

# Internetwork og Web-programmering

## JS Funktioner & Objekter

Forelæsning 2  
Brian Nielsen

Distributed, Embedded, Intelligent Systems



# Agenda: JavaScript Intro 2

- Lektion 2: Grundlæggende
  - Bindinger
  - Iteration, selection
  - Arrays
- Lektion 2
  - Funktioner
  - Funktions-parametre
  - JavaScript Objekter
  - Constructor funktioner
  - Objekt serialisering
  - Array metoder
  - Videregående om objekter, prototype, og klasser

```
let diceRoll=[1,6,6,2,3,4,6];

function count6es(roll){
  let found6=0;
  for(let i=0;i<roll.length;i++){
    if(roll[i]===6) found6++;
  }
  console.log(`found ${found6} sixes!`);
  return found6;
}

console.log("Found6: "+count6es(diceRoll));

found 3 sixes!
Found6: 3
```

# Funktioner i JS

Funktioner er vigtige og anvendes MEGET!

3 måder at definere dem på:

- Funktions-erklæringer
  - Hyppigst anvendte
  - Kan kaldes inden de erklæres ("hoistes");
- Funktions-udtryk
  - Skaber en funktions-værdi
  - Anvendes i udtryk eller sætninger hvor man har brug for en funktions-værdi
  - Kan bindes til et navn via `let` eller `const` som alle andre værdier
- Pile-funktions-udtryk
  - Kompakt alternative til alm. funktions udtryk
  - Anvendes typisk til at lave en "lille" funktion, der overføres som parameter
  - Relateret til "lambda-udtryk"
  - Har visse [forskelle](#) og begrænsninger ifbm. metoder og objekter

```
// funktions kald/applikation/anvendelse/invokation
let b=bmi(1.80,90); //27.78

// funktions erklæring
function bmi(h,w){
    return w/(h*h);
}

//funktions udtryk (anonymt)
let bmi2= function(h,w){
    return w/(h*h);
};
b=bmi2(1.80,100); //30.86

//funktions udtryk (navngivet)
bmi2= function calcBMI(h,w){return w/(h*h)};
b=bmi2(1.80,110); //33.95

let bmix=bmi2;
b=bmix(1.80,120); //37.03

//pile-funktioner
const bmi3 = (h,w) =>{ return w/(h*h)};
b=bmi3(1.80,130); //40.12

//hvis kun 1 parameter: parentes i param liste unødvendig
//hvis kun 1 statement: fjern {}, og "return" er implicit
const incr = a => a+1;
b=incr(5); //6
```

# Funktions parametre


- Vi ønsker at udskrive nummererede overskrifter i et antal sproglige varianter
  - print\_N\_DK og print\_N\_UK er jo helt ens, bortset fra hvilken print funktion den kalder: så skal vi jo parameterisere
- funktioner kan overføres som parametre til funktioner som alle andre værdier
- Den overførte funktion kan kaldes på alm vis.

```
function printHeadingDK(no){
  console.log("Jeg er overskrift nummer " + no);
}
function printHeadingUK(no){
  console.log("I am heading "+ no);
}
function print_N_DK(N){
  for(let i=0;i<N;i++) printHeadingDK(i);
}
function print_N_UK(N){
  for(let i=0;i<N;i++) printHeadingUK(i);
}

print_N_DK(5);
print_N_UK(5);

function repeatHeading(N,print){
  console.log(`Will repeat ${N} time the function`);

  for(let i=0;i<N;i++)
    print(i);
}
repeatHeading(5,printHeadingDK);
repeatHeading(5,printHeadingUK);
repeatHeading(5,no => console.log("<H1> heading "+no+"
</H1>"));
```



Jeg er overskrift nummer 0  
Jeg er overskrift nummer 1  
Jeg er overskrift nummer 2  
Jeg er overskrift nummer 3  
Jeg er overskrift nummer 4  
I am heading 0  
I am heading 1  
I am heading 2  
I am heading 3  
I am heading 4

# Funktions parametre

- Parameteriserede funktioner: mere genanvendelige
  - én funktion til sortering istedet for 100 varianter:
    - Sort\_strings(..)
    - Sort\_ints(..)
    - Sort\_score(..)
    - ...
    - => sort(array, compare)
- Call-backs
  - Event-handlers:

```
function handleClick(){...}  
htmlElem.addEventListener("click", handleClick);
```

# Funktions parametre i C.

- I kender qsort i C!

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))
```

## C version

```
const char * array[] = { "eggs", "bacon", "cheese", "mushrooms",  
"spinach", "potatoes", "spaghetti"};
```

```
/* Compare the strings. */
```

```
static int compare(const void * a, const void * b) {  
    /* The pointers point to offsets into "array", so we need to dereference  
       them to get at the strings. */
```

```
    return strcmp(*(const char **) a, *(const char **) b);  
}
```

```
/* n_array is the number of elements in the array. */
```

```
qsort(array, n_array, sizeof(const char *), compare);
```

## JavaScript version

```
let arr = [ "eggs", "bacon", "cheese", "mushrooms", "spinach", "potatoes",  
"spaghetti" ];
```

```
function compare(a,b) {  
    if (a<b) return -1;  
    if (a>b) return 1;  
    return 0;  
}  
arr.sort(compare);  
//eller bare arr.sort();
```

# Funktioner kan nestes

- En funktion danner sit eget synlighedsfelt (lexical scope);
- Bindinger indenfor "functionDemo" er ikke synlige udenfor

```
function functionDemo(){  
  //funktioner kan nestes, synlighedsfelt  
  
  function printHeadingDK(no){  
    console.log("Jeg er overskrift nummer " + no);  
  }  
  function printHeadingUK(no){  
    console.log("I am heading "+ no);  
  }  
  function print_N_DK(N){  
    for(let i=0;i<N;i++) printHeadingDK(i);  
  }  
  function print_N_UK(N){  
    for(let i=0;i<N;i++) printHeadingUK(i);  
  }  
  ...  
}
```

# (Funktioner som retur-værdier)

- Funktioner er værdier, derfor kan de returneres!
- `nth`-funktionen genererer en funktion der kalder `f` hver `n`'te gang
- Bemærk at `nth`'s parametre anvendes i `fn`, og disse binder overlever i `fn`, efter den er returneret!
  - "Closures"
- Nævnes her FYI, vi får nok ikke brug for det i web-programmering

```
function printHeadingDK(no){
  console.log("Jeg er overskrift nummer " + no);
}
function repeatHeading(N,print){
  for(let i=0;i<N;i++){
    print(i);
  }
  repeatHeading(5,printHeadingDK);

function nth(n,f){
  let fn=function(no){
    if(no%n===0)
      return f(no);
  }
  return fn;
}
//function that only prints an even heading
let evenHeading=nth(2,printHeadingDK);
//even headings between 0..4
repeatHeading (5, evenHeading);
```

```
Jeg er overskrift nummer 0
Jeg er overskrift nummer 2
Jeg er overskrift nummer 4
```



# Javascript Objekter

# JavaScript Objekter: Literaler

- En samling af "properties" / "egenskab"
  - Map fra nøgle (unik navn) til værdi ("key-value")
  - Et literal laves vha. { }
  - **Punktum notation** til adgang til nøglens værdi
  - Keys kan være strenge med mellemrum \*)
- Et objekt kan bindes til et navn via const og let.
  - Navnet "peger på" objektet.
- En egenskab kan oprettes dynamisk (og fjernes!)
- Mange indbyggede objekter: FX
  - Math, String, Array, Map, Set, JSON, ...
  - Function, Object, ...
- \*) kan være behændigt, men misbruges ofte hvor "Map" er en bedre løsning

```
let student1={
  name:"Gynter",
  studentID:12345678,
  "Ynglings Sang": "Højt på træets grønne"
}
console.log(student1.name);
console.log(student1["Ynglings Sang"]);

student1.age=42;
console.log(student1);
```

```
{
  name: 'Gynter',
  studentID: 12345678,
  'Ynglings Sang': 'Højt på træets grønne top',
  age: 42
}
```

# JavaScript Objekter: Metoder

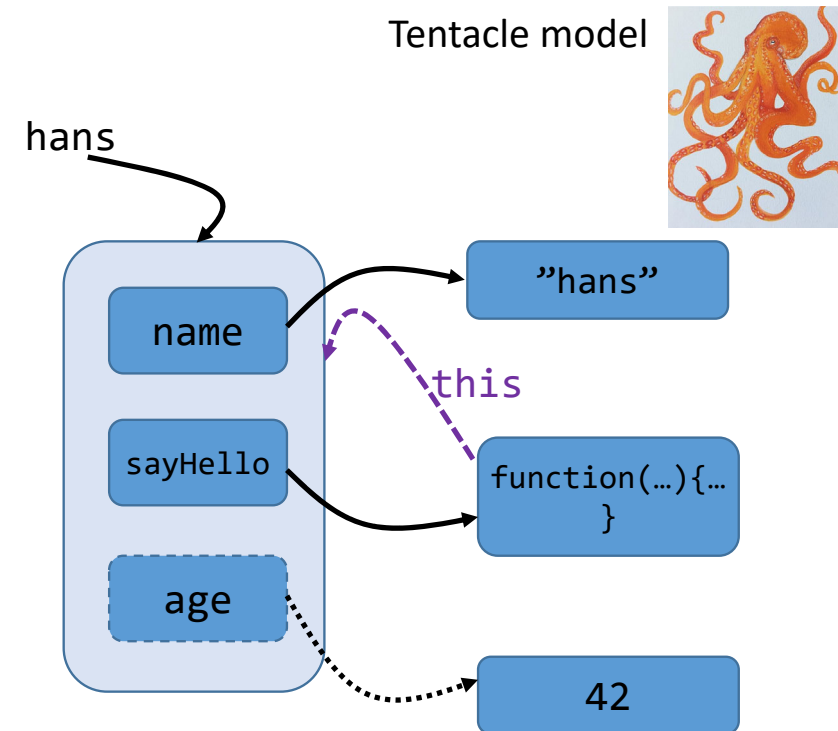
- En egenskab kan også være en funktion
- Kan operere på objektets andre egenskaber
- "Metoder"
- "**this**" er en implicit binding i en funktion til det objekt, som "metode-funktionen" tilhører
- Objekter kan anvendes kapsle relateret data og de funktioner, der arbejder på de data sammen i en logisk enhed

```
let hans={  
  name:"Hans",  
  sayHello: function(toName) {  
    console.log(`Hej ${toName}, jeg hedder ${this.name}`);  
  }  
}  
hans.sayHello("Brian");  
// Hej Brian, jeg hedder Hans
```

# Objekter: tentakel-modellen

- Objekter kan forstås som et "hovedet", der samler armene, som griber egenskabernes værdi.

```
let hans={  
  name:"Hans",  
  sayHello: function(toName) {  
    console.log(`Hej ${toName}, jeg hedder ${this.name}`);  
  }  
}  
hans.age=42;  
hans.sayHello("Brian");  
// Hej Brian, jeg hedder Hans
```



# Objekter: Foranderlighed

- **Tal, Streng, Booleans** er primitive typer som er "immutable"
  - Operationer på strenge genererer nye strenge.
- **Objekter, Arrays** er "mutable"
- Hvis en binding er erklæret **const**, kan *bindingen* ikke ændres
  - **hans={}** giver fejl, da hans er bundet til andet objekt
  - Men det udpegede objekt **kan** ændres
- Som parametre til funktioner
  - de primitive typer opfører sig som "value" parametre
  - Konstruerede typer som referencer (pointers)
- `Object.freeze(hans);`

```
let text1 = "Jeg er uforanderlig";
let text2 = text1.toUpperCase();
//text1[2] = 'c'; //run time error
console.log(text1); //Jeg er uforanderlig;

const hans={
  name:"Hans",
  sayHello: function(toName) {
    console.log(
      `Hej ${toName}, jeg hedder ${this.name}`);
  }
}

hans.sayHello("Brian");//Hej Brian, jeg hedder Hans

hans.name="Peter";
hans.sayHello("Brian");//Hej Brian, jeg hedder Peter
hans.name="Hans"; //ændr tilbage

let peter=hans; //nu bundet til same objekt
peter.sayHello("Brian");//Hej Brian, jeg hedder Hans
hans.sayHello("Brian"); // Hej Brian, jeg hedder Hans
```



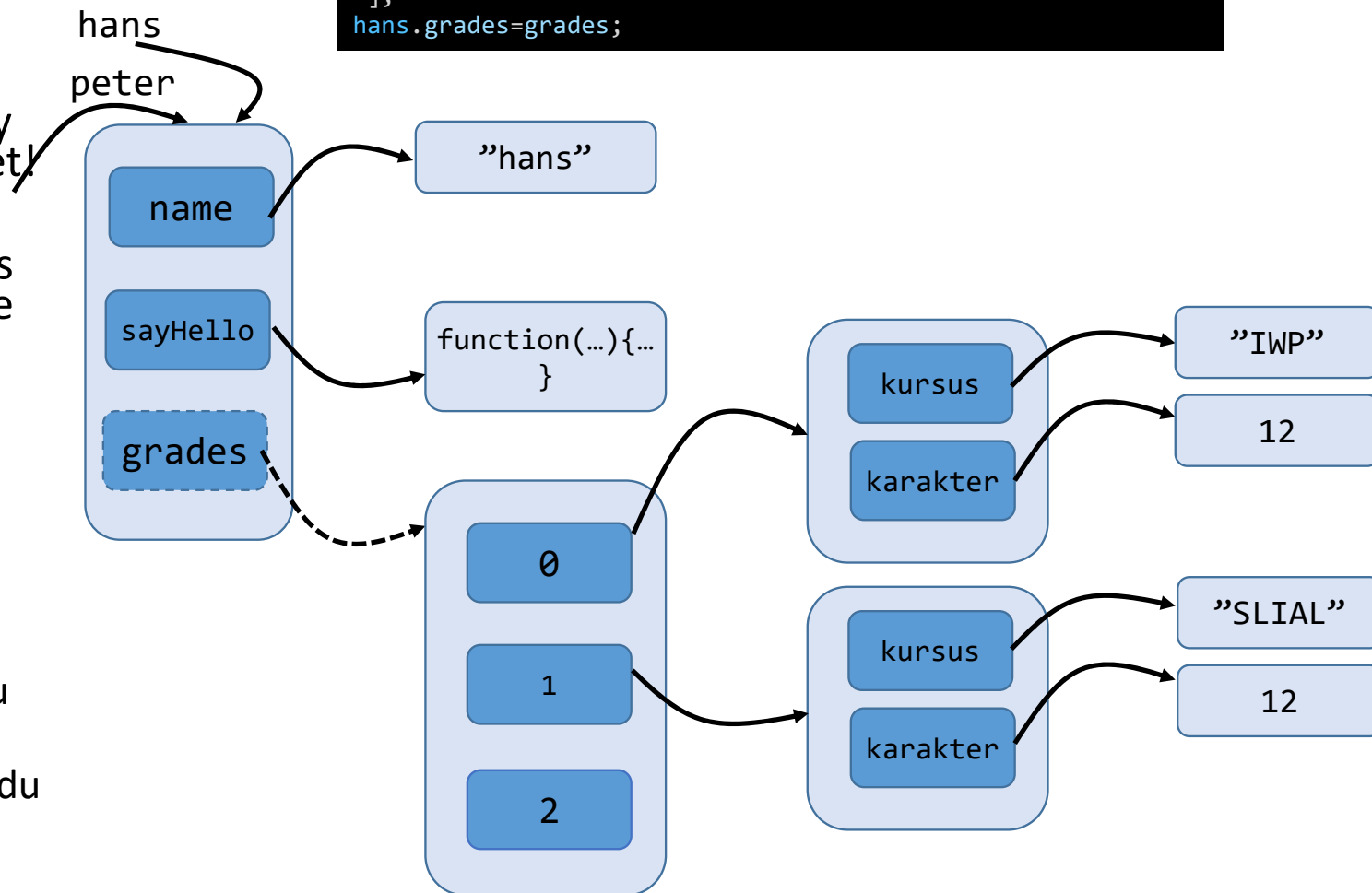
# JavaScript Objekter

- En egenskab kan også være et objekt eller array
- Assignment til objekt skaber ny binding, ikke ny kopi af objektet!
  - `let peter=hans`
- Sammenligning bliver sand hvis operander er bundet til samme objekt
  - `peter===hans //true`
- NB!
  - `deepCopy`, `deepComparison`, `deepFreeze` finde ikke!

```
let kurt={};
Object.assign(kurt,hans);
```

- Kun egenskaber på øverste niveau kopieres
- Kan dog programmers selv – hvis du virkelig har behov

```
let grades=[
  {kursus: "IWP", karakter: 12},
  {kursus: "SLIAL", karakter: 4},
  {kursus: "Imperativ Programmering:", karakter: 7},
];
hans.grades=grades;
```



# En objekt fabrik?

- Hvordan kan vi få mange studenter i vores program?
- En mulig "studenter" objekt fabrik?
  - Objekterne her fungerer som uafhængige objekter, som ikke deler arvemasse (se senere)
- En bedre og systematiseret løsning: Constructor funktioner

```
function studentFactory(studentName){
  let template={
    name:studentName,
    sayHello: function(name){
      console.log(
        `Hello ${name}, my name is ${this.name}!`);
    },
    think: function() {
      return this.name+ " is thinking hard ... ";
    }
  };
  return template;
}
let hans=studentFactory("Hans");
let peter=studentFactory("Peter");
hans.sayHello("Brian");
peter.sayHello("Brian");
```

# Constructor Funktioner og "new" operator

- **New** operatoren opretter en instans af et bruger-defineret (eller indbygget) objekt.
- Anvendes på en funktion.
  - En såkaldt konstruktor funktion, der opretter de nødvendige egenskaber i funktionen
- Konstruktor funktioner skrives pr. konvention med stort begyndelsesbogstav

```
function Student(name){  
  this.name=name;  
  this.sayHello=function(name){  
    console.log(  
      `Hello ${name}, my name is ${this.name}!`);  
  };  
}  
let hans=new Student("Hans");  
hans.sayHello("Brian");
```



# Eksempel fra BMI-JS.JS

- Når bruger indsender nyt sæt BMI data oprettes et BMIEntry objekt

```
//Constructor Function to create BMI-objects
function BMIEntry(name,height,weight){
  console.log(`NEW ${name}, ${height}, ${weight}, `);
  this.userName=name;
  this.weight=weight;
  this.height=height;
  this.calcBMI=function(){... return bmi;};

  this.calcBMIChange=function (otherBMIEntry){
    return round2Decimals(this.calcBMI() -
                          otherBMIEntry.calcBMI());
  };
}
```

```
//create a new unique ID for the game
//name and diceCount comes from user input

let entry=new BMIEntry("Mickey",180,98);
let usersBMI=entry.calcBMI();
...
```

# Objekt Serialisering

- Serialisering er en betegnelse for lave struktureret data om til en "flad" (bit-streng).

- Gem objekt i filer
- Send over netværk

- JSON JavaScript Object Notation,

- Standardiseret data-udvekslingsformat for JS objekter
- Kun "data" egenskaber inkluderes, ikke funktioner
- Som læsbar tekst

- JSON objekt med metoderne

- stringify
- parse

Også brugt i diverse konfiguration filer til VisualStudio

```
let serializedHans=JSON.stringify(hans);
let hans2=JSON.parse(serializedHans);
console.log("Serialized object: "+serializedHans);
console.log(hans2);
```

```
Serialized object: {"name":"Hans","grades":[{"kursus":"IWP","karakter":-3}, {"kursus":"SLIAL","karakter":4}, {"kursus":"Imperativ Programmering:", "karakter":7}]}
```

```
{
  name: 'Hans',
  grades: [
    { kursus: 'IWP', karakter: -3 },
    { kursus: 'SLIAL', karakter: 4 },
    { kursus: 'Imperativ Programmering:', karakter: 7 }
  ]
}
```

# JS Arrays

- Objekter, hvor properties/nøglerne er tal!
- Vi bruger [] til array literaler.
- Alternativt Array constructor
  - Mange og fleksible måder at skabe arrays på, se litt.
- Elementer indsættes/fjernes dynamisk
- UTALLELIGE metoder til at ændre på arrays!
  - Brug bog som opslagsværk, eller
  - web (fx Mozilla Developer Network )
- Kan være "tyndt populeret" (sparse)
  - Afsættes kun plads til de elementer der bruges
- "Itererbare"

```
let diceRoll=[1,6,6,2,3,4,6];
let dice2=new Array(1,6,6,2,3,4,6);
//prepared to hold 10000 elems.
let dice3=new Array(10000);

let sl=dice2.slice(2, 4); //[ 6, 2 ]
console.log(sl);          //[6,2]

sl.push(1);                //[6,2,1]
sl.push(2);                //[6,2,1,2]
sl.push(3);                //[6,2,1,2,3]
console.log(sl.pop());     //3
console.log(sl);           //[6,2,1,2]

console.log("Index på første 6er: "+
            diceRoll.indexOf(6));//1
```

# JS Arrays: Itererbare

- Nogle objekter i JS repræsenterer serier af elementer som kan "oplistes" og "itereres"
  - Elementerne tages et efter et, og behandles
- Fx. **for** (*var of coll*) {krop}
  - Binder et element fra coll til "var", og udfører kroppen
  - Gentages serielt indtil alle elementer er behandlet
- Array-metoden **forEach** kalder en *call-back funktion* for hvert element efter tur.
  - Call-back funktionen tager elementet som første parameter

```
arr.forEach(callback(currentValue[, index[, array]]) {})
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/forEach](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)

```
let diceRoll=[1,6,6,2,3,4,6];
for(let d of diceRoll){
    console.log("Dice " +d);
}

let diceSum=0;
for(let d of diceRoll){
    diceSum+=d;
}
console.log(diceSum); //28

function printDiceHeading(dice){
    console.log(`<h3> Dice ${dice}</h3>\n`)
}
diceRoll.forEach(printDiceHeading);
```

Dice 1  
Dice 6  
Dice 6  
Dice 2  
Dice 3  
Dice 4  
Dice 6

```
printDiceHeading(diceRoll[0]);
printDiceHeading(diceRoll[1]);
...
printDiceHeading(diceRoll[6]);
```

<h3> Dice 1</h3>  
<h3> Dice 6</h3>  
<h3> Dice 6</h3>  
<h3> Dice 2</h3>  
<h3> Dice 3</h3>  
...

# JS Arrays (sparse)

- Arrays kan være tyndt befolkede
  - De fleste JS implementationer afsætter kun plads til de elementer der bruges
  - Ubrugt plads kaldes et "hul"
- Vær dog obs på iteratorer
  - Inkluderer "huller" (undefined værdi)
  - Metoden forEach, skipper huller.

```
let sparse=[];
sparse[7]="Hejsa";
sparse[42]="med";
sparse[100]="dig!";
console.log(sparse.length);//101!

console.log(sparse[10]); //undefined
sparse.forEach(s=> console.log(`heading ${s}`));

//prints all values, incl holes.
for(let [i,d] of sparse.entries()){
    console.log("index "+i+" value "+d);
}
//iterates over non-holes.
let s="";
diceRoll.forEach(d=>s+=`heading ${d}\n`);
console.log(s);
```

```
index 0 value undefined
index 1 value undefined
...
index 6 value undefined
index 7 value Hejsa
```

```
heading Hejsa
heading med
heading dig!
```

# Videregående om arrays og funktioner

- Map og reduce er velkendte indenfor funktions-orienteret programmering
- (også kendte inden for cluster computing til big-data beregning)
  - Map: skaber et nyt array hvor hvert element er "behandlet" af map
  - Reduce: reducerer en serie af værdier til en (typisk)

```
let diceRoll=[1,6,6,2,3,4,6];
```

```
function sum(prevSum,newValue) {  
    return prevSum+newValue;  
}
```

```
let res=diceRoll.reduce(sum,0);  
console.log("SUM="+res); //28!
```

```
sum(0,1)  
sum(1,6)  
sum(7,6)  
sum(13,2)  
sum(15,3)  
sum(18,4)  
sum(22,6)=28
```

```
let strings=["Hejsa", "med", "dig!"];  
let avgStrLen=strings.map(s=>s.length).reduce(sum,0)/strings.length;  
console.log("Avg Len="+avgStrLen); //4!
```

```
["Hejsa", -map---> [5,  
  "med",   -map---> 3,  
  "dig!"]  -map---> ,4] -reduce ---> 12 / 3;
```

```
avgStrLen=strings.map(s=>s.length).reduce((s,v)=> s+v,0)/strings.length;
```

# Eks "BMI Database"

- Array af BMIEntry objekter
- Brug af **filter** metoden til at lave et nyt array med BMIEntries hvor entry.userName matcher userName argument.
- Funktionen til filter returnerer sand/falsk

```
function BMIDB() {
  this.bmiData=[];
  //Add test data
  this.bmiData.push(new BMIEntry("Mickey",180, 90));

  this.lookup= function(userName){... };
  this.calcDelta= function (name){... };
  this.recordBMI=function(bmiRecord){...};

  this.selectUserEntries=function(userName){
    return this.bmiData.filter(e=>e.userName===userName);
  };
}

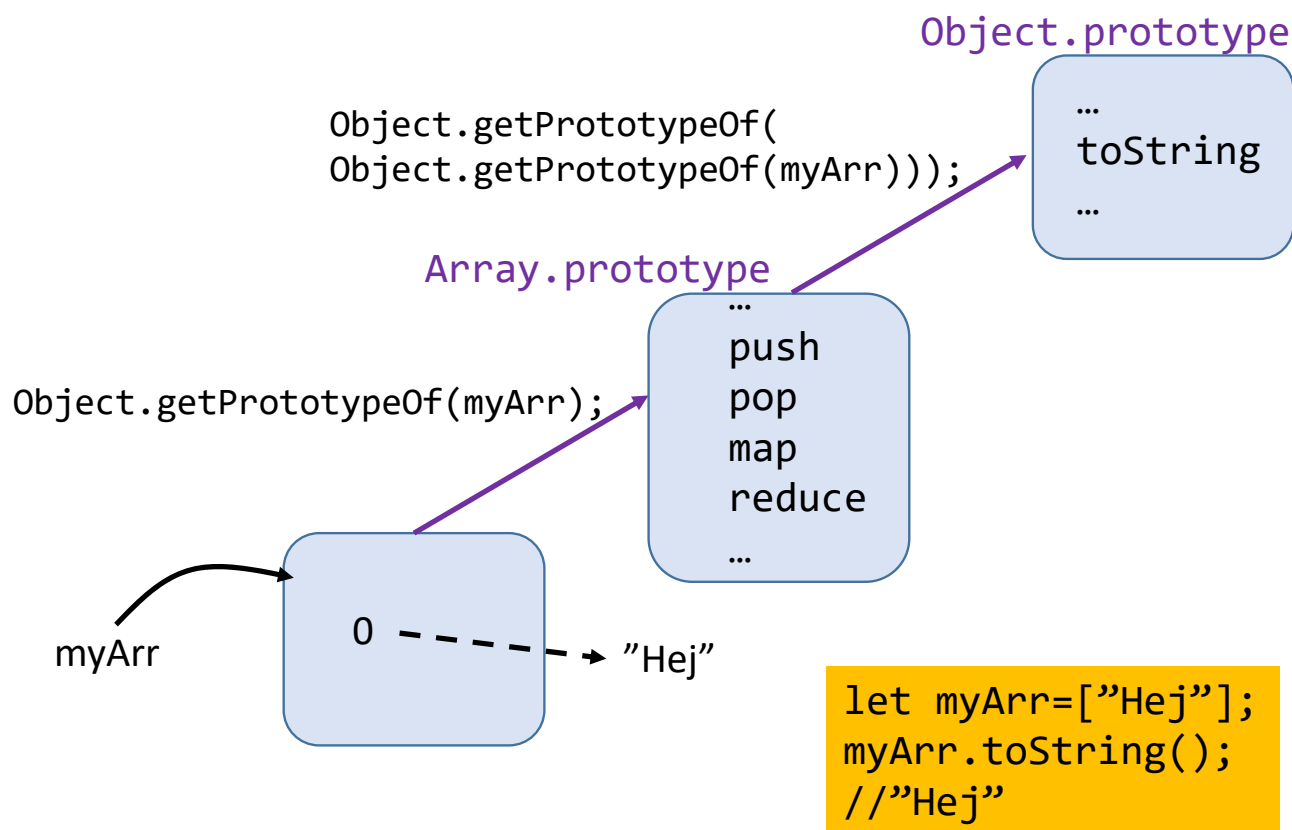
const theBMIDB=new BMIDB();
let mickeyData=theBMIDB.selectUserEntries("Mickey");
```

Videregående / Perspektivering



# JavaScript Objekter: Prototyper

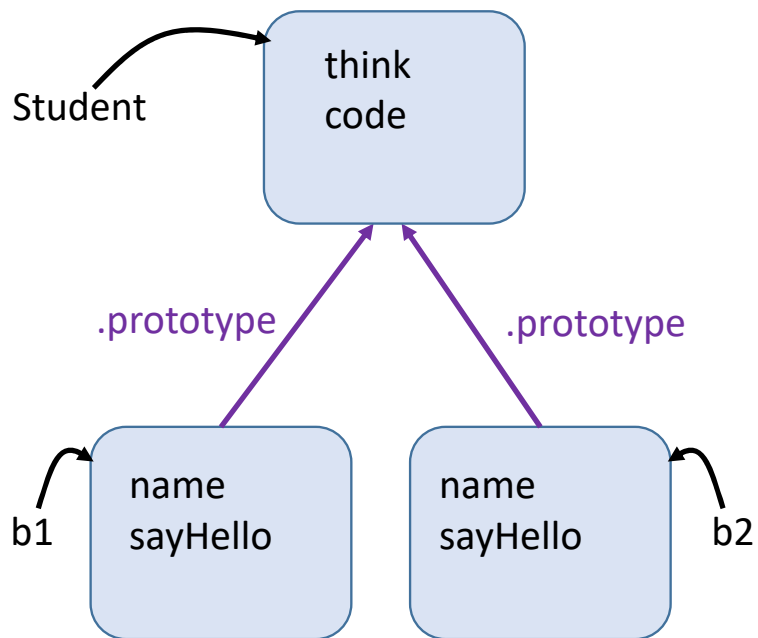
- De fleste JS objekter er skabt ud fra en "prototype", og tilføjer egenskaber til denne
- Adgang til en "property" delegeres via prototype kæden til den mest specifikke erklæring



Du kan lave dit eget objekt baseret på en prototype med `Object.create()`

```
let peter =Object.create(hans);
peter.sayHello("Brian;")
```

# JavaScript: prototyper



Extending objekt at tilføje properties

Overriding metoder ved at erstatte den m. specialiseret version

```
function Student(name){
  this.name=name;
  this.sayHello=function(name){
    console.log(`Hello ${name}, my name is ${this.name}!`);
  };
}
let b1=new Student("Brian");

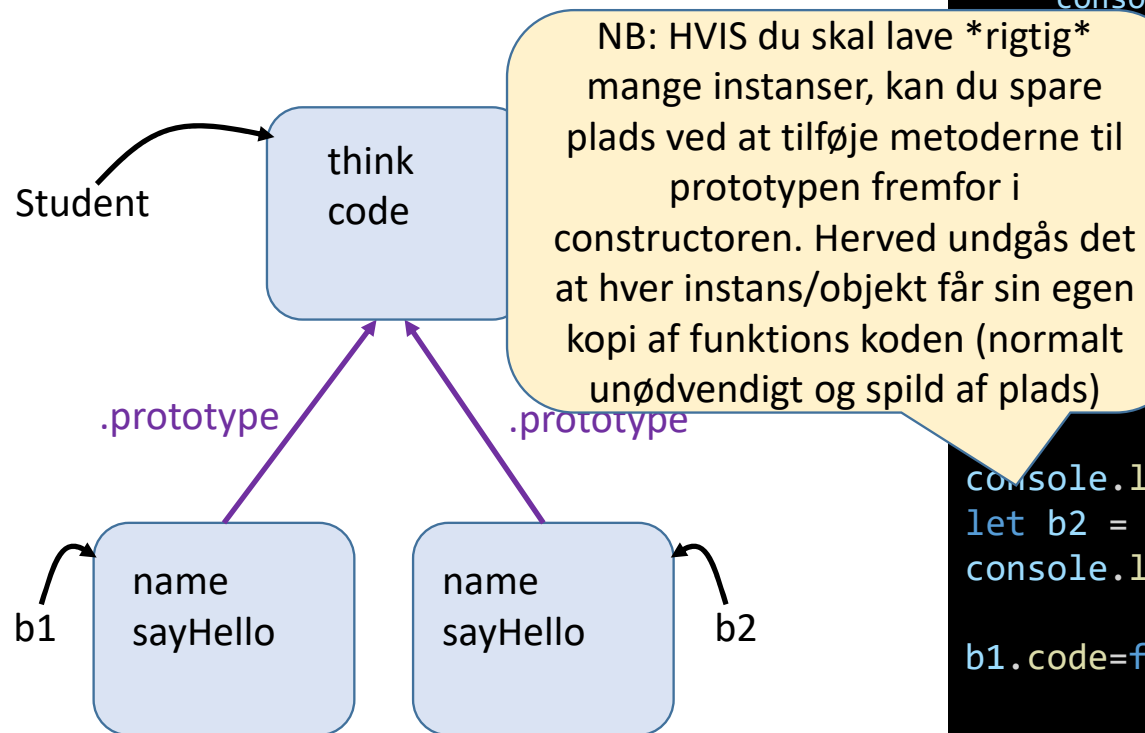
//Add properties to the prototype; now shared by all Students
Student.prototype.think=function() {
  return this.name+ " is thinking hard ... ";};
Student.prototype.code=function() {
  return this.think()+ " and is coding the pants off";}

console.log(b1.code());
let b2 = new Student("Birte");
console.log(b2.code());

b1.code=function(){
  return this.think()+ " and is coding rather lazily";}
console.log(b1.code());
console.log(b2.code());
```

*Brian is thinking hard ... and is coding the pants off  
Birte is thinking hard ... and is coding the pants off  
Brian is thinking hard ... and is coding rather lazily  
Birte is thinking hard ... and is coding the pants off*

# JavaScript: prototyper



NB: HVIS du skal lave \*rigtig\* mange instanser, kan du spare plads ved at tilføje metoderne til prototypen fremfor i constructoren. Herved undgås det at hver instans/objekt får sin egen kopi af funktions koden (normalt unødvendigt og spild af plads)

```
function Student(name){
  this.name=name;
  this.sayHello=function(name){
    console.log(`Hello ${name}, my name is ${this.name}!`);
  };
}

Student("Brian");

// ties to the prototype; now shared by all Students
Student.prototype.think=function() {
  return this.name+ " is thinking hard ... ";};
Student.prototype.code=function() {
  return this.think()+ " and is coding the pants off";};

console.log(b1.code());
let b2 = new Student("Birte");
console.log(b2.code());

b1.code=function(){
  return this.think()+ " and is coding rather lazily";};
console.log(b1.code());
console.log(b2.code());
```

Extending objekt at tilføje properties

Overriding metoder ved at erstatte den m. specialiseret version

*Brian is thinking hard ... and is coding the pants off*  
*Birte is thinking hard ... and is coding the pants off*  
*Brian is thinking hard ... and is coding rather lazily*  
*Birte is thinking hard ... and is coding the pants off*

# Lidt generelt om objekter

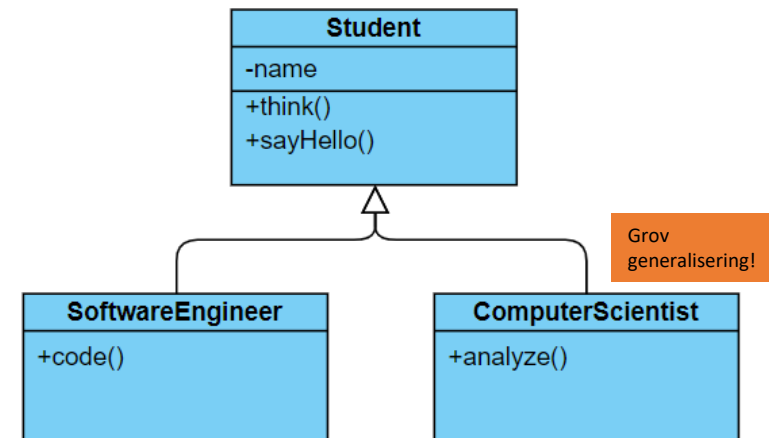
- Der er flere synspunkter på hvad objekter er (og især hvad OO er!) HER:
- ***En samling af tæt-relaterede variable og funktioner, der opererer derpå***
  - Fungerer som en selvstændig enhed i større programmer.
  - En "abstraktion":
    - kan repræsentere en "ting" fra en virkelig eller simuleret verden ("student", "bil", "betaling", "lønseddel")
    - eller en abstrakt datatype (fx "stak", "kø", "træ", "graf"):
- **Indkapsling:** Vi ønsker kun at "annoncere" information om, hvordan en bruger (programmøren/"kalderen") skal bruge objektet, ikke interne detaljer
  - Fx om bruger vi array eller liste til at lave "stak"?
  - Kalderen skal kun kende operationer "push", "pop"
  - => vi kan ændre intern repræsentation og algoritmer uden at ændre alle steder i kode objektet bliver brugt
  - => vi kan arbejde uafhængigt af hinanden
- **Interface:** De egenskaber (fortrinsvist funktioner) som må kendes udad til.
  - I nogle sprog kan funktioner og variable erklæres som "private" eller "public"
  - Nøgleord `private` tilføjet i JS2022

# Lidt generelt om objekter

NB! I OO skelnes egenskaber (properties)

- variable ~ "attributter",
- funktioner ~ "metoder"

- **Class:** En beskrivelse af formen for lignende objekter
- **Objekt / "instans":** et konkret eksemplar af klassen
- **Inheritance:** Et objekt kan "arve" egenskaber fra et andet objekt.
  - Typisk: specialisering/generalisering
- **Polymorfi:** Evnen for et objekt at optræde under forskellig "type".
  - Hvis objektet mindst har "rette egenskaber" (interface) skal det kunne bruges alle steder i programmet, som forventer den egenskab (interface).
  - `University.examine(Student s) {s.think()...}`  
`University.examine(michael);`



I et hypotetisk klasse-baseret OO sprog

```
class Student(){
    String name;
    void think(){...};
    void sayHello();
};
class SoftwareEngineer inherits Student{
    void code() {doCode...};
};

Student hans, peter;
SoftwareEngineer michael;
ComputerScientist kurt;
micheal.think();
```

# JavaScript "Classes"

- En overbygning på prototype begrebet, som *simulerer* et klasse-baseret oo-sprog

```
class Student{
  constructor(name) {this.name=name}
  sayHello (name){console.log(`Hello ${name}, my name is ${this.name}!`);}
  think() {return this.name+ " is thinking hard ... "};}
}

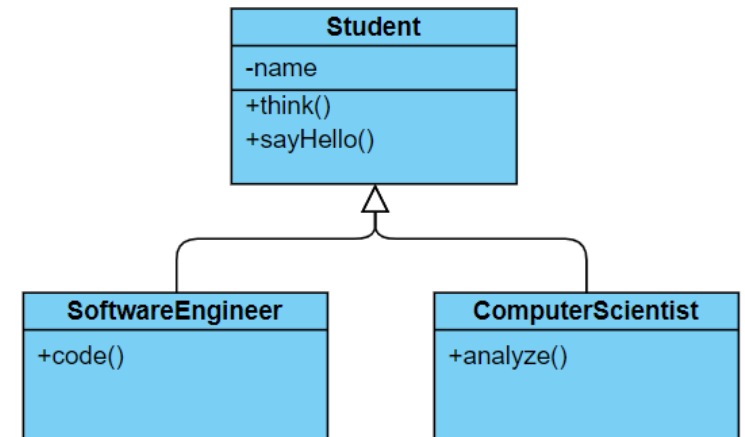
class SoftwareStudent extends Student {
  code() {return this.think()+ " and is coding the pants off";}
}

let b1 = new Student("Brian1");
let b2 = new SoftwareStudent("Brian2");
console.log(b2.code());
console.log(b1.think());
console.log(b1.code()); //Students don't generally code; so trying code()
on a student fails.
}
```

*Brian2 is thinking hard ... and is coding the pants off*

*Brian1 is thinking hard ...*

*Error: b1.code is not a function*



# JavaScript og OO

- Constructor funktioner er nyttigt
- Brug OO og inheritance sparsommeligt:  
Kommer på dat3/sw3 med fuld blæs
  - OOA
  - OOD
  - OOP
- Der er mange finurligheder af JS objekt model, som vi ikke kommer ind på
  - [DF chap 9] har yderligere info
  - For særligt interesserede: Se JS bøger under supplerende materiale, fx <http://speakingjs.com/es5/ch17.html>
  - [Understanding the four layers of JavaScript](#)
  - <https://javascript.info/prototype-inheritance>

