaper. Primitiv verdier brukes ofte som grunnleggende byggesteiner i programmering og er det første nivået for datatyper som programmerere arbeider med.

##### `valueOf()` returnerer

1. **Number-objekter:** For Number-objekter returnerer `valueOf()` selve tallverdien som en vanlig numerisk verdi.
2. **String-objekter:** For String-objekter returnerer `valueOf()` strengen i sin primitive form.
3. **Boolean-objekter:** For Boolean-objekter returnerer `valueOf()` selve boolsk verdien (true eller false) som en vanlig boolsk verdi.
4. **Date-objekter:** For Date-objekter returnerer `valueOf()` datoen som en Unix-tidsstempel, som er antall millisekunder siden 1. januar 1970.
5. **Egendefinerte objekter:** Du kan tilpasse hvordan `valueOf()` oppfører seg for egendefinerte objekter ved å definere metoden for ditt eget objekt.

Kode valueOf() eksempler

class CustomNumber extends Number {

  constructor(*value*) {

*super*(value);

  }

  valueOf() {

    return *super*.valueOf();

  }

}

class CustomString extends String {

  constructor(*value*) {

*super*(value);

  }

  valueOf() {

    return *super*.toString();

  }

}

class CustomBoolean extends Boolean {

  constructor(*value*) {

*super*(value);

  }

  valueOf() {

    return *super*.valueOf();

  }

}

class CustomDate extends Date {

  constructor(*dateString*) {

*super*(dateString);

  }

  valueOf() {

    return *super*.getTime();

  }

}

class CustomObject {

  constructor(*value*) {

    this.value = value;

  }

  valueOf() {

    return this.value \* 2;

  }

}

const numObj = new CustomNumber(42);

const strObj = new CustomString("Hello");

const boolObj = new CustomBoolean(true);

const dateObj = new CustomDate("2023-09-27");

console.log(numObj.valueOf()); *// 42 (vanlig tall)*

console.log(strObj.valueOf()); *// "Hello" (vanlig streng)*

console.log(boolObj.valueOf()); *// true (vanlig boolsk verdi)*

console.log(dateObj.valueOf()); *// Unix-tidsstempel*

const customObj = new CustomObject(100);

console.log(customObj.valueOf()); *// 200*

Dette kodeeksempelet tar for seg en del programmeringskonsept som vi kan være til senere.

I moderne JavaScript, spesielt med bruk av primitiver og typekonvertering, brukes `valueOf()` sjelden, men det kan være nyttig i spesifikke situasjoner der du trenger å hente den grunnleggende verdien av et objekt.

#### toString()

I JavaScript er `toString()` en innebygd metode som finnes i de fleste objekter. Denne metoden brukes til å konvertere et objekt til en streng. Resultatet av `toString()` er en representasjon av objektet som en lesbar tekststreng.

C++, C# og java har metoden toString(). Smalltalk er et tidlig objektorientert språk som ikke har en standard toString()-metode. Imidlertid gir det deg full kontroll over hvordan objekter skal presenteres ved å tilpasse metoder som printOn:. Python og Ruby har noen spesielle varianter. Java på den andre siden har toString() innebygd overalt.

##### Objekttype VS toString()

Avhengig av objekttypen kan `toString()` oppføre seg på forskjellige måter:

* **Number-objekter:** `toString()` konverterer Number-objekter til deres numeriske verdier som tekststrenger.
* **String-objekter:** `toString()` returnerer den originale strengen som en tekststreng.
* **Boolean-objekter:** `toString()` returnerer "true" for `true`-verdier og "false" for `false`-verdier.
* **Datoobjekter:** `toString()` returnerer en tekststreng som representerer datoen og klokkeslettet i en lesbar form.
* **Array-objekter:** `toString()` konverterer alle elementene i arrayen til tekststrenger og returnerer dem som en komma-separert liste.
* **Egendefinerte objekter:** Du kan tilpasse hvordan `toString()` oppfører seg for egendefinerte objekter ved å definere din egen `toString()`-metode for objektet.

Når du kaller toString()-metoden på et egendefinert objekt i JavaScript uten å lage en spesifikk toString()-metode i objektets prototypekjede, vil JavaScript automatisk søke etter toString()-metoden i objektets prototypekjede, inkludert i den øverste prototypen, som er Object.prototype.

Hvis toString()-metoden ikke er funnet i objektets prototypekjede, vil JavaScript falle tilbake til Object.prototype.toString()-metoden, som er en standardmetode som gir en enkel tekstrepresentasjon av objektet. Denne standardmetoden returnerer en streng som inneholder objektets typenavn, for eksempel "[object Object]". Denne representasjonen er generisk og gir ingen spesifikk informasjon om objektets egenskaper eller tilstand.

Kode Eksempler på toString()

Her er noen eksempler som viser hvordan `toString()` fungerer for ulike objekter:

class CustomNumber extends Number {

  constructor(*value*) {

*super*(value);

  }

  toString() {

    return *super*.toString();

  }

}

class CustomString extends String {

  constructor(*value*) {

*super*(value);

  }

  toString() {

    return *super*.toString();

  }

}

class CustomBoolean extends Boolean {

  constructor(*value*) {

*super*(value);

  }

  toString() {

    return *super*.toString();

  }

}

class CustomDate extends Date {

  constructor(*dateString*) {

*super*(dateString);

  }

  toString() {

    return *super*.toString();

  }

}

class CustomObject {

  constructor(*value*) {

    this.value = value;

  }

  toString() {

    return "Custom Object: " + this.value;

  }

}

const numObj = new CustomNumber(42);

const strObj = new CustomString("Hello");

const boolObj = new CustomBoolean(true);

const dateObj = new CustomDate("2023-09-27");

const customObj = new CustomObject(100);

console.log(numObj.toString()); *// "42"*

console.log(strObj.toString()); *// "Hello"*

console.log(boolObj.toString()); *// "true"*

console.log(dateObj.toString()); *// En tekstrepresentasjon av datoen*

console.log(customObj.toString()); *// "Custom Object: 100"*

#### valueOf() VS toString()

Mens `valueOf()` er vanligvis brukt for å hente den "primitive" verdien av et objekt, brukes `toString()` for å få en tekstlig representasjon av objektet, noe som ofte er nyttig for å vise eller kommunisere data.

### Egenskaper og this-nøkkelordet

Egenskaper representerer data eller tilstanden til objektene som er opprettet fra klassen. Klassens egenskaper er variabler som er tilgjengelige i hele objektet/klassen.

constructor()-metoden brukes til å initialisere objektets egenskaper. Den tar imot parametere som brukes til å sette objektets tilstand når det blir opprettet.

`this` er et nøkkelord i refererer til gjeldende objekt. Det vil si objektet som er i ferd med å bli manipulert eller påvirket av koden. `this` spesielt nyttig for å referere til objektets egenskaper, men kan også brukes til å referer til klassens metoder innenfor klassens definisjon.

Sammenlikne det med deg selv. this kan vi bytte ut med min.

min.sekk = nye sekken;

min.vektIKg = 76;

Du er nå et objekt, men de andre i klassen er sine egne objekt og vil da ha lignende, men ikke de samme egenskapen som blir satt igjennom this (min) nøkkelordet til hver enkelt.

I `constructor()`-metoden kommer det inn argumenter som vi vil lagre i objektet som egenskaper. `this` blir brukt i constructor()-metoden for å referere til objektet som blir opprettet, og for å initialisere eller sette medlemsvariabler.

I en metode kan det være at argumentene har samme navn som egenskapene til klassen. Dette er fult lov, men får å få tak i klassens egenskaper bruker vi this-nøkkelordet, legger inn et punktum og skriver inn navnet på egenskapen vi vil få tak i.

this.name = name;

*JS*

1 class Person {

2   let name;

3   let age;

4

5   constructor(*name*, *age*) {

6   this.name = name; *// Bruker 'this' til å referere til objektets*

7 *// egenskap 'name'*

8     this.age = age;   *// Bruker 'this' til å referere til objektets*

9 *// egenskap 'age'*

10   }

11

12   greet() {

13     console.log(`Hei, jeg heter ${this.name} og er ${this.age} år gammel.`);

14   }

15 }

16

17 *// Oppretter en ny instans av Person-klassen*

18 const person = new Person("Alice", 30);

19

20 *// Kaller greet-metoden for å vise en hilsen*

21 person.greet(); *// Utskrift: Hei, jeg heter Alice og er 30 år gammel.*

Linje 1 defineres Person-klassen ved hjelp av class-nøkkelordet. Dette er grunnlaget for å lage objekter av typen "Person".

Linje 2 og 3 deklarerer du to egenskaper (name og age) ved hjelp av let-nøkkelordet.

Linje 5 er constructor()-metoden. Den tar imot to parametere, name og age.

Linje 6 og 8 initialiserer vi de to egenskapene med disse verdiene fra argumentene. Her bruker vi this-nøkkelordet for å spesifisere at vi vil lagre argumentene i medlemsvariablene fra linje 2 og 3.

Linke 12 har vi greet()-metoden. Dette er en vanlig metode i klassen som ikke tar noen parametere. Den bruker de to egenskapene (name og age) for å generere og skrive ut en hilsen i konsollen når den blir kalt.

Linje 18 utenfor klassen, opprettes en ny instans av Person-klassen ved å kalle new Person("Alice", 30);. Dette oppretter et nytt objekt med de angitte egenskapene name og age.

Linje 21 blir greet()-metoden kalt på den opprettede instansen (person.greet();). Dette fører til at hilsenen blir skrevet ut i konsollen, som i dette tilfellet vil være "Hei, jeg heter Alice og er 30 år gammel."

#### Normalen i JS

Dette er vanlig å se i programmeringsspråk at vi lager medlemsvariabler på klasse nivået (globale variabler for klassen) og setter dem i konstruktøren.

Men i JS er det vanlig å IKKE lage medlemsvariabler, men at vi oppretter den direkte i konstruktøren.

*JS*

1 class Person {

2

3   constructor(*name*, *age*) {

4   this.name = name; *// Bruker 'this' til å referere til objektets*

5 *// egenskap 'name'*

6     this.age = age;   *// Bruker 'this' til å referere til objektets*

7 *// egenskap 'age'*

8   }

9

10   greet() {

11     console.log(`Hei, jeg heter ${this.name} og er ${this.age} år gammel.`);

12   }

13 }

Her er `this`-nøkkelordet brukt til å referere til objektets egenskaper `name` og `age` både inne i `constructor()`-metoden og i `greet()`-metoden. Dette gir tilgang til objektets tilstand og gjør det mulig å vise informasjonen om objektet i hilsenen.

`this` er en kritisk del av OOP. De fleste objektorientere programmeringsspråk har en eller annen variant `this`. I JavaScript tillater `this` riktig referanse til objektets medlemsvariabler og metoder, og sikrer at riktig objekt blir manipulert. Det er viktig å forstå hvordan `this` fungerer når du jobber med ES6-klasser og OOP i JavaScript.

### Bruke objektets egenskaper

Vi bruker følgende kode i eksemplene under:

class Person {

    constructor(name, age) {

      this.name = name;

      this.age = age;

    }

}

// Opprett et objekt fra Person-klassen

const person1 = new Person("Alice", 30);

#### Lese

Vi kan hente (lese) objektets egenskap slik:

// Lese objektest egenskaper

console.log(person1.name);

// Skriver ut "Alice" i consolet

#### Skrive

Vi kan endre (skrive) til objektets egenskap slik:

// Skrive til objektets egenskap

person1.age = 31;

// person1 sin alder er endret til 31

#### Slette egenskap

I JS kan vi kan til og med slette et objekts egensap(er) ved å benytte nøkkelordet delete. Det ser slik ut:

// Slette egenskap

delete person1.age;

// Objektet har nå ikke age egenskapen

#### Opprette egenskap

I motsatt retning kan vi i JS opprette en egenskap på et objekt som det ikke hadde til å begynne med. Vi benytter samme fremgangsmåte som for å skrive til et objekts egenskap, men vi bruker da navnet på en egenskap som ikke allerede er i objektet. Som i dette eksempelet:

// Legg til nye egenskaper

person1.email = "alice@example.com";

#### Egenskaper av typen array

Vi kan du opprette egenskaper som er av typen array (tabell/matrise) i en klasse eller et objekt ved å definere egenskapen som en variabel og tilordne den en arrayverdi.

Under er et eksempel som lar man gjøre:

* Legge til en hobby addHobby()
* Returnere siste element og fjerne det returAndRemoveLastHobby()
* Returnere første element og fjerne det returAndRmoveFirstHobby()
* Legge til et element først i listen addToBeginningHobby()
* Fjerne et element removeHobby()
* Bytte ut et element med et annet replaceHobby()

Kode Eksempel på array i et objekt

class Person {

    constructor(name, age, hobbies) {

        this.name = name;

        this.age = age;

        this.hobbies = []; // Tom array hvis ingen hobbies er oppgitt.

    }

    // Legg til et element i slutten av "hobbies"-arrayen.

    addHobby(hobby) {

        this.hobbies.push(hobby);

    } // addHobby

    // Fjern og returner det siste elementet i "hobbies"-arrayen.

    returAndRemoveLastHobby() {

        if (this.hobbies.length === 0) {

            return undefined; // Returnerer undefined hvis arrayen er tom.

        }

        return this.hobbies.pop();

    } // returAndRemoveLastHobby()

    // Fjern og returner det første elementet i "hobbies"-arrayen.

    returAndRmoveFirstHobby() {

        if (this.hobbies.length === 0) {

            return undefined; // Returnerer undefined hvis arrayen er tom.

        }

        return this.hobbies.s // returAndRmoveFirstHobby()hift();

    } // returAndRmoveFirstHobby()

    // Legg til et element i starten av "hobbies"-arrayen.

    addToBeginningHobby(hobby) {

        this.hobbies.unshift(hobby);

    } // addToBeginningHobby()

    // Fjern et element fra "hobbies"-arrayen basert på indeksen.

    removeHobby(index) {

        if (index >= 0 && index < this.hobbies.length) {

            return this.hobbies.splice(index, 1)[0];

        }

    } // removeHobby()

    // Bytt ut et element i "hobbies"-arrayen basert på indeksen.

    replaceHobby(index, newHobby) {

        if (index >= 0 && index < this.hobbies.length) {

            this.hobbies[index] = newHobby;

        }

    } // replaceHobby()

} // class Person

// Opprett en person med noen hobbies

const alice = new Person("Alice", 30, ["Lesing", "Sykling", "Maling"]);

// Legg til en ny hobby

alice.pushHobby("Fotografering");

// Fjern den siste hobbyen

const sisteHobby = alice.popHobby();

console.log("Fjernet siste hobby:", sisteHobby);

// Fjern den første hobbyen

const førsteHobby = alice.shiftHobby();

console.log("Fjernet første hobby:", førsteHobby);

// Legg til en ny hobby i starten

alice.unshiftHobby("Hiking");

// Fjern hobbyen på indeks 1

const fjernetHobby = alice.removeHobby(1);

console.log("Fjernet hobby på indeks 1:", fjernetHobby);

// Bytt ut hobbyen på indeks 0

alice.replaceHobby(0, "Svømming");

// Skrive ut oppdater hobby på indeks 0

console.log("Oppdaterte hobbies:", alice.hobbies);

### Bruke egenskaper med Getter og Setter

Dette er noen av de mest brukte spesielle metodene i JavaScript-klasser. De gir fleksibilitet og muligheter for tilpasning i hvordan objekter oppfører seg og hvordan de representeres som tekststrenger eller sammenlignes med andre objekter.

I ES6-klasser gir getter- og setter-metoder en mekanisme for å kontrollere tilgangen til egenskaper (properties) i objekter. De gir deg muligheten til å legge til logikk for lesing og skriving av egenskaper, noe som kan være nyttig for å validere data eller legge til ekstra behandling når en egenskap blir lest eller satt.

get og set er nøkkelord som angir om funksjonen er en Getter eller Setter.

Med en Setter metode kan vi kontrollere dataene før de blir lagret i egenskapen. Eksempel på dette er å sjekke om et isbn-nummer eller en epost er korrekt formatert, eller om dataene er av riktig datatype (selv om JS er mer «slepphendt» med dataene i variabler så kan det være viktig med tanke på at dataene senere skal inn i en database).

Med Getter- og Setter metoder kan du også begrense dataene til:

* Read/Write (Lese/Skrive). Både Getter og Setter er opprettet
* Read-Only (bare lesing). Bare Getter metoden er opprettet. Eksempelvis en klasse med astronomiske konstanter (se kode eksempel under), når klassens er opprettet er det ikke behov for å endre dem, men vi har behov for å hente den ut
* Write-only (bare skriving). Bare Setter metoden er opprettet. Eksempelvis applikasjonsinnlogging, sikkerhetstokens, kommando m.m. (Se under for kode eksempel)

Kode Eksempel hvor dataene blir verifisert før lagring

class Circle {

  constructor(*radius*) {

    this.\_radius = radius;

  }

  get radius() {

    return this.\_radius;

  }

  set radius(*value*) {

    if (value >= 0) {

      this.\_radius = value;

    }

  }

}

const circle = new Circle(5);

console.log(circle.radius); *// Leser egenskapen*

circle.radius = 7;          *// Endrer egenskapen*

console.log(circle.radius); *// Leser egenskapen*

#### Setter (Settemetode):

En *setter*-metode lar deg endre verdien av en egenskap i en klasse, og det gir deg muligheten til å utføre validering eller ekstra logikk før verdien blir satt.

Kode Write-only eksempel

class LoginSystem {

  constructor() {

    this.\_username = "";

    this.\_password = "";

  }

*// Setter-metode for å angi brukernavn*

  set username(*newUsername*) {

    this.\_username = newUsername;

  }

*// Setter-metode for å angi passord*

  set password(*newPassword*) {

    this.\_password = newPassword;

  }

*// Metode for å forsøke innlogging*

  login() {

    if (this.\_username === "bruker" && this.\_password === "passord") {

      console.log("Innlogging vellykket!");

    } else {

      console.log("Feil brukernavn eller passord. Prøv igjen.");

    }

  }

}

const loginSystem = new LoginSystem();

loginSystem.username = "bruker"; *// Setter brukernavn*

loginSystem.password = "passord"; *// Setter passord*

loginSystem.login(); *// Prøver innlogging*

Kode Personklasse med både getter- og setter-metode

class Person {

  constructor() {

    this.\_age = 0;

  }

  get age() {

    return this.\_age;

  }

  set age(*newAge*) {

    if (newAge >= 0 && newAge <= 120) {

      this.\_age = newAge;

    } else {

      console.log("Ugyldig alder.");

    }

  }

}

const person = new Person();

person.age = 30; *// Setter verdien ved å kalle setter-metoden*

console.log(person.age); *// Henter verdien ved å kalle getter-metoden*

I dette eksemplet bruker vi en setter-metode `age` for å endre aldersegenskapen, men før vi endrer verdien, validerer vi at den er innenfor rimelige grenser. Hvis valideringen mislykkes, gir *setter*-metoden en feilmelding.

#### Getter (Hentemetode)

En *getter*-metode er en metode som lar deg hente verdien av en egenskap i en klasse som om den var en vanlig egenskap, men den gir deg muligheten til å utføre ekstra logikk før du returnerer verdien. Legg merke til at metoden har samme navn som egenskapen (med unntak at egenskapen/medlemsvariabelen starter understrek).

Kode Read-only - Astronomiske konstanter

class AstronomyConstants {

  constructor() {

*// Definer astronomiske konstanter*

    this.\_gravitationalConstant = 6.67430e-11; *// m³ kg⁻¹ s⁻²*

    this.\_speedOfLight = 299792458; *// meter per sekund (m/s)*

    this.\_astronomicalUnit = 149597870700; *// meter (m)*

    this.\_earthMass = 5.972e24; *// kilogram (kg)*

    this.\_solarMass = 1.989e30; *// kilogram (kg)*

  }

*// Read-only getter for gravitasjonskonstanten G*

  get gravitationalConstant() {

    return this.\_gravitationalConstant;

  }

*// Read-only getter for lyshastigheten*

  get speedOfLight() {

    return this.\_speedOfLight;

  }

*// Read-only getter for astronomisk enhet (astronomical unit)*

  get astronomicalUnit() {

    return this.\_astronomicalUnit;

  }

*// Read-only getter for jordens masse*

  get earthMass() {

    return this.\_earthMass;

  }

*// Read-only getter for solens masse*

  get solarMass() {

    return this.\_solarMass;

  }

}

const astronomy = new AstronomyConstants();

console.log("Gravitasjonskonstanten G: " + astronomy.gravitationalConstant + " m³ kg⁻¹ s⁻²");

console.log("Lyshastigheten: " + astronomy.speedOfLight + " m/s");

console.log("Astronomisk enhet (AU): " + astronomy.astronomicalUnit + " meter");

console.log("Jordens masse: " + astronomy.earthMass + " kg");

console.log("Solens masse: " + astronomy.solarMass + " kg");

I dette eksemplet bruker vi en getter-metode `fullName` for å hente den fulle navneegenskapen. Dette lar oss konstruere den fullstendige strengen ved hjelp av både `firstName` og `lastName`-egenskapene.

#### Bruk av Underscore (\_)

Merk at i mange JavaScript-kodebaser brukes konvensjonen med å legge til et understreksymbol (for eksempel `\_firstName` eller `\_age`) for å indikere at egenskapen er beregnet å bli aksessert ved hjelp av getter- og setter-metoder. Dette er en praksis som hjelper til med å unngå uendelige rekursive samtaler mellom *getter*- og *setter*-metoder.

Det er viktig å forstå at *getter*- og *setter*-metoder gir en ekstra grad av kontroll og fleksibilitet når du arbeider med egenskaper i JavaScript-klasser, og de er nyttige for å opprettholde integriteten og validiteten til dataene i klassene dine.

Mer om dette nedenfor i Acces modifiers (tilgangsmodifikator) delen.

### 🗹 Oppgaver om klasser og objekt

I disse oppgavene bruker vi Visual Studio Code sammen med Node.js. Sørg for at VSC og Node.js er installert på datamaskinene.

#### Enkle oppgaver

##### Enkel klasse og instans

Lag en klasse `Person` med en constructor() som tar imot navn og alder. Opprett deretter en instans av `Person` og skriv ut informasjonen om personen.

##### Metode i klassen

Utvid `Person`-klassen fra oppgave 1 ved å legge til en metode som lar personen si "Hei!". Kall metoden på instansen du opprettet og skriv ut meldingen.

##### Personinformasjon

Bruk Person klassen og opprett deretter flere instanser av Person med forskjellig informasjon, og skriv ut denne informasjonen.

#### Middels oppgaver

##### Produktliste

Opprett en klasse Produkt med en constructor() som tar imot produktnavn og pris. Lag så en Produktliste klasse som lager flere produktinstanser og lag en funksjon som finner det dyreste produktet.

##### Flere egenskaper

Opprett en klasse `Bil` med en constructor() som tar imot merke, modell og årstall. Legg til metoder for å få informasjon om bilen, for eksempel "Mercedes-Benz C200, 2022".

##### Temperaturkonvertering

Lag en kalt `TemperatureConverter` som skal håndtere konvertering mellom forskjellige temperaturskalaer. Klassen skal ha følgende egenskaper:

* `celsius`: Temperaturen i Celsius.
* `fahrenheit`: Temperaturen i Fahrenheit.

Implementer *getter*- og *setter*-metoder for både `celsius` og `fahrenheit` som sørger for at verdiene er gyldige og at konverteringen mellom Celsius og Fahrenheit er korrekt.

Bruk følgende formel for å konvertere mellom Celsius og Fahrenheit:

1. Fahrenheit = (Celsius \* 9/5) + 32
2. Celsius = (Fahrenheit - 32) \* 5/9

##### Tidspunkt håndtering

Lag en klasse kalt `Time` som skal håndtere tidspunkt (time, minutter og sekunder). Klassen skal ha følgende egenskaper:

* `hours`: Timer (0-23).
* `minutes`: Minutter (0-59).
* `seconds`: Sekunder (0-59).

Implementer getter- og setter-metoder for hver av disse egenskapene for å validere verdiene og sørge for at de er innenfor gyldige områder.

Lag også `toString()`-metoden for å gi en tekstlig representasjon av tidspunktet i formatet "HH:MM:SS".

Lag `valueOf()`-metoden som returnerer totalt antall sekunder som et tall.

#### Utfordrende oppgaver

##### Lånesystem bibliotek

Du skal lage et system for et lånesystem bibliotek. Systemet skal håndtere bøker, lånere, lån og bibliotekets samling. Opprett følgende fire klasser: Bok, Person, Låner, Lån og Bibliotek.

* Bok-klassen skal ha egenskaper for boktittel, ISBN og forfatter (skal bruke Person-klassen). Den skal også ha en metode for å sjekke om boken er tilgjengelig for utlån.
* Låner-klassen skal bruker Person-klassen og ha egenskaper for lånenummer. Den skal også ha en metode for å låne en bok.
* Person-klassen skal ha egenskaper for fornavn, etternavn, adresse, telefon-nummer og e-post. Skal også ha en toString() metode for å returnere informasjon om personen.
* Lån-klassen skal inneholde informasjon om en lånesituasjon, inkludert boken som lånes, låneren og utlånsdato. Den skal ha en metode for å bekrefte lånet og en for å returnere boken.
* Bibliotek-klassen skal ha en *liste over bøker* i samlingen og en liste over aktive lån. Den skal ha metoder for å legge til bøker i samlingen, sjekke tilgjengeligheten av bøker og håndtere lån og retur av bøker.

##### Bankkonto

Lag en ES6-klasse kalt `BankAccount` som har egenskapene `balance` og `accountType`. Implementer getter og setter metoder for `balance` som hindrer at saldoen går under 0, og setter metoder for `accountType` som kun aksepterer verdiene "Savings" eller "Checking."

## Abstraksjon (Abstraction)

### Avklaring av uttrykket grensesnitt

Grensesnitt i abstraksjon og innkapsling refererer til den måten eksterne deler av programmet kan samhandle med en klasse eller et objekt, samtidig som de er begrenset til å bruke de offentlige metoder og egenskaper som er eksponert av klassen (se Access modifiers (tilgangsmodifikator) nedenfor). Grensesnitt i denne sammenhengen handler om å etablere en klar og begrenset tilgang til klassens indre tilstand og funksjonalitet. Det vil si at funksjoner (metoder) og egenskaper (medlems variabler) som en klasse tilbyr for bruk av andre deler av programmet. Offentlige metoder og egenskaper er tilgjengelige fra utsiden av klassen og brukes for å utføre operasjoner eller hente informasjon fra klassen.

### Abstraksjon er

Abstraksjon er en av de sentrale prinsippene innen OOP og representerer en viktig teknikk for å organisere og utvikle programvare.

Den grunnleggende ideen bak abstraksjon er å forenkle kompleksiteten i et system ved å isolere og representere de mest relevante egenskapene og funksjonalitetene, samtidig som man skjuler eller abstraherer unødvendige detaljer. Vi eksponerer detaljene gjennom medlemsfunksjoner.

### Hva og ikke hvordan

I praksis innebærer abstraksjon å definere klasser, grensesnitt og metoder som gir et høyt nivå av abstraksjon over de underliggende implementeringsdetaljene. Dette tillater utviklere å arbeide på et mer konseptuelt nivå, der de fokuserer på **"hva" et objekt gjør, i stedet for "hvordan" det gjør det**. Ved å skjule detaljene om hvordan en klasse fungerer, kan utviklere bruke objekter av klassen uten å trenge å forstå alle de interne operasjonene.

### Fordelene med abstraksjon

* **Klarhet**: Koden mer lesbar og forståelig, da den tillater utviklere å fokusere på det essensielle og ignorere unødvendige detaljer.
* **Gjenbruk**: klasser og metoder kan gjenbrukes i ulike deler av programmet, noe som sparer tid og innsats.
* **Vedlikehold**: Endringer i implementeringsdetaljer påvirker ikke koden som bruker abstrakte klasser eller metoder, noe som gjør vedlikehold enklere.
* **Sikkerhet**: Bidra til å beskytte data ved å kontrollere tilgangen til dem gjennom grensesnitt og metoder.
* **Skalerbarhet**: Abstrakte strukturer tillater utviklere å bygge komplekse systemer ved å kombinere og utvide abstrakte komponenter.

### Eksempel

Eksempler på abstraksjon inkluderer opprettelse av abstrakte klasser som definerer grunnleggende funksjonalitet for en rekke relaterte klasser, bruk av grensesnitt for å fastsette kontrakter som klasser må følge, og bruk av metoder som utfører handlinger uten at brukeren trenger å kjenne til de detaljerte trinnene som kreves.

#### Drivstoffsystem

Når en person kjører en bil, bruker de gasspedalen for å akselerere og bremsen for å stoppe. De fleste bilførere trenger ikke å forstå de detaljerte tekniske aspektene ved hvordan drivstoffsystemet fungerer under panseret. Drivstoffsystemet er abstrahert, og føreren trenger bare å forholde seg til de grunnleggende kontrollene.

Abstraksjonen i dette tilfellet lar føreren fokusere på det viktigste aspektet av kjøringen, nemlig styring og kontroll, uten å trenge å bekymre seg for kompliserte motorprosesser som forbrenning, innsprøytning og tenning. Dette gjør det mulig for alle å kjøre en bil, uavhengig av deres tekniske kunnskaper om bilens indre arbeid

#### Abstraksjon av Databaser

Innen datahåndtering og databasesystemer kan abstraksjon brukes til å representere databaser som abstrakte enheter. Dette kan gjøres ved å opprette et grensesnitt for databaseoperasjoner som inkluderer funksjoner som leggTilData, hentData, og oppdaterData. Flere konkrete databaseimplementeringer, for eksempel MySQL, PostgreSQL eller NoSQL-databaser, kan deretter implementere dette grensesnittet. Dette tillater utviklere å jobbe med databaser som abstrakte lag uten å bekymre seg for de spesifikke detaljene i hver database.

### Kort sagt

Abstraksjon er en kritisk komponent i utviklingen av OOP-programvare hvor fokuset er HVA og ikke HVORDAN. Det fremmer modularitet, fleksibilitet og enkelhet i design og implementasjon.

## Innkapsling (Encapsulation)

Innkapsling refererer til konseptet om å pakke inn data (egenskaper) og metoder (funksjoner) som opererer på disse dataene i en enkelt enhet, kalt et objekt. Hensikten med innkapsling er å skjule de interne detaljene av et objekt og tillate tilgang til data og funksjoner gjennom veldefinerte grensesnitt.

### Hovedelementer i Innkapsling

1. **Egenskaper (Data)**: Egenskaper, også kalt medlemmer eller variabler, er de dataene som er relevante for objektet. Disse egenskapene kan være offentlige, private eller beskyttede. Offentlige, private eller beskyttede er elementer som vi går tar for oss i temaet Access modifiers (tilgangsmodifikator).
2. **Metoder (Funksjoner)**: Metoder er funksjoner som utfører operasjoner på egenskapene til objektet. De er også pakket inn i objektet og kan være offentlige, private eller beskyttede, avhengig av ønsket tilgangsnivå.

### Fordeler med Innkapsling

* **Sikkerhet**: Ved å gjøre egenskaper private eller beskyttede, kan utviklere kontrollere tilgangen til objektets data. Dette bidrar til å forhindre utilsiktede endringer eller feil bruk av dataene.
* **Abstraksjon**: Innkapsling gjør det mulig å abstrahere de interne detaljene av et objekt. Eksterne brukere trenger ikke å vite hvordan dataene lagres eller hvordan metodene fungerer internt; de trenger bare å forstå de offentlige grensesnittene.
* **Fleksibilitet**: Ved å tilby offentlige grensesnitt, kan utviklere endre den interne implementeringen av et objekt uten å påvirke eksterne brukere. Dette gir stor fleksibilitet og letter vedlikehold og videreutvikling.

### Access modifiers (tilgangsmodifikator)

I OOP er "access modifiers" (tilgangsmodifikatorer) nøkkelord som brukes til å definere tilgangsnivået til metoder og egenskaper i en klasse. Disse tilgangsmodifikatorene kontrollerer hvem som kan bruke og endre disse medlemmene fra ekstern kode, og de er viktige for å oppnå konseptene om innkapsling og sikkerhet i OOP.

#### De vanligste tilgangsmodifikatorene i OOP er

* **Public (Offentlig)**: Dette er standardtilgangen hvis ingen tilgangsmodifikator er spesifisert. Offentlige medlemmer er tilgjengelige fra hvor som helst i koden, både innenfor klassen og eksternt. De kan leses og endres uten begrensninger.
* **Private (Privat)**: Private medlemmer er bare tilgjengelige innenfor klassen som definerer dem. De kan ikke nås eller endres utenfor klassen. Dette gir en høy grad av sikkerhet og oppnår prinsippet om innkapsling.
* **Protected (Beskyttet)**: Beskyttede medlemmer er tilgjengelige i den samme klassen samt i avledede klasser som arver fra den overordnede klassen. De er ikke tilgjengelige eksternt. Dette gir en mellomliggende grad av sikkerhet og støtter arv.

##### Module (modul) / Library (bibliotek) / Pakker (packages) / Namespace

En modul/bibliotek og alle de andre navnene som blir brukt på dette er en organisatorisk enhet som brukes i mange programmeringsspråk for å gruppere sammen beslektede klasser, funksjoner, eller andre ressurser. Hensikten med en pakke er å gi struktur og organisasjon til koden, slik at den kan bli mer oversiktlig og lettere å administrer, spesielt i store prosjekter.

Her er noen viktige aspekter ved modul/bibiliotek/pakke:

* organisering
* unngå navnekonflikter
* modularitet
* sikkerhet og tilgangskontroll samt
* kildekodeorganisering.

Programmeringsspråk som Java, C#, Python og mange andre bruker konseptet med modul/bibliotek/pakke. Hver av disse språkene har sine egne måter å opprette, administrere og bruke dem på, men formålet er generelt det samme: å hjelpe med organisering og strukturering av kode for å gjøre den mer lesbar og vedlikeholdbar.

##### Tilgangsmodifikator for modul / bibliotek / pakke

I tillegg har vi en tilgangsmodifikator disse. Nøkkelordet for dette varierer fra språk til språk slik som default, internal og friend.

Når man bruker tilgangsmodifikatoren for modul/bibliotek/pakke gir man tilgang til klassemedlemmene til all kode som hører til samme modul/bibliotek/pakke.

##### Tilgangsmodifikator i JavaScript

I JavaScript (ES6 og senere) er det ingen offisielle nøkkelord som "public," "private," eller "protected" for tilgangsmodifikatorer slik som i noen andre objektorienterte programmeringsspråk som Java eller C++.

Tilgangskontroll i JavaScript er i stor grad basert på konvensjoner og noen nye funksjoner som private egenskaper og metoder som bruker #.

Her er hvordan tilgangsmodifikatorer og # kan brukes i JavaScript:

* **Public (Offentlig)**: medlemsvariabler og medlemsmetoder skrives som vanlig uten noe endringer.
* **Private (Privat)**: medlemsvariabler og medlemsmetoder kan man fra ES6 starte med et hash-tegn # for å gjøre dem private.

class MyClass {

    #privateProperty = 42; // Privat egenskap

    #privateMethod() { // Privat metode

      return "Dette er en privat metode.";

    }

    // Offentlig metode som bruker de private metodene og egenskapene

    publicMethod() {

      return this.#privateMethod();

    }

  }

* **Protected (Beskyttet)**: JavaScript har ingen innebygde tilgangsmodifikatorer for beskyttede egenskaper og metoder. Du kan bruke konvensjoner som for eksempel \_ (underscore) for å indikere at noe skal behandles som beskyttet (privat), men dette er kun basert på konvensjon og ikke en streng tilgangskontroll. En medlemsvariable eller en medlemsmetoder som starter med underscore \_ er i JavaScript sine øyne public, det er opp til oss som programmerere å ikke bruke dem.

class MyClass {

    \_protectedProperty = 42; // Beskyttet egenskap (konvensjon)

    \_protectedMethod() { // Beskyttet metode (konvensjon)

      return "Dette er en beskyttet metode.";

    }

  }

#### Kort sagt

Tilgangsmodifikatorer gir utviklere muligheten til å definere grensesnittet til en klasse, og de er viktige for å opprettholde dataintegritet og sikkerhet. Ved å bruke tilgangsmodifikatorer kan du begrense tilgangen til sensitive data og sørge for at de kun endres på de ønskede måtene.

Kode innkapslings eksempel

I JavaScript kan innkapsling oppnås ved bruk av konsepter som closures (forklart under) og private medlemsvariabler (privatscopet) variabler. Her er et enkelt eksempel:

function Circle(*radius*) {

*// Privat egenskap*

  var \_radius = radius;

*// Offentlig metode for å hente radius*

  this.getRadius = function () {

    return \_radius;

  };

*// Offentlig metode for å beregne areal*

  this.calculateArea = function () {

    return Math.PI \* \_radius \* \_radius;

  };

}

var myCircle = new Circle(5);

console.log(myCircle.getRadius()); *// 5*

console.log(myCircle.\_radius); *// undefined (privat egenskap er utilgjengelig)*

console.log(myCircle.calculateArea()); *// 78.54*

I dette eksemplet er `\_radius` en privat egenskap, og tilgang til radius og beregning av areal er tilgjengelig via offentlige metoder.

**Ekstra**: En "**closure**" er en funksjon i JavaScript som har tilgang til variabler fra et overordnet kodeblokk/omfang, selv etter at det overordnede omfanget har blitt avsluttet.

function createCounter() {

  let count = 0;

  function increment() {

    count++;

    console.log(count);

  }

  return increment;

}

const counter = createCounter();

counter(); *// Utskrift: 1*

counter(); *// Utskrift: 2*

## 🗹 Oppgaver for abstraksjon og innkapsling + tilgangsmodifikator

#### Enkel oppgave

##### Abstraksjon av Dyr

Lag to separate klasser, "Dog" og "Cat," som har egenskapen "navn." Implementer metoder i hver klasse for å returnere lyden de lager.

#### Middels oppgave

##### Identifiser Abstraksjon og Innkapsling i Flyreiseapplikasjon

*Gå i sammen to og to.*

Du jobber med å utvikle en programvareapplikasjon som skal hjelpe brukere med å søke etter og bestille flyreiser. Applikasjonen skal kunne håndtere flere flyselskaper, ruter, og reisedetaljer. Du er ansvarlig for å utforme klassene og objektstrukturen for dette systemet.

I denne oppgaven er det flere konsepter relatert til objektorientert programmering, spesielt abstraksjon og innkapsling, som du må vurdere. Identifiser:

**Abstraksjon**: Hvilke egenskaper og metoder bør være en del av klassene som representerer flyreiser, flyselskaper, og ruter i applikasjonen din? Hva er de viktigste aspektene av en flyreise som du vil abstrahere?

**Innkapsling**: Hvilke av egenskapene og metodene du har identifisert i punkt 1 bør være offentlige og tilgjengelige fra utsiden av klassene, og hvilke bør være private og beskyttet for å sikre korrekt funksjonalitet?

Skriv ned dine tanker om abstraksjon og innkapsling i sammenheng med denne flyreiseapplikasjonen, og diskuter hvordan du vil implementere dem i koden din.

#### Utfordrende oppgave

##### Weblagring

Lag to separate klasser for lagring i nettleseren, "LocalStorage" og "SessionStorage," som implementerer lagringsmetoder for henholdsvis varig og midlertidig nettleserlagring.

##### Dataspill med Abstraksjon og Innkapsling

*Gå i sammen to og to.*

Ditt oppdrag er å lage et enkelt dataspill ved hjelp av prinsippene om abstraksjon og innkapsling. Spillet skal inkludere følgende komponenter: spillfiguren (f.eks., en karakter eller et kjøretøy), fiender, et mål og en spillverden.

Spillet handler om:

"Ocean Racer" er et spennende racingspill som tar spilleren med på en adrenalinfylt konkurranse på åpent hav. Spilleren har kontroll over en kraftig racerbåt og må kjempe mot motstandere i en serie intense båtraser. Spilleren styrer en rask racerbåt som han/hun styrer. Spillet tilbyr en rekke forskjellige løp. Spilleren kan velge mellom forskjellige typer racerbåter. Målet med spillet er å konkurrere mot andre racerbåter, vinne løp, tjene poeng og oppgraderinger, og til slutt bli en mester i Ocean Racer-verdenen.

* Finn klassene/objektene som dere mener som trengs for å lage spillet
* Tenk abstraksjon og finn egenskapene og metodene til disse
* Vurder tilgangsmodifikatorer for forskjellige egenskapene og metodene

## Arv (Inheritance)

A diagram of a diagram

Description automatically generatedArv er en grunnleggende konseptuell mekanisme der en klasse (eller objekt) kan arve egenskaper og metoder fra en annen klasse. Dette innebærer opprettelsen av en hierarkisk struktur der en klasse, kjent som en "**subklasse**" eller "**barnklasse**," kan arve attributter og metoder fra en annen klasse, kjent som en "**superklasse**" eller "**foreldreklasse**."

**Merk**: at en subklasse ofte kan bli referert til som en *avledet klasse*.

Fordeler med arv:

* Gir kode struktur, lager en logisk hierarkisk oppsett av klassene
* Gjenbrukbarhet der koden i superklassene ikke nødvendigvis trengs å skrives de arves og er tilgjengelige
* Utvidelse og tilpassing, subklassen kan utvide funksjonaliteten ved å legge til nye egenskaper og metoder. Subklassen kan tilpasse eksisterende metoder ved å overskrive dem for å gi en annen implementasjon
* Overføring av **egenskaper** og **metoder,** subklassen arver alle egenskaper og metoder fra superklassen (som ikke er gjort private i superklassen). Dette inkluderer både offentlige og beskyttede egenskaper/metoder, avhengig av programmeringsspråket

**Eksempel: *Kjøretøy***

La oss vurdere et eksempel med klassene "Vehicle" som superklasse og "Car" som subklasse. Superklassen "Vehicle" kan ha egenskaper som "hastighet" og metoder som "start()" og "stopp()". Subklassen "Car" kan arve disse egenskapene og metoder og legge til egne, som "horn()".

class Vehicle {

  constructor(speed) {

    this.speed = speed;

  } // ctor()

  start() {

    console.log("Vehicle started.");

  } // start()

  stop() {

    console.log("Vehicle stopped.");

  }// stop()

} // Vehicle

class Car extends Vehicle {

  constructor(speed, brand) {

    super(speed); // Arver fra superklassen

    this.brand = brand; // Egen egenskap

  } // ctor()

  horn() {

    console.log("Car horn activated.");

  } // horn()

} // Car

**Eksempel fra oppgaven om abstraksjon av dyr**

I oppgaven om å lag to separate klasser, "Dog" og "Cat," som har egenskapen "navn." Implementer metoder i hver klasse for å returnere lyden de lager vil vi nok har laget en superklasse kaldt Animal. Og dyrene hund (Dog) og katt (Cat) blir subklassene.

A diagram of a cat and dog

Description automatically generated

class Animal {

  constructor(name) {

    this.name = name;

  } // cotr()

  makeSound() {

    // Denne metoden skal overskrives i subklassene

  } // makeSound()

} // Animal

class Dog extends Animal {

  makeSound() {

    return "Woof!"; // Hunden lager bjeffelyd

  } // makeSound()

} // Dobg

class Cat extends Animal {

  makeSound() {

    return "Meow!"; // Katten lager mjauelyd

  } // makeSound()

} // Cat

const myDog = new Dog("Buddy");

const myCat = new Cat("Whiskers");

console.log(`${myDog.name} sier: ${myDog.makeSound()}`);

console.log(`${myCat.name} sier: ${myCat.makeSound()}`);

*Resultatet*

Buddy sier: Woof!

Whiskers sier: Meow!

## Polymorfisme (Polymorphism)

Ordet "polymorfisme" kommer fra gresk og betyr "mange former," og i programmering refererer det til evnen til å behandle objekter av forskjellige klasser på en enhetlig måte. Med andre ord tillater polymorfisme at du kan bruke et generelt grensesnitt eller overordnet klasse til å arbeide med objekter av ulike klasser, selv om de har forskjellige implementasjoner. Dette gjør koden mer fleksibel og gjenbrukbar.

Det er to hovedtyper av polymorfisme:

* Run-Time Polymorphism (Dynamisk Polymorfisme)
* Compile-Time Polymorphism (Statisk Polymorfisme)

### Overriding (overskriving/metodeoverstyring)

Overriding også kjent som Run-Time Polymorphism (Dynamisk Polymorfisme) gjør det mulig for en avledet klasse å gi sin egen implementasjon av en metode som allerede er definert i dens overordnede (super) klasse. Dette betyr at når du har en metode i både superklassen og subklassen med samme navn og signatur (parameterliste), så vil subklassens implementasjon overskrive superklassens implementasjon når metoden blir kalt fra en instans av subklassen.

I koden over om dyr og arv ser vi at metoden makeSound() ikke er implementert, det vil si at det ikke er skrevet noen kode. Vi har også en kommentar om at metoden blir overskrevet i superklassen.

  makeSound() {

    // Denne metoden skal overskrives i subklassene

  } // makeSound()

Å gjøre *Overriding (oveskriving)* lar oss overstyre metoden som ble arvet i fra superklassen. I dette tilfellet er det meningen at den metoden makeSound() skal overskrives av subklassene.

Vi kan også utføre *Overriding* på en metode som gjør noe.

class Animal {

  constructor(name) {

    this.name = name;

  } // cotr()

  makeSound() {

    return "Silence";

  } // makeSound()

} // Animal

class Dog extends Animal {

  makeSound() {

    return "Woof!"; // Hunden lager bjeffelyd

  } // makeSound()

} // Dog

class Cat extends Animal {

} // Cat

const myDog = new Dog("Buddy");

const myCat = new Cat("Whiskers");

console.log(`${myDog.name} sier: ${myDog.makeSound()}`);

console.log(`${myCat.name} sier: ${myCat.makeSound()}`);

*Resultatet*

Buddy sier: Woof!

Whiskers sier: Silence

Samme kode men vi lager et array og loop-er igjennom funksjonen for hver enkelt av dem:

// Lager array med forskellige dyr

const animals = [new Dog("Buddy"), new Cat("Whiskers"), new Cat("Bird killer")];

// Loop-er igjennom alle og kaller metoden

animals.forEach((animal) => {

  console.log(`${animal.name} says: ${animal.makeSound()}`);

});

**Resulat**:

Buddy says: Bark

Whiskers says: Meow

Tweetie says: Chirp

### Overloading/method overloading (metodeoverbelastning)

"Overloading" kan også være kjent som Compile-Time Polymorphism (Statisk Polymorfisme). Overloading er muligheten til å definere flere funksjoner eller metoder med samme navn i en klasse eller et program, men med ulike parameterlister og/eller retur type. Når en funksjon eller metode blir kalt, velger programmet hvilken versjon som skal utføres basert på antall og typer argumenter som blir gitt. Dette gjør det mulig å ha flere funksjoner med samme navn, men som utfører forskjellige oppgaver avhengig av inndataene.

### Nøkkelpunkter ved overloading

#### Signatur

Samme navn, forskjellige parametere og/eller retur type: Overloaded-metoder deler samme navn, men de må ha forskjellige parametere, enten i form av antall parametere eller typer og kan også returnere data av forskjellige typer.

class Calculator {

    int Add(int a, int b) {

      return a + b;

    } // Add()

    double Add(double a, double b) {

      return a + b;

    } // Add()

} // Calculator

int Add(int a, int b)

double Add(double a, double b)

Her er de 2 metodene fra klassen over. Vi ser at de har samme navn men jeg er forskjellige da de tar inn forskjellige parametere og også returnerer forskjellige typer data. Dette kaller vi for metoden sin signatur eller også kaldt metodens fingerprint.

**MERK** bedre at dette er ikke JavaScript. JavaScript tar ikke høyde for hvilken datatype som kommer inn i parameterne og JavaScript trenger heller ikke og si hvilken datatype som returneres, da javaskrift bruker return nøkkelordet og returnerer den datatypen som eventuelt blir kalkulert ut.

I programmeringsspråk som C# og java, slik som dette eksemplet over viser, kan vi ha flere varianter av en metodesignatur.

int Add(int a, int b)

int Add(int a, int b, int c)

double Add(double a, double b)

double Add(int a, double b)

double Add(double a, int b)

Disse metodesignaturene er alle gyldige. Dersom vi kaller Add() metoden og sender inn en int for parametere a og double for parameter b, vil programmet kalkulerer og returnere en dobbel. Det som skjer er at for å kunne kalkulere to forskjellige datatyper vil programmet hjelpe ved å konvertere den ene slik at de har samme datatype. Og her vil da int verdien blir konvertert dobbel og programmet returnerer svaret som dobbel.

#### Automatisk valg

Når en overloaded-funksjon blir kalt, velger programmeringsspråket automatisk hvilken versjon som skal brukes basert på signaturen som er gitt.

#### Fleksibilitet

Overloading gir fleksibilitet ved å tillate at funksjoner med lignende funksjonalitet kan organiseres på en konsistent måte.

### Overloading og JavaScript

Som nevnt over og også vist i eksemplet vil ikke overloading med bruk av signaturer som har samme antall parametere bare funksjonelt da JavaScript ikke bruker datatyper for å presisere hvilken data som kommer inn i parameterne.

Men vi kan bruke overloading JavaScript der vi oppgir med 2 til signaturer med forskjellige antall parametere.

For eksempel et over vil dette være noe vi kan gjøre i JavaScript.

int Add(int a, int b)

int Add(int a, int b, int c)

### Constructor og overloading

Vi kan også overlade konstruktøren til en klasse.

class Person {

    public string Name { get; set; }

    public int Age { get; set; }

    public Person(string name, int age)

    {

        Name = name; // setter propertien navn

        Age = age; // setter propertien age

    } // ctor med navn og alder som parameter

    public Person(string name)

    {

        this(name, 0)// kaller ctor med bare navn og alder

    } // ctor() med navn som parameter

    public Person()

    {

        this("Unknown") // kaller ctor med bare navn som få kaller

                        //  konstruktor med navn og alder

    } // ctor() uten parameter

} // Person

**MERK** at i JavaScript ES6 kan en klasse bare ha én konstruktørmetode. Dette er begrenset av språket. Du kan imidlertid simulere overbelastning av konstruktører ved å bruke alternative tilnærminger. For eksempel kan du bruke standardparametere eller overvåke antall argumenter og behandle dem forskjellig inne i den ene konstruktøren.

Her er et eksempel som viser hvordan du kan bruke standardparametere for å simulere overbelastning:

class Person {

  constructor(name = "Unknown", age = 0) {

    this.name = name;

    this.age = age;

  }

  describe() {

    console.log(`Name: ${this.name}, Age: ${this.age}`);

  }

}

const person1 = new Person("Alice", 30);

const person2 = new Person("Bob");

const person3 = new Person();

person1.describe(); // Name: Alice, Age: 30

person2.describe(); // Name: Bob, Age: 0

person3.describe(); // Name: Unknown, Age: 0

Og her er et eksempel hvor vi sjekker overvåke antall argumenter/parametere.

class Person {

  constructor(name, age) {

    if (arguments.length === 2) {

      this.name = name;

      this.age = age;

    } else if (arguments.length === 1) {

      this.name = name;

      this.age = 0;

    } else {

      this.name = "Unknown";

      this.age = 0;

    }

  }

  describe() {

    console.log(`Name: ${this.name}, Age: ${this.age}`);

  }

}

const person1 = new Person("Alice", 30);

const person2 = new Person("Bob");

const person3 = new Person();

person1.describe(); // Name: Alice, Age: 30

person2.describe(); // Name: Bob, Age: 0

person3.describe(); // Name: Unknown, Age: 0

## 🗹 Oppgaver for Arv og Polymorfism + overriding og overloading

##### Shapes

Lag en klasse Shape med egenskaper width og height. Lag en metode calculateArea() som returnerer arealet.

Lag to underklasser: Rectangle og Triangle. Rectangle skal ha en konstruktør som tar inn bredde og høyde, mens Triangle skal ha en konstruktør som tar inn lengden på de to kortsidene og beregner høyden selv. Overstyr calculateArea() i begge underklassene for å beregne riktig areal.

Opprett instanser av både Rectangle og Triangle og kall calculateArea() for begge.

#### Middels oppgave

##### Flere nivåer av arv

Lag en klasse Animal med en egenskap name og en metode speak() som viser en melding om dyrets lyd.

Lag to underklasser: Mammal og Bird. Mammal skal ha en konstruktør som tar inn name og en metode greet() som viser en hilsen. Bird skal ha en metode fly() som viser en melding om at fuglen flyr.

Lag en underklasse Dog som arver fra både Mammal og Animal. Overstyr speak() for å vise en bjeffelyd og greet() for å hilse som en hund.

Opprett en instans av Dog og kall både speak(), greet(), og fly().

#### Utfordrende oppgave

##### Virtuelt hotellreservasjonssystem

Gå i grupper og finn ut hvilke klasser, egenskaper og metoder som trengs, tenk på arv, polymorfisme, overriding og overloading.

Du er tildelt oppgaven med å utvikle et virtuelt hotellreservasjonssystem ved hjelp av prinsippene for objektorientert programmering. Systemet skal tillate gjester å søke etter ledige rom, reservere rom, og administrere reservasjonene sine.