

TOM HEINE NÄTT
JOSTEIN NORDENGEN

Programmering i JavaScript

{ }

© Gyldendal Norsk Forlag as og Dataservice as 2016

Redaktør: Øystein Falch

Design og layout: Odin Media / Claus Gulbrandsen

Omslagsdesign: 07 Media – 07.no / Mette Tønsberg og
Odin Media / Claus Gulbrandsen

Trykk og innbinding: 07 Media – 07.no

Omslagsfoto: Shutterstock

ISBN: 978-82-05-49009-3

Alle henvendelser om utgivelsen kan rettes til:

Dataservice as
Storgata 4C
1767 HALDEN

Det må ikke kopieres fra denne boka i strid med åndsverkloven og fotografiloven eller i strid med avtaler om kopiering inngått med KOPINOR, Interesseorgan for rettighetshavere til åndsverk. Kopiering i strid med lov eller avtale kan medføre erstatningsansvar og inndragning og kan straffes med bøter eller fengsel.
Forfatterene har mottatt støtte fra Det faglitterære fond.

Alle Gyldendal bøker er produsert i miljøsertifiserte trykkerier.
Se www.gyldendal.no/miljo



www.it-2.no

Forord

JavaScript er i dag det rådende programmeringspråket for bruk i interaktive nettsider og såkalt client-side-programmering. Med JavaScript kan du blant annet lage kalkulatorer, bildegallerier, spill og andre interaktive nettsider.

Hvem er denne boka for?

Denne boka er primært utviklet som en del av et læreverk til programfaget Informasjonsteknologi 2. Benyttes den i dette programfaget vil den sammen med *basisbok IT-2* dekke læreplanmålene. Basisboka tar for seg mer overordnede sider av programmering, planlegging og dokumentasjon, mens denne boka viser hvordan programmering faktisk foregår i JavaScript.

Boka kan imidlertid benyttes i all grunnleggende opplæring i JavaScript-programmering. Den vil for eksempel være en god ressurs for studenter som tar et innføringskurs i programmering på en høgskole eller et universitet.

Boka tar som utgangspunkt at du ikke har programmert før, men at du har grunnleggende kunnskaper om å lage nettsider med HTML og CSS.

Hvordan bruker du boka?

De fleste kapitlene inneholder ny teori og små eksempler, for så å oppsummere med større og mer komplette eksempler som også tar med seg teori fra tidligere kapitler.

Programkoden vises med fargekoding for å lette lesingen. Denne fargekodingen kan være ulik i forskjellige verktøy.

Symbolet ↪ brukes der kode i boka er delt over flere linjer på grunn av sidebredden, men egentlig skal skrives på én linje.

I boka finnes det to typer informasjonsbokser, nemlig *merk* og *tips*.

TIPS

Disse boksene inneholder ekstra forklaringer og informasjon som bygger opp under den omliggende teksten. Informasjonen her vil typisk være for dem som ønsker å fordype seg noe mer i temaet.

MERK

Disse boksene inneholder informasjon om vanlige feil og fallgruver man må passe på å unngå når man skriver programkode.

Bokas nettsider

Adressen til læreverkets nettsider er www.it-2.no.

For lærere som skal undervise programfaget IT-2, er det laget et lukket nettsted.

For å få tilgang til dette nettstedet kreves det en lisens som man bestiller ved å klikke på "Faglæreres sider" på www.it-1.no. På nettstedet finner man f.eks. forslag til årsplan, eksamensoppgaver, løsningsforslag på eksamensoppgaver og forslag til prosjektoppgaver.

Oppgaver til boka

Oppgaver til denne boka finnes på våre nettsider: <http://it-2.no/?CatID=1209>.

Klikk på «Oppgaver» ved siden av forsidebildet og oppgavesamlingen lastes ned.

Oppgavene ligger åpent og krever ikke brukernavn og passord.

Vi håper at du vil få stor glede og nytte av å lære å programmere, og at denne boka vil hjelpe deg på vei. Lykke til!

Tom Heine Nätt og Jostein Nordengen

April 2016

Innholdsfortegnelse

{01} Introduksjon til programmering S.10

Dataprogrammer og programmering	10
Programmeringslogikk	11
Pseudokode	11
Programmeringsspråk	12
Hendelsesorientert programmering	13
Utviklingsmiljø	14
Eksemplene i boka	15
Ditt første prosjekt	16
Kodestiler og JavaScript	18
Skrive kode	19
Case-sensitivitet og spesialtegn	19
Lagring	20
Design og funksjonalitet	20
Hurtigfunksjoner	20
Fargekoding	21
Ryddighet	21
Kommentarer	22
Feil i koden	24
Hovedtyper av feil	24
Presentere feil i nettleseren	25
Hjelp og ressurser	27

{02} HTML, CSS og JavaScript S.28

Troikaen	28
JavaScript	30
Grunnskjelettet	30
Hendelser	32
Eksempel - Vis svar på gåte	33
Hendelsesparameter	34
Andre typer musehendelser	34
Manipulere HTML	35
Endre strukturen	36
Eksempel - Legge til rundetider i liste	37
Manipulere CSS	38
Eksempel - Velge farge på tekst	39

Skjemaer	41
Navngiving av skjemaelementer	41
Knapper	41
Tekstbokser	42
Eksempel - Velkomstmelding	42
Hendelsen onchange	46
Eksempel - Tegnteller	46
Avkrysningsboks (Checkbox)	48
Eksempel - Vise melding dersom avkrysset	48
Nedtrekksliste	50
Eksempel - Finn språk	50
Radioknapper	52
Eksempel - Bildefremviser	53
Canvas	55
Tegne linjer	56
Tegne rektangler	57
Tegne en sirkel	57
Eksempel - Sirkeltegner	58

{03} Variabler og operatorer s.60

Hvorfor bruke variabler?	62
Variabelnavn	62
Forandre verdien til variabler	63
Typer data	64
String	64
Number	66
Boolean	66
Tekst og tall	67
Virkningsområde	68
Eksempel - Klikkteller	69
Operatorer	70
Tilordningsoperatoren	70
Matematiske operatorer	71
Eksempel - BMI-kalkulator	74
Konkateneringsoperatoren	77
Eksempel - Historiegenerator	78
Konvertere mellom datatyper	80
Avrunde tall	80
Konstanter	81
Objektvariabler	82

{04} Valgsetninger s.83

Valgsetninger	85
Valgsetninger med et alternativt utfall	87
Valgsetninger med alternative utfall	88
Valgoperatoren	90
Eksempel - Gjett på tall	91
Eksempel - Ballflytter	93

Logiske uttrykk	95
Relasjonsoperatører	96
Logiske operatører	98
AND-operatøren	98
OR-operatøren	99
Negeringsoperatøren	99
Kompliserte uttrykk	100

{05} Løkker s.101

While-løkker	102
Uendelige løkker	103
For-løkker	104
Benytte telleren til mer enn en teller	106
Eksempel - Tegne rutenett	107
Nestede kontrollstrukturer	109
Eksempel - 5 little monkeys jumping...	110
Kontrollere en løkke	112
Break	112
Continue	113
Eksempel - Primtallsgenerator	114

{06} Arrayer s.118

Arrayer (tabeller)	118
Jobbe med arrayer	119
Lengden av en array	121
Legge til elementer i en array	121
For-løkker og arrayer	122
Eksempel - Statistikk over terningkast	123
Array-funksjoner	126
Flerdimensjonale arrayer	128
Eksempel - Finne avstander	129
Assosiativne arrayer	132
Lage en assosiativ array	132
Iterere gjennom en assosiativ array	133
Sjekke om en nøkkel finnes	133
Rader og felt	134
Tekststrenger og arrayer	134
Eksempel - Spørreprogram	135

{07} Funksjoner s.139

- Enkle funksjoner **141**
 - Parametere **143**
 - Eksempel - Sirkelfunksjon **144**
 - Funksjoner som returnerer verdier **146**
- Programflyt **148**
- Globale og lokale variabler **149**
- Rekursive funksjoner **150**
- Funksjon som verdi **151**
- Retningslinjer for funksjoner **153**
- Innebygde funksjoner **155**
 - Matematikkfunksjoner **156**
 - String-funksjoner **158**
 - Eksempel - Kryptering/Dekryptering **159**
 - Eksempel - Palindrom **163**

{08} Finne og rette feil s.165

- Feiltyper **166**
 - Syntaktiske feil **166**
 - Semantiske feil **166**
 - Logiske feil **167**
 - Programmet gir feil resultater **167**
 - Programmet henger eller krasjer **167**
 - Grensesnittfeil **168**
 - Programmet er for ressurskrevende **168**
 - Sikkerhetkritiske feil **168**
- Rette og forhindre feil **169**
- Feilsøkingssverktøy **172**
 - Kommentere vakk kode **172**
 - Skrive ut verdier **172**
 - Debuggeren **173**
 - Utføre kode fra nettleseren **176**
 - Fange opp kjørefasefeil **179**

{09} Hendelser s.180

- Sender-lytter-modell **181**
 - Steg 1 - Lage en lytterfunksjon **181**
 - Steg 2 - Registrere lytterfunksjonen **181**
- Elementer i hendelseshåndteringen **182**
 - Sender **182**
 - Lytter **182**
 - Hendelsesobjekter **182**
 - Flere lyttere til én sender **183**
 - Flere sendere til én lytter **183**
 - Virkning av hendelser **184**
 - Avregistrere en lytter **184**

- Musehendelser **185**
 - Museklikkhendelser **185**
 - Musebevegelser **185**
 - Informasjon i hendelsesobjektet **186**

- Drag and drop **186**
 - Tastaturhendelser **188**
 - Informasjon i hendelsesobjektet **189**
 - Sjekke om en tast er nedtrykket **190**
- Tidsstyrte hendelser **191**
 - Stoppe tidsstyrte hendelser **192**
 - Eksempel - Tegneprogram **193**
 - Eksempel - Drag and drop game **196**

{10} Lyd, video og animasjon s.200

- Lyd og video **200**
 - Spille av lyd **200**
 - HTML audio-tagg **201**
 - HTML video-tagg **202**
- Styre lyd/videoavspilling **203**
- Programmert animasjon **204**
 - Lage animasjon på et canvas **204**
 - Lage animasjon ved hjelp av CSS-aposisjoner **206**
 - Bevegelse og hastighet **206**
 - Konstant hastighet **207**
 - Akselerasjon **208**
 - Bestemme retning **209**
 - Sprett **210**
 - Kollisjonstesting **212**
 - Eksempel - Kanonkulespill **213**

{11} Eksternt innhold s.220

- Lokal testserver **221**
- Lese inn data fra en ekstern tekstfil **222**
- Lese inn strukturerte data **223**
 - XML **225**
 - Lese og behandle XML **226**
 - Eksempel - Oppskriftskatalog **228**
 - JSON **232**
 - Lese og behandle JSON **232**
 - Lese og lagre data lokalt **233**
 - Eksempel - Bakgrunnsfarge **234**
 - Lagre data til server **236**
 - Skrive data **238**
 - Eksterne XML- og JSON-kilder **241**
 - Eksempel - Værdata fra YR **242**
 - Databaser **245**
 - Hente ut data fra database **246**
 - Lagre data i en database **247**
 - Eksempel - Highscorespillet **248**

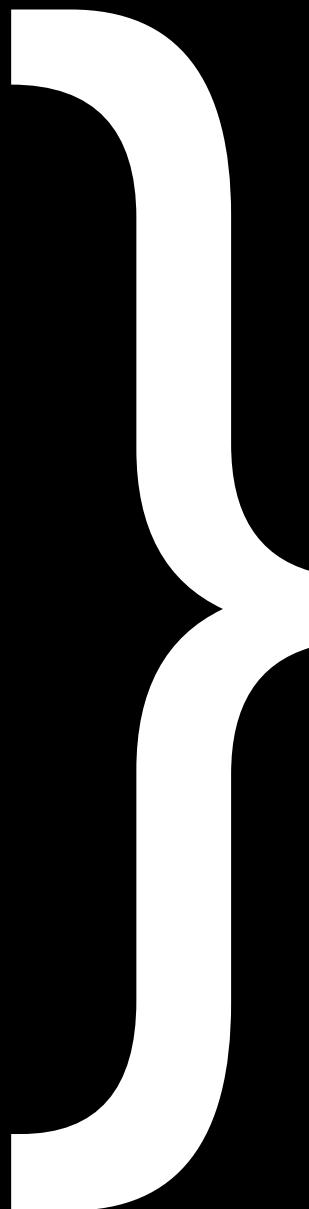
{12} Objektorientering s.252

Klasser og objekter **253**
Arv **255**
Hvorfor objektorientering? **255**
Objektorientering i JavaScript **256**
Egne klasser **257**
Arv **258**

Hurtigreferanse s.260

Variabler og konstanter **262**
Tekst **263**
Skjemaelementer **264**
Kontrollstrukturer **266**
Funksjoner **268**
Arrayer **269**
Assosiative arrayer **270**
Hendelser **271**
Grafikk **273**
Endre HTML-dokumentet **274**
Endre CSS **275**
Lyd og video **275**
Dynamisk innhold **276**
Matte **278**

Index s.279



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

1 Introduksjon til programmering

I dette kapitlet vil du lære

- om begrepene dataprogrammer og programmering
- om JavaScript som programmeringsspråk
- å bruke editorer for å skrive kode
- å lage ditt første JavaScript-program
- om feil i koden

Dataprogrammer og programmering

Det er ikke lett å forklare på noen få linjer hva et dataprogram er, men en definisjon kan være:

Et dataprogram er et sett med entydige instruksjoner, som forteller maskinen steg for steg hvordan den løser et gitt problem.

En *instruksjon* er noe datamaskinen allerede vet hvordan den skal utføre, slik som å legge sammen to tall eller lese en tekstlinje fra en fil. Vår oppgave når vi programmerer, er å sette sammen ulike instruksjoner slik at de løser en bestemt oppgave.

Legg spesielt merke til begrepet *entydige instruksjoner* i definisjonen. Det er viktig å være klar over at et dataprogram ikke lever sitt eget liv og tar egne beslutninger. En datamaskin utfører kun de instruksjonene programmereren har gitt den. Programkoden du skriver, kan derfor ikke basere seg på at datamaskinen skal ta egne beslutninger eller har kjennskap til problemet som skal løses. Et dataprogram blir ikke smartere enn den som har skrevet programmet...



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Programmeringslogikk

Det å sette sammen ulike instruksjoner på en slik måte at programmet gjør det vi ønsker, kalles *programmeringslogikk*. Det er denne kunsten vi må mestre når vi ønsker å programmere. Programmering kan ses på mer som en ferdighet enn kunnskap. Selvfølgelig må man også forstå de ulike instruksjonene og reglene for hvordan de settes sammen, men ofte er dette den enkleste delen av programmeringsfaget.

For å bli en god programmerer kreves det at man løser mange oppgaver og hele tiden trener på den logiske ferdigheten. På samme måte som det er vanskelig å bli en mester i å sjonglere kun ved å lese en bok, er det øvelse som teller også når det gjelder programmering.

Pseudokode

Når man skal løse programmeringsproblemer, er det viktig å ha en plan for hvordan man skal gå fram, før man setter seg ned foran maskinen og begynner på selve programmeringen. Et godt tips vil være å forklare hvordan problemet skal løses steg for steg på en slik måte at de ulike stegene er umulig å misforstå. En slik beskrivelse av programmet kalles ofte *pseudokode* – altså en blanding av forklarende tekst og programkode. Pseudokode kan også være ren tekst som er strukturert i de stegene som løsningen består av.

Når du skal løse problemer, kan det derfor være lurt å starte med å lage pseudokode før du går i gang med selve programmeringen.



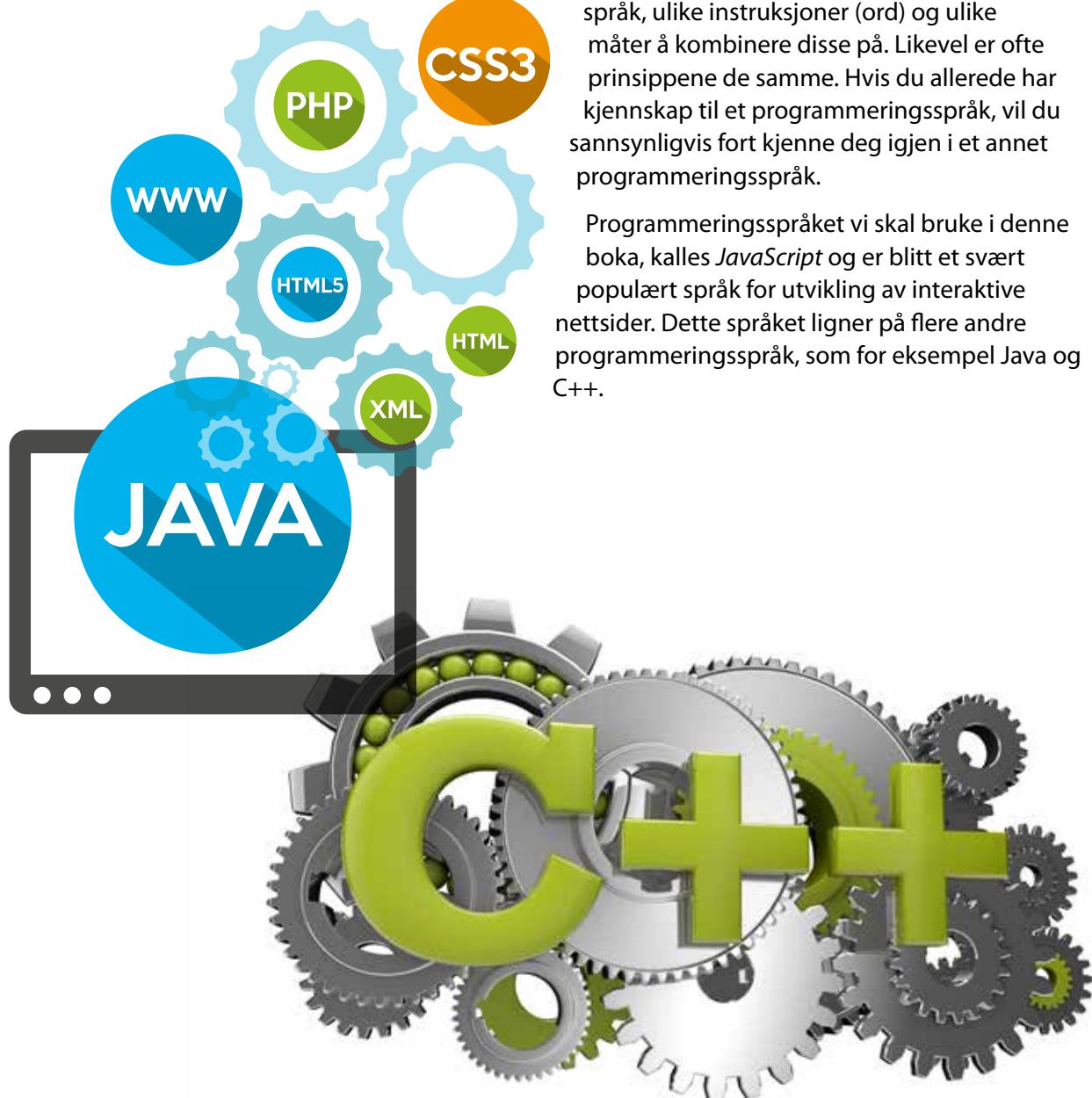
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Programmeringsspråk

Et *programmeringsspråk* bestemmer hvilke instruksjoner vi kan bruke, og hvordan disse instruksjonene kan settes sammen. Dette minner mye om naturlige språk som har både ord og gramatikk.

Ulike programmeringsspråk har, som naturlige språk, ulike instruksjoner (ord) og ulike måter å kombinere disse på. Likevel er ofte prinsippene de samme. Hvis du allerede har kjennskap til et programmeringsspråk, vil du sannsynligvis fort kjenne deg igjen i et annet programmeringsspråk.

Programmeringsspråket vi skal bruke i denne boka, kalles *JavaScript* og er blitt et svært populært språk for utvikling av interaktive nettsider. Dette språket ligner på flere andre programmeringsspråk, som for eksempel Java og C++.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Hendelsesorientert programmering

Når vi programmerer, vil vi ofte benytte oss av det som kalles *hendelsesorientert programmering*. Det vil si at vi kjører kode ut fra hendelser som oppstår, for eksempel at brukeren klikker på en knapp.

Vanligvis er det brukeren som er opphavet til hendelsene når han eller hun klikker på knapper, bruker tastaturet eller musa. Men vi kan også kjøre kode ut fra andre typer hendelser, for eksempel når en fil er ferdig lastet ned, eller tidsstyrte hendelser. Tidsstyrte hendelser er hendelser som oppstår med regelmessige intervaller slik at vi kan for eksempel animere et element ved å flytte det for hvert intervall.

De fleste programmer i dag er hendelsesorienterte. Grunnen til dette er at de vanligvis kjøres under operativsystemer som har et grafisk brukergrensesnitt som brukeren kan styre på forskjellige måter.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Utviklingsmiljø

En av fordelene med JavaScript er at det ikke er knyttet opp mot noe bestemt utviklingsmiljø eller noen spesiell editor der vi skriver programkoden, retter feil og tester programmet. Så lenge du har en teksteditor som lagrer ren tekst, kan den benyttes til å skrive JavaScript-kode med. Dette betyr f.eks. at Windows sin Notepad/Notisblokk kan benyttes.

Vi anbefaler imidlertid å arbeide med en noe mer avansert editor. Fargelegging av programkode (såkalt *syntax highlighting*) og muligheten til å arbeide med flere filer samtidig (faner) er en stor fordel.

Enkelte foretrekker også verktøy som gir forslag på programkode (såkalt *code completion*), men vår anbefaling er å vente litt med slike funksjoner. Som nybegynner er det veldig lett å gå ukritisk for forslagene som dukker opp, og vanskelig å finne ut av feilene som oppstår dersom det ikke var riktig valg.

I denne boka kommer vi til å benytte verktøyet *Notepad++* som kan lastes ned fra <http://notepad-plus-plus.org>. Du kan imidlertid fint benytte din favoritteditor i stedet. Andre muligheter er f. eks. Dreamweaver, Brackets, WebStorm og Sublime Text.

For å se på resultatet av JavaScript-programmer trenger du kun en nettleser. Det er imidlertid ikke alle nettlesere som er like gode på å vise all slags JavaScript-kode, samt ikke alle nettlesere som er like behjelpeelige med å feilsøke. I denne boka kommer vi til å benytte *Google Chrome* som nettleser, men du står fritt til å benytte noe annet. Kapitelet om feilretting kommer imidlertid til å omtale funksjoner og vise skjermbilder fra Chrome.



JavaScript utvikles stadig. Enkelte ting vi viser i denne boka, kan kreve en nyere nettleser for å kunne kjøres. Vi vil anbefale å benytte en nettleser som oppdaterer seg selv, og som generelt sett ligger langt fremme i å støtte nye standarder.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksemplene i boka

Så langt det lar seg gjøre, vil vi forsøke å holde eksemplene i boka så små og enkle som mulig. Spesielt gjelder dette i starten av boka. For å få til dette er det kanskje ikke alle programmene vi lager, som kan klassifiseres som like fornuftige. Ofte vil en utskrift av noen tall eller endring av noen farger være nok til å illustrere prinsipper i programmeringen.

Vi anser det lettere å forstå noen få linjer programkode i stedet for et stort og avansert eksempel med en mengde linjer kode hvor kun én eller to av linjene illustrerer prinsippene eksempelet forsøker formidle. I tillegg reduserer vi mengden feil som ellers vil kunne gjøre det vanskelig å få eksempelet til å fungere på din maskin.

Alt du lærer gjennom disse mindre eksemplene, kan du ta med deg og lage mer fornuftige programmer senere, noe vi også vil fokusere på mot slutten boka.

Med dette prinsippet følger også at utseende på eksemplene blir noe begrenset. Vi kommer ikke til å "pynte" på nettsidene via CSS der det ikke er nødvendig for hva vi ønsker å vise. Mer design kan du gjerne legge på selv, etter at du har fått programkoden til å fungere.

Som tidligere nevnt kommer vi til å benytte Notepad++ som editor i denne boka. All programkode vil vises med fargekoding (syntax highlighting) fra dette verktøyet. Vi har imidlertid valgt å endre fargen tekststrenger (alt mellom " og ") vises med, fra grått til oransje. Dette av hensyn til lesbarheten på trykk.

Der eksemplene består av flere skritt og programkode som skal utvides, har vi valgt å markere programkode som allerede skal være skrevet inn, med en grå farge. Dette for å gjøre det lettere for deg å orientere deg om hvor den nye koden skal plasseres.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

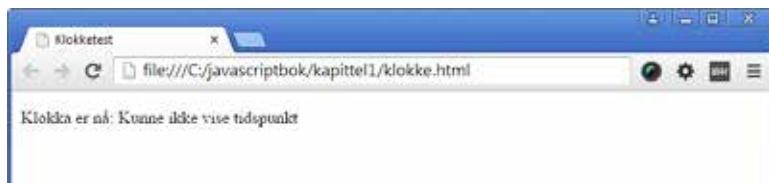
Ditt første prosjekt

For å komme i gang med å forstå hvordan vi programmerer JavaScript, skal vi her gå gjennom et eksempel som lager en svært enkel nettside med en tekstlig klokke. Ikke bekymre deg så mye over hvorfor programkodene ser ut som de gjør. Dette kommer vi tilbake til senere. Tenk mer over prinsippene rundt hvordan programkoden fikk ting til å skje, samt prosessen med å skrive kode og teste nettsiden.

Først lager vi en svært enkel nettside i HTML5. Nettsiden inneholder ikke noe spesielt, annet enn en ``-tag med *ID* satt til **klokkeutskrift** der hvor vi ønsker at tidspunktet skal stå. Det er svært viktig at vi gir elementer vi etter hvert skal styre fra JavaScript, en slik ID. Vi kan ikke i programkoden si "det elementet du ser ved siden av teksten klokka er nå skal få en ny verdi". I programkoden omtales kun ting med ID-er (navn).

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Klokketest</title>
</head>
<body>
    <p>Klokka er nå: <span id="klokkeutskrift">Kunne ikke vise tidspunkt</span></p>
</body>
</html>
```

Velg å lagre denne fila som *klokke.html* et sted lokalt på din maskin. Åpne så fila i en nettleser og se at den viser teksten i paragrafen. Som regel åpnes fila i nettleseren om du dobbeltklikker på den. Du kan også åpne fila gjennom nettleserens *open-kommando* (vanligvis Ctrl+O).



Vi skal nå legge til JavaScript-koden som faktisk genererer klokka. Denne koden skal erstatter teksten som nå står inne i ``-taggen, med det faktiske tidspunktet. Dette er en smart oppbygning av et JavaScript-prosjekt, der feilmeldingen er standardvisning, og dersom alt fungerer, erstattes denne med det faktiske resultatet.

For å få til denne funksjonaliteten legger vi til en `<script>`-tagg i `<head>`-seksjonen av HTML-fila og fyller denne med JavaScript-kode. Denne blokken plasseres i `<head>` ettersom programkoden ikke er en del av selve innholdet på nettsiden. Den manipulerer bare innholdet.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Klokke&nbsp;test</title>
    <script>

        window.onload = oppdater;

        function oppdater() {
            var tid = new Date();
            document.getElementById("klokkeutskrift").innerHTML = tid. ↪
                getHours() + ":" + tid.getMinutes() + ":" + tid.getSeconds();

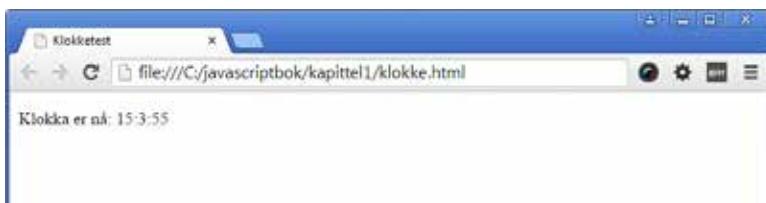
            setTimeout(oppdater, 1000);
        }

    </script>
</head>
<body>
    <p>Klokka er nå: <span id="klokkeutskrift">Kunne ikke vise ↪
        tidspunkt</span></p>
</body>
</html>
```

Lagre nettsiden, og vis den på nytt i nettleseren. Har du nettsiden åpen fra tidligere, må du oppdatere den. Feilmeldingen skal nå ha blitt erstattet av korrekt tidspunkt, og klokka skal oppdatere seg av seg selv. Se bort fra at tall under 10 vises uten en 0 foran. Dette vil du lære hvordan du kan ordne senere i boka, men det blir litt for mye koder nå.

Selv om vi ikke skal forklare koden i detalj her, og det heller ikke er meningen at du skal forstå detaljene, så tar vi oss tid til en liten gjennomgang. Først forteller vi at når nettsiden er ferdig hentet ned (`window.onload`), så skal blokken med kode (funksjonen) kalt `oppdater` utføres.

Deretter definerer vi hva blokken/funksjonen `oppdater` skal innebære. Først skal den hente ut systemtiden og kalle denne `tid`. Så skal koden få fatt i elementet med `ID` satt til `klokkeutskrift` i HTML-koden, og bytte teksten mellom taggene (`innerHTML`) med `tid` sin time-verdi, et kolon, `tid` sin minutt-verdi, et kolon og `tid` sin sekund-verdi. Til slutt skal det registreres at funksjonen `oppdater` skal utføres på nytt etter 1000 millisekunder (tilsvarer ett sekund)



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Kodestiler og JavaScript

Som du vil oppdage når du leser om JavaScript på nett eller ser andres koder og eksempler, finnes det en rekke ulike stiler og oppsett å skrive JavaScript etter. Til en viss grad er de ulike stilene mer eller mindre likeverdige, men det finnes allikevel visse grunner til å foretrekke noen fremfor andre.

Vi skal ikke gå inn på detaljer så tidlig i boka, men vil kort redegjøre for noen av de større valgene vi har gjort.

Et av de store skillene er om det er JavaScript-koden som refererer til HTML-koden eller omvendt. I eksempelet du alt har sett, er det ingen ting mellom `<body>` og `</body>` som refererer til noe av JavaScript-koden. Vi refererte derimot til en tagg med en bestemt *ID* ved hjelp av `document.getElementById` fra JavaScript-koden i `<head>`.

Den alternative metoden er å referere til funksjoner i JavaScript direkte fra HTML-koden:

```
<body onload="oppdater()">
<div onclick="sendData()">
```

Vi har bevisst valgt å aldri si noe om JavaScript i HTML-koden. Dette gjør at koblingen mellom HTML og Javascript kun blir enveis, og at HTML-koden kan utvikles uavhengig av om man kan JavaScript eller ikke. Det er også enklere å benytte den samme JavaScript-koden på flere nettsider, uten å nødvendigvis måtte forandre noe på selve koden. Denne stilens å skrive JavaScript på virker også å bli mer og mer dominerende, selv om ingen av metodene er "forbudt".

For å unngå å ha flere filer å forholde seg til vil vi i eksemplene i denne boka skrive JavaScript og CSS som en del av HTML-koden, plassert i `<head>`. Etter hvert er det imidlertid lurt å skille ut CSS og JavaScript i egne filer, noe som er god praksis å gjøre:

```
<head>
  <script src="script.js"></script>
  <link rel="stylesheet" type="text/css" href="stil.css">
</head>
```

Det finnes også mer formelle regelsett for hvordan kode skal skrives. *JSLint* er et slikt regelsett som er utarbeidet for å skrive kode som mange mener er mer sikret mot feil og mer oversiktlig. Editoren *Brackets* har JSLint skrudd på som standard. Vi kommer ikke til å følge noen slike regelsett i denne boka, men du står fritt til å gjøre det på egen hånd.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Skrive kode

Når du skal skrive programkode, er jo dette i prinsippet ren tekst. Programkode har imidlertid noen ekstra utfordringer ved seg som vi vil omtale her, og som kan redde deg unna mange potensielle feil og problemer. I tillegg vil de kunne gi deg en mer ryddig og mer oversiktlig kode.

Case-sensitivitet og spesialtegn

Vær klar over at programkode for JavaScript er såkalt *case-sensitiv*. Dette betyr at kommandoer og referanser skrevet med ulike case (store/små bokstaver) ikke refererer til det samme.

Koden `document.getElementById("KlokkeUtskrift")` henter altså ikke ut elementet dersom ID-en er stavet `klokkeutskrift` i HTML-dokumentet. Feilstavinger i selve kommandoene vil også gjøre at `Document.getElementById ("klokkeutskrift")` vil gi en feilmelding ettersom begrepene `Document` og `getElementById` ikke er kjente.

Bruk av æ, ø, å, mellomrom, prosent og andre spesialtegnbør også unngås i alt som ikke er tekst og informasjon som brukeren skal se. I de fleste tilfeller vil tegnene fungere, men i de systemoppsettene det ikke fungerer, lager det feil som er svært vanskelige å finne.

Husk at case-sensitivitet og anbefalingen om å unngå spesialtegn også gjelder filnavn. På de fleste webservere (alle Linux-baserte) vil filen `minSide.html` være en annen fil enn `minside.html` eller `minside.HTML`.

Selv om det kan virke litt sært i det hele tatt å benytte store bokstaver i programmeringskode, er ”regelen” enkel. Der hvor et begrep eller navn egentlig burde bestått av flere ord, vil vi heller alltid skrive det i ett ord, med stor bokstav der hvert ord egentlig skulle begynt.

Med andre ord blir *get element by ID* skrevet `getElementById`. Tilsvarende vil *antall ledige plasser* bli `antallLedigePlasser`. Denne teknikken kalles *camelCase*.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lagring

Et annet velkjent problem når vi skriver kode, er at vi glemmer å lagre fila/filene i editoren vi benytter. Ved stadige vekslinger mellom editor for å endre koden og nettleseren for å se resultatet er det fort gjort å glemme dette steget. Når vi da ser på resultatet i nettleseren, kan det være vanskelig å forstå hvorfor koden vi nettopp skrev, ikke ga det resultatet vi forventet.



Du synes sikkert dette høres ut som en helt opplagt ting å huske på, og en opplagt ting å tenke på dersom problemet oppstår, men vent og se. Er man langt inne i kodetankengangen, vil koden også være det første stedet vi leter etter årsaken til problemet.

Design og funksjonalitet

Mange velger å begynne prosjekter med innholdet og designet for deretter å legge på funksjonalitet og programkode til slutt. Dette er en naturlig måte å angripe problemet på for mange, da programkoden ofte er det vanskeligste og dermed det man velger å utsette.

Allikevel er det en anbefaling å starte med programkoden. Det er mye enklere å skrive og feilsøke koden dersom det er få andre elementer å forholde seg til.

Hurtigfunksjoner

Så godt som alle editorer støtter hurtigtaster. Du kan med en gang lære deg følgende:

Ctrl + C	Kopier markert tekst
Ctrl + V	Lim inn kopiert tekst
Ctrl + X	Kopier og slett market tekst (klipp ut)
Ctrl + Z	Angre (undo)
Ctrl + Y	Angre siste angring (redo)
Ctrl + A	Marker alt
Ctrl + F	Åpne verktøy for å søke etter tekst
Ctrl + S	Lagre

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Alle editorer har også sine egne hurtigtaster. Følgende gjelder for Notepad++. Du finner hurtigtaster for andre editorer i manualen eller ved et nettsøk:

Ctrl + D	Dupliser linjen du har markøren på
Ctrl + L	Sletter aktiv linje
Ctrl + K	Gjør om markert tekst til en kommentar
Ctrl + Shift + K	Fjerner kommentar
Ctrl + U	Gjør om markert tekst til lower case
Ctrl + Shift + U	Gjør om markert tekst til upper case

Pass imidlertid på at du ikke benytter slike funksjoner ukritisk. En vanlig kilde til feil er den såkalte copy/paste-syken der man kopierer kode man har til hensikt å endre litt på, men glemmer et par av endringene og introduserer feil som kan være vanskelige å finne.

Faktisk skal du svært sjeldent være nødt til å kopiere kode, da behovet for dette svært ofte indikerer at du burde benyttet funksjoner eller objektorientering, som vi skal komme tilbake til senere i boka.

Fargekoding

Selv om fargekodingen (syntax highlighting) som de fleste editorer inneholder, er et godt hjelpemiddel for å få oversiktig kode, er det viktig å være klar over at denne funksjonaliteten ikke har noen form for intelligens. Fargekodingen leter kun etter kjente ord, tegn og sekvenser.

Altfor mange som lærer seg å programmere, tror at koden de skriver, er riktig kun fordi den "blir blå". Sørg derfor for å lære deg å skrive riktig kode og forstå hvordan kode bygges opp, uavhengig av hvordan koden fargelegges.

Ryddighet

Selv om det kanskje ikke virker slik når programkoden til stadighet feiler, så er maskinen svært lite nøyne på hvordan programkoden struktureres. Linjeskift, innrykk og mellomrom kan stort sett plasseres der du vil, så lenge det ikke splitter selve kodebegrepene.

Faktisk kan du skrive all programkoden på en eneste lang linje. Det som er viktig, er imidlertid å lære seg å skrive programkode med en god struktur. Innrykk (såkalt indentering), mellomrom og linjeskift gjør koden lett å lese, reduserer sannsynligheten for feil og gjør det lettere å finne feil som allikevel dukker opp.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

En utfordring er at få ser hensikten med å ha en ryddig struktur på alle de små eksemplene de lager i starten av sin programmeringskarriere. Har man først lært seg å skrive dårlig kode, er dette vanskelig å avvenne når kompleksiteten øker. Bruk derfor mye tid og energi nå i starten for å forstå hvordan god og ryddig kode skal skrives.

Kommentarer

Ofte ønsker vi å legge inn forklarende tekst i koden i form av *kommentarer*. Disse kommentarene ignoreres helt av maskinen, men skal snarere være til hjelp for mennesker som skal lese koden for å forstå hva som er gjort. For å skille kommentarer i koden fra koden selv er det innført flere ulike måter å skrive kommentarer på.

Om vi ønsker å ha kommentarer som kun går over én linje, kan vi starte kommentaren med //. Maskinen vil da skjønne at alt som står etter disse tegnene og frem til linjeskift, er kommentarer den ikke skal bry seg om. For eksempel:

```
//Henter ut systemtiden og kaller denne for tid  
var tid = new Date();
```

Vi kunne også skrevet kommentaren på slutten av linja:

```
var tid = new Date(); //Henter ut systemtiden og kaller denne for tid
```



Hvilken av de to stilene å kommentere på som man velger, er en smakssak, men det kan lønne seg å være konsekvent med den ene eller den andre metoden.

Det finnes også en måte å markere at tekst som går over flere linjer, skal være én kommentar. Vi starter da kommentaren med /* og avslutter med */. For eksempel:

```
/* Her hentes systemtiden ut.  
Denne verdien tas vare på  
og gis betegnelsen tid */  
var tid = new Date();
```



Det er viktig å huske på å avslutte en kommentar som går over flere linjer, ved hjelp av */. Gjør du ikke dette eller skriver avslutningen feil, vil maskinen tro at resten av fila er en kommentar.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Ofte prøver man også å få en litt finere formatering på lengre kommentarer ved hjelp av en ny stjerne på hver linje. Dette har ingen praktisk betydning og er kun gjort for å øke lesbarheten:

```
/* Her hentes systemtiden ut.  
 * Denne verdien tas vare på  
 * og gis betegnelsen tid  
 */  
var tid = new Date();
```

Etter hvert trenger man selvfølgelig ikke å kommentere så opplagte ting, slik som det eksempelet over viser, men det er viktig å kommentere koden godt. Det kan både hjelpe deg selv til å forstå hva du har tenkt (spesielt om det er lenge siden du skrev koden), og i alle fall hjelpe andre som ikke synes din løsning er like opplagt.

Det er også viktig å få en god vane med å kommentere samtidig med at du lærer deg å programmere. Det er vanskeligere å lære seg til denne vanen i ettertid.

Du kan også midlertidig "deaktivere" deler av koden din ved å sette kommentartegn rundt koden. Maskinen vil da ikke utføre koden fordi den tror det er en kommentar. Dette er en god måte å feilsøke kode på.

TIPS



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Feil i koden

Det er nok ikke noe sted uttrykket "alle kan gjøre feil" passer så godt inn som i programmering. Selv de mest erfarne programmerere gjør masser av feil, så du må ikke fortvile om programmet ikke kjører ved første forsøk.

Det som er viktig, er å ikke bare rette feilen, men også forstå hvorfor det ble feil. Man lærer faktisk mer av å lage programmer som feiler, enn av programmer som virker på første forsøk. Lærer du deg årsaken til feilen, vil dette også hindre deg i å gjøre samme type feil igjen.

Dette poenget illustreres godt i en historie som fortelles om Thomas Edison. Totalt gjorde han nærmere 10 000 forsøk på å lage en lyspære. Underveis ble han spurt om hvordan han kunne orke å fortsette etter så mange mislykkede forsøk. Edison svarte da at han ikke hadde hatt et eneste mislykket forsøk. Alle forsøkene var særdeles vellykkede i å bevise hvordan man ikke kunne lage en lyspære.

Hovedtyper av feil

Litt forenklet kan vi si at vi skiller mellom to hovedtyper av feil: de *syntaktiske* og de *logiske*. De syntaktiske feilene omfatter skrivefeil og strukturelle feil i koden. Sammenligner vi med naturlig språk, vil dette være en type feil der stavemåten i ord eller grammatikken i setningene er feil. I slike tilfeller vil ikke datamaskinen forstå hva den skal gjøre. Programkoden blir derfor ikke kjørt i det hele tatt.

De logiske feilene er feil som kun oppdages når programmet kjører. Programmet utfører de instruksjonene du har skrevet i koden, men programmet virker ikke slik som du hadde tenkt deg. Sammenligner vi igjen med naturlig språk, er det setninger som høres/ser fine ut, men som ikke har den meningen eller oppfattes av andre slik vi ønsker. Senere i boka skal vi se på hvordan vi forhindrer, finner og retter slike feil.



I tillegg til å gi feil resultater kan logiske feil også få programkoden til å stoppe opp under kjøring, rett og slett fordi det du ber maskinen om, ikke er mulig å gjennomføre. Logiske feil kan også få maskinen til å utføre den samme operasjonen om igjen og om igjen, noe som oppleves som at programmet har "hengt seg".

De logiske feilene kan være av mange typer og er så knyttet opp i problemet som skal løses, at det er vanskelig å si noe generelt om disse feilene. Vanligvis er det disse feilene som det er vanskeligst å finne ut av. Syntaktiske feil finner maskinen for oss når vi prøver å kjøre programmet. Vi må imidlertid selv rette dem, da maskinen kun vet at det er feil, og ikke hva som er det riktige.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Presentere feil i nettleseren

I vanlig modus vil de syntaktiske feilene oppleves som at koden ikke utføres, og at nettsiden ikke gjør det den var forventet å gjøre. De fleste nettlesere i dag inneholder derimot funksjoner for å presentere syntaktiske feil i JavaScript. Denne boka vil fokusere på hvordan dette gjøres i Google Chrome.

Forsøk å fjerne kommaet mellom de to verdiene i denne kodelinja i eksempelet vi lagde tidligere:

Fra:

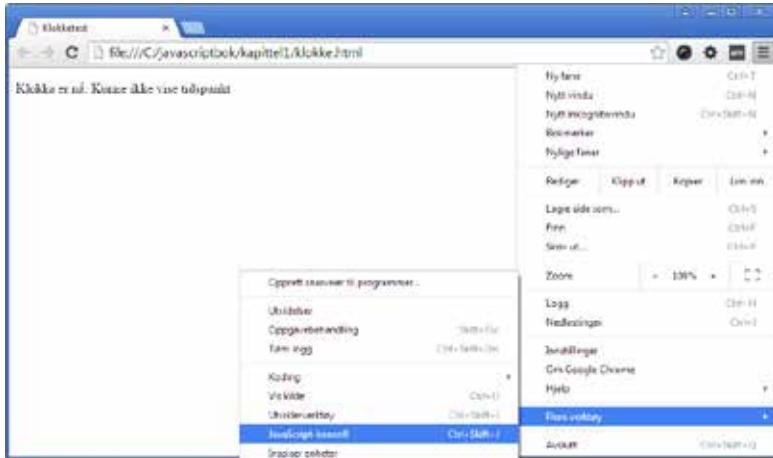
```
setTimeout(oppdater, 1000);
```

Til:

```
setTimeout(oppdater 1000);
```

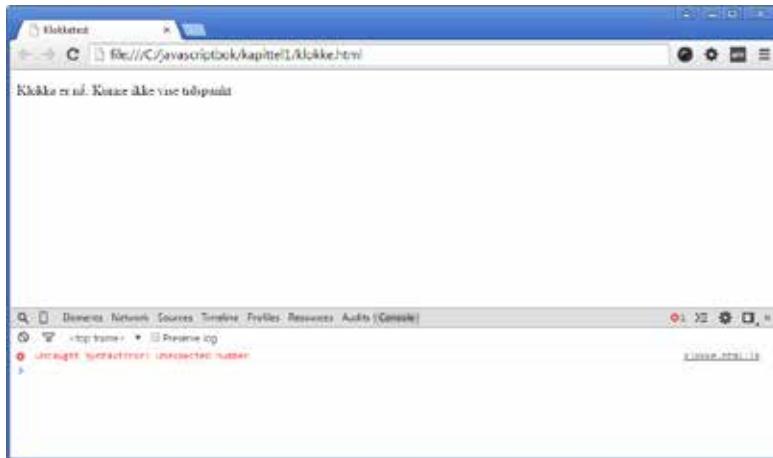
Lagre filen og vis den i Chrome. Resultatet vil nå være at nettsiden vises på samme måte som før vi la til programkoden. Med andre ord at vi ser teksten "Kunne ikke vise tidspunkt", uten noe mer informasjon om hva som var galt.

Velg så menyen i Chrome og deretter valget **flere verktøy** og **JavaScript-konsoll**.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Du får nå opp et panel som du så kan skjule/vise ved hjelp av **F12** eller **Ctrl+Shift+J**. Dette panelet vil vise alle syntaktiske feil i koden.



Den noe kryptiske feilmeldingen forteller at den støtte på et nummer når den egentlig forventet noe annet. Den forteller også at feilen ble oppdaget på linje nummer 16 i fila *klokke.html*. Merk deg at vi skrev "ble oppdaget" og ikke "er". Det er nemlig ikke helt sikkert at feilen oppdages av systemet på den linja som du må lete etter den på. I mange tilfeller ligger feilen et par linjer før. Det er imidlertid det som står på angitt linje, som avslørte problemet.

Ofte kan vi lure på hvorfor systemet ikke bare retter koden for oss, dersom den er så sikker på hva som er feil og hvor feilen er. Grunnen er at det ofte er mange mulige løsninger på en feil. Vi ønsker ikke at en maskin skal "gjette" på hva som er riktig. Tenk deg selv om det var en programmeringsfeil i autopiloten til et fly eller ABS-bremsene til en bil. Vi ønsker ikke at systemet skal gjette på hva programmereren egentlig mente å gjøre.

TIPS

En enkelt feil i koden kan ofte føre til mange feilmeldinger. Det kan være lurt å rette den øverste feilen først, for så å teste på nytt og se hvilke feil som vi da står igjen med.

Det er lurt å gjøre seg kjent med dette systemet for feilmeldinger nå mens programmene er små og oversiktlige, så det ikke blir for mye å forholde seg til når programmene blir mer kompliserte. Et godt tips vil være å ta feilfri kode og selv introduisere ulike typer feil ved å gjøre endringer i koden, for så å se hvilke typer feilmeldinger som da blir resultatet.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Hjelp og ressurser

Selv om vi vil ta for oss mye grunnleggende JavaScript-programmering i denne boka, vil du garantert få behov for å finne mer utdypende materiale etter hvert som du ønsker å lage prosjekter med en egen vri.

Det finnes flere oppslagsverk, referanser og online-kurs for JavaScript. Et sted å starte er f. eks. w3schools sin seksjon for JavaScript:

<http://www.w3schools.com/js/>

Som del av din opplæring er det også svært effektivt å ta for seg eksisterende eksempelkode og endre på denne. De gangene endringene resulterer i feil og utilsiktet oppførsel, er det viktig å bruke god tid på å forstå hvorfor dette skjedde. Som tidligere nevnt lærer man mer av å gjøre feil enn å gjøre riktig, så lenge man bruker tid på å forstå hvorfor det ble feil.

Det å "google problemet" er imidlertid ofte den mest effektive løsningen for konkrete problemer. Sett sammen søkestrenget av ordet "JavaScript" og det du ønsker å gjøre (formulert på engelsk), og du vil så godt som alltid få svar:

"JavaScript format number with two decimals"

"JavaScript show current time"

"JavaScript draw circle"



Oppgaver til denne boka finnes på våre nettsider: <http://it-2.no/?CatID=1209>
Klikk på «Oppgaver» ved siden av forsidebildet og oppgavesamlingen lastes ned.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

2 HTML, CSS og JavaScript

I dette kapitlet vil du lære

- mer om forholdet mellom JavaScript, HTML og CSS
- å manipulere HTML og CSS ved hjelp av programkode
- om hendelser
- å benytte skjemaelementer
- å tegne enkle tegninger via kode

Troikaen

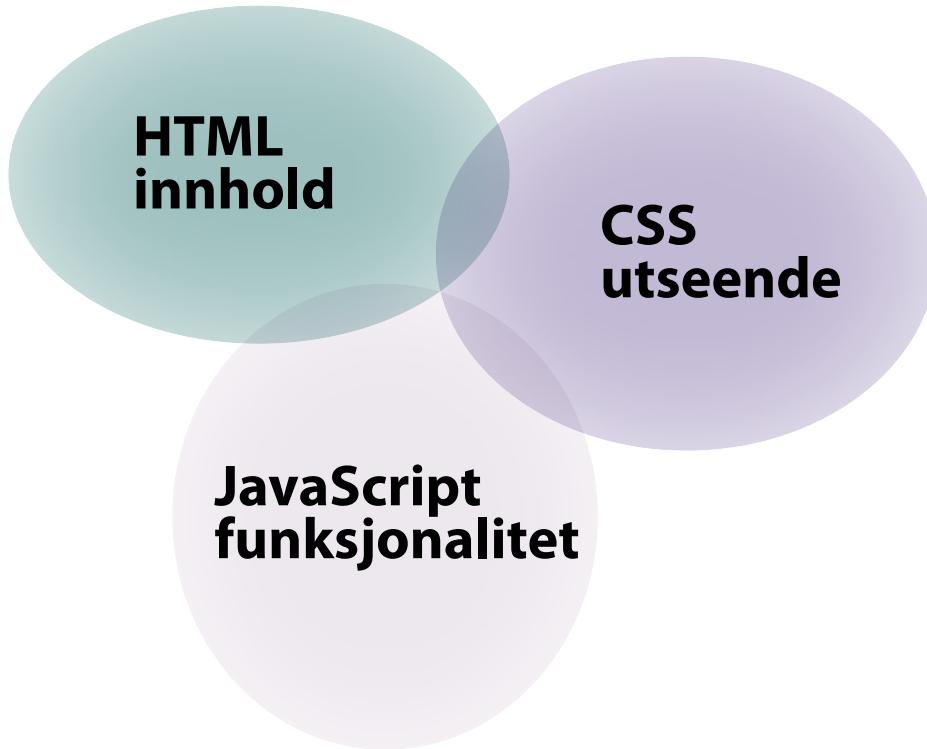
De aller fleste nettsider består av de tre delene *innhold, utseende og funksjonalitet*. Noen nettsider har mye, andre har lite av de ulike delene. For brukeren smelter disse tre delene sømløst sammen og gir en helhetlig nettside. Det er imidlertid svært viktig for en utvikler å skille dem fra hverandre.

Innholdet er teksten og de innholdsrelaterte bildene, og dette er gitt gjennom HTML. Innholdet har en gitt rekkefølge. Det har også en semantikk ved at overskrifter, tabeller, lister osv. er markert ved hjelp av tagger.



Alt av utseende, slik som farger, posisjoner, skyggelegging og bakgrunnsbilder, styres av CSS. Funksjonalitet, slik som utregninger, dataprosessering, dynamisk innhold og interaktivitet, er overlatt til JavaScript.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Enkelte ting i en nettside kan være vanskelig å plassere i en av disse kategoriene. Er f.eks. en logo en del av innholdet eller en del av utseendet? Er en animasjon en del av utseendet eller en del av funksjonaliteten? I disse tilfellene må man gjøre et valg på hva som ligger nærmest.

I tillegg til å plassere elementer i riktig del er det også om å gjøre å få koblingene mellom de tre delene så løse som mulig. Vi ønsker enkelt å kunne gjenbruke utseendet på flere nettsider eller enkelt bytte ut utseendet med et annet.

Tilsvarende ønsker vi å kunne gjenbruke funksjonalitet på mange nettsider eller kunne utvikle nettsider uten å måtte ta hensyn til funksjonaliteten før senere.

Det er ikke gitt at de tre teknologiene som benyttes, må være HTML, CSS og JavaScript. I dette systemet kan f.eks. teknologien for utseende enkelt byttes fra CSS til noe annet. I dag er imidlertid HTML, CSS og JavaScript så godt som enerådende, og få alternativer finnes.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

JavaScript

JavaScript har sine røtter tilbake til nettleseren Netscape i 1995. Det ble opprinnelig kalt *LiveScript*, men byttet snart navn til JavaScript. Navnet er noe uheldig, da det umiddelbart får mange til å tro at det har noe med Java å gjøre, noe det ikke har annet enn også å være et programmeringsspråk. Navnevalget ble trolig gjort for å få en spinoff-effekt på populariteten Java hadde den gang.

JavaScript har først virkelig blomstret som teknologi de senere årene da det stadig har blitt et større krav om funksjonalitet i nettsider. Nettskyapplikasjoner, spill og krav om interaktivitet er pådriverne.

JavaScript er blitt så godt som standard for såkalt *client-side* av nettsiders funksjonalitet. Dette er alt av funksjonalitet i nettsiden som utføres av nettleseren. Motsatsen er *server-side* som da logisk nok foregår på webserveren før vi har mottatt nettsiden. Typisk server-side programmeringsspråk er PHP og Python, og vi vil komme innom dette i forbindelse med *dynamisk innhold* senere i boka.

Ettersom client-side foregår i nettleseren, uten at nettsiden må oppdateres, egner det seg godt til interaksjon fra brukeren, typisk for å lage brukergrensesnitt og enkle nettsidebaserte applikasjoner. Server-side egner seg godt for sikkerherhetskritisk funksjonalitet, datalagring og kommunikasjon med andre systemer.



Benytt aldri client-side-funksjonalitet som sikkerhetsmekanismer. Brukeren kan både se og manipulere koden gjennom sin nettleser.

Grunnskjelettet

I denne boka vil stort sett alle eksempler vi lager, benytte samme oppbygning på JavaScript-koden. Vi skal derfor ta for oss denne litt grundigere nå.

En av utfordringene vi ofte må løse når vi skal programmere med JavaScript, er at programkoden helst skal starte først etter at nettsiden er ferdig lastet. Starter programkoden på et tidligere tidspunkt, kan det være at elementer i nettsiden som koden skal arbeide med, ennå ikke er opprettet.

Vi må derfor forklare i programkoden at selv om den står først (i `<head>`-taggen), så skal den vente med å utføres til nettsiden er ferdig lastet. Dette gjøres ved å lage en blokk med kode (en såkalt *funksjon*) som vi heretter kommer til å kalle **oppstart**. I tillegg må vi si at denne koden skal utføres under `window.onload`, som er et teknisk begrep på at nettsiden er ferdig lastet.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Testside</title>
    <script>

        window.onload = oppstart;

        function oppstart() {
            //Programkode her...
        }

    </script>
    <style>
    </style>
</head>
<body>
    <p>Testside</p>
</body>
</html>
```



Ettersom stort sett alle eksempler i denne boka vil ta utgangspunkt i denne grunnstrukturen, kan det være lurt å lagre den som mal.html og benytte den som en basis videre. Da slipper du skrive inn den samme koden gang på gang. Vi har også lagt inn en `<style>`-blokk i denne malen som vi kommer til å benytte for CSS i de eksemplene der det er påkrevd.

Egentlig burde koden mellom `<script>` og `</script>` hatt et innrykk for å følge et godt oppsett. Vi har imidlertid utelatt dette, da det ville gitt flere linjebrudd på lange kodelinjer utover i boka.

TIPS

Et eksempel på bruk kan være å legge inn programkoden for å lage en meldingsboks inn i funksjonen `oppstart`. Her vises kun `<script>`-taggen, noe vi kommer til å gjøre ofte fremover i boka. Kode du allerede har, er også grået ut:

```
<script>

window.onload = oppstart;

function oppstart() {
    alert("Hei verden");
}

</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Dersom du nå lagrer og åpner fila i en nettleser, vil den vise en meldingsboks med teksten *Hei verden*. Vi kommer til å omtale dette som å teste nettsiden videre i boka.



TIPS

En meldingsboks vil vises forskjellig av de ulike nettleserne. Det er også et element man generelt bør unngå i mer profesjonelle brukergrensesnitt.

På tilsvarende måte som vi her kun viste koden inne i **<script>**-taggen, vil vi i videre eksempler kun presentere **<body>**-taggen for innhold og kun vise **<style>**-taggen for design. Hva som skal være innholdet i **<title>**-taggen, kan du i eksemplene velge selv.

Hendelser

Til nå har vi bare sett kode som utføres idet nettsiden er ferdig lastet. Vi har ikke hatt noen mulighet til å påvirke når koden skal utføres. Den eneste hendelsen vi har knyttet kode til, er med andre ord lasting av nettsiden gjennom **window.onload**.

Vi skal ta for oss en rekke ulike typer hendelser gjennom boka, men her begrenser vi oss til å koble hendelser til *klikk* på ulike elementer. For å få til dette må vi ha noe som er klikkbart på nettsiden. I prinsippet kan alle elementer bli klikkbare.

Dersom vi f.eks. har følgende paragraf mellom **<body>**-taggene

```
<body>
    <p id="klikktekst">Klikk her</p>
</body>
```

kan vi gjøre denne klikkbar gjennom JavaScript-kode. Når siden er ferdig lastet, må vi koble en ny blokk/funksjon med kode mot paragrafens **onclick**-hendelse. Denne koblingen må/bør foregå etter at siden er lastet, altså inne **oppstart**-blokken. Vi skal lage mer profesjonelle knapper senere i boka.

```
<script>
    window.onload = oppstart;

    function oppstart() {
        document.getElementById("klikktekst").onclick = trykket;
    }

    function trykket() {
        alert("Au!");
    }
</script>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Ordinære lenker med `href` satt til # kan benyttes som mer synlig klikkbar objekter.



Eksempel - Vis svar på gåte

I dette eksempelet skal vi lage en enkel nettside som viser en gåte og teksten *klikk her for å se svaret*. Når teksten klikkes på, vises svaret på gåten.

1. Lag et nytt HTML-dokument du kaller *gaate.html* ut fra *mal.html* (som vi lagde tidligere i dette kapitlet).
2. Legg til en paragraf for selve gåten og en paragraf som skal bli klikkbar.

```
<body>
    <p>Hva er svaret på livet, universet og alt mulig?</p>
    <p id="svar">Klikk her for å se svaret</p>
</body>
```

3. Legg til koden som lar paragrafen med ID **svar** bli klikkbar, og som setter innholdet i denne paragrafen til å være *svaret er 42* når den blir klikket på.

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("svar").onclick = visSvar;
}

function visSvar() {
    document.getElementById("svar").innerHTML = "Svaret er 42";
}

</script>
```

4. Test nettsiden, og sjekk at programkoden virker.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Hendelsesparameter

Hver gang vi lager en blokk med kode (funksjon) som skal knyttes til en hendelse, kan vi også plukke ut ekstra informasjon om hendelsen. Dette gjøres ved å inkludere en såkalt parameter i funksjonen. Denne parameteren kan ha et hvilket som helst navn, men det er vanlig å kalle den **evt**.

```
function trykket(evt) {  
    alert("Au!");  
}
```

For hendelser som har med musen å gjøre, vil denne parameteren blant annet ha egenskaper som forteller oss om musens X- og Y-koordinat i nettsiden. Koordinaten X blir målt i antall piksler fra venstre kant, Y i antall piksler fra toppen.

```
function trykket(evt) {  
    alert("Du trykket på koordinaten " + evt.clientX + " , " + evt.clientY);  
}
```

Meldinger slik som "Du trykket på koordinaten 14 , 23" vil nå skrives ut i meldingsboksen. Vi kommer mer tilbake til en grundigere forklaring og mer praktisk bruk av denne *hendelsesparameteren* senere i boka.

Andre typer musehendelser

I tillegg til hendelsen **onclick** som utføres hver gang vi klikker på et element, finnes det også flere andre musehendelser som kan benyttes i JavaScript.

Hendelsen **onmouseover** oppstår idet du flytter musepekeren over elementet, mens **onmouseout** oppstår når musepekeren forlater elementet. Du kan enkelt benytte disse på samme måte som **onclick**.

```
<script>  
  
window.onload = oppstart;  
  
function oppstart() {  
    document.getElementById("klikktekst").onmouseover = over;  
}  
  
function over() {  
    alert("Du traff!");  
}  
  
</script>
```

Hendelsen **onmousemove** oppstår kontinuerlig mens du flytter musepekeren over et element. Å benytte denne kan derfor medføre at hendelsen blir utført mange ganger etter hverandre.

Vi kommer tilbake til praktisk bruk av disse hendelsene, og flere andre, senere i boka.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Manipulere HTML

For å få koblet funksjonaliteten til innholdet i nettsiden er vi avhengig av at JavaScript-koden kan lese og endre HTML-koden vår. Vi så bl.a. et eksempel på dette i forrige kapittel, da vi erstattet en tekst i en paragraf med et klokkeslett.

Den enkleste måten å få tak i elementer fra HTML-dokumentet på i JavaScript er å gi elementene en ID – rett og slett ved å legge til et ID-attributt på taggen vi ønsker å arbeide mot. Dette kan vi gjøre på alle type tagger, slik som bilder, paragrafer og overskrifter:

```

<p id="ingress">Tekst...</p>
<h1 id="sideoverskrift">Overskriften...</h1>
```

Du velger selv hva ID skal være, men naturlig nok kan du ikke ha mer enn ett element på hver nettside med samme ID. Du bør også unngå mellomrom og spesialtegn.



Dersom det ikke finnes noen naturlig tagg å legge ID-en på, kan vi sette inn en **<div>**-tagg for å markere en ny blokk/avsnitt eller en ****-tagg for å markere en del av teksten:

```
<div id="resultat"></div>

<p>Jeg er <span id="alder">25</span> år gammel.</p>
```

For å referere til et element som har en gitt ID, benytter vi følgende kode:

```
document.getElementById("alder")
```

Dette refererer til taggen i seg selv, og vi kan så referere til egenskaper ved taggen ved å legge til punktum og egenskapsnavnet. For eksempel kan vi endre teksten i en tagg ved å gi **innerHTML** en ny verdi:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("alder").innerHTML = "35";
}

</script>
```



Vi kan fjerne teksten i et element ved å sette det lik en tom tekststreng. Altså

```
document.getElementById("alder").innerHTML = "";
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tilsvarende kan vi endre hvilket bilde som vises i en ``-tagg, ved å forandre egenskapen `src`:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("mittBilde").src = "bilde2.jpg";
}

</script>
```

Endre strukturen

Det er ikke bare innholdet i HTML-taggene vi kan manipulere, men også strukturen i seg selv. Vi kan fjerne, flytte eller legge til tagger på gitte steder. Foreløpig skal vi holde oss til å utvide dokumentet med nye elementer.

For å kunne legge til en ny tagg må vi først opprette taggen, og så koble den på en eksisterende tagg vi kan referere til.

Vi oppretter nye tagger ved hjelp av `document.createElement`, slik som dette:

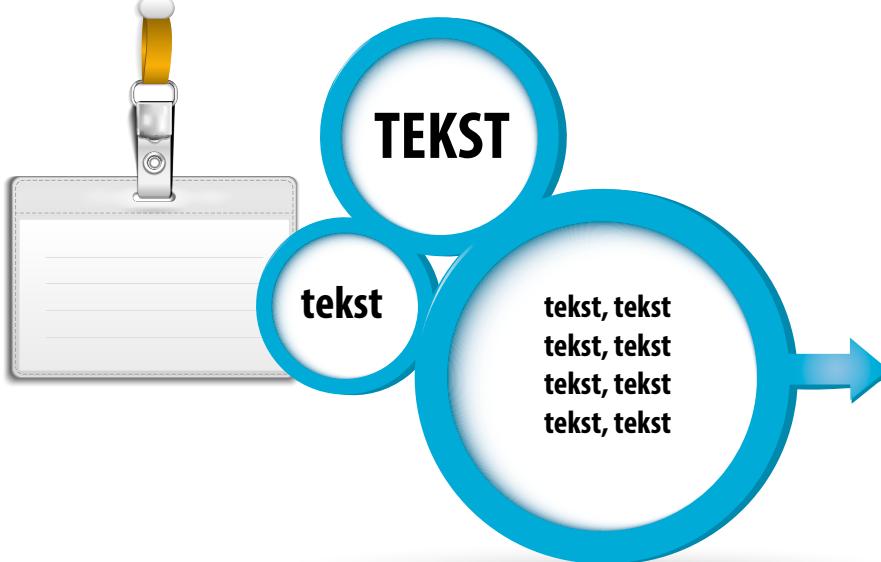
```
var nyttElement = document.createElement("p");
```

Navnet `nyttElement` kan erstattes med andre navn, men vi må gi det nye elementet et temporært navn for å kunne arbeide videre med det. Det neste vi ofte ønsker å gjøre, er å fylle elementet med innhold:

```
nyttElement.innerHTML = "Litt mer tekst...";
```

Deretter må vi koble elementet på en eksisterende tagg. Elementet blir da et barn/underelement av denne taggen. Har vi f.eks. en `<div>`-tagg med ID `tekst`, kan vi koble paragrafen vi nettopp lagde på denne ved hjelp av:

```
document.getElementById("tekst").appendChild(nyttElement);
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Legge til rundetider i liste



I dette eksempelet skal vi lage en liste der vi legger til et nytt tidspunkt hver gang vi trykker på en tekst.

1. Lag et nytt HTML-dokument du kaller *rundetider.html* ut fra *mal.html*.
2. Legg til en paragraf som skal bli klikkbar, en ledetekst og en liste med ID **tider**:

```
<body>
    <h1>Rundetider</h1>
    <p id="leggtil">Legg til ny tid</p>
    <h3>Tider:</h3>
    <ol id="tider">
    </ol>
</body>
```

3. Skriv programkoden som gjør paragrafen klikkbar, og ved klick oppretter nye ****-elementer fylt med gjeldende tidspunkt, og kobler disse på lista med ID **tider**:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("leggtil").onclick = leggtil;
}

function leggtil() {
    var tid = new Date();
    var nyttElement = document.createElement("li");

    nyttElement.innerHTML = tid.getHours() + ":" + 
                           tid.getMinutes() + ":" + tid.getSeconds();

    document.getElementById("tider").appendChild(nyttElement);
}

</script>
```

4. Test nettsiden og kontroller at du får opp nye tidspunkt hver gang du trykker *Legg til ny tid*.



Rundetider

Legg til ny tid

Tider:

1. 15:6:16
2. 15:6:43
3. 15:7:1
4. 15:7:24
5. 15:7:48

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Manipulere CSS

Elementer vi henter ut via `document.getElementById`, har en `style`-egenskap som igjen har underegenskaper med tilsvarende navn som i CSS. Vi kan f.eks. endre fargen på en tekst i en paragraf med ID `ingress` til rød ved hjelp av følgende kode:

```
document.getElementById("ingress").style.color = "red";
```

Her blander vi imidlertid sammen CSS og JavaScript mer enn vi kanskje ønsker. En mer ryddig fremgangsmåte er heller å lage en ny klasse som en egen CSS-beskrivelse. Dette gjøres ved å legge inn følgende kode i `<style>` (eller som et eget eksternt stilark):

```
<style>
    .roedtekst {color:red;}
</style>
```

Deretter settes denne klassen på paragrafen via JavaScript:

```
document.getElementById("ingress").className = "roedtekst";
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Velge farge på tekst



I dette eksempelet skal vi lage en melding, samt en liste med fargenavnene grønn, blå og rød. Når en av fargene trykkes på, skal meldingen presenteres i denne fargen. Det som i bakgrunnen skjer, er at meldingen blir tilordnet en av tre ferdigdefinerte CSS-klasser.

1. Lag et nytt HTML-dokument du kaller *fargeknapper.html* ut fra *mal.html*.
2. Legg til en overskrift som skal bli klikkbar, en ledetekst og en liste med fargene:

```
<body>
    <h1 id="melding">Hei verden</h1>
    <p>Klikk på ønsket farge:</p>
    <ul>
        <li id="groenn">Grønn</li>
        <li id="blaa">Blå</li>
        <li id="roed">Rød</li>
    </ul>
</body>
```

3. Legg til stilene som skal benyttes:

```
<style>
    .groenn {color:green;}
    .blaa {color:blue;}
    .roed {color:red;}
</style>
```

4. Legg til programkoden som lar hvert listeelement bli klikkbart, og som for hvert element gir elementet melding den tilhørende klassen:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("groenn").onclick = visGroenn;
    document.getElementById("blaa").onclick = visBlaa;
    document.getElementById("roed").onclick = visRoed;
}

function visGroenn() {
    document.getElementById("melding").className = "groenn";
}

function visBlaa() {
    document.getElementById("melding").className = "blaa";
}

function visRoed() {
    document.getElementById("melding").className = "roed";
}

</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

5. Test nettsiden og kontroller at overskriften forandrer farge etter hvert som du klikker på de ulike fargenavnene.

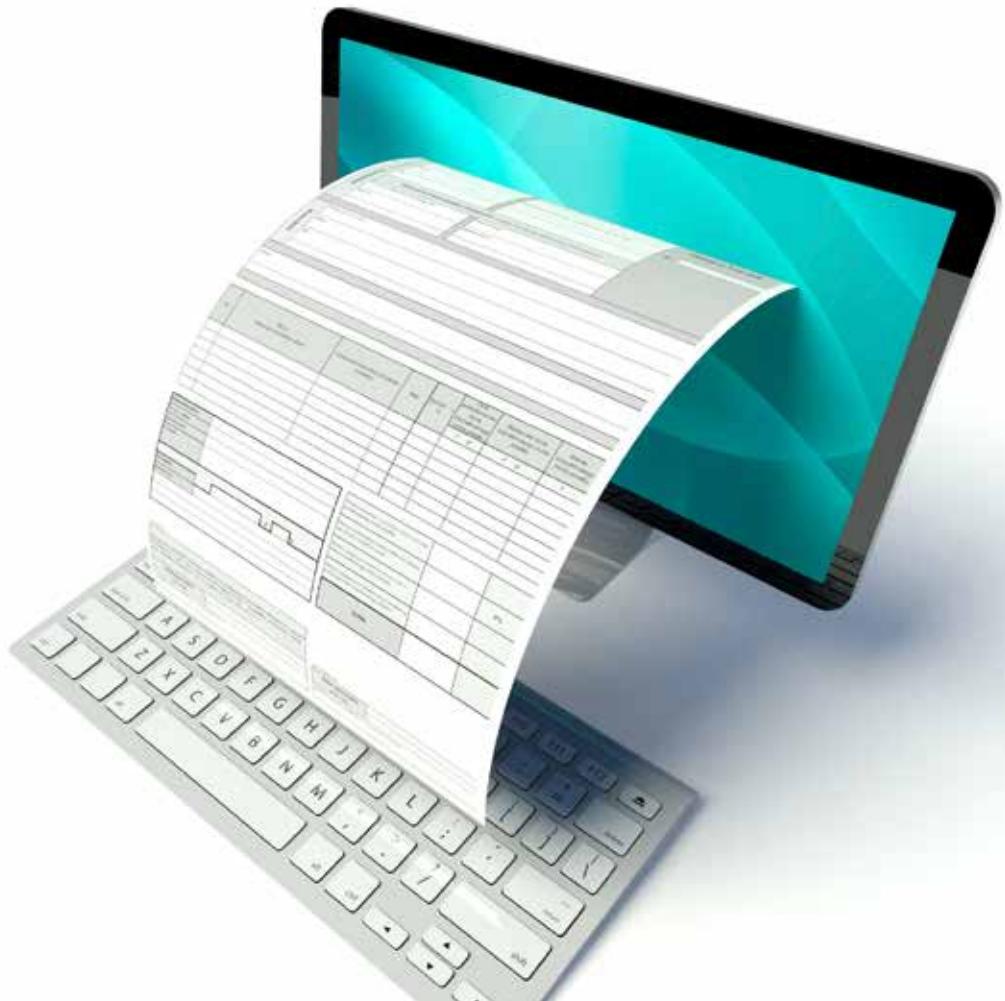
Hei verden

Klikk på ønsket farge:

- Grønn
- Blå
- Rød

6. For å få de tre listeelementene til å følge de samme fargene, kan vi knytte dem opp mot klassene som inneholder fargekodingen:

```
<body>
    <h1 id="melding">Hei verden</h1>
    <p>Klikk på ønsket farge:</p>
    <ul>
        <li id="groenn" class="groenn">Grønn</li>
        <li id="blaa" class="blaa">Blå</li>
        <li id="roed" class="roed">Rød</li>
    </ul>
</body>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Skjemaer

Svært mange bruksområder for programmering vil kreve en annen type input fra brukere enn klikk. Skal vi f.eks. lage en søkerboks, vil det være fornuftig at brukeren kan skrive inn tekst, eller om vi skal lage et påmeldingsskjema, kan det være fornuftig å kunne velge kjønn blant to ferdigdefinerte valg.

Vi vil i denne seksjonen ta for oss noen av de vanligste *skjemaelementene*, egenskaper og hendelser, samt et lite eksempel på bruk av hver av dem.

Navngiving av skjemaelementer

Som du vil legge merke til, bruker vi et såkalt *prefiks* til ID-en på skjemaelementene. For eksempel begynner alle navn på knapper med *btn* og alle tekstbokser med *txt*. Dette er strengt tatt ikke nødvendig å ha med, men vil gjøre koden mer lesbar for nybegynnere. Det er mye lettere å se hvilken type element vi arbeider med, når dette går frem av navnet. I tillegg får vi ofte problemet med at flere elementer i koden og nettsiden egentlig er til for det samme og burde hatt samme navn. Dette løses ved at prefiksene gir dem unike navn, selv om hoveddelen er den samme.

Her er en kort oppsummering av prefiksene som benyttes i denne boka:

Knapp	btn
Tekstboks	txt
Avkrysningsboks	chk
Radioknapper	rbtn
Nedtrekksliste/Lista	lst

Knapper

Tidligere har vi benyttet ordinære HTML-elementer slik som paragrafer og listepunkter som klikkbare elementer. For å synliggjøre at noe kan trykkes på for brukeren, kan det være fornuftig å lage knapper. Dette gjøres ved å benytte følgende HTML-kode:

```
<button id="btnKnapp" type="button">Klikk meg...</button>
```

Klikk meg...

Den aktuelle hendelsen for en knapp vil være **onclick**. Denne fungerer på samme måte som for andre HTML-elementer.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tekstbokser

Tekstbokser/tekstfelt er kanskje et av de mest anvendelige skjemaelementene vi har. I sin enkleste form må vi sette type til å være text, og gi den en ID.

Ditt navn: <input type="text" id="txtNavn" />

Ditt navn:

I JavaScript kan vi så plukke ut teksten som brukeren har skrevet gjennom egenskapen **value**.

`document.getElementById("txtNavn").value`



Eksempel - Velkomstmelding

I dette eksempelet skal vi lage en enkel nettside bestående av en tekstboks og en knapp. Når knappen trykkes på, skal en meldingsboks vises med en hilsen til brukeren.

1. Lag et nytt HTML-dokument du kaller *velkomstmelding.html* ut fra *mal.html*.
2. La nettsiden inneholde en tekstboks der brukeren skriver inn sitt navn, og en knapp:

```
<body>
    Ditt navn: <input type="text" id="txtNavn" />
    <button type="button" id="btnVelkomst">OK</button>
</body>
```

3. Skriv programkoden som gjør knappen trykkbar, og som viser en melding basert på brukerens navn når knappen trykkes på:

```
<script>

window.onload = oppstart;

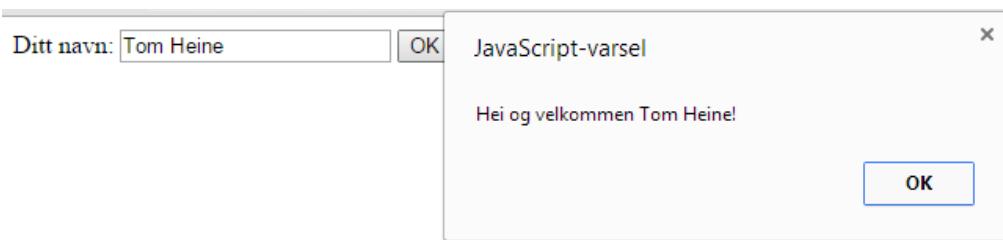
function oppstart() {
    document.getElementById("btnVelkomst").onclick = visHilsen;
}

function visHilsen() {
    alert("Hei og velkommen " + ↵
        document.getElementById("txtNavn").value + "!");
}

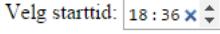
</script>
```

4. Test nettsiden og kontroller at velkomstmeldingen vises når brukeren klikker på OK.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



For å forenkle input til tekstbokser finnes det en rekke ulike input-typer. Fra JavaScript vil alle disse fungere likt, og gi deg en tekst, men utseendet og funksjonaliteten for brukerne vil være forskjellige. De mest aktuelle input-typene vil være:

<code>date</code>		Lar brukeren velge en dato ut fra en kalender, og gjør begrensninger på hvilke verdier som er en gyldig dato.
<code>datetime-local</code>		Fungerer som <code>date</code> , men lar brukeren også sette et tidspunkt.
<code>number</code>		Lar brukeren skrive inn tallverdier. Verdien kan også justeres ved hjelp av såkalte <i>spin-buttons</i> .
<code>password</code>		Kamuflerer input fra brukeren.
<code>range</code>		Lar brukeren velge verdi på en skala. Krever bruk av egenskapene <code>max</code> og <code>min</code> , som forklares om litt. Hva som står i hver ende av skalaen, er ren tekst før og etter skjemaelementet.
<code>time</code>		Lar brukeren angi et tidspunkt

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

TIPS

På mobile enheter vil tastaturet endre seg basert på hvilken input-type vi benytter.

MERK

Eksakt hvordan skjemaelementene ser ut og fungerer, styres av nettleseren. Ikke alle nettlesere støtter alle input-typene og viser da en ordinær tekstboks i stedet.

MERK

Selv om input-boksene begrenser muligheten for feil input, kan de ikke forhindre bevisst manipulering. All slik feilkontroll mot hacking må derfor gjøres på serveren og ikke i klienten.

I tillegg til at vi kan justere egenskapen **type**, kan vi også endre mange andre egenskaper. De viktigste er:

maxlength

Maksimalt antall tegn som tekstboksen skal tillate, f.eks:

```
<input type="text" id="txtBrukernavn" maxlength="8" />
```

max, min og step

For typene **range** og **number** kan vi sette hva største og minste verdi skal være gjennom egenskapene **max** og **min**. Vi kan også sette hvor store steg endringen skal medføre gjennom egenskapen **step**:

```
<input type="range" name="points" min="0" max="100" step="5" />
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

placeholder

Angir en hjelpetekst som skal vises i tekstboksen inntil brukeren skriver inn sin egen tekst. Dette er et alternativ til etiketter som vi skal se på senere.

```
<input type="text" id="txtNavn" placeholder="Navn" /> <br />
<input type="text" id="txtTelefon" placeholder="Telefonnummer" />
```

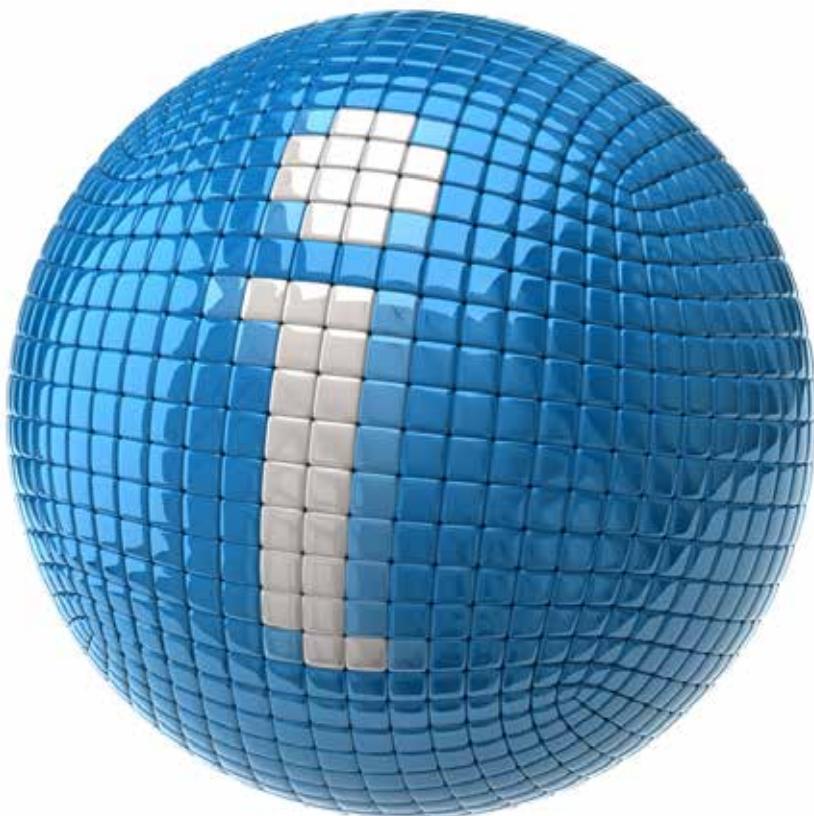
Navn
Telefonnummer

value

Value refererer til innholdet i tekstboksen. Om dette settes gjennom HTML, vil det bli en slags standardverdi som vises ved lasting av nettsiden.

```
<input type="text" id="txtNavn" value="Ole Olsen" />
```

Forskjellen mellom **placeholder** og **value** er at **placeholder** viser en hjelpetekst som forsvinner når brukeren skriver inn sin egen tekst, mens **value** er en faktisk verdi i tekstboksen.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Hendelsen `onchange`

Vi har tidligere sett på ulike typer musehendelser, og hendelsene slik som `mouseover` og `onclick` fungerer også for skjemaelementer. Det er imidlertid mer aktuelt å gjøre noe i det øyeblikket et skjemaelement endrer verdi, fremfor når den blir klikket på.

Derfor kan de fleste skjemaelementene dra nytte av en hendelse som heter `onchange`. Denne utføres nettopp når elementet har fått en ny verdi. For en tekstboks vil hendelsen utføres idet vi flytter fokus ut av tekstboksen, og ikke mens vi skriver.



Eksempel - Tegnteller

Vi skal nå lage en svært enkel versjon av Twitter sin melding som viser hvor mange tegn brukeren har igjen av sine maksimalt 140 tegn. Først skal vi benytte hendelsen `onchange`, deretter en annen hendelse som utføres hver gang vi skriver et tegn, kalt `onkeydown`.

1. Lag et nytt HTML-dokument du kaller `tegteller.html` ut fra `mal.html`.
2. Lag tekstboksen og en paragraf som skal vise meldingen med antall tegn igjen. Selve antallet er omsluttet av en ``, slik at vi kan endre dette. Vi lar også tekstboksen ha en maksgrense på 140 tegn:

```
<body>
    <input type="text" id="txtMelding" maxlength="140" />
    <p>Antall tegn igjen: <span id="antTegn">140</span></p>
</body>
```

3. Legg inn koden som gjør at beregningen utføres hver gang teksten i tekstboksen settes, og som viser dette i ``-feltet `antTegn`:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("txtMelding").onchange = tellTegn;
}

function tellTegn() {
    document.getElementById("antTegn").innerHTML = 140 - ↪
        document.getElementById("txtMelding").value.length;
}

</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

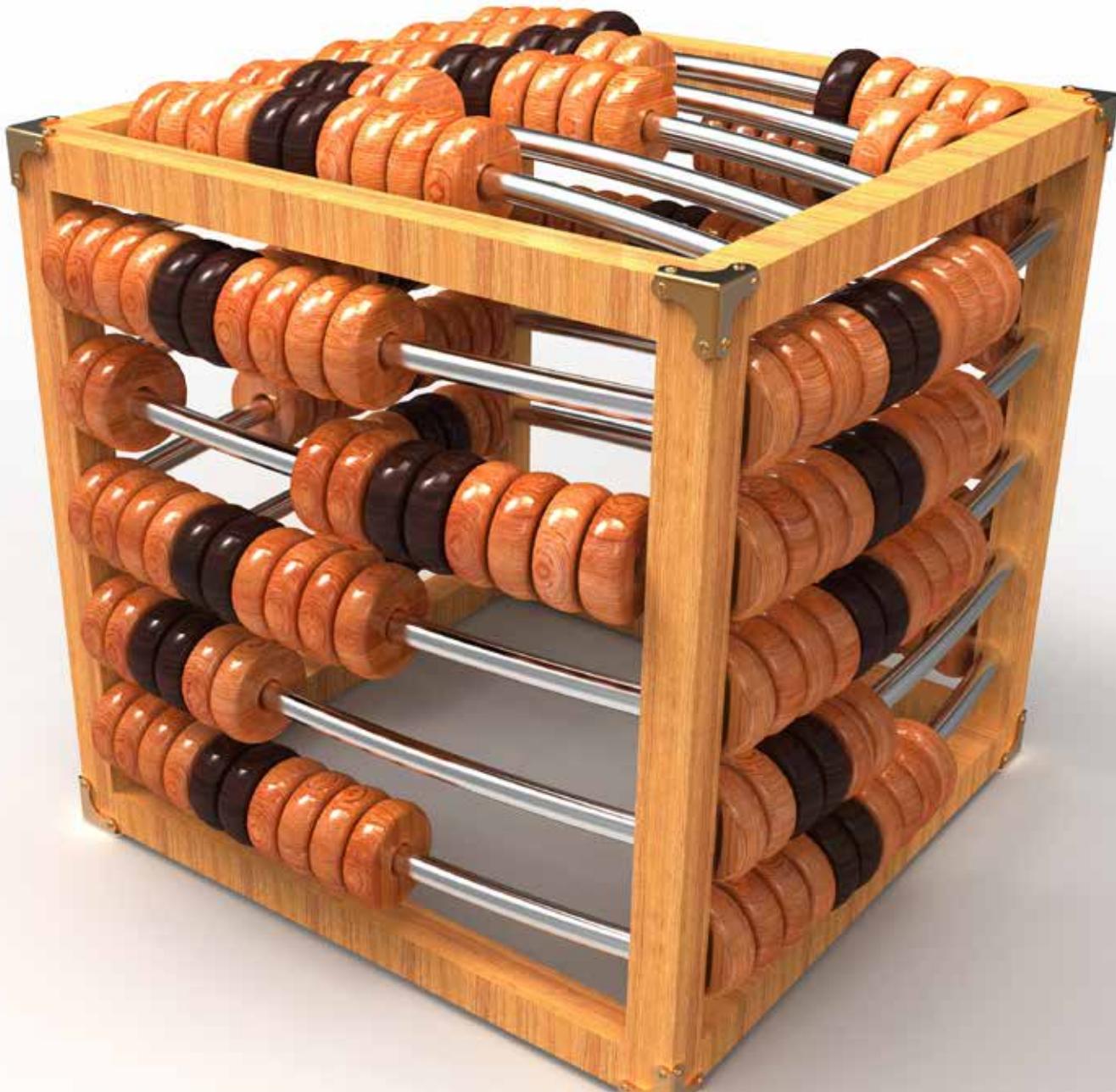
4. Test nettsiden, og kontroller at antall gjenstående tegn blir oppdatert når fokus flyttes ut av tekstboksen (f.eks. ved et klikk på et annet element).
5. Gjør en endring fra hendelsen **onchange** til **onkeydown**, slik at antallet oppdateres mens vi skriver:

```
function oppstart() {  
    document.getElementById("txtMelding").onkeydown = tellTegn;  
}
```

6. Test nettsiden på nytt, og kontroller at antallet endres mens vi skriver.

Hei! Dette er en

Antall tegn igjen: 125



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Avkrysningsboks (Checkbox)

En avkrysningsboks er et skjemaelement som har to tilstander. Enten er den avkrysset, eller så er den ikke det. Disse tilstandene samsvarer med egenskapen **checked** som forteller om den er krysset av eller ikke.

```
<input type="checkbox" name="chkPizza" />Jeg like pizza<br />
<input type="checkbox" name="chkPasta" />Jeg liker pasta
```

- Jeg like pizza
- Jeg liker pasta

Dersom vi ønsker at en avkrysningsboks skal være avkrysset som standard, må vi legge til egenskapen **checked**, som kun har **checked** som mulig verdi.

```
<input type="checkbox" name="chkPasta" checked="checked" />Jeg liker pasta
```

Egenskapen **value** holder på en verdi som ikke er synlig for brukeren, men som vi kan benytte videre i programkoden vår.

```
<input type="checkbox" name="chkPasta" value="Pasta" />Jeg liker pasta
```



Eksempel - Vise melding dersom avkrysset

I dette eksempelet ønsker vi å ha en nettside med en melding i en paragraf. Meldingen skal skjules når nettsiden laster, men når vi endrer om avkrysningsboksen er avkrysset eller ikke, skal meldingen henholdsvis vises eller skjules. Dette eksempelet vil introdusere et nytt programmeringselement kalt *valgsetninger* (if-test) som vi kommer tilbake til i kapittel 4. Du kan foreløpig se om du forstår konseptet med *valgsetninger*, så tar vi detaljene senere.

1. Lag et nytt HTML-dokument du kaller *vismelding.html* ut fra *mal.html*.
2. Legg inn en melding i en paragraf og en avkrysningsboks:

```
<body>
  <p id="melding">Dette er en test</p>
  <p><input type="checkbox" id="chkVisMelding" />Vis melding</p>
</body>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Legg inn programkoden som forteller at under lasting av nettsiden skal meldingen skjules og det skal kobles en hendelse kalt `visMelding` til forandringer i status på avkrysningsboksen. Når denne hendelsen utføres, skal det gjøres et valg basert på om avkrysningsboksen er avkrysset. Er den det, skal meldingen vises, er den ikke det, skal meldingen skjules.:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("melding").style.visibility = 'hidden';
    document.getElementById("chkVisMelding").onchange = visMelding;
}

function visMelding() {
    if (document.getElementById("chkVisMelding").checked === true) {
        document.getElementById("melding").style.visibility = 'visible';
    }
    else {
        document.getElementById("melding").style.visibility = 'hidden';
    }
}

</script>
```

4. Test nettsiden og kontroller at meldingen kun vises når avkrysningsboksen er avkrysset.

Dette er en test

Vis melding



Vi kommer som sagt tilbake til detaljer rundt valgsettninger, men i all enkelhet sjekkes det i `if`-delen av koden om egenskapen `checked` tilsvarer verdien `true` (altså sann). I så fall utføres koden som setter meldingen til synlig.

Dersom ikke denne testen slår til, utføres koden i `else`-delen.

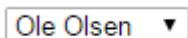
De tre likhetstegnene angir at vi ønsker å sjekke likhet, og ikke endre en verdi slik som ett likhetstegegn gjør.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Nedtrekksliste

En nedtrekksliste benyttes til å vise en liste av valg. Nedtrekkslister bygges opp av et `<select>`-element og flere `<option>`-elementer. Det er mye på samme måte som vi bygger opp en ordinær liste på ved hjelp av `` og ``.

```
<select id="lstKunder">
    <option value="34212">Ole Olsen</option>
    <option value="5514">Per Persen</option>
    <option value="21345">Nils Nilsen</option>
</select>
```



Egenskapen `value` på hvert `<option>`-element inneholder en verdi som brukeren ikke ser, men som vi kan benytte videre i programkoden vår for det aktuelle valget. Den valgte verdien kan i JavaScript plukkes ut fra nedtrekkslisten som egenskapen `value` på selve boksen.



Eksempel - Finn språk

I dette eksempelet skal vi lage en nettside der brukeren får spørsmål om hvilket land han/hun er fra. Når dette blir valgt, vises en melding med språket det er antatt at brukeren snakker.

1. Lag et nytt HTML-dokument du kaller `finnspraak.html` ut fra `mal.html`.
2. Lag nedtrekkslisten der tilhørende språk er satt som `value` på hvert element. Inkluder også en tom paragraf der meldingen vil vises:

```
<body>
    <p>Hvilket land er du fra?</p>
    <select id="lstLand">
        <option value="norsk">Norge</option>
        <option value="svensk">Sverige</option>
        <option value="dansk">Danmark</option>
        <option value="finsk">Finland</option>
    </select>
    <p id="melding"></p>
</body>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Skriv programkoden som kobler en hendelse til endring i nedtrekkslisten, og som plukker ut egenskapen **value** fra nedtrekkslisten og viser denne i paragrafen:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("lstLand").onchange = visSpraak;
}

function visSpraak() {
    document.getElementById("melding").innerHTML = "Da snakker " + document.getElementById("lstLand").value;
}

</script>
```

4. Test nettsiden, og kontroller at tilhørende språk vises.

Hvilket land er du fra?

Danmark ▾

Da snakker du mest sannsynlig dansk

Nedtrekkslister viser som standard kun ett element av gangen, unntatt under selve valgprosessen. Vi kan påvirke dette ved å endre egenskapen **size**. Endres denne til noe annet enn 1, vises det i stedet en såkalt listebox, der flere elementer er synlige av gangen.

```
<select id="lstKunder" size="4">
    <option value="34212">Ole Olsen</option>
    <option value="5514">Per Persen</option>
    <option value="21345">Nils Nilsen</option>
</select>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Radioknapper

En radioknapp ligner mye på en avkrysningsboks som vi så på tidligere, men med radioknapper er det kun ett av elementene i en gruppe av radioknapper som kan være valgt av gangen. Velger vi en, vil med andre ord markøren forsvinne i en annen.

```
<input type="radio" name="kjonn" id="rbtnKjonnMann" value="mann" />Mann  
<input type="radio" name="kjonn" id="rbtnKjonnKvinne" value="kvinne" />Kvinne
```

Ettersom en nettside kan bestå av flere ulike grupper radioknapper, er det viktig å inkludere egenskapen **name** og sette denne til det samme for de radioknappene som hører sammen, slik at de danner en gruppe. Dessverre finnes det ingen veldig rett-frem-metode for å plukke ut verdien til valgte element i en gruppe, uten å lære litt mer om JavaScript enn det vi har gjort til nå.

Egenskapen **value** fungerer som for checkboxer, ved at vi kan plukke ut en bakerforliggende verdi for et element.



Selv om det virker mest naturlig å benytte **onchange** for å detektere når en radioknapp blir markert eller markert bort, er det imidlertid slik at **onclick** fungerer best til dette i de fleste nettlesere. Hendelsen **onclick** vil også fungere når man markerer boksen ved hjelp av tastaturet.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Eksempel - Bildefremviser

I dette eksempelet skal vi lage en svært enkel bildefremviser. Nettsiden skal bestå av en ``-tagg og tre radioknapper, en for hvert bilde vi ønsker å vise. Når brukeren klikker på en radioknapp, skal ``-taggens `src`-egenskap settes til verdien som ligger lagret i egenskapen `value` på valgte radioknapp.

1. Lag et nytt HTML-dokument du kaller `bildefremviser.html` ut fra `mal.html`.
2. Legg tre bilder i samme mappe som HTML-fila. For enkelhets skyld kaller vi dem `bilde1.jpg`, `bilde2.jpg` og `bilde3.jpg`.
3. Lag en ``-tagg med ID `bilde`, og en radioknapp for hvert av bildene:

```
<body>
    
    <p><input type="radio" name="bildeliste" id="rbtnBilde1" value="bilde1.jpg" />Bilde 1</p>
    <p><input type="radio" name="bildeliste" id="rbtnBilde2" value="bilde2.jpg" />Bilde 2</p>
    <p><input type="radio" name="bildeliste" id="rbtnBilde3" value="bilde3.jpg" />Bilde 3</p>
</body>
```

4. Lag en `onclick`-hendelse til hver av radioknappene, som setter bildets `src`-egenskap til valgte radioknapps filnavn:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("rbtnBilde1").onclick = visBilde1;
    document.getElementById("rbtnBilde2").onclick = visBilde2;
    document.getElementById("rbtnBilde3").onclick = visBilde3;
}

function visBilde1() {
    document.getElementById("bilde").src = document.getElementById("rbtnBilde1").value;
}

function visBilde2() {
    document.getElementById("bilde").src = document.getElementById("rbtnBilde2").value;
}

function visBilde3() {
    document.getElementById("bilde").src = document.getElementById("rbtnBilde3").value;
}

</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

5. Test nettsiden, og kontroller at du får byttet bilde som vises ved hjelp av radioknappene. Når nettsiden lastes første gang, vises det ikke noe bilde.



Bilde 1

Bilde 2

Bilde 3

6. Vi kan få det første bildet til å vises automatisk, samtidig som første radioknapp hukes av, ved å legge til følgende JavaScript-kode under lastingen av nettsiden:

```
function oppstart() {  
    document.getElementById("rbtnBilde1").onclick = visBilde1;  
    document.getElementById("rbtnBilde2").onclick = visBilde2;  
    document.getElementById("rbtnBilde3").onclick = visBilde3;  
  
    document.getElementById("bilde").src =   
    document.getElementById("rbtnBilde1").value;  
    document.getElementById("rbtnBilde1").checked = true;  
}
```

Programkoden i dette eksempelet kan lages mye mer ryddig og elegant. Vi skal se at vi kan lage enklere kode som også lett håndterer å utvide bildegalleriet senere i boka, vi har bare ikke lært alt som skal til ennå.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Canvas

En av endringene i HTML5 var innføringen av taggen `<canvas>`. Denne taggen markerer et område i nettsiden som kan tegnes på ved hjelp av JavaScript. Uten JavaScript er `<canvas>` kun et hvitt område på nettsiden.

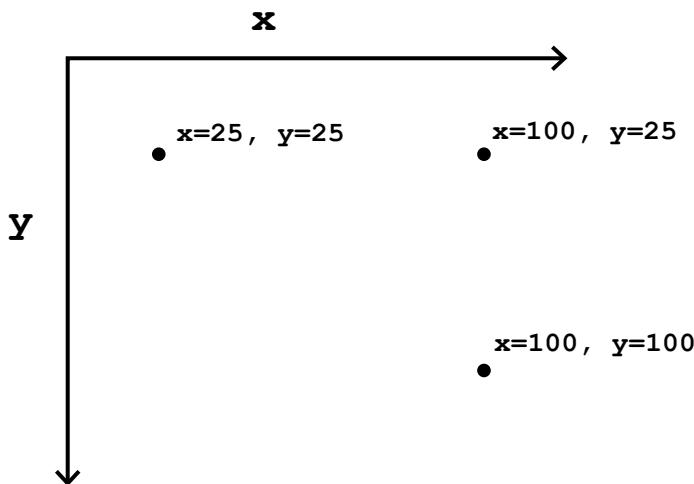
I HTML-koden setter vi inn taggen med ønsket vidde og bredde:

```
<body>
  <canvas id="tegneflate" width="200" height="200"></canvas>
</body>
```

Deretter må vi i JavaScript hente ut dette elementet og sette en såkalt *context*, der vi velger om vi vil tegne i 2D eller 3D. Vi skal begrense oss til kun 2D-tegning. Denne contexten gis ofte navnet `ctx`.

```
var ctx = document.getElementById("tegneflate").getContext("2d");
```

Deretter kan vi begynne å tegne på canvasen ved hjelp av kommandoer der vi angir koordinater. Det er viktig å være klar over at alle koordinater angis fra øverste venstre hjørne. En økning i *x* vil derfor bety at vi går mot høyre, en økning i *y* vil bety at vi går nedover mot bunnen.



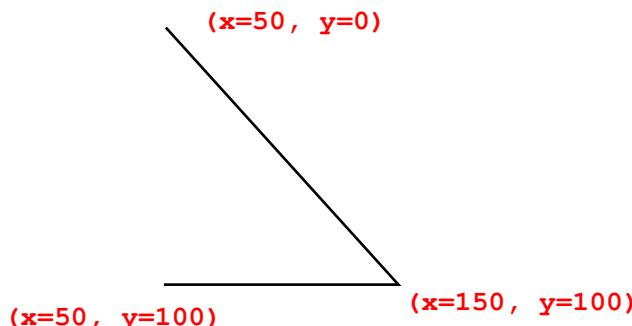
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tegne linjer

For å tegne en linje må vi først flytte oss til posisjonen der vi ønsker linjen skal starte ved hjelp av kommandoene `beginPath` og `moveTo`. Deretter flytter vi ”pennen” til neste punkt ved hjelp av kommandoen `lineTo`. Disse kan vi ha en eller flere av, ettersom om vi ønsker en linje bestående av flere segmenter eller ikke. Så slutføres tegningen ved hjelp av kommandoen `stroke`.

```
ctx.beginPath();
ctx.moveTo(50, 0);
ctx.lineTo(150, 100);
ctx.lineTo(50, 100);
ctx.stroke();
```

I dette tilfellet flytter vi pennen til punktet $x=50, y=0$, deretter tegner vi en strek til $x=150, y=100$ og en ny strek til $x=50, y=100$. Dette gir oss følgende figur:



Fargen på linja kan justeres gjennom egenskapen `strokeStyle`. Denne tar som input de samme typene fargekoder og fargenavn som CSS. Tykkelsen kan settes gjennom `lineWidth` som tar antall piksler som verdi.

```
ctx.strokeStyle = "red";
ctx.lineWidth = 10;
```



Endringer på vidde og farge må gjøres før linja tegnes.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tegne rektangler

For å tegne et rektangel benytter vi kommandoen `rect`. Denne kommandoen forventer verdiene x og y, som angir øverste høyre hjørne, i tillegg til en bredde og en høyde.

```
ctx.beginPath();
ctx.rect(10, 10, 150, 100);
ctx.stroke();
```

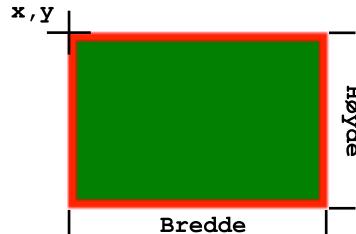
Her settes x og y til 0, bredden til 150 og høyden til 100. Som standard tegnes rektangelet med sort kantlinje. Linjefargen kan vi endre ved hjelp av `strokeStyle` som vi tidligere har omtalt.

For å tegne et fylt rektangel bruker vi kommandoen `fill` i stedet for `stroke`. Vi må da også angi en fyllfarge ved hjelp av egenskapen `fillStyle` dersom vi ønsker noe annet enn sort.

```
ctx.fillStyle = "green";
```

Kommandoene kan også kombineres for å tegne et fylt rektangel med kantlinje. Pass da på å tegne fyllet først.

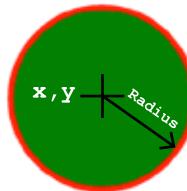
```
ctx.beginPath();
ctx.rect(10, 10, 150, 100);
ctx.fillStyle = "green";
ctx.fill();
ctx.lineWidth = 5;
ctx.strokeStyle = "red";
ctx.stroke();
```



Tegne en sirkel

Tegning av sirkler gjøres ved hjelp av kommandoen `arc`. Denne angir senterpunkt, radius og start-/sluttpunkt på buen. Ettersom vi ønsker å tegne en hel sirkel, setter vi startpunktet til 0 radianer og sluttpunktet til 2π radianer. Foruten dette fungerer `arc` på samme måte som `rec` når det gjelder `fill` og `stroke`.

```
ctx.beginPath();
ctx.arc(100, 110, 80, 0, 2 * Math.PI);
ctx.fillStyle = "green";
ctx.fill();
ctx.lineWidth = 5;
ctx.strokeStyle = "red";
ctx.stroke();
```



For mange er nok radianer en ukjent benevning. Radianer er en alternativ måte å måle vinkler på, og 360 grader er det samme som 2π radianer, altså "helt rundt" i en sirkel.

TIPS

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Eksempel - Sirkeltegner

Vi skal nå lage en nettside med et `<canvas>`-element. Når brukeren beveger musepekeren over canvaset, skal det tegnes en sirkel på koordinatene til musepekeren. Ettersom vi får ut koordinatene til musepekeren angitt fra øverste venstre hjørne på nettsiden, og ikke i forhold til `<canvas>`-elementet, velger vi å flytte canvaset til posisjon 0,0 for å unngå for mange utregninger.

1. Lag et nytt HTML-dokument du kaller `sirkeltegner.html` ut fra `mal.html`.
2. Legg inn en canvas med ID `tegneflate`:

```
<body>
    <canvas id="tegneflate" width="500" height="500"></canvas>
</body>
```

3. Flytt canvaset til øverst i venstre hjørne. Legg også til en kantlinje slik at vi ser tegneområdet:

```
<style>
    #tegneflate {
        position: absolute;
        top: 0;
        left: 0;
        border-style: solid;
    }
</style>
```

4. Legg inn programkoden som gjør at en funksjon vi kaller `flyttMus`, utføres hver gang musepekeren flyttes over canvaset. Denne funksjonen skal tegne en sirkel med diameter 80 piksler og senterpunkt i koordinatene til musen:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("tegneflate").onmousemove = flyttMus;
}

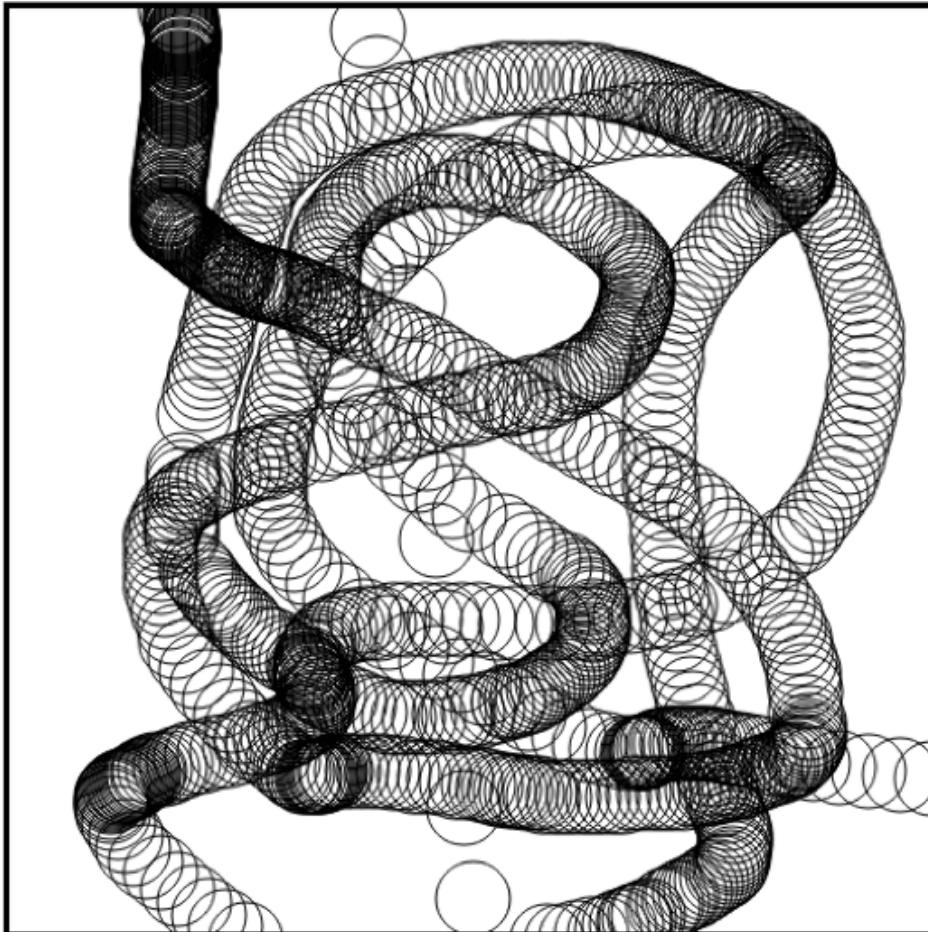
function flyttMus(evt) {
    var ctx = document.getElementById("tegneflate").getContext("2d");

    ctx.beginPath();
    ctx.arc(evt.clientX, evt.clientY, 20, 0, 2 * Math.PI);
    ctx.stroke();
}

</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

5. Test nettsiden, og forsøk tegne ved å flytte musepekeren rundt på canvas-området i ulikt tempo.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

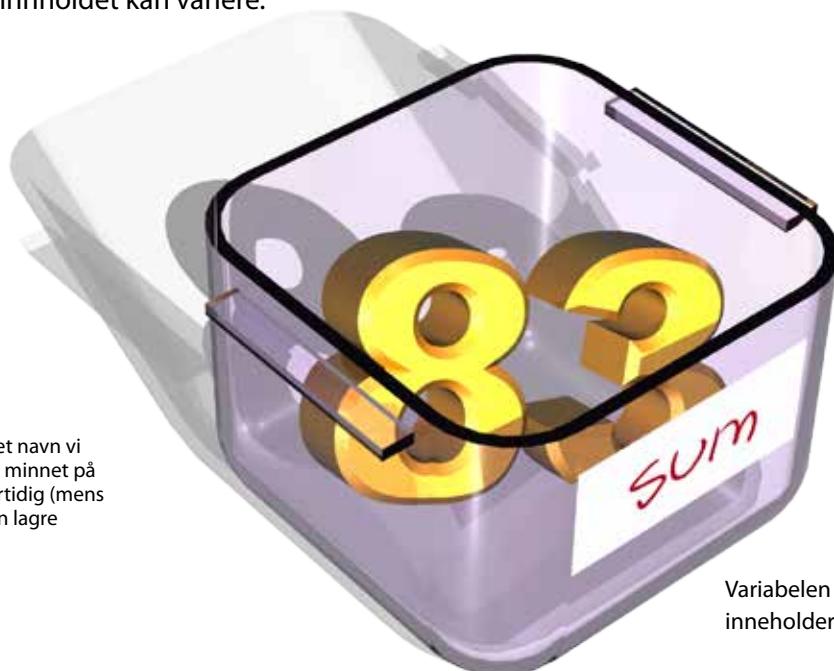
3 Variabler og operatorer

I dette kapitlet vil du lære

- hva en variabel er
- å hente og sette verdier i en variabel
- om ulike type data
- om operatorer
- å konvertere mellom datatyper

Det finnes utallige måter å forklare hva en variabel er. La oss imidlertid forsøke med en enkel forklaring, og si at en variabel rett og slett er et navn vi gir på et bestemt sted i minnet på maskinen. Her kan vi midlertidig (mens programmet/nettsiden kjører) lagre verdier fra for eksempel mellomregninger eller input fra brukeren.

I praksis er systemet for å håndtere variabler ganske mye mer komplisert, men vi kan foreløpig tenke på minnet i maskinen som navngitte beholdere på en hylle. Hver beholder kan inneholde informasjon. Beholderen er den samme hele tiden, men innholdet kan variere.



Egentlig er variablene et navn vi gir på et bestemt sted i minnet på maskinen, der vi midlertidig (mens programmet kjører) kan lagre verdier.

Variablen **sum** inneholder her talet 83.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Med programkode kan vi da for eksempel si at vi ønsker å sette av plass i minnet til å lagre et tall, og gi denne plassen navnet **sum**. Dette kalles å **deklarere** en variabel.

```
var sum;
```



Ettersom vi ikke har gitt variabelen noen verdi ennå, kan vi tenke oss variabelen som en tom beholder.

For å gi variabelen en verdi (innhold) kan vi i tillegg skrive:

```
var sum;  
sum = 4;
```



Her setter vi verdien til å være tallet 4 etter at variabelen **sum** er deklarert. Første gang vi setter en verdi i en variabel, sier vi at vi **initialiserer** variabelen.



Vi bruker tegnet = for å tilordne verdier til variablene. Vi leser dette som *settes lik* i stedet for *er lik*. Merk deg at variabelen alltid må stå på venstre side og verdien på høyre side av likhetstegnet.

Det er vanlig både å deklarere og initialisere en variabel samtidig:

```
var sum = 4;
```

Et program kan inneholde så mange variabler som vi måtte ønske. La oss si at vi ønsker å bruke variabler for å summere to tall. Vi kan da skrive:

```
var tall1 = 4;  
var tall2 = 3;  
var sum = tall1 + tall2;  
alert(sum); //Viser verdien 7 i en meldingsboks
```



Her settes verdien av **tall1** til 4 og **tall2** til 3. Variabelen **sum** settes lik summen av **tall1** og **tall2** som er 7.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Hvorfor bruke variabler?

Det er mange grunner til at vi bruker variabler:

- Holde rede på verdier som forandrer seg, for eksempel antall poeng i et spill eller input fra brukeren
- Hvis den samme verdien skal brukes flere steder i programmet, bruker vi en variabel i stedet for å skrive selve verdien. Hvis vi ønsker å forandre verdien, trenger vi bare forandre ett sted. Vi sier vi unngår *hardkoding*. Hardkoding betyr å skrive tall rett inn i koden uten å bruke variabler.
- Vi kan bruke variabler i mellomregninger slik at koden blir mer oversiktlig.
- Variablene vil også gjøre koden mer lettles. For eksempel gjør variabelen **sum** det tydelig at vi beregner summen av noe.

Variabelnavn



For systemets del står vi ganske fritt til å velge navn på variabler. Det finnes allikevel visse begrensninger i at variabelnavn ikke kan inneholde mellomrom og punktum, ei heller andre tegn enn tall, bokstaver, understrek og dollartegn.

I tillegg må variabelnavnet starte med en bokstav, en understrek eller et dollartegn. Det vil være mulig å benytte andre bokstaver/tegn enn bare de engelske (a-z), men dette frarådes, da det ofte skaper problemer.



Det finnes noen reserverte nøkkelord, slik som *function*, *if*, *while*, *for* osv., som ikke kan benyttes som variabelnavn alene. Du vil gjenkjenne slike nøkkelord ved at de som oftest blir markert med en annen farge/stil av editorens syntax highlighting.

I tillegg til begrensningene i selve programmeringsspråket er det også vanlig å skrive variabelnavn på en bestemt måte for å gjøre programmet mer lesbart. Dette kalles en kodekonvensjon, eller altså et sett med regler for hvordan koden bør skrives for å være mest mulig lesbar, og for at alle skal skrive kode på mer eller mindre samme måte.

Ifølge den vanligste kodekonvensjonen for JavaScript skal alle variabelnavn skrives på *camelCase*-formatet. Det vil si at variabelnavnet er skrevet med små bokstaver (*lowercase*). Dersom variabelnavnet skulle være satt sammen av flere ord, skrives imidlertid første bokstav i hvert påfølgende ord med stor forbokstav (*uppercase*). Eksempler på variabelnavn i *camelCase*-formatet er:

```
var antall = 34;
var antallTimer = 23;
var antallTimerPerUke = 15;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Det er også lurt å venne seg til å gi variablene gode og beskrivende navn basert på hva verdien de inneholder, skal brukes til. Det er ofte fristende når man programmerer, å gi variabler navn som *temp3*, *i*, *p1* og tall for å spare tid. Dessverre medfører dette ofte at man glemmer hva variablene egentlig har for funksjon og innhold, samtidig som koden blir nær sagt ulesbar. Det er dermed bedre heller å bruke litt lengre tid på å skrive gode variabelnavn enn å bruke lang tid på å finne feil i programmet siden.

Variabelnavnene skiller også mellom store og små bokstaver (*case sensitive*). Variabelen *dato* er for eksempel en annen variabel enn *Dato*.



Forandre verdien til variabler

Når du har tilordnet en verdi til en variabel, har du full mulighet til å forandre verdien i etterkant. For eksempel:

```
var tall = 4;  
alert(tall); //Viser tallet 4  
tall = 3;  
alert(tall); //Viser tallet 3
```

Vær oppmerksom på at du ikke skal lage den samme variabelen på nytt. I JavaScript vil denne koden bli godtatt, men det er ikke noen god programmeringsskikk.

```
var tall = 4;  
var tall = 3; // Deklarerer variabelen på nytt
```

I mange tilfeller ønsker vi å forandre en tallverdi ved å bruke den nåværende verdien til å beregne en ny verdi. For eksempel:

```
var tall = 5;  
tall = tall + 1;  
alert(tall); //Viser tallet 6
```

Her setter vi først verdien til 5. Deretter setter vi **tall** lik seg selv, altså 5, og plussr på 1. Resultatet blir da 6.

Variabelen må imidlertid ha en startverdi før vi kan sette den lik seg selv. Hvis ikke, blir resultatet ugyldig:

```
var tall;  
tall = tall + 1; //Feil!  
alert(tall);
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Typer data

Variabler kan inneholde data av forskjellig type, for eksempel tall eller tekst. I JavaScript finnes det tre typer data av det som kalles *primitive* datatyper. Dette er *string*, *number* og *boolean*. I tillegg finnes det to spesialverdier som er *null* (ingen verdi) og *undefined* (udefinert). Det er også en kompleks datatype som kalles *object*, og som i prinsipp er ”alt annet”. En versjon av denne kalles arrayer og er en liste med verdier.

Det er viktig å ha god oversikt over hvilken type data vi arbeider med, da ulike typer oppfører seg og behandles ulikt. Vi skal derfor ta en kort gjennomgang av de primitive datatypene her.

String

Verdier av typen *string* er det vi på godt norsk kaller en tekst eller tekststreg. For at systemet skal forstå hva som er en tekst og hva som er programkode, må all tekst omsluttet av anførselstegn. En deklarasjon og initialisering av en variabel med en string-verdi vil derfor se slik ut:

```
var minHilsen = "Hei på deg";
```

I en string er det visse tegn som ikke uten videre er tillatt å benytte. Det mest naturlige av disse er tegnet `",`, ettersom anførselstegn benyttes for å markere hva som er starten og slutten på selve tekststrengen.

Vi kan imidlertid sette inn en såkalt *escaped* versjon av tegnet, som skrives `\"`, for å angi at vi ikke ønsker programmeringsbetydningen, men tegnets tekstlige betydning.

```
var tekst = "Dette tegnet \" , er et anførselstegn.";
```

Ettersom tegnet `\` nå brukes til å escape andre spesialtegn, får vi et problem når vi skal bruke dette tegnet i teksten. Dermed må tegnet escape seg selv, og vi setter inn `\\"` for hver gang vi vil ha en `\` i teksten.

```
var filsti = "C:\\temp\\\\minmappe";
```

Vi har også enkelte systemtegn som er vanskelige å skrive som vanlige tegn. Dette gjelder for eksempel tabulator og linjeskift. Disse tegnene skrives derfor som `\t` for tabulator og `\n` for linjeskift.

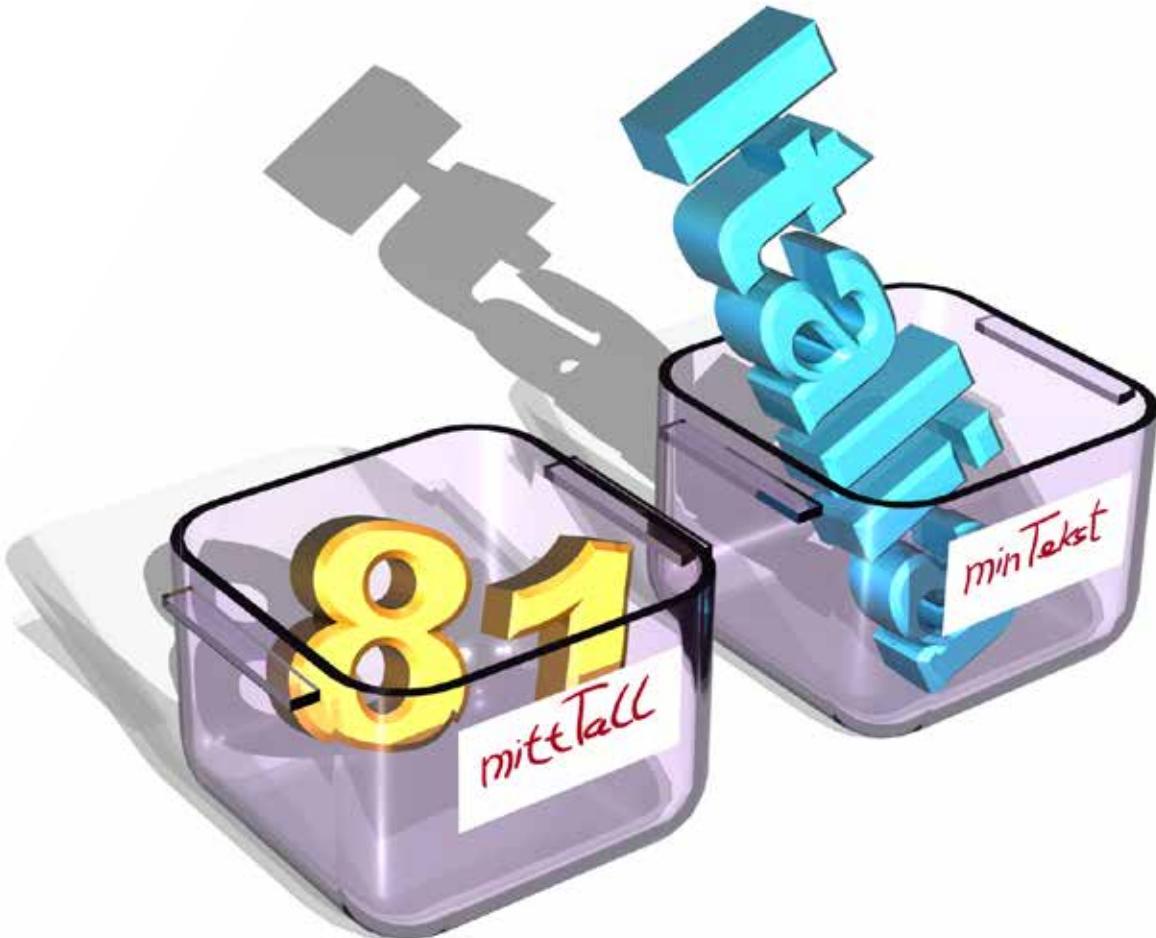
```
var tekst = "Linje nr 1\n\tLinje nr 2 med innrykk.;"
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

En oversikt over de vanligste såkalte escape characters finner du i følgende tabell:

Escape character	Betydning
\n	Ny linje
\'	Enkelt anførselstegn
\"	Anførselstegn
\\"	Backslash
\t	Tabulator

Spesialtegnene \n og \t har en effekt i situasjoner slik som en meldingsboks eller ved skriving til en fil fra programkoden. Dersom teksten skal vises som en del av en nettside, vil ikke disse to tegnene ha noen effekt. Da benyttes
 for nye linjer og CSS-formatering for å få inntrykk.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Number

En verdi av typen *number* er en tallverdi. Dette kan både være en heltallsverdi (*integer*) og et desimaltall (*flyttall / float*). Eksempel på deklarasjon og initialisering av en tallverdi vil være:

```
var antallAnsatte = 45;  
  
var temperaturUte = -34;  
  
var stortTall = 3475894375892648690214367863263871648.0037467  
  
var breddePaaEtBlyatomIMilliMeter = 1 / (7.1 * 1000000000000000.0);
```



Det er punktum og ikke komma som angir desimalskillet i et desimaltall, ettersom så godt som alle programmeringsspråk er basert på det amerikanske tallsystemet.

Boolean

Datatypen *boolean* er den enkleste datatypen, ettersom den kun har de to tillatte verdiene **true** og **false**. Dette er på norsk hva vi ville kalt en *boolsk verdi*. Verdien **true** tilsvarer det norske *sann*, og **false** tilsvarer *usann*.

Det virker kanskje litt underlig å ha en egen datatype som kun baserer seg på to slike *logiske verdier*, men den er faktisk veldig flittig brukt i programmering, noe du vil få se etter hvert. Typiske situasjoner vil være når man ønsker å angi om en *betingelse* er oppfylt eller ikke, slik som en avkrysningsboks for et ja/nei-spørsmål.

Eksempel på deklarasjon og initialisering av en variabel med en boolsk verdi vil være:

```
var erMann = true;  
var harBetalt = false;
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tekst og tall

I programmering kan man angi tallverdier både som en tekst og som et tall. I dagliglivet ellers har vi ikke noe spesielt forhold til dette skillet.

```
var alder1 = "45";  
var alder2 = 45;
```

Angir vi det som en string, slik som i variabelen **alder1**, vil systemet anse det for å være tegnene 4 og 5. Angir vi det som et tall, slik som i variabelen **alder2**, vil systemet anse det som verdien 45.

I de fleste tilfeller vil *konvertering* mellom slike *datatyper* foregå nærmest "automatisk", slik at vi kan regne med tekst og la tall bli en del av teksten på en nettside. Du bør allikevel ha et bevisst forhold til om det er et tall du skal regne med, eller en tekst du angir. Kundenummer, telefonnummer og bankkontonummer er typiske verdier som bør defineres som en tekst. Alder, overskudd, timelønn og pris er typiske verdier som bør defineres som et tall.

For å avgjøre om en verdi skal angis som et tall eller en tekst, kan du stille deg spørsmålet: *Vil det noen gang kunne være aktuelt å dele verdien på to?* Er svaret ja, er det mest sannsynlig et tall som er riktig; er svaret nei, er det en tekst.

TIPS

En forskjell vi allerede nå kan se på, er om tallet starter med 0-er, noe typisk kundenummer og telefonnummer kan gjøre.

```
var kundenummer1 = "003242";  
var kundenummer2 = 003242  
  
alert(kundenummer1); // Viser 003242  
alert(kundenummer2); // Viser 3242
```

Den første meldingsboksen som presenterer verdien i **kundenummer1**, vil vise 003242, mens den andre meldingsboks vil vise 3242.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Virkningsområde

En ting som kompliserer bruk av variabler, er at plasseringen av variablene deklarasjon avgjør hvor og hvor lenge den er gyldig. Dersom vi deklarerer variabelen inne i en funksjon, vil variabelen kun være tilgjengelig inne i denne funksjonen og bli en såkalt *lokal variabel*. Dersom du forsøker å referere til variabelen fra en annen funksjon enn der den er definert, vil du få en feilmelding om at variabelen ikke er definert.

```
<script>

window.onload = oppstart;

function oppstart() {
    var test = 5;
    document.getElementById("btnUtfør").onclick = skrivUt;
}

function skrivUt() {
    alert(test); //Feilmelding
}

</script>
```

For å få flere funksjoner til å dele variabelen må vi deklarere variabelen utenfor funksjonene, som en såkalt *global variabel*.

```
<script>

var test = 0;
window.onload = oppstart;

function oppstart() {
    test = 5;
    document.getElementById("btnUtfør").onclick = skrivUt;
}

function skrivUt() {
    alert(test); // Viser 5
}

</script>
```

Et annet skille mellom de lokale og de globale variablene er at en lokal variabel oppstår når funksjonskoden starter, og forsvinner når funksjonskoden er ferdig. Skal en variabel beholde sin verdi fra gang til gang som funksjonen utføres, må den defineres som global.



I god kode bør vi unngå globale variabler, og heller sende inn verdier i funksjoner som parametre, samt få verdier tilbake som retur-verdier. Vi skal se nærmere på dette i kapitelet om funksjoner.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Klikkteller



Vi skal her lage en svært enkel nettside bestående av en knapp og en paragraf som viser antall ganger knappen er klikket på.

1. Lag et nytt HTML-dokument du kaller *klikkteller.html* ut fra *mal.html*.
2. Legg inn knappen med ID **btnKlikk** og et ****-element med ID **antKlikk**:

```
<body>
    <button id="btnKlikk">Klikk meg!</button>
    <p>Knappen er trykket <span id="antKlikk">0</span> ganger</p>
</body>
```

3. Skriv programkoden som definerer en global variabel for antall og knytter en hendelse kalt **klikket** til knappens **onclick**. I denne hendelsesfunksjonen skal variabelen **antall** bli én mer enn den var, samt at det nye antallet skal vises i ****-elementet:

```
<script>

window.onload = oppstart;

var antall = 0;

function oppstart() {
    document.getElementById("btnKlikk").onclick = klikket;
}

function klikket() {
    antall = antall + 1;
    document.getElementById("antKlikk").innerHTML = antall;
}

</script>
```

4. Test nettsiden, og kontroller at paragrafen viser antall ganger du har klikket på knappen.

Klikk meg!

Knappen er trykket 31 ganger



Legg ekstra merke til at variabelen **antall** må deklarereres globalt (altså utenfor funksjonene). Det er fordi vi ønsker at den skal beholde verdien fra gang til gang som hendelsen **klikket** utføres. Hadde den blitt definert inne i hendelsen, vil den blitt satt til 0 hver gang.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Operatorer

For å jobbe med variabler og verdier finnes det innebygd i JavaScript en rekke *operatorer* som utfører handlinger på disse. Du har allerede benyttet flere av dem i eksemplene uten å vite om det.

Tilordningsoperatoren

Jobben til tilordningsoperatoren (*likhetstegnet*) er å ta verdien på høyre side og legge denne i variabelen/egenskapen på venstre side.



Det er viktig å ikke blande sammen tilordningsoperatoren med *likhetsoperatoren* som vi kjenner fra matematikken, selv om den i matematikken uttrykkes med samme tegn. Som vi skal se i neste kapittel, er likhetsoperatoren i programmering uttrykt med ===.

Vi har allerede sett en mengde eksempler på bruk av tilordningsoperatoren, men merk deg spesielt i følgende eksempel at en variabel på høyre side av tilordningsvariablene oppfører seg som en verdi.

```
var tallA = 5;  
var tallB = tallA;
```

Eksempelet over medfører at **tallA** får verdien 5 og **tallB** deretter får verdien til **tallA**, eller altså 5.



Tilordningsoperatoren tar alltid verdien på høyre side og legger i variablene på venstre side. Det er altså ikke mulig å ha en verdi på venstre side som skal legges i en variabel på høyre side.

Samme variabel kan, som vi har sett tidligere, også bli brukt både på venstre og høyre side av tilordningsoperatoren.

```
var tallA = 5;  
tallA = tallA + 2;
```

Det vi her sier, er at vi først setter **tallA** til 5. Deretter vil vi at **tallA** skal få en ny verdi, og denne nye verdien er den gamle verdien til **tallA** økt med 2. Dermed vil **tallA** ende opp med verdien 7 etter at disse to instruksjonene er utført.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Matematiske operatorer

I programmeringsspråk finner vi selvfølgelig også igjen operatorer vi kjenner fra matematikken, slik som *addisjon* (pluss), *subtraksjon* (minus), *multiplikasjon* (gange) og *divisjon* (dele). Disse operatorene krever en verdi på begge sider og vil erstatte hele uttrykket med den nye verdien.

Eksempler på bruk av de *matematiske operatorene* vil være som følger:

```
var tallA = 5 + 1;  
var tallB = tallA - 2;  
var tallC = 10 * 6;  
var tallD = tallC / tallA;  
var tallE = 26 / 4;
```

I eksemplene over vil **tallA** bli 6, **tallB** bli 4, **tallC** bli 60, **tallD** bli 10 og **tallE** bli 6.5.

De unære operatorene

Vi har også en gruppe operatorer som kalles de *unære* operatorene. Navnet er nok mer fancy enn det selve operatorene er, ettersom de ganske enkelt styrer fortegn på et tall.

Vi kan skrive følgende for å gange den negative verdien av 5 med 7:

```
var svar = -5 * 7;
```

Det er også mulig å angi positive fortegn ved hjelp av en pluss, men det er ikke vanlig i bruk, da det er gitt at det er en positiv verdi om fortegn mangler.

TIPS



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Alternative operatorer for tilordning mellom samme variabel

I programmering støter man på operasjoner som benyttes ofte. Ettersom programmerere nærmest av natur ønsker den enkleste måten å gjøre ting på, har det blitt laget hurtigere skrivemåter for de vanligste operasjonene.

Dersom man for eksempel vil øke verdien til variablene `tall` med 5, må man skrive det slik:

```
tall = tall + 5;
```

Altså sier vi at maskinen skal ta verdien til `tall` og legge til 5. Deretter skal variablene `tall` settes til denne nye verdien.

For å forkorte slike operasjoner, der en variabel gjentas på begge sider av tilordningsoperatoren, har man laget operatoren `+=`. Med denne operatoren kan instruksjonen heller skrives som følger:

```
tall += 5;
```

Det samme blir fortsatt utført, men skrevet på en enklere/raskere måte. Tilsvarende finnes det operatorer for subtraksjon, multiplikasjon og divisjon på samme variabel:

```
tall -= 7; // tilsvarer tall = tall - 7;
tall *= 2; // tilsvarer tall = tall * 2;
tall /= 2; // tilsvarer tall = tall / 2;
```

Inkrementerings- og dekrementeringsoperatorer

I programmering kommer man overraskende ofte over tilfeller der man ønsker å øke eller minke en tallvariabel med verdien 1. Vi kunne, som vi har sett tidligere, skrevet dette slik:

```
tall1 = tall1 + 1;
tall2 = tall2 - 1;
```

Bruker vi operatorene fra forrige avsnitt, kunne vi også ha skrevet det på denne måten:

```
tall1 += 1;
tall2 -= 1;
```



Men late/effektive som programmerere er, har man funnet på en enda raskere måte å gjøre dette på gjennom operatorene for *inkrementering* (`++`) og *dekrementering* (`--`):

```
tall1++;
tall2--;
```

Det kan som sagt virke litt pussig å ha operatorer som er spesialtilpasset øking og minking med verdien 1. Vi skal imidlertid snart se at disse operasjonene er veldig hyppig forekommende.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Presedens

Noen operatorer utføres før andre. Dette kalles *presedens*. For eksempel blir multiplikasjon utført før addisjon. Hvis vi vil overstyre dette, kan vi bruke parenteser. Det som står inne i parenteser, blir alltid utført før det som står utenfor.

```
var svar1 = 2 + 3 * 4; //Her får vi verdien 14  
var svar2 = (2 + 3) * 4; //Her får vi verdien 20
```

Modulo-operatoren

En operator det ikke er så vanlig å omtale i matematikkurs, men som er svært mye brukt i programmering, er den såkalte *modulo-operatoren*. Modulo-operatoren representeres i JavaScript med tegnet %, men har ingenting med prosenter å gjøre.

Det modulo-operatoren gjør, er å finne ut hvor mye det hadde blitt til overs dersom vi hadde utført en *heltallsdivisjon*. Modulo omtales derfor også ofte som *rest*. For å utføre heltallsdivisjon gjør vi en vanlig divisjon, men kutter av desimalene ved hjelp av **Math.floor**.

Et forklarende eksempel kan være følgende:

```
var epler = 47;  
var personer = 3;  
var eplerPrPers = Math.floor(epler / personer);  
var tilOvers = epler % personer;  
  
alert("Dersom man er " + personer + " personer, og har " + epler + ↪  
    " epler, blir det " + eplerPrPers + " epler per person, og " + ↪  
    tilOvers + " epler til overs");
```

Her ser vi at vi bruker modulo-operatoren til å finne ut hvor mange epler som blir igjen når vi deler et antall epler rettferdig (heltallsdivisjon) mellom et antall personer. Eksemplet over vil dermed produsere utskriften *Dersom man er 3 personer og har 47 epler, blir det 15 epler per person og 2 epler til overs*.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Oversikt over matematiske operatorer

Nedenfor følger en liten liste der vi oppsummerer de matematiske operatorene:

Operator	Betydning
+	addisjon
-	subtraksjon
*	multiplikasjon
/	divisjon
=	sette lik
++	øke med én
--	minsk med én
+=	sette ny verdi lik gjeldende verdi + en verdi
-=	sette ny verdi lik gjeldende verdi – en verdi
*=	sette ny verdi lik gjeldende verdi * en verdi
/=	sette ny verdi lik gjeldende verdi / en verdi
%	modulo - gir restverdien av en divisjon, for eksempel $10 \% 8 = 2$



Eksempel - BMI-kalkulator

Vi skal i dette eksempelet lage en enkel kalkulator for å beregne BMI (Body Mass Index). Formelen for å beregne BMI er følgende, der vekt angis i kg og høyde i meter:

$$\text{BMI} = \text{vekt} / \text{høyde}^2$$

Dette kan også skrives som:

$$\text{BMI} = \text{vekt} / (\text{høyde} * \text{høyde})$$

1. Lag et nytt HTML-dokument du kaller `bmi.html` ut fra `mal.html`.
2. Lag to tekstfelt med ID `txtVekt` og `txtHoyde`. Legg også inn en knapp med ID `btnBeregn` og en paragraf med ID `resultat`:

```
<body>
    <p>Vekt: <input type="text" id="txtVekt" /><br />
    Høyde: <input type="text" id="txtHoyde" /></p>
    <button id="btnBeregn">Beregn</button>
    <p id="resultat"></p>
</body>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Legg inn programkoden som kobler en funksjon vi kaller **beregn**, til knappens klikk-hendelse. Denne skal plukke ut verdiene fra tekstboksene, beregne BMI og så vise dette i paragrafen **resultat**:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("btnBeregn").onclick = beregn;
}

function beregn() {
    var hoyde = document.getElementById("txtHoyde").value;
    var vekt = document.getElementById("txtVekt").value;

    var bmi = vekt / (hoyde * hoyde);

    document.getElementById("resultat").innerHTML = "Din BMI er: " + bmi;
}

</script>
```

4. Test nettsiden og kontroller at utregningen fungerer. Husk at du må benytte punktum som desimalskille.

Vekt:

Høyde:

Din BMI er: 25.381468541909282



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

5. For også å få komma som et mulig desimalskille kan vi erstatte alle forekomster av komma med punktum idet teksten overføres til variablene. Det samme kan vi gjøre ved å erstatte punktum med komma i utskriften av BMI:

```
function beregn() {  
    var hoyde = document.getElementById("txtHoyde").value.replace(".", ".");  
    var vekt = document.getElementById("txtVekt").value.replace(".", ".");  
  
    var bmi = vekt / (hoyde * hoyde);  
  
    document.getElementById("resultat").innerHTML = "Din BMI er: " + bmi.toString().replace(".", ",");  
}
```

6. Vi kan også avrunde antall desimaler i utskriften til kun ett ved hjelp av kommandoen **toFixed**:

```
document.getElementById("resultat").innerHTML = "Din BMI er: " + bmi.toFixed(1).replace(".", ",");
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Konkateneringsoperatoren

De forrige avsnittene omhandlet operatorer som jobbet med tall. For tekst finnes det kun én operator som er aktuell å omtale på dette tidspunktet, og det er **konkateneringsoperatoren**. Igjen er kanskje navnet stiligere enn selve funksjonaliteten, men operatoren er i hvert fall ofte i bruk og viktig å kunne.

Jobben til konkateneringsoperatoren er å danne en ny tekststreng ut fra flere andre tekststrenger. La oss starte med å se på et eksempel:

```
var fornavn = "Ola";
var etternavn = "Normann";
var fulltNavn = fornavn + " " + etternavn;
```

Som vi ser her, definerer og initialiserer vi variablene **fornavn** og **etternavn**.

Deretter konkatenerer vi sammen innholdet i variablen **fornavn** med en string som inneholder et mellomrom, og konkatenerer dette igjen sammen med innholdet i variablen **etternavn**. Tekststrengen vi nå har fått, lagrer vi i en ny variabel som kalles **fulltNavn**.

Selv om konkateneringsoperatoren er uttrykt ved tegnet +, er dette en helt annen operator enn addisjonsoperatoren vi så på tidligere. De deler kun samme tegn for operatoren. Hvilken operator det er snakk om, ser systemet ut fra datatypene på hver side.



En snedig funksjon ved konkateneringsoperatoren i JavaScript, er at den også automatisk kan konvertere verdier fra tallverdier til string-verdier.

```
var alder = 28;
var tekst = "Min alder er " + alder;
```

Eksempelet over vil først sette variablen **alder** til tallverdien 28. Deretter lages en ny variabel med navn **tekst**, som får som innhold teksten *Min alder er* sammensatt med verdien i variabelen **alder**.

Dette skulle egentlig ikke fungere ettersom variablen **alder** er et tall og ikke en string. Konkateneringsoperatoren er imidlertid så snill at den automatisk konverterer tallverdien 28 (som ble hentet ut fra variabelen **alder**) til teksten 28 før den setter sammen delene.

Ved bruk av konkateneringsoperatoren må minst én av verdiene være tekst. Dersom begge verdiene er et tall, vil + tolkes som addisjon. Dermed får vi heller summen av verdiene som resultat.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tilordning mellom samme variabel

På tilsvarende måte som for tall kan vi sette en tekstvariabel lik seg selv + en tekst, for eksempel:

```
var handleliste = "";
handleliste = handleliste + "Brød,";
handleliste = handleliste + "Melk,";
handleliste = handleliste + "Kaffe";
```

På denne måten er det mulig å samle opp mer og mer tekst i en variabel, for eksempel hver gang brukeren trykker på en knapp. Vi kunne også ha skrevet:

```
var handleliste = "";
handleliste += "Brød,";
handleliste += "Melk,";
handleliste += "Kaffe";
```



Eksempel - Historiegenerator

I dette eksempelet skal vi lage en historiegenerator. Historien skal bli generert ved å hente verdier fra brukeren gjennom skjemaelementer på nettsiden, og så sette disse verdiene sammen med ulike ferdige tekststrenger for å få en komplett historie. Hensikten med eksempelet er å repetere hvordan man henter ut verdier fra skjemaelementer, samt å se hvordan man kan bruke disse verdiene, som da har ulike typer data, til å bygge opp en større tekststrengh.

1. Lag et nytt HTML-dokument du kaller *historiegenerator.html* ut fra *mal.html*.
2. Legg til skjemaelementer for navn, kjønn og alder. Disse skal ha ID **txtNavn**, **lstKjonn** og **txtAlder**:

```
<body>
  <p>Navn: <input type="text" id="txtNavn" /><br />
  Kjønn: <select id="lstKjonn"><option value="gutt">Gutt <input checked="" type="radio"/>
    </option><option value=" jente">Jente</option></select><br />
  Alder: <input type="number" min="0" max="120" id="txtAlder" /><br />
  <button id="btnGenerer" type="button">Generer historie</button></p>
  <p id="historie"></p>
</body>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Skriv kode som henter ut verdiene fra skjemaelementene, syr de sammen til en tekststreng og så viser denne i paragrafen:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("btnGenererer").onclick = beregn;
}

function beregn() {
    var navn = document.getElementById("txtNavn").value;
    var kjonn = document.getElementById("lstKjonn").value;
    var alder = document.getElementById("txtAlder").value

    var aarTill100 = 100 - alder;

    var historie = "Det var en gang en " + kjonn + " som het " + navn + " og var " + alder + " år gammel.<br />";
    historie = historie + "Denne personen hadde " + aarTill100 + " år igjen til 100-årsdagen.';

    document.getElementById("historie").innerHTML = historie;
}

</script>
```

4. Test nettsiden og kontroller at du får generert historier ut fra valgene du gjør.

Navn:

Kjønn:

Alder:

Det var en gang en gutt som het Tom og var 34 år gammel.
Denne personen hadde 66 år igjen til 100-årsdagen.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Konvertere mellom datatyper

Selv om datatyper ikke er så tydelig i JavaScript, er det som tidligere nevnt svært viktig å ha et bevisst forhold til dem. Ofte kommer imidlertid data i en annen type enn det vi ønsker. Leser vi f.eks. innholdet i en tekstboks, vil dette alltid være tekst. Skal vi regne med dataene, er det ofte lurt bevisst å konvertere dem fra en tekst til et tall først, i stedet for å satse på at den automatiske konverteringen fungerer som vi ønsker.

Slik konvertering gjøres med `parseInt` for heltall og `parseFloat` for desimaltall.

```
var alder = parseInt(document.getElementById("txtAlder").value);  
var nyAlder = alder + 100;
```

Hadde vi ikke benyttet `parseInt` i eksempelet, ville `nyAlder` blitt alderen med tegnene 100 bak, f.eks. 23100 om alderen var oppgitt til å være 23.

Ønsker vi å gå fra en annen datatype til en string, kan vi benytte `toString`, som alle verdier inneholder.

```
var alder = 5;  
var alderTekst = alder.toString() + " år";
```

Denne konverteringen gjøres imidlertid automatisk under konkatenering.

Avrunde tall

Hvis vi ønsker å avrunde et kommatall, kan vi konvertere det til en string med et visst antall desimaler. For å gjøre dette bruker vi `toFixed`. Antall desimaler angir vi inne i parentesen:

```
var tall = 2 / 3;  
var avrundetTall = tall.toFixed(2);  
alert(avrundetTall); //Viser teksten 0.66
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Konstanter

Konstanter er egentlig helt vanlige variabler med en spesiell egenskap ved at de ikke kan endre verdi etter at de er initialisert. De brukes dermed for å angi verdier som alltid skal være det samme, slik som for eksempel sekunder i en time eller dager i en uke.

Tenk deg at du lager et program som bruker antall dager i uken, dvs. 7, i ulike beregninger. Hvis du uforvarende angir verdien til noe annet i en beregning, vil resultatet bli feil. Hvis vi i stedet lager en konstant for antall dager i uken og bruker denne i beregningene, vil vi være sikre på at vi bruker riktig tall. På denne måten unngår vi feil i programmet.

En konstant defineres ved å skrive nøkkelordet **const** i stedet for **var** foran en vanlig variabeldeklarasjon. Vanligvis navngir man konstanter med store bokstaver og skiller eventuelle ord med understrek.

```
const DAGER_I_UKEN = 7;
```

Ettersom variablene DAGER_I_UKEN nå er definert som en konstant, er det ikke mulig å endre verdi ved å skrive DAGER_I_UKEN = 8 senere. Man kan altså benytte verdien i konstanten så mange ganger man måtte ønske, men ikke endre den.

Konstanter er forholdsvis nytt i JavaScript. Derfor vil det være en del nettleseere som ikke kjenner igjen konseptet eller behandler det som en variabel. Du kan også benytte en ordinær variabel, angitt med store bokstaver i navnet, for å selv "imitere" en konstant, men mister da kontrollen på om den endrer verdi.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Objektvariabler

I programmering skiller vi ofte på variabler med en *primitiv* datatype og variabler med en *kompleks* datatype. Variabler med en primitiv datatype holder på en verdi, mens de med en kompleks datatype holder på en *referanse* til et objekt. Derfor kalles også variablene med en kompleks datatype for *objektvariabler*.

Oppretter vi f.eks. en variabel med en tallverdi, kan vi kopiere verdien over i en annen variabel. Endres en av variablene etter det, er det kun verdien i den aktuelle variablene som endres:

```
var tall = 5;
var tall2 = tall;
tall = 7;
alert(tall2); // gir 5
```

Kopierer vi innholdet i en variabel som holder på et objekt, vil vi derimot kun kopiere referansen (som kan anses å være "verdien"). Endrer vi på objektet denne referansen peker på gjennom en av variablene, vil også objektet i den andre variablene endres. Det er jo tross alt samme objekt.

```
var paragraf = document.getElementById("utskrift");
var paragraf2 = paragraf;
paragraf.innerHTML = "Hei";
alert(paragraf2.innerHTML); // gir "hei"
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

4 Valgsetninger

I dette kapitlet vil du lære

- hva en kontrollstruktur er
- hvordan man setter opp valgsetninger
- om ulike former for valgsetninger
- om logiske uttrykk
- om relasjonsoperatorer og logiske operatorer

Hittil i denne boka har vi i all hovedsak sett på programkode som utføres *sekvensielt* (om vi ser bort fra lytterne/hendelsene). Med sekvensielt mener man at kjøringen av koden starter på første instruksjon og deretter går instruksjon for instruksjon nedover til vi når slutten. Dette og neste kapittel skal ta for seg såkalte *kontrollstrukturer*, som kan benyttes for å påvirke hvordan instruksjonene blir utført.

Vi skal i denne boka se på to ulike typer kontrollstrukturer, nemlig *valgsetninger* som utfører kode dersom visse betingelser stemmer, og løkker som utfører kode så lenge visse betingelser stemmer.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

For de små kodeeksemplene i dette kapitlet vil vi benytte et enkelt brukergrensesnitt med en tekstboks med ID **txtVerdi**, en knapp med ID **btnUtfør** og en paragraf med ID **utskrift**. All programkode i eksemplene kommer til å bli skrevet i en hendelsesfunksjon kalt **utfør** som er koblet til knappens **onclick**-hendelse.

```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title>Testside</title>
    <script>

        window.onload = oppstart;
        function oppstart() {
            document.getElementById("btnUtfør").onclick = utfør;
        }

        function utfør() {
            //Programkoden til eksemplene
        }

    </script>
<style>
</style>
</head>
<body>
    <p><input type="text" id="txtVerdi" /><br />
    <button id="btnUtfør">Utfør</button></p>
    <p id="utskrift"></p>
</body>
</html>
```

Grensesnittet blir seende slik ut (paragrafen synes ikke da den ikke har noe tekst):

A screenshot of a web browser window. At the top, there is a small red rectangular button with the text "Oppstart". Below it is a text input field with the ID "txtVerdi". Underneath the input field is a button with the ID "btnUtfør" containing the text "Utfør". To the right of the button is a blank paragraph element with the ID "utskrift".

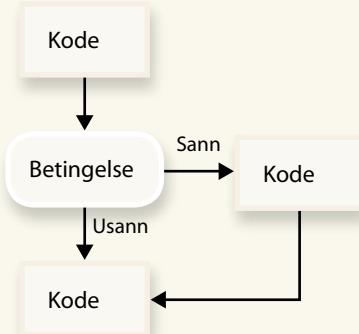
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Valgsetninger

Dersom vi ønsker at en instruksjon eller en gruppe instruksjoner kun skal utføres dersom visse *betingelser* er oppfylt, kan vi benytte en *valgsetning*. I programmering omtales dette også ofte som en *if-test*.

En valgsetning omslutter programkoden som kun skal utføres dersom en betingelse stemmer. Konseptuelt ser strukturen slik ut:

```
Kode før
if (betingelse) {
    Kode som skal utføres kun om betingelsen stemmer
}
Kode etter
```



En betingelse er da et *logisk uttrykk* som enten gir **true** (sann) eller **false** (usann) som verdi. Betingelsen vil bli evaluert av systemet, og programkoden inne i selve valgsetningen vil som sagt kun bli utført dersom betingelsen evaluerer til **true** på gitte tidspunkt i kodeutførelsen. Programkode som måtte stå før og etter valgsetningen, derimot, vil kjøre som normalt.

Med andre ord er det to mulige veier videre når man kommer til en valgsetning. Dersom betingelsen evaluerer til **false**, går vi rett til eventuell programkode etter valgsetningen. Skulle betingelsen derimot evaluere til **true**, utfører vi først programkoden som hører til valgsetningen, før vi på samme måte som ved **false** fortsetter med instruksjonene etter valgsetningen.

Legg merke til krøllparentesene som omslutter alt som skal høre til valgsetningen. Denne programkoden kan bestå av en eller flere instruksjoner og kalles en **kodeblokk**.

Grunnen til at vi må samle instruksjonene i slike kodeblokker, er at en valgsetning egentlig kun er knyttet til neste instruksjon. En kodeblokk blir imidlertid regnet som én instruksjon, og dermed kan valgsetningen i praksis benyttes til å utføre flere instruksjoner samtidig.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

TIPS

Man kan unnlate krøllparentesene rundt programkoden til en valgsetning dersom den kun består av én instruksjon. Det kan imidlertid være en god vane alltid å benytte dem for å øke programkodens lesbarhet. I tillegg vil du da få kun én fremgangsmåte å forholde deg til.

La oss si at vi ønsker at teksten i paragrafen utskrift skal forandres til *Du er myndig* hvis vi har skrevet en verdi i tekstboksen som er større eller lik 18. Da kan vi skrive følgende programkode i hendelsesfunksjonen **utfører**:

```
function utfører() {  
    var alder = document.getElementById("txtVerdi").value;  
    document.getElementById("utskrift").innerHTML = "";  
    if (alder >= 18) {  
        document.getElementById("utskrift").innerHTML = "Du er myndig";  
    }  
}
```

Kjører du koden, vil du se at dersom du skriver inn tall som er mindre enn 18 i tekstboksen, forblir utskriften blank. I alle andre tilfeller slår derimot valgsetningen til, og teksten vil endres til *Du er myndig*. Vi har altså skrevet kode som utføres kun dersom en betingelse er oppfylt.

MERK

Det skal ikke være noe semikolon etter noen av kontrollstrukturene, da de ikke er instruksjoner i seg selv, men snarere en mekanisme som styrer utførelsen av instruksjoner.

Vi vil senere i dette kapitlet gi en grundigere gjennomgang av de ulike operatorene som benyttes for å lage testene/betingelsene til valgsetninger.

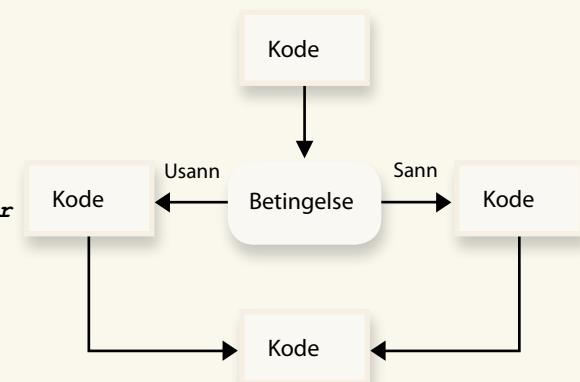
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Valgsetninger med et alternativt utfall

Ofte ønsker vi også å ha et alternativt utfall, som utføres dersom ikke betingelsen i valgsetningen evaluerer til `true`. Dette kan vi oppnå ved å utvide valgsetningen med en `else`-blokk.

Dersom vi har en valgsetning med en `else`-blokk, får vi følgende konseptuelle struktur.

```
Kode før
if (betingelse) {
    Kode som utføres om betingelsen stemmer
}
else {
    Kode som utføres om betingelsen ikke stemmer
}
Kode etter
```



Avhengig av om betingelsen i valgsetningen evaluerer til `true` eller `false`, utføres én av de to kodeblokkene, men aldri begge.

Kodeordet `else` har aldri noen betingelse. Den utføres i *alle andre tilfeller*.



Vi kan nå utvide valgsetningen i eksempelet over med en `else`-blokk. Vi kan også fjerne det at vi tømmer tekstboksen, ettersom det uansett verdi blir en utskrift.

```
function utfør() {
    var alder = document.getElementById("txtVerdi").value;
    if (alder >= 18) {
        document.getElementById("utskrift").innerHTML = "Du er myndig";
    }
    else {
        document.getElementById("utskrift").innerHTML = "Du er ikke myndig";
    }
}
```

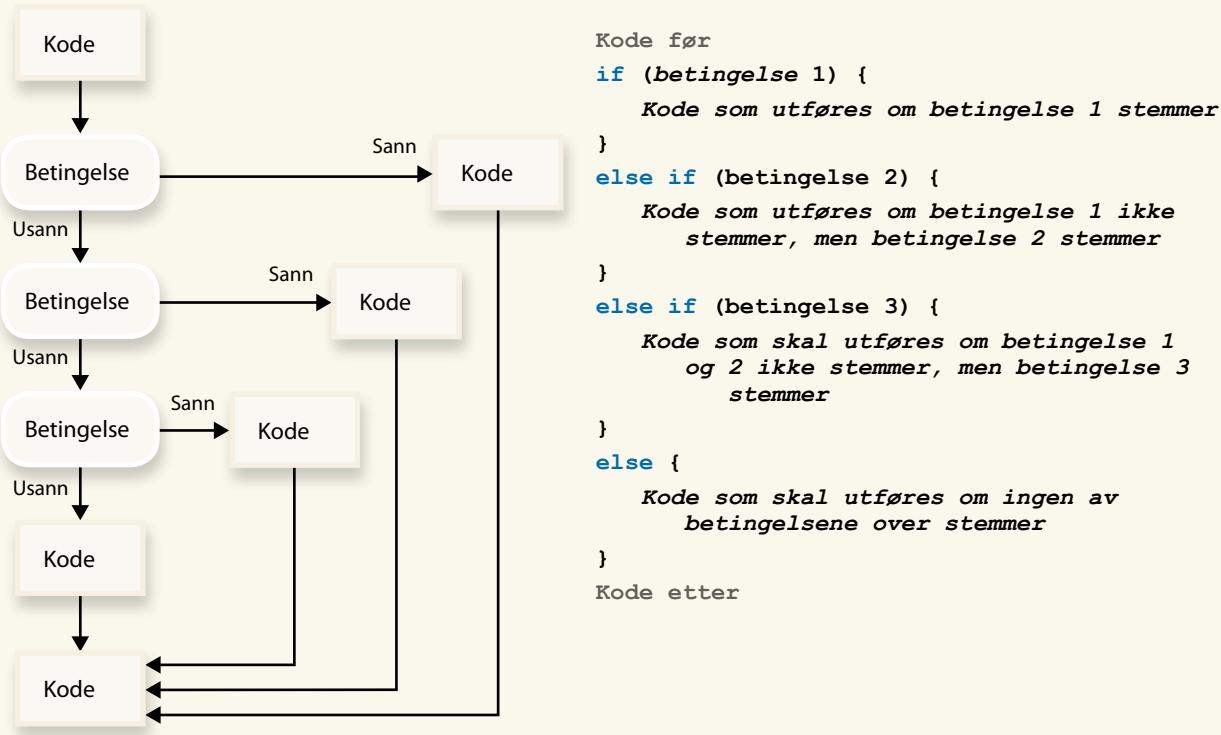
Dersom betingelsen i valgsetningen nå ikke blir oppfylt, eller med andre ord at verdien i variablen `alder` er mindre enn 18, vil utskriften bli *Du er ikke myndig*. På denne måten får brukeren en melding uansett utfall.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Valgsetninger med alternative utfall

Til nå har vi sett på valgsetninger som har hatt ett eller to mulige utfall. Det finnes imidlertid enda en variant av valgsetninger som har flere betingelser og dermed også flere mulige utfall. Denne typen valgsetninger lager vi ved å legge til en eller flere *else if*-blokker, avhengig av hvor mange alternative utfall vi ønsker.

Dersom vi benytter oss av to *else if*-blokker i valgsetningen, vil den konseptuelle strukturen se slik ut:



Legg merke til at det kun er kodeblokken som tilhører den første av betingelsene som evaluerer til `true`, som blir utført. I tilfelle ingen av betingelsene slår til, blir koden under `else` utført. Det er imidlertid valgfritt om `else`-blokken skal være med eller ikke.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Vi kan utvide eksempelet vårt på følgende måte:

```
function utfør() {  
    var alder = document.getElementById("txtVerdi").value;  
    if (alder >= 150) {  
        document.getElementById("utskrift").innerHTML = "Du er et mirakel";  
    }  
    else if (alder >= 67) {  
        document.getElementById("utskrift").innerHTML = "Du er pensjonist";  
    }  
    else if (alder >= 18) {  
        document.getElementById("utskrift").innerHTML = "Du er myndig";  
    }  
    else if (alder >= 0) {  
        document.getElementById("utskrift").innerHTML = "Du er ikke myndig";  
    }  
    else {  
        document.getElementById("utskrift").innerHTML = "Du er ikke født...";  
    }  
}
```

Her har vi nå fem ulike meldinger som kan bli gitt til brukeren. De fire første styres av ulike betingelser, mens den siste blir gitt dersom ingen av de fire førstebetingelsene slår til.

Hadde vi her snudd om på rekkefølgen til betingelsene og deres kodeblokker, slik at den første testen hadde hatt betingen `alder >= 0`, ville denne slått til på alle gyldige verdier for en alder. Dermed ville vi alltid fått utskriften *Du er ikke myndig*, fordi alle mulige aldere som er større enn 0, ville dekkes av denne testen.

Det er viktig at de mest spesialiserte betingelsene står først og de mest generelle til slutt. Dette er fordi programmet under kjøring vil begynne med den første betingen og så arbeide seg nedover helt til en betingelse som er sann, blir funnet.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Valgoperatoren

Ofte ønsker vi at en variabel eller egenskap skal få én av to mulige verdier avhengig av om en betingelse er sann eller ikke. På tross av et ganske avskrekkende navn er en *ternary conditional operator*, eller *valgoperator* på godt norsk, en veldig smart måte å utføre slike operasjoner på.

Rent konseptuelt kan vi si at denne strukturen ser ut som følger, og at hele konstruksjonen blir byttet ut med enten *sann-* eller *usannverdien* basert på hva betingelsen blir evaluert til:

```
( betingelse ? sannverdi : usannverdi )
```



Det er spørsmålsteget og kolonet som gjør at denne egentlig helt vanlige parentesen blir oppfattet av systemet som nettopp en valgoperator.

La oss ta utgangspunkt i eksempelet fra tidligere, som setter paragrafen **utskrift** sin tekst til *Du er myndig* eller *Du er ikke myndig* avhengig av verdien til variabelen **alder**.

Vi kan spare oss for mye arbeid ved heller å skrive programkoden på følgende måte:

```
function utfør() {
    var alder = document.getElementById("txtVerdi").value;
    document.getElementById("utskrift").innerHTML = "Du er " + ↗
        (alder >= 18 ? "myndig" : "ikke myndig");
}
```

Her vil da teksten *Du er* bli satt sammen med teksten *myndig*, om verdien til variabelen **alder** virkelig er større enn eller lik 18, og teksten *ikke myndig* i alle andre tilfeller.

Denne typen verdiangivelse er også svært nyttig ved utskrift av flertallsformer. Ofte ser man programmer som inneholder meldinger som *Det er 45 sekund(er) igjen...* Dette er egentlig bare latskap, ettersom det er kjedelig å skrive en valgsetning som gjør at man får spesialiserte utskrifter med for eksempel *sekund* og *sekunder*.

Valgoperatoren kan imidlertid gjøre dette ganske enkelt for oss.

```
var sekunder = document.getElementById("txtVerdi").value;
document.getElementById("utskrift").innerHTML = "Det er " + ↗
    sekunder + " " + (sekunder === 1 ? "sekund" : "sekunder") + " igjen";
```

Her vil systemet under utførelsen av programkoden enten sette inn teksten *sekund* eller *sekunder* i stedet for valgoperatoren, avhengig av om variabelen **sekunder** inneholder verdien 1 eller ikke.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Gjett på tall

Vi skal her lage en enkel variant av spillet "Gjett på et tall mellom 0 og 100".

Systemet skal trekke et tilfeldig tall, og brukeren får 10 forsøk på å gjette. Etter hvert forsøk skal brukeren få beskjed om tallet var for høyt eller lavt.



1. Lag et nytt HTML-dokument du kaller *gjettall.html* ut fra *mal.html*.
2. Lag en tekstboks med ID **txtTall** og en knapp med ID **btnGjett**. Få også med en paragraf med ID **melding** der resultatet skal vises:

```
<body>
    <p>Tall: <input type="text" id="txtTall" /><br />
    <button id="btnGjett">Gjett</button></p>
    <p id="melding"></p>
</body>
```

3. Skriv inn programkoden som styrer spillokikken:

```
<script>

window.onload = oppstart;
var hemmeligTall;
var forsoekIgjen;

function oppstart() {
    hemmeligTall = Math.floor(Math.random() * 101);
    forsoekIgjen = 10;
    document.getElementById("btnGjett").onclick = gjett;
}

function gjett() {
    forsoekIgjen--;
    var tall = document.getElementById("txtTall").value;
    if (tall === hemmeligTall) {
        document.getElementById("melding").innerHTML = "Gratulerer!!!";
        document.getElementById("btnGjett").disabled = true;
    }
    else if (forsoekIgjen <= 0) {
        document.getElementById("melding").innerHTML = "Du klarte ikke å finne tallet" + hemmeligTall;
        document.getElementById("btnGjett").disabled = true;
    }
    else if (tall < hemmeligTall) {
        document.getElementById("melding").innerHTML = "Tallet er for lavt";
    }
    else if (tall > hemmeligTall) {
        document.getElementById("melding").innerHTML = "Tallet er for høyt";
    }
}

</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

4. Test spillet, og kontroller at de ulike utfallene fungerer.

Tall:

Tallet er for høyt

I programkoden til eksempelet setter vi først noen globale variabler som spillet trenger. Vi setter blant annet **hemmeligTall** til å være et tilfeldig heltall mellom 0 og 100. Vi kommer tilbake til **Math.random()** i et senere kapittel, men kan foreløpig si at denne koden produserer et tilfeldig tall fra og med 0 til (men ikke til og med) 1. Ved å gange dette med 101 og så fjerne desimalene (**Math.floor**) får vi et tilfeldig tall i intervallet vi ønsker.

I hendelsen **gjett**, som er knyttet til knappen, minker vi først antall forsøk brukeren har igjen. Deretter leser vi inn tallet fra brukeren og tar en del sjekker. Først ser vi om brukeren har gjettet riktig tall eller har gått tom for forsøk. Deretter sjekker vi om tallet var høyere eller lavere enn det hemmelige tallet. Til alle utfall gir vi brukeren en melding tilbake. Merk deg at rekkefølgen på testene er svært viktig, da det kun er den første testen som er sann i en **if-else if**-struktur som utfører sin tilhørende kode.

Dersom brukeren svarer riktig eller går tom for forsøk, deaktiverer vi knappen slik at brukeren ikke kan spille videre. Dette gjøres ved å sette egenskapen **disabled** til **true**.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Ballflytter



I dette eksempelet skal vi lage et canvas der vi i et ganske hyppig tempo fjerner eksisterende innhold og tegner en ny sirkel. Posisjonen sirkelen tegnes på, skal endres og skal gradvis gå mot posisjonen til musepekeren. Totalt unyttig, men eksempelet inneholder et par programmeringsteknikker det er verdt å merke seg.

1. Lag et nytt HTML-dokument du kaller *ballflytter.html* ut fra *mal.html*.

2. Legg inn et canvas på nettsiden med ID **tegneflate**:

```
<body>
    <canvas id="tegneflate" width="800" height="800"/>
</body>
```

3. La canvaset bli plassert øverst i høyre hjørne og få en kantlinje så vi ser det:

```
<style>
    #tegneflate {
        border-style:solid;
        position:absolute;
        top: 0;
        left: 0;
    }
</style>
```

4. Legg inn programkoden som skal til for å oppnå funksjonaliteten til eksempelet:

```
<script>

window.onload = oppstart;
var ballX = 400;
var ballY = 400;
var musX = 0;
var musY = 0;

function oppstart() {
    document.getElementById("tegneflate").onmousemove = musFlyttet;
    setInterval("flyttBall()",100);
}

function musFlyttet(evt) {
    musX = evt.clientX;
    musY = evt.clientY;
}

function flyttBall() {
    if (ballX > musX) {
        ballX -= 5;
    }
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```

        else {
            ballX += 5;
        }
        if (ballY > mouseY) {
            ballY -= 5;
        }
        else {
            ballY += 5;
        }

        var canvas = document.getElementById("tegneflate");
        var ctx = canvas.getContext("2d");

        ctx.clearRect(0, 0, canvas.width, canvas.height);

        ctx.beginPath();
        ctx.arc(ballX, ballY, 40, 0, 2 * Math.PI);
        ctx.fillStyle = "green";
        ctx.fill();
        ctx.lineWidth = 5;
        ctx.strokeStyle = "red";
        ctx.stroke();

    }

</script>

```

Dette eksempelet inneholder mye kode og noe helt ny kode. Allikevel består det av forholdsvis få deler. Vi definerer fire globale variabler: to for å holde på posisjonen til ballen og to for å holde på den siste kjente posisjonen til musepekeren.

Under lasting av nettsiden forteller vi at hendelsesfunksjonen **musFlyttet** skal utføres mens musepekeren flytter seg over canvaset. Denne funksjonen henter ut musekoordinatene fra hendelsesparameteren og oppdaterer variablene **musX** og **musY** med disse verdiene. Vi forteller også at en blokk med kode (funksjon) med navn **flyttBall** skal utføres med 100 millisekunders (0,1 sekunders) mellomrom gjennom kommandoen **setInterval**.

Det som utføres hvert 0,1 sekund er å justere ballens posisjon 5 punkter/pixler nærmere siste kjente museposisjon ved hjelp av valgsetninger. Deretter fjerner vi innholdet i canvaset ved hjelp av kommandoen **clearRect** og tegner ut ballen på sin nye posisjon. Denne fjerningen og tegningen går så fort at når vi ser på animasjonen, oppfatter vi det som at det er den samme sirkelen som flytter seg.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Logiske uttrykk

Et *logisk uttrykk*, eller ofte også kalt en *betingelse*, er en påstand som enten er oppfylt eller ikke. Vi kan dermed si at det logiske uttrykket evaluerer til sant (`true`) eller usant (`false`).

Disse to verdiene stemmer godt overens med datatypen *boolean* som vi så på i forrige kapittel. Med andre ord kan man ta vare på resultatet av et logisk uttrykk i en variabel som datatypen *boolean*.

I tillegg er det logiske uttrykk som vi benytter som betingelse/test i valgsetninger og løkker (neste kapittel). Logiske uttrykk er faktisk en av de viktigste byggesteinene i programmering.

Vi skal i denne gjennomgangen benytte logiske uttrykk sammen med variabler, men eksakt de samme uttrykkene kunne vært benyttet som en betingelse/test i en valgsetning eller løkke.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Relasjonsoperatorer

JavaScript har mange spesielle operatorer for å uttrykke en betingelse. En gruppe av disse er relasjonsoperatorene. Disse brukes for å lage en påstand om hvordan to verdier forholder seg til hverandre, og vi har alt sett disse i bruk tidligere i dette kapitelet.

Vi kan for eksempel påstå at $5 < 4$, noe som medfører **false**, ettersom 5 ikke er mindre enn 4. Derimot vil påstanden $5 > 4$ gi oss **true**, ettersom betingelsen nå er **sann**. Dersom vi ser på dette som programkode, kan vi skrive som følger:

```
var svarA = 5 < 4;    //Lagrer false i variabelen
var svarB = 5 > 4;    //Lagrer true i variabelen
```

Benyttet i en valgsettning kunne det sett slik ut:

```
if (5 < 4) {
    document.getElementById("utskrift").innerHTML = "Fem er mindre enn fire";
}
else if (5 > 4) {
    document.getElementById("utskrift").innerHTML = "Fem er større enn fire";
}
```

Eller vi kunne benyttet variablene **svarA** og **svarB** i koden:

```
if (svarA === true) {
    document.getElementById("utskrift").innerHTML = "Fem er mindre enn fire";
}
else if (svarB === true) {
    document.getElementById("utskrift").innerHTML = "Fem er større enn fire";
}
```



Større enn eller lik og mindre enn eller lik skrives slik som vi uttaler det.
Det finnes ikke noen $=>$ og $=<$.

De vanligste relasjonsoperatorene:

$==$	lik
$!=$	ulik
$>$	større enn
$<$	mindre enn
\geq	større enn eller lik
\leq	mindre enn eller lik

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

I programmering er det stor forskjell på ett og tre likhetstegn. Ett likhetstegn (=) benyttes for å tilordne variabelen på venstre side verdien på høyre side. Tre likhetstegn (==), derimot, benyttes for å sammenligne om venstre og høyre side er like eller ei.



Du vil sikkert oppdage at operatorene == og != (altså ett likhetstegn mindre) også kan benyttes for likhet og ulikhet i JavaScript . Disse gjør imidlertid en automatisk typekonvertering som kan skape mange potensielle feil, og omtales av mange programmerere som “the evil twin”.



Ettersom variabler inneholder verdier, kan vi også benytte variabler i logiske uttrykk. Vi kan for eksempel basert på verdien i variablen alder la variablen **myndig** få verdien **true** eller **false**.

```
var alder = 26;  
var myndig = (alder >= 18);
```

Variablen **myndig** vil nå inneholde verdien **true** dersom variablen **alder** er større eller lik 18. I alle andre tilfeller, altså om **alder** er mindre enn 18, vil den få verdien **false**.

Man trenger ikke å ha parenteser rundt det logiske uttrykket, men det hjelper mye på lesbarheten av koden.



Vi kan også se på et eksempel med lik og ulik. Her ønsker vi både å sjekke om to personer er like gamle, og om de ikke er like gamle.

```
var alderPersonA = 26;  
var alderPersonB = 43;  
var likeGamle = (alderPersonA === alderPersonB);  
var ikkeLikeGamle = (alderPersonA !== alderPersonB);
```

I dette tilfellet vil de boolske variablene **likeGamle** og **ikkeLikeGamle** alltid ha motsatt verdi av hverandre.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Logiske operatorer

I tillegg til relasjonsoperatorene, som benyttes for å uttrykke påstander om hvordan to verdier forholder seg til hverandre, har vi også et sett med *logiske operatorer*. De logiske operatorene kjennetegnes ved at de jobber med betingelser omtrent på samme måte som matematiske operatorer jobber med tall.

I dette avsnittet skal vi se på to logiske operatorer som kalles *AND* og *OR*. Disse brukes for å koble sammen selvstendige betingelser til større uttrykk. I tillegg skal vi se på *negeringsoperatoren* som snur verdien til et logisk uttrykk.

AND-operatoren

AND (og på norsk) benyttes for å knytte sammen betingelser slik at begge betingelsene må være sanne for at hele uttrykket skal bli sant. *AND* uttrykkes i JavaScript ved operatoren **&&**.

Vi kan for eksempel oversette betingelsen *du er kvalifisert om alderen er mindre enn 45 OG høyden er mer enn 189* til følgende logiske uttrykk.

```
var kvalifisert = ( alder < 45 && hoyde > 189 );
```

Vi ser her at det logiske uttrykket egentlig består av to selvstendige betingelser, nemlig **alder < 45** og **høyde > 189**, men at disse er knyttet sammen til ett logisk uttrykk ved hjelp av **&&**.

Man kan også forklare de logiske operatorene ved hjelp av såkalte *sannhetstabeller*. Rader og kolonner vil da være resultatet av de to betingelsene, og der den aktuelle rad og kolonne møtes står resultatet av operatoren.

&&	true	false
true	true	false
false	false	false

Ut av tabellen ser vi at den eneste kombinasjonen som gir **true** som resultat, er **true & true**.



Det som står på hver side av AND- og OR-operatorene, må være selvstendige betingelser som vi kan si sant eller usant til. Det er dermed ikke tillatt å skrive **alder >= 3 && <= 45**. I stedet må vi skrive **alder >= 3 & alder <= 45** for å oppfylle kravet om selvstendige betingelser.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

OR-operatoren

OR (eller på norsk) benyttes for å knytte sammen betingelser slik at det holder at en av betingelsene er sanne for at uttrykket skal bli sant, selv om også begge kan være det. *OR* uttrykkes ved operatoren `||`.

Tegnet | finner du som oftest på tasten som er til venstre for ett-tallstasten, eller alt + 7 på Mac.



Vi kan dermed oversette betingelsen *du er kvalifisert om alderen er mindre enn 45 ELLER høyden er mer enn 189* til følgende logiske uttrykk:

```
var kvalifisert = ( alder < 45 || hoyde > 189 );
```

Vi kan også lage en sannhetstabell for OR, tilsvarende den vi lagde for AND.

	true	false
true	true	true
false	true	false

Som vi ser, gir alle kombinasjoner **true**, unntatt **false || false**.

I vanlig språk er det ikke noen klar definisjon av ordet *eller*. Det blandes ofte sammen med begrepet *enten eller*. Husk at i programmering betyr operatoren OR begge eller én.



Negeringsoperatoren

Blant de logiske operatorene finner vi også *negeringsoperatoren*. Denne snur verdien av en betingelse slik at **true** blir **false** og **false** blir **true**. Man kan si at negeringsoperatoren uttrykker begrepet *ikke*.

Tar vi utgangspunkt i eksemplet om kvalifisering, kan vi finne ut om man *ikke* er kvalifisert, ved å negere verdien i variabelen `kvalifisert`.

```
var kvalifisert = ( alder < 45 && hoyde > 189 );
var ikkeKvalifisert = !(kvalifisert);
```

Alternativt kunne vi funnet denne verdien direkte ved å negere selve uttrykket.

```
var ikkeKvalifisert = !( alder < 45 && hoyde > 189 );
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Ofte, men ikke alltid, vil det også være mulig ganske enkelt å snu relasjonsoperatorene i betingelsen i stedet for å benytte negeringsoperatoren. Vær imidlertid forsiktig med å gjøre dette dersom det logiske uttrykket består av flere delbettingelser satt sammen med logiske operatorer, da dette ikke alltid er så rett fram som det kan virke.



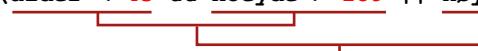
Dersom man ønsker å negere et logisk uttrykk, må man benytte seg av parenteser for å si at negeringen skal gjelde hele uttrykket.

Kompliserte uttrykk

Man kan selvfølgelig lage så kompliserte logiske uttrykk man måtte ønske. Det er ingen begrensning på hvordan man setter sammen ulike betingelser ved hjelp av de logiske operatorene vi har sett på i dette kapitlet.

Normalt vil et logisk uttrykk evalueres fra venstre mot høyre, men alle AND evalueres før alle OR.

```
var kvalifisert = (alder < 45 && hoyde > 189 || hoyde < 120);
```



Dette logiske uttrykket beskriver altså at *man skal være yngre enn 45 år og høyere enn 189 cm ELLER lavere enn 120 cm.*

Dersom det ikke var dette man ønsket, kan man benytte seg av parenteser for å gruppere betingelser som hører sammen, og dermed påvirke rekkefølgen uttrykket evalueres etter.

```
var kvalifisert = (alder < 45 && ( hoyde > 189 || hoyde < 120 ));
```



Ved å legge til et sett parenteser har vi nå endret opptakskravet til at *man skal være yngre enn 45 år OG høyere enn 189 cm eller lavere enn 120 cm.*



Det vil alltid være to måter å finne et komplisert logisk uttrykk på. Enten kan man finne uttrykket slik vi ønsker det, eller man kan finne det motsatte og så negere dette. Ofte vil en av disse veiene være lettere enn den andre, så det er lurt å prøve begge om du står fast.

De vanligste logiske operatorene:

- && logisk og
- || logisk eller
- ! logisk ikke (negeringsoperator)

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

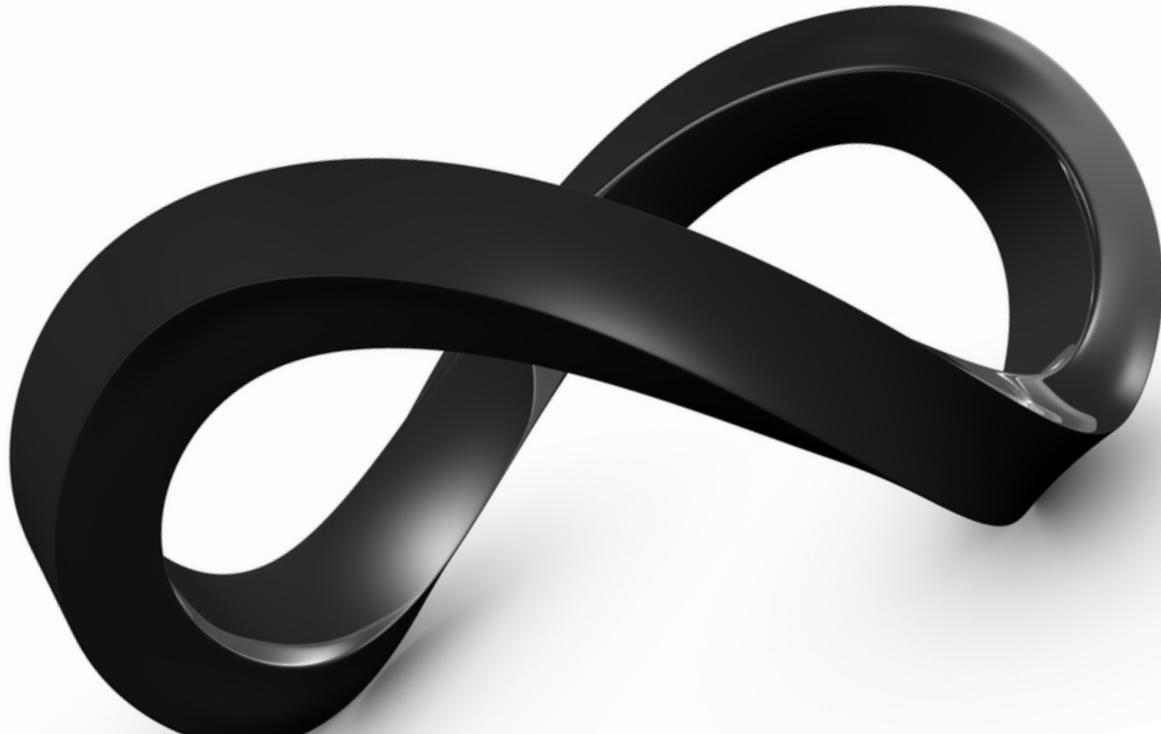
5 Løkker

I dette kapitlet vil du lære

- hva løkker er
- å benytte løkkens teller
- hvordan du kombinerer kontrollstrukturer
- om nøkkelordene `break` og `continue`

Vi skal i dette kapitelet fortsette gjennomgangen av kontrollstrukturer. Som vi husker, er en kontrollstruktur noe som påvirker hvordan instruksjoner blir utført. Mens vi i forrige kapittel så på ulike former for valgsettninger, som utfører programkode *dersom* en betingelse stemmer, skal vi i dette kapitelet se på en type kontrollstrukturer som utfører programkode *så lenge* en betingelse stemmer.

Slike kontrollstrukturer kalles som oftest *løkker*, og de har som sagt mye til felles med valgsettninger. Valgsettninger kan imidlertid utføre en gruppe instruksjoner maksimalt én gang, mens løkker ikke har noen slik begrensning.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

While-løkker

Den mest grunnleggende formen for denne typen kontrollstrukturer er den såkalte while-løkken. Ser vi på strukturen til denne, kan den illustreres som følger:



Legg merke til at en *while*-løkke minner svært mye om en *if-test* i oppbygningen. Den store forskjellen ligger i hva som skjer etter at kodeblokken er utført. I stedet for å gå videre til eventuelle instruksjoner etter strukturen går *kontrollflyten* i løkker tilbake til betingelsen.

Her utføres det en ny test på om betingelsen fortsatt er *sann*, og er den det, vil kodeblokken utføres nok en gang før vi igjen sjekker betingelsen. Slik fortsetter løkka helt til betingelsen blir *usann*. Vi kaller hver utførelse av kodeblokken i en løkke for en *itasjon*.



Når du arbeider med løkker, er det spesielt viktig å tenke nøye gjennom hva som skal utføres én gang før løkka starter, hva som skal utføres flere ganger mens løkka kjører, og hva som skal utføres én gang etter løkka er ferdig. Det å plassere instruksjoner feil her vil gi ganske så pussige resultater.

Ved å benytte samme brukergrensesnitt som vi introduserte i kapitelet om valgsetninger, kan et enkelt eksempel på bruk av en *while*-løkke være som følger:

```

function utfør() {
    var antall = document.getElementById("txtVerdi").value;
    while (antall > 0) {
        document.getElementById("utskrift").innerHTML += "Nok en gang<br />";
        antall--;
    }
}
  
```

Først setter vi variabelen **antall** til å være det tallet vi skriver i tekstboksen. Deretter kommer vi til selve løkka, som vil teste at antallet er større enn 0. Blir dette **false**, er vi ferdige ettersom det ikke står noen instruksjoner etter while-løkka.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Dersom antallet derimot skulle være større enn 0, utfører vi innholdet i kodeblokken som tilhører løkka. Det første som da gjøres, er ved hjelp av operatoren `+=` å legge til teksten *Nok en gang* og et linjeskift i paragrafen. Deretter minker vi variabelen `antall` sin verdi med én.

Når kodeblokken er utført, er imidlertid ikke løkka ferdig. Kontrollflyten går tilbake til betingelsen og evaluerer denne på nytt. Slik fortsetter det helt til betingelsen blir **false** og koden vår er ferdig utført.

Dette medfører at paragrafen **utskrift** nå vil inneholde et antall linjer med teksten *Nok en gang*, som tilsvarer det antallet vi skrev i tekstboksen.

Uendelige løkker

Hvis testen i en løkke forblir sann, vil løkka i teorien gå i det uendelige. I praksis vil dette ha *effekten* av at programmer ser ut til å ha låst seg. Dette omtales som *uendelige løkker*. Det virker som programkoden ikke gjør noe, men i praksis står den og jobber så mye at den ikke har tid til å behandle kommandoer fra brukeren.

I nettlesere er det imidlertid bygd inn en funksjon som gjør at programkoden avsluttes etter et forhåndsdefinert tidsinterval, og du vil få en feilmelding.

Du unngår uendelige løkker ved å være sikker på at testen blir **false** etter et visst antall iterasjoner. En forutsetning for dette er at vi tester på en variabel som forandrer verdi i løkka. Løkka i eksempelet nedenfor vil gå i det uendelige ettersom variablene vi tester på, ikke forandrer verdi:

```
var teller = 0;
while (teller < 3) { // Feil! - uendelig løkke
    document.getElementById("utskrift").innerHTML += "Nok en gang<br />";
}
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Selv om vi forandrer verdi på variabelen vi tester på, kan testen i seg selv føre til at den alltid slår til. I eksempelet nedenfor vil `teller` alltid være større enn 3 slik at testen alltid slår til:

```
var teller = 4;
while (teller > 3) { // Feil! - uendelig løkke
    document.getElementById("utskrift").innerHTML += "Nok en gang<br />";
    teller = teller + 1;
}
```



Når det gjelder løkker, er det svært viktig at instruksjonene i kodeblokken en eller annen gang gjør at betingelsen blir usann. Ellers vil løkka utføre instruksjonene i kodeblokken et uendelig antall ganger, og brukeren vil oppfatte dette som at systemet har hengt seg. En vanlig feil av denne typen vil være å bevege seg bort fra verdien som gjør at betingelsen blir `false`, i stedet for mot den.

For-løkker

Svært ofte får vi *while-løkker* der vi initialiserer en variabel før selve løkka, samtidig som denne variabelen er en del av betingelsen og endres inne i løkka. Et typisk eksempel på dette vil være følgende løkke:

```
var teller = 1;
while (teller < 10) {
    document.getElementById("utskrift").innerHTML += "Nok en gang<br />";
    teller++;
}
```

Dersom kodeblokken hadde inneholdt flere instruksjoner enn i eksempelet over, blir det lett å miste oversikten over de tre viktigste delene av løkka. Vi tenker da på instruksjonen som setter startverdien på variabelen, selve betingelsen og den instruksjonen som gjør at utfallet av betingelsen etter hvert vil endres til `false`.

Derfor har man konstruert en annen måte å skrive slike tilfeller på, som kalles *for-løkker*. Det fine med denne typen løkker er at den samler de tre viktigste delene av løkka på ett sted, slik at det er lett å beholde oversikten.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tar vi for oss det samme eksempelet, vil dette skrevet med en *for-løkke* i stedet for en *while-løkke* se slik ut:

```
for (var teller = 1; teller < 10; teller++) {
    document.getElementById("utskrift").innerHTML += "Nok en gang<br/>";
}
```

Som en tommelfingerregel er det ofte lurt å benytte *for-løkker* dersom betingelsen består av en numerisk variabel som øker eller minker for hver iterasjon, og *while-løkker* om betingelsen er en annen type logisk uttrykk.

TIPS

Legg merke til at *for-løkker* og *while-løkker* egentlig er samme kontrollstruktur, bare skrevet på ulike måter. Alle *while-løkker* av denne typen (med en startverdi, en betingelse og en endringsdel til slutt) kan lett skrives som *for-løkker*, og omvendt, etter følgende omformingsregel:

<pre>start while (bettingelse) { Kode og endring }</pre>		<pre>for (start; bettingelse; endring) { Kode }</pre>
--	--	---

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Benytte telleren til mer enn en teller

Variabelen vi benytter i betingelsen til en løkke, kalles ofte en **teller** dersom den inneholder heltall. Ettersom telleren er en helt vanlig variabel, kan vi også benytte denne i kodeblokken som tilhører løkka. I følgende eksempel bruker vi derfor telleren som en del av utskriften.

```
for (var teller = 1; teller < 5; teller++) {
    document.getElementById("utskrift").innerHTML += teller + ". gang<br />";
}
```

Som du ser, er det ingenting spesielt med instruksjonen i denne kodeblokken. Vi vil imidlertid oppnå at løkka skriver ut en melding der tallet som **teller** inneholder, er en del av utskriften. Dette medfører at tekstboksen vil få fire linjer med tekst, der innholdet vil være *1. gang*, *2. gang*, *3. gang* og *4. gang*.



Selv om det også er tillatt å endre tellervariabelen inni kodeblokken til en for-løkke, frarådes dette sterkt. Endringsdelen av for-løkka mister da litt av sin mening, og det kan lett oppstå logiske feil i koden.

Vi kan også lage et litt større eksempel som skriver ut en bestemt gangetabell, ved hjelp av en *for-løkke* og dens teller:

```
function utfør() {
    var faktor1 = document.getElementById("txtVerdi").value;
    for(var faktor2 = 1; faktor2 <= 10; faktor2++) {
        var produkt = faktor1 * faktor2;
        document.getElementById("utskrift").innerHTML += faktor1 + " * "
            + faktor2 + " = " + produkt + "<br />";
    }
}
```

Vi lar her variabelen **faktor1** være et tall vi skriver i tekstboksen. Telleren i løkka er **faktor2** og får som vi ser, verdier fra og med 1 til og med 10.

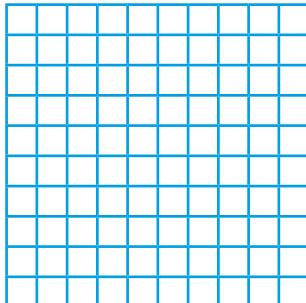
Inne i kodeblokken til løkka lager vi så en ny variabel som heter **produkt**. Denne variabelen vil bli opprettet hver gang kodeblokken skal utføres. Innholdet i variabelen settes til å være **faktor1** multiplisert med **faktor2**.

Etter at vi har regnet ut hva produktet av de to faktorene er, legger vi til nok en linje i utskriften. For å sette sammen denne nye linja benytter vi **faktor1** som ble gitt i tekstboksen, **faktor2** som er telleren i løkka, og **produkt** som kalkuleres for hver *iterasjon*.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Tegne rutenett

I dette eksempelet skal vi bruke løkker for å tegne et rutenett på et canvas:



1. Lag et nytt HTML-dokument du kaller *rutenett.html* ut fra *mal.html*.

2. Legg til et canvas på nettsiden med ID **tegneflate**:

```
<body>
    <canvas id="tegneflate" width="400" height="400"></canvas>
</body>
```

3. Skriv inn programkoden som tegner rutenettet ved hjelp av to løkker:

```
<script>

window.onload = oppstart;

function oppstart() {
    var antallKolonner = 10;
    var antallRader = 10;
    var bredde = document.getElementById("tegneflate").width;
    var hoeyde = document.getElementById("tegneflate").height;
    var kolonneBredde = bredde / antallKolonner;
    var radHoeyde = hoeyde / antallRader;

    var ctx = document.getElementById("tegneflate").getContext("2d");

    ctx.lineWidth = 2;
    ctx.strokeStyle = "blue";

    for (var tellerK = 0; tellerK <= antallKolonner; tellerK++) {
        ctx.beginPath();
        ctx.moveTo(tellerK * kolonneBredde, 0);
        ctx.lineTo(tellerK * kolonneBredde, hoeyde);
        ctx.stroke();
    }

    for (var tellerR = 0; tellerR <= antallRader; tellerR++) {
        ctx.beginPath();
        ctx.moveTo(0, tellerR * radHoeyde);
        ctx.lineTo(bredde, tellerR * radHoeyde);
        ctx.stroke();
    }
}

</script>
```

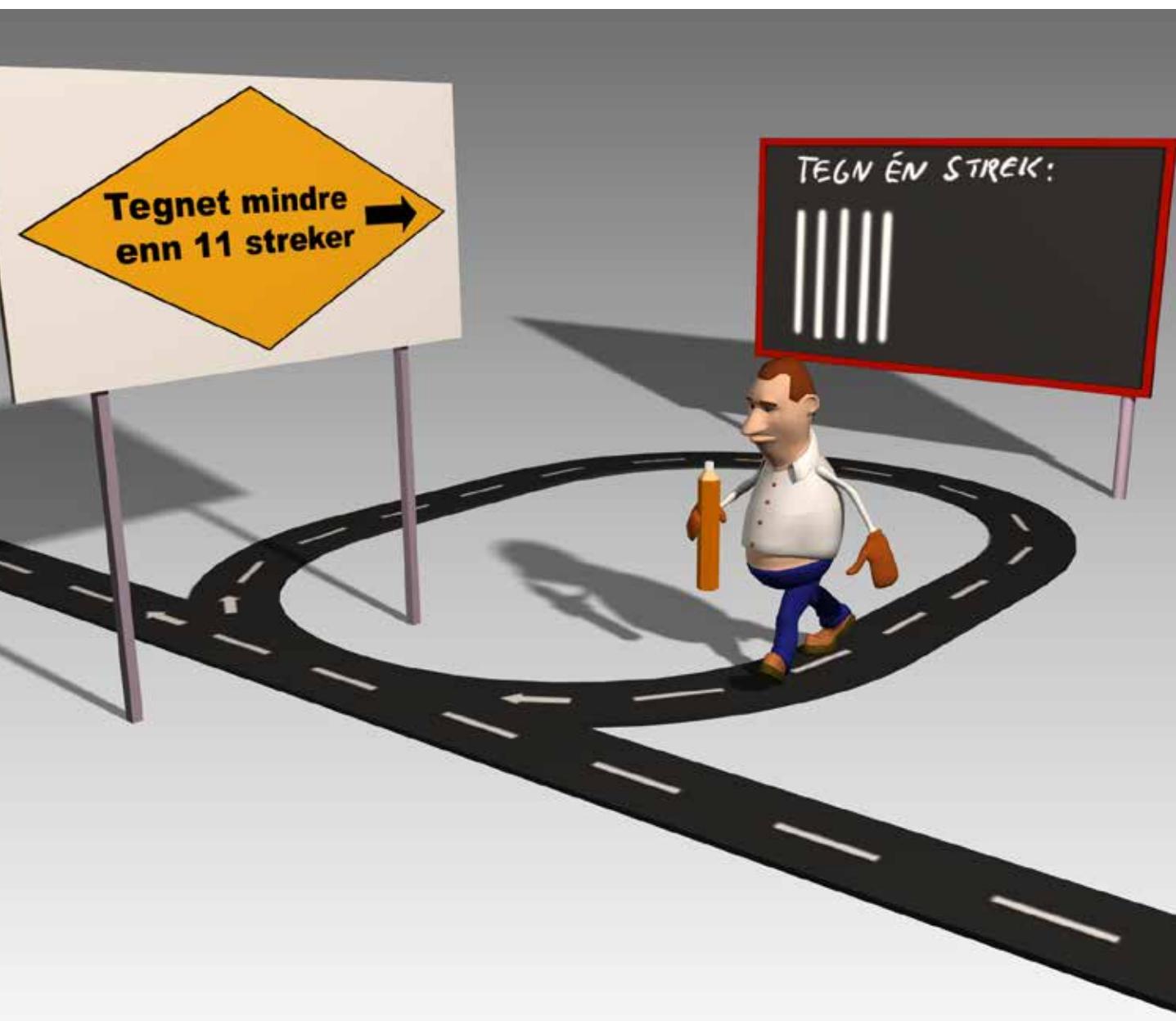
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

- Test nettsiden og kontroller at den tegner rutenettet korrekt.

I førsten av **oppstart**-funksjonen setter vi en del variabler som holder på utregninger vi får behov for senere. Legg spesielt merke til at bredde og høyde på kolonner/rader beregnes ut fra canvasets høyde/bredde og ønsket antall. Dette gjør det lettere å oppdatere senere, da vi kun trenger endre målene på selve **<canvas>**-taggen.

I den første løkken setter vi **tellerK** til 0. Deretter går vi gjennom løkka så lenge **tellerK** er mindre enn eller lik antall kolonner. Første gang vi går gjennom løkka, er **tellerK** lik 0 slik at vi tegner en linje fra (0,0) til (0,400). Andre gang vi går gjennom løkka, er **tellerK** lik 1 slik at vi tegner en linje fra (40,0) til (40,400). Tredje gang går linja fra (80,0) til (80,400) osv.

Tilsvarende gjøres for de vannrette linjene i den andre løkka.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Nestede kontrollstrukturer

Ettersom kodeblokken til en kontrollstruktur kan inneholde ordinær programkode, kan vi også ha nye kontrollstrukturer som en del av denne. Dette kalles da *nestede kontrollstrukturer*.

I eksempelet som følger, kombinerer vi en løkke som teller fra 1 til 25, med en valgsetning:

```
for (var teller = 1; teller <= 25; teller++) {
    if(teller % 2 === 0) {
        document.getElementById("utskrift").innerHTML += teller + ↗
            "<br />";
    }
}
```

For hver gang kodeblokken til løkka blir utført, vil valgsetningen sjekke om **teller** for øyeblikket er et partall (altså 0 i rest etter en divisjon på 2). Skulle det vise seg at betingelsen til valgsetningen stemmer, legges tallet i **teller** til som en linje i utskriften. Dersom betingelsen derimot ikke stemmer, går vi rett og slett bare videre til neste iterasjon i løkka, ettersom det ikke er andre instruksjoner i kodeblokken.

Ofte får vi også nytte av å ha en løkke inni en annen, eller såkalte *nestede løkkers*, som er en spesialversjon av *nestede kontrollstrukturer*. Et vanlig eksempel på en slik form for nestede kontrollstrukturer er det å generere den lille gangetabellen.

Vi tar her utgangspunkt i programkoden vi hadde for å skrive ut en bestemt gangetabell. Imidlertid lar vi nå **faktor1** være telleren i en egen løkke i stedet for en variabel som henter sin verdi fra en tekstboks.

```
function utfør() {
    for(var faktor1 = 1; faktor1 <= 10; faktor1++) {
        for(var faktor2 = 1; faktor2 <= 10; faktor2++) {
            var produkt = faktor1 * faktor2;
            document.getElementById("utskrift").innerHTML += ↗
                faktor1 + "*" + faktor2 + "=" + produkt + "<br />";
        }
        document.getElementById("utskrift").innerHTML += "<br />";
    }
}
```

Ved første øyekast ser nok dette ganske forvirrende ut, men det er viktig å huske på at for hver iterasjon utføres all programkode i kodeblokken til den ytterste løkka. At denne kodeblokken inneholder en annen løkke, er ikke engang den ytterste løkka klar over.

Ser vi tilbake på eksempelet denne programkoden baserer seg på, er det ikke store endringene. Faktisk er den innerste løkka som styrer **faktor2**, eksakt den samme.

Forskjellen er at **faktor1** nå er en teller i en løkke som går fra 1 til 10, i stedet for et fast tall. For hver gang denne telleren øker, kjører vi kodeblokken og dermed også den innerste løkka. Med andre ord får vi 10 utskrifter for hver iterasjon i den ytterste løkka.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Helt til slutt i den ytterste løkkas kodeblokk har vi også lagt til et ekstra linjeskift i utskriften. Dette vil medføre at for hver iterasjon av den ytterste løkka får vi en blank linje før neste verdi av telleren **faktor1** genererer sin gangetabell.



Eksempel - 5 little monkeys jumping...

En barnesang som for noen kanskje er kjent, er *5 little monkeys jumping on the bed*. Vi skal nå forsøke å generere versene ved hjelp av en løkke i stedet for å skrive dem manuelt.

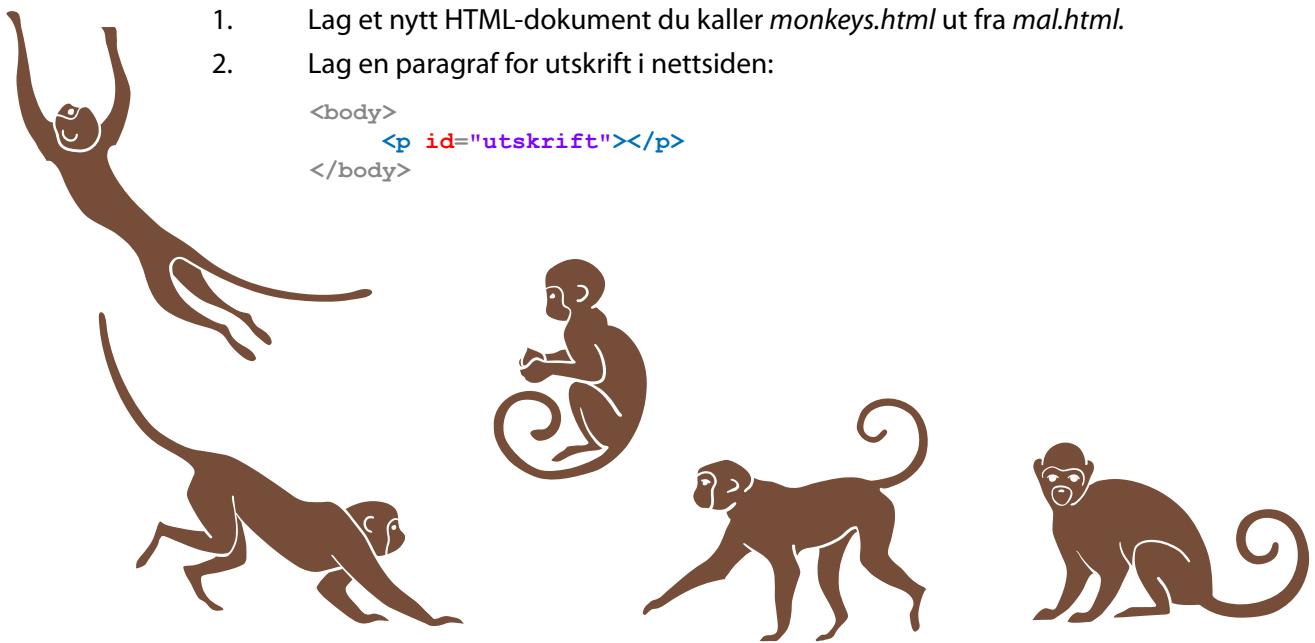
For dem som ikke kjenner visa eller husker teksten, består den av fem vers. For hvert vers reduserer vi antallet med én.

```
<tall> little monkeys jumping on the bed,  
One fell out and bumped his/her head;  
Momma called the doctor, the doctor said,  
That's what you get for jumping on the bed!  
<tall-1> little monkeys jumping on the bed!
```

Hensikten med dette eksempelet er å se praktisk bruk av løkker, der det ville være en stor jobb å gjøre det samme manuelt. I tillegg inneholder eksempelet nestede kontrollstrukturer.

1. Lag et nytt HTML-dokument du kaller *monkeys.html* ut fra *mal.html*.
2. Lag en paragraf for utskrift i nettsiden:

```
<body>  
  <p id="utskrift"></p>  
</body>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Lag en løkke som teller ned fra 5 til 1, og instruksjoner i kodeblokken til løkka, som hver og en produserer en ny linje for aktuelle vers:

```
<script>

window.onload = oppstart;

function oppstart() {
    var tekst = "";
    for (var antall = 5; antall >= 1; antall--) {
        tekst += antall + " little monkeys jumping on the bed,<br />";
        tekst += "One fell out and bumped his head;<br />";
        tekst += "Momma called the doctor, the doctor said,<br />";
        tekst += "That's what you get for jumping on the bed!<br />";
        tekst += (antall-1) + " little monkeys jumping on the bed!<br /><br />";
    }
    document.getElementById("utskrift").innerHTML = tekst;
}

</script>
```

4. Test nettsiden, og kontroller at visa skrives ut korrekt.
 5. Vi bør imidlertid legge til et alternativt siste vers, slik at sangen blir komplett. Dette gjøres ved å benytte en valgsetning på linje nummer to og den siste linja:

```
for (var antall = 5; antall >= 1; antall--) {
    tekst += antall + " little monkeys jumping on the bed,<br />";
    tekst += (antall > 1 ? "One" : "He") + " fell out and bumped his head;<br />";
    tekst += "Momma called the doctor, the doctor said,<br />";
    tekst += "That's what you get for jumping on the bed!<br />";
    if(antall > 1) {
        tekst += (antall-1) + " little monkeys jumping on the bed!<br /><br />";
    }
    else {
        tekst += "Put those monkeys straight to bed!<br /><br />";
    }
}
```

6. Test nettsiden igjen og kontroller at det siste verset også ble korrekt.

Selv om denne visa kanskje ikke er av de største poetiske underverkene som er lagd, illustrerer dette eksempelet i alle fall noen viktige konsepter. Merk deg spesielt hvordan vi bruker telleren i løkka til å skrive versene med, og at vi med en valgsetning kan få alternative hendelser i løkkas kodeblokk. Dersom du øker tellerens startverdi fra 5 til 100, ser du noe av effekten med løkker enda tydeligere.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Kontrollere en løkke

I tillegg til den kontrollen vi får fra betingelsen, har vi enda noen muligheter for å styre en løkke. Dette kan gjøres gjennom de to nøkkelordene **break** og **continue**, som begge påvirker løkkas naturlige gang.

Break

Løkke

```
{
  kode
  kode
break; —
  kode
}
```

Nøkkelordet **break** medfører at løkka stopper så snart denne kommandoen påtreffes, uansett hvor mange iterasjoner av løkka som måtte gjenstå. Vi kan illustrere **break** på den måten som vises i margen.

Dersom vi tar for oss dette i et faktisk eksempel, skulle det kunne se slik ut:

```
for (var teller = 0; teller < 20; teller++) {
    if (teller * teller * 1.3 > 125) {
        break;
    }
    document.getElementById("utskrift").innerHTML += teller + "<br />";
}
```

Løkka skulle nå egentlig ha skrevet ut tallene fra 0 til 19 på hver sin linje. Allikevel stopper løkka etter utskriften av tallet 9, ettersom $10 \times 10 \times 1.3$ er mer enn 125 og kommandoen **break** da blir utført.



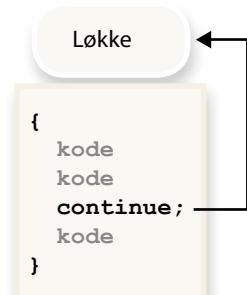
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Continue

Nøkkelordet **continue** medfører at løkka hopper direkte til neste iterasjon.

Man unnlater med andre ord å utføre resterende instruksjoner i kodeblokken, og kontrollflyten går tilbake til selve løkka.

På samme måte som med **break** kan vi illustrere **continue** på den måten som er vist i margen.



Et lite eksempel på bruk av **continue** kan være som følger.

```
for (var teller = 0; teller < 20; teller++) {\n    if (teller % 3 === 0) {\n        continue;\n    }\n    document.getElementById("utskrift").innerHTML += teller + "<br />";\n}
```

Dette eksempelet skriver ut alle tall mellom 0 og 19 på hver sin linje, unntatt de som er delige med 3. Valgsetningen slår nemlig til på disse tallene, og kommandoen **continue** blir utført. I sin tur medfører dette at vi hopper rett til neste iterasjon i stedet for å produsere utskriften av tallet.

Det kan være lurt begrense bruken av nøkkelordene **break** og **continue** til kun der det ikke er mulig å få samme funksjonalitet ved å skrive om betingelsen eller endringsdelen av løkka. Disse nøkkelordene kan lett introdusere logiske feil som er vanskelige å finne ut av.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Eksempel - Primtallsgenerator

Å finne alle primtall innenfor et gitt intervall er et problem som enkelt lar seg løse ved hjelp av nestede løkker. Vi benytter her regelen om at et tall er et primtall dersom det ikke er delelig på noen tall mellom 2 og seg selv.

Hensikten med dette eksempelet er å vise hvilken nytte vi kan ha av nestede løkker. Eksempelet vil også illustrere en teknikk vi kan benytte for å utføre en handling kun dersom alle iterasjoner i en løkke gir samme resultat i en nestet valgsettning.

1. Lag et nytt HTML-dokument du kaller *primtall.html* ut fra *mal.html*.
2. Legg til et enkelt brukergrensesnitt bestående av to tekstfelt, en knapp og en paragraf for utskrift:

```
<body>
    <p>
        Fra: <input type="text" id="txtFra" />
        Til: <input type="text" id="txtTil" />
        <button id="btnGenerer" type="button">Generer</button>
    </p>
    <p id="utskrift"></p>
</body>
```

3. Registrer en **onclick**-hendelse på knappen, der du henter ut verdiene fra tekstboksene og legger dem i variabler. Vi lager også en variabel kalt **tekst** som skal fylles med en utskrift og deretter presenteres i paragrafen:

```
<script>

window.onload = oppstart;

function oppstart() {
    document.getElementById("btnGenerer").onclick = generer;
}

function generer() {
    var fraVerdi = document.getElementById("txtFra").value;
    var tilVerdi = document.getElementById("txtTil").value;

    var tekst = "";

    document.getElementById("utskrift").innerHTML = tekst;

}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

4. Lag deretter en for-løkke i samme hendelse, med en teller som går fra **fraVerdi** til **tilVerdi**:

```
function generer() {  
    var fraVerdi = document.getElementById("txtFra").value;  
    var tilVerdi = document.getElementById("txtTil").value;  
  
    var tekst = "";  
  
    for (var hovedtall = fraVerdi; hovedtall <= tilVerdi; hovedtall++) {  
    }  
  
    document.getElementById("utskrift").innerHTML = tekst;  
}
```

5. Det neste vi ønsker, er en nestet løkke, som går fra tallet 2 og opp til verdien av **hovedtall**. Denne løkka skal også sjekke om telleren **hovedtall** er delelig med telleren **deletall**, og dette gjøres ved hjelp av modulusoperatoren:

```
function generer() {  
    var fraVerdi = document.getElementById("txtFra").value;  
    var tilVerdi = document.getElementById("txtTil").value;  
  
    var tekst = "";  
  
    for (var hovedtall = fraVerdi; hovedtall <= tilVerdi; hovedtall++) {  
        for (var deletall = 2; deletall < hovedtall; deletall++) {  
            if (hovedtall % deletall === 0) {  
            }  
        }  
    }  
  
    document.getElementById("utskrift").innerHTML = tekst;  
}
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

6. Det siste som gjenstår, er å skrive ut de verdiene til telleren **hovedtall**, som ikke ved noen tilfeller er delelige på telleren **deletall**. Her benytter vi et lite triks med en boolsk variabel:

```
function generer() {
    var fraVerdi = document.getElementById("txtFra").value;
    var tilVerdi = document.getElementById("txtTil").value;

    var tekst = "";

    for (var hovedtall = fraVerdi; hovedtall <= tilVerdi; hovedtall++) {
        var erPrimtall = true;
        for (var deletall = 2; deletall < hovedtall; deletall++) {
            if (hovedtall % deletall === 0) {
                erPrimtall = false;
                break;
            }
        }
        if (erPrimtall === true && hovedtall > 1) {
            tekst += "Tallet " + hovedtall + " er et primtall!  
";
        }
    }

    document.getElementById("utskrift").innerHTML = tekst;
}
```

7. Test nettsiden, og finn for eksempel alle primtall mellom 45 og 300.

Dette eksempelet illustrerer stort sett det meste du trenger å vite om løkker. Det første vi ser, er at vi kan benytte verdiene til variabler som start- og sluttverdi, noe som gjør løkkene mye mer fleksible.

Primtallsgeneratoren er også et godt eksempel på bruk av nestede løkker. Legg merke til hvordan stoppverdien i den innerste løkka hele tiden er basert på nåverdien til telleren i den ytterste løkka.



Noe du virkelig bør merke deg, er bruken av variabelen **erPrimtall**. For hver iterasjon i den ytterste løkka setter vi **erPrimtall** til **true**. Dette betyr da at vi går ut fra at tallet er et primtall inntil det motsatte er bevist.

Deretter forsøker den innerste løkka å sette variablen til **false**, dersom den skulle finne noe som motbeviser dette. Ettersom det jo holder med ett bevis på at verdien i telleren **hovedtall** ikke er et primtall, utføres det også en **break** slik at den innerste løkka stopper.

Når den innerste løkka så er ferdig, sjekker vi om variabelen **erPrimtall** fortsatt er **true**. Er den det, og tallet er større enn 1, betyr jo dette at vi aldri fant et tall som verdien i telleren **hovedtall** var delelig på, og tallet må være et primtall. Vi kan i så tilfelle foreta en utskrift av tallet, før vi fortsetter til neste iterasjon i den ytterste løkka.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

En veldig vanlig feil vil være å plassere valgsetningen som utfører utskriften, inn i kodeblokken til den innerste løkka. Problemet er at den da foretar utskriften hver gang den finner et tall som telleren **hovedtall** ikke er delelig på. Dem vil det være mange av, selv på de tallene som ikke er primtall.

For bedre å forstå hva dette problemet innebærer, kan du gjøre følgende endring i koden og teste selv. Prøv imidlertid med et veldig lite og lavt intervall, for her vil det bli mange utskrifter.

```
function generer() {
    var fraVerdi = document.getElementById("txtFra").value;
    var tilVerdi = document.getElementById("txtTil").value;

    var tekst = "";

    for (var hovedtall = fraVerdi; hovedtall <= tilVerdi; hovedtall++) {
        for (var deletall = 2; deletall < hovedtall; deletall++) {
            if (hovedtall % deletall !== 0) {
                tekst += "Tallet " + hovedtall + " er et primtall!<br />";
            }
        }
    }

    document.getElementById("utskrift").innerHTML = tekst;
}
```

Legg deg deretter dette på minnet, for du vil garantert komme borti flere slike problemer, der en boolsk variabel vil være et godt hjelpemiddel.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

6 Arrayer

I dette kapitlet vil du lære

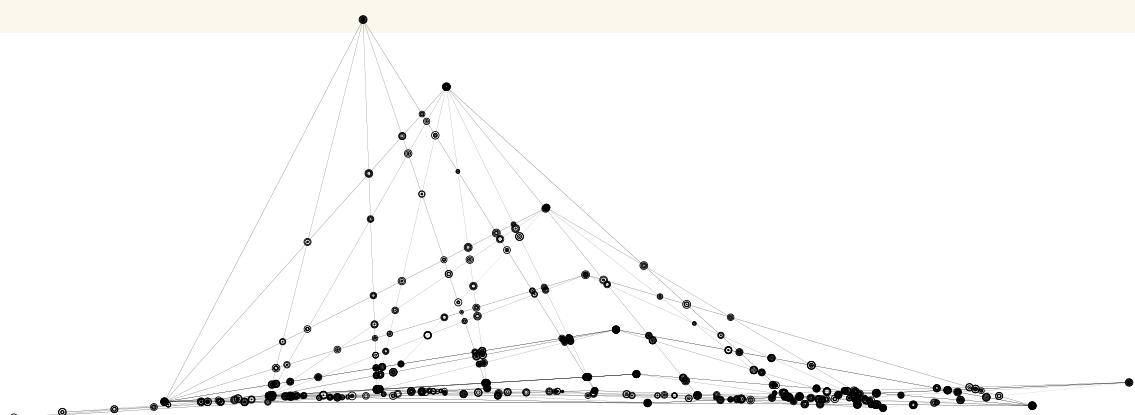
- hva en array er
- om funksjonaliteten som ligger innebygd i en array
- hvordan man benytter løkker sammen med arrayer
- om arrayer med flere dimensjoner
- om assosiative arrayer
- om behandling av tekststrenger

Arrayer (tabeller)

Da vi snakket om variabler tidligere i denne boka, sa vi at dette egentlig er navnet på et område i minnet på datamaskinen der man midlertidig kan lagre én verdi. En *array* er egentlig ikke noe annet enn navnet på en teknikk som lar oss lagre flere verdier under samme navn.

TIPS

Ofte kalles arrayer for tabeller i norske lærebøker. Vi velger allikevel å benytte navnet arrayer, ettersom begrepet tabell gir assosiasjoner til en struktur som har både rader og kolonner, noe en array vanligvis ikke har. Liste kan imidlertid være et alternativt begrep på norsk.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Jobbe med arrayer

La oss starte denne introduksjonen med en sammenligning mellom arrayer og ordinære variabler. Når vi deklarerer og initialiserte en ordinær variabel, så det ut som følger:

```
var poeng = 4;  
var land = "Norge";
```

Disse variablene kan holde på én verdi av gangen. Dersom vi i stedet ønsker å deklarerere og initialisere en array som kan holde på flere verdier av gangen, ser det slik ut:

```
var poengListe = [4, 73, 56, 13, 24];  
var landListe = ["Norge", "Sverige", "Danmark", "Kina", "Italia"];
```

Det er firkantparentesene rundt verdiene skilt med komma som angir at vi nå ikke ønsker å lagre én verdi, men snarere en array bestående av flere verdier.

En forenklet forklaring på arrayer er at det er en teknikk som samler mange verdier til én, som dermed får plass i én variabel.

TIPS

Går vi igjen tilbake til eksempelet med de ordinære variablene, kunne vi for eksempel benyttet verdiene som ligger lagret i disse i meldinger:

```
var tekst1 = "Jeg fikk " + poeng + " poeng";  
var tekst2 = "Jeg bor i " + land;
```

Vi kan på en tilsvarende måte benytte verdiene i en array. Vi må imidlertid angi hvilken av verdiene vi ønsker å jobbe med, ettersom systemet ikke kan vite dette. Verdiene i en array er nummerert fra 0 og oppover, og vi angir hvilken verdi vi mener ved hjelp av dette nummeret.

```
var tekst3 = "Jeg fikk " + poengListe[0] + " poeng"; // Jeg fikk 4 poeng  
var tekst4 = "Jeg bor i " + landListe[2]; // Jeg bor i Danmark
```

Her angir vi at vi ønsker å benytte verdien på *indeks* 0 og indeks 2 i arrayen. Dette betyr da at vi ønsker å jobbe med den første verdien i **poengListe** og den tredje verdien i **landListe**.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Arrayer begynner nummereringen av verdiene på 0. Dette medfører at første verdi har indeks 0, andre verdi har indeks 1, osv.



Hvis du forsøker å hente ut eller referere til en verdi på en indeks som ikke finnes, vil du få ut den spesielle verdien **`undefined`**.

0	Norge
1	Sverige
2	Danmark
3	Kina
4	Italia

Det kan være praktisk å tenke på arrayen som en nummerert liste.

Det å endre en ordinær variabel sin verdi ble gjort på følgende måte:

```
poeng = 12;
land = "Finland";
```

Å gjøre den tilsvarende operasjonen med en array krever igjen at vi må angi hvilken av verdiene vi ønsker å arbeide med.

```
poengListe[2] = 12;
landListe[1] = "Finland";
```

I eksempelet over ville vi da endret verdien i `poengListe` på indeks 2, eller altså verdi nummer tre, fra 56 til 12. I `landListe` ville vi endret verdien på indeks 1, altså verdi nummer to, fra *Sverige* til *Finland*.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Til forskjell fra en del andre programmeringsspråk kan vi ha forskjellige datatyper i den samme arrayen i JavaScript. For eksempel vil dette gå greit:

```
var minArray = ["Hans Hansen", 20];
```

For å unngå programmeringsfeil er det imidlertid lurt å la en array kun inneholde elementer av samme datatype.

Lengden av en array

Hvis vi vil finne ut hvor mange elementer det er i arrayen, kan vi bruke egenskapen **length**:

```
var antallElementer = minArray.length;
```

Siden nummereringen i en array starter på 0, vil indeksen til det siste elementet alltid være én mindre enn lengden av arrayen.



Legge til elementer i en array

Dersom vi har en eksisterende array, kan vi legge til nye elementer i denne ved å referere til plasser som er etter siste posisjon.

```
var medlemmer = ["Ole", "Per", "Nils", "Kari"];
medlemmer[4] = "Line";
```

Arrayen **medlemmer** vil nå ha fem elementer, der det siste, på indeks 4, er *Line*. Det kan være lurt å benytte seg av egenskapen **length**, slik at vi er sikre på at elementet havner på siste posisjon. Tallverdien for lengde tilsvarer alltid neste ledige indeks, ettersom vi starter å telle på 0.

```
var medlemmer = ["Ole", "Per", "Nils", "Kari"];
medlemmer[medlemmer.length] = "Line";
```

Enkelte synes også det er mer oversiktlig å lage en helt tom array, og deretter fylle denne med verdier instruksjon for instruksjon:

```
var medlemmer = [];
medlemmer[0] = "Ole";
medlemmer[1] = "Per";
medlemmer[2] = "Nils";
medlemmer[3] = "Kari";
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

For-løkker og arrayer

Ofte har vi lyst til å gjøre noe med alle elementene i en array. Ettersom hvert element er referert til med en indeks av typen heltall (integer), er for-løkker en god kandidat til denne jobben. Om vi for eksempel ønsker å skrive ut alle elementene i en array, kan vi gjøre som følger:

```
var tallListe = [45, 67, 34, 54, 23, 42];
var tekst = "";

for (var i = 0; i < tallListe.length; i++) {
    tekst += "Element nr " + i + " er " + tallListe[i] + "<br />";
}

document.getElementById("utskrift").innerHTML = tekst;
```

For-løkka lager nå en teller med navn `i`, som etter tur får alle de mulige verdiene en indeks i arrayen `tallListe` kan ha. Legg merke til at vi stopper når telleren `i` ikke lenger er mindre enn `tallListe.length`. Dette er fordi høyeste indeks er én mindre enn antall elementer i arrayen. For mange vil det være fristende å skrive tallet 6 i stedet for `tallListe.length`, men da måtte vi oppdatert informasjonen i løkka dersom vi endret antall elementer i arrayen.



Det er ikke noe krav om at telleren skal hete `i`, men det er vanlig ettersom det er en forkortelse for *indeks*.

Et annet eksempel kan være å summere tallene i arrayen:

```
var tallListe = [45, 67, 34, 54, 23, 42];
var sum = 0;

for (var i = 0; i < tallListe.length; i++) {
    sum += tallListe[i];
}

document.getElementById("utskrift").innerHTML =
    "Summen av elementene i arrayen er " + sum;
```



Det er mye bedre å benytte `length` som stoppverdi enn det faktiske antallet. På den måten slipper vi å huske på og endre stoppverdien om vi skulle endre antallet elementer i arrayen. Vi unngår dermed mange potensielle feil.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Statistikk over terningkast



I dette eksempelet skal vi automatisk slå en tenkt terning 1000 ganger og så benytte en array for å telle opp hvor mange ganger vi fikk de ulike antall øyne.

Vi skal også utvide eksempelet noe og presentere resultatet som et liggende søylediagram.

1. Lag et nytt HTML-dokument du kaller *terningkast.html* ut fra *mal.html*.
2. Legg inn en enkel `<div>`-tagg der utskriften skal komme:

```
<body>
  <div id="utskrift"></div>
</body>
```

3. Skriv koden som produserer de 1000 terningkastene, og som skriver ut en paragraf for hvert av utfallene 1–6. Hver paragraf inneholder en `` med selve teksten. Utskriften her er gjort ganske tungvint i forhold til det vi har sett før, ved at vi lager nye elementer, men dette gjør det lettere når vi senere i eksempelet skal legge til flere elementer som en del av utskriften:

```
<script>

window.onload = oppstart;

function oppstart() {

    var opptelling = [0,0,0,0,0,0];

    for (var i = 0; i < 1000; i++) {
        var oeyne = Math.floor(Math.random() * 6) + 1;
        opptelling[oeyne - 1]++;
    }

    for (var j = 0; j < opptelling.length; j++) {
        var nyttElement = document.createElement("p");
        nyttElement.innerHTML = "Antall " + (j + 1) + "'ere: " + opptelling[j];
        document.getElementById("utskrift").appendChild(nyttElement);
    }
}

</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

4. Test nettsiden, og kontroller at du fikk en utskrift av fordelingen.

Antall 1'ere: 172

Antall 2'ere: 157

Antall 3'ere: 183

Antall 4'ere: 161

Antall 5'ere: 155

Antall 6'ere: 172

5. Lag nå en stil som forteller at ``-tagger med classen **graf** skal være røde og ha høyden 20 piksler. La også visningstypen være **inline-block**, slik at vi kan justere størrelsen på ``-taggen uten at den følger innholdets størrelse:

```
<style>
    span.graf {
        background-color: red;
        height: 20px;
        display: inline-block;
        margin-left: 10px;
    }
</style>
```

6. Legg til kode som lager nye ``-elementer i hver paragraf vi skriver ut, og som setter klassen til disse til å være **graf**, samt vidden til å være det antall piksler som antall slag viser:

```
for (var j = 0; j < oppelling.length; j++) {
    var nyttElement = document.createElement("p");
    nyttElement.innerHTML = "Antall " + (j + 1) + "'ere: " + oppelling[j];

    var nyGraf = document.createElement("span");
    nyGraf.className = "graf";
    nyGraf.style.width = oppelling[j] + "px";
    nyttElement.appendChild(nyGraf);

    document.getElementById("utskrift").appendChild(nyttElement);
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

7. Test nettsiden, og kontroller at du nå også får ut en graf for hvert element.

Antall 1'ere: 179	
Antall 2'ere: 166	
Antall 3'ere: 175	
Antall 4'ere: 163	
Antall 5'ere: 149	
Antall 6'ere: 168	



Programkoden i eksempelet består i hovedsak av tre deler.

Først oppretter vi en array med navn **oppstelling**, som inneholder 6 plasser. Dette er en plass for hver side av terningen. Antallet setter vi foreløpig til å være 0 for hvert element, ettersom vi ennå ikke har begynt å slå terningen.

Neste del av koden inneholder en løkke som utfører noe 1000 ganger. Det denne løkka gjør, er å lage en variabel som heter **øyne**, og sette denne til et tilfeldig tall mellom 1 og 6. Deretter øker vi verdien/antallet i arrayen på indeksen som samsvarer med antallet øyne, men siden arrayen starter på 0 og går til 5, må vi trekke fra 1 på antall øyne.

Til slutt skriver vi ut verdien som ligger lagret på de ulike indeksene i arrayen. Her må vi legge til 1 på indeksen under utskrift, slik at vi får verdier fra 1 til 6.

Det mest kompliserte i denne koden er nok utskriften. Vi genererer her opp **<p>**-tagger som vi fyller med en tekst og med en ****. ****-elementet styrer vi i hovedsak gjennom en CSS-regel definert i **<style>**-delen av dokumentet, men vidden setter vi i selve koden.

HTML-koden som produseres, vil bli som følger:

```
<div id="utskrift">
  <p>Antall 1'ere: 179<span class="graf" style="width: 179px;"></span></p>
  <p>Antall 2'ere: 166<span class="graf" style="width: 166px;"></span></p>
  <p>Antall 3'ere: 175<span class="graf" style="width: 175px;"></span></p>
  <p>Antall 4'ere: 163<span class="graf" style="width: 163px;"></span></p>
  <p>Antall 5'ere: 149<span class="graf" style="width: 149px;"></span></p>
  <p>Antall 6'ere: 168<span class="graf" style="width: 168px;"></span></p>
</div>
```

Om du synes utskriftsdelen av koden var komplisert, så er det selve bruken av arrayen til oppstelling som er det viktigste på dette tidspunktet.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Array-funksjoner

Arrayer har mange såkalte funksjoner (metoder) som gir oss forskjellige muligheter for å manipulere på innholdet, for eksempel:

- **splice (startindex, antall)** – sletter antall elementer fra *startindex*.
- **splice (startindex, antall, ...verdi, ...verdi, ...)** – *splice* kan også brukes til å legge til nye eller erstatte elementer ved å oppgi kommaseparerte verdier etter *antall*.
- **push (verdi)** – legger til et element i slutten av arrayen og setter innholdet lik *verdi*.
- **pop ()** – fjerner det siste elementet i arrayen og gir tilbake verdien som resultat.
- **unshift (verdi)** – legger til et element i begynnelsen av arrayen og setter innholdet lik *verdi*. De andre elementene blir forskjøvet ett hakk bakover.
- **shift ()** – fjerner det første elementet i arrayen og gir verdien som resultat. De andre elementene blir forskjøvet ett hakk fremover.
- **sort ()** – sorterer elementene i arrayen i stigende rekkefølge.
- **reverse ()** – reverserer rekkefølgen i arrayen.
- **indexOf (verdi, startindex)** – søker etter verdi i arrayen fra *startindex*. Hvis verdien blir funnet, returneres indeksen til dette elementet, hvis ikke, returneres -1. *Startindex* kan droppes, og da starter søket på første av arrayen.
- **concat (array2)** – setter sammen arrayen med *array2* og returnerer resultatet som en ny array. Hvis du ikke angir noen *array2*, returneres en kopi av arrayen.
- **join (skilletegn)** – konverterer arrayen til en sammenhengende tekststrengh. I tekststrenghen blir hvert element skilt med et angitt skilletegn.

Nedenfor skal vi se på noen eksempler på faktisk bruk av funksjonene. Du bruker en funksjon ved å skrive **arraynavnet.funksjonsnavnet()**. Alle eksemplene tar utgangspunkt i følgende array, og hvert eksempel går tilbake til "originalen":

```
var minArray = [3, 6, 2, 4, 1, 3];
```

Kode som legger til elementet 7 på slutten av arrayen:

```
minArray.push(7);
```

Kode som fjerner elementer på index 3 og to elementer videre, altså elementene 4 og 1:

```
minArray.splice(3, 2);
```

Kode som reverserer arrayen slik at den blir 3, 1, 4, 2, 6 og 3:

```
minArray.reverse();
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Kode som får tak i indeksen til elementet som har verdien 6. Søket starter i begynnelsen av tabellen. Resultatet blir at variabelen **posisjon** inneholder tallet 1:

```
var posisjon = minArray.indexOf(6);
```

Lager en kommasseparert tekststreng for utskrift, der arrayen er blitt gjort om til teksten "3,6,2,4,1,3":

```
var tekst = minArray.join(",");
```

Kort fortalt er en funksjon (metode) en navngitt samling med kode som vi kan kjøre når vi ønsker det. De innebygde funksjonene i JavaScript hjelper oss med å gjøre vanlige operasjoner, slik at vi slipper å skrive kode for dette selv. I neste kapittel vil vi gi en mer grundig innføring i hva funksjoner er, og hvordan du lager dine egne.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Flerdimensjonale arrayer

Noen ganger ønsker vi å lagre verdier i tabellform, slik som vi kan gjøre i et regneark – altså i rader og kolonner. Vi kan da lage en array som inneholder andre arrayer. Nedenfor lager vi en array på med tre elementer, der hvert element består av en ny array på fire elementer.

```
var billiste = [ ];
billiste[0] = ["Volvo", "Fiat", "Lada", "Renault"];
billiste[1] = ["Blå", "Rød", "Gul", "Hvit"];
billiste[2] = [110, 150, 68, 100];
```

Vi kan tenke oss arrayen visualisert på denne måten, med rader og kolonner:

Indeks	0	1	2
0	Volvo	Blå	110
1	Fiat	Rød	150
2	Lada	Gul	68
3	Renault	Hvit	100

Helt korrekt er det imidlertid at arrayen lagres i minnet på maskinen slik:

0				1				2			
0	1	2	3	0	1	2	3	0	1	2	3
Volvo	Fiat	Lada	Renault	Blå	Rød	Gul	Hvit	110	150	68	100



Det er ikke gitt at den ytterste arrayen tilsvarer kolonnen og den innerste arrayen tilsvarer raden. Dette er kun et valg vi har gjort når vi skal tegne opp en datastruktur som egentlig bare finnes i minnet på maskinen.

La oss si vi ønsker å hente ut innholdet i andre kolonne, fjerde rad. Vi kan da bruke:

```
var farge = billiste[1][3]; // Hvit
```

Vi kan også endre en verdi i arrayen ved å bruke samme skrivemåte. Nedenfor setter vi innholdet i andre kolonne, tredje rad til *rustrød*:

```
billiste[1][2] = "Rustrød";
```

Arrayen i eksempelet ovenfor er en *tedimensjonal array*. Det er mulig å lage arrayer med flere dimensjoner ved å ha en array som inneholder arrayer, som igjen inneholder arrayer, osv. Disse er det imidlertid vanskelig å tenke seg hvordan ”ser ut”.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Finne avstander



I dette eksempelet skal vi bruke en todimensjonal array for å finne avstanden mellom ulike steder. Vi har allerede fått oppgitt avstanden i kilometer mellom stedene i oversikten nedenfor:

	Arendal	Grimstad	Lillesand	Kristiansand	Mandal
Arendal	0	22	41	71	113
Grimstad	22	0	21	51	93
Lillesand	41	21	0	30	72
Kristiansand	71	51	30	0	43
Mandal	113	93	72	43	0

To steder velges fra nedrekkslister som er fylt med stedsnavnene, og avstanden mellom stedene vises.

1. Lag et nytt HTML-dokument du kaller *avstand.html* ut fra *mal.html*.
2. Lag selve brukergrensesnittet bestående av to nedtrekkslister, en knapp og en paragraf:

```
<body>
  <p>
    Fra: <select id="lstFra"></select><br />
    Til: <select id="lstTil"></select><br />
    <button id="btnBeregn">Beregn</button>
  </p>
  <p id="utskrift"></p>
</body>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Skriv programkoden som oppretter en todimensjonal array for avstandene, samt en array for stedsnavn. I lastingen av siden fyller vi nedtrekkslistene med stedsnavnnene. I hendelsen til knappen henter vi ut stedsnavn og avstand tilsvarende valgene som er gjort i nedtrekkslistene:

```
<script>

window.onload = oppstart;

var avstandListe = [];
avstandListe[0] = [0, 22, 41, 71, 113];
avstandListe[1] = [22, 0, 21, 51, 93];
avstandListe[2] = [41, 21, 0, 30, 72];
avstandListe[3] = [71, 51, 30, 0, 43];
avstandListe[4] = [113, 93, 72, 43, 0];

var sted = ["Arendal", "Grimstad", "Lillesand", "Kristiansand", "Mandal"];

function oppstart() {
    for (var i = 0; i < sted.length; i++) {
        var valg1 = document.createElement("option");
        valg1.innerHTML = sted[i];
        valg1.value = i;
        document.getElementById("lstFra").appendChild(valg1);

        var valg2 = document.createElement("option");
        valg2.innerHTML = sted[i];
        valg2.value = i;
        document.getElementById("lstTil").appendChild(valg2);
    }

    document.getElementById("btnBeregn").onclick = beregn;
}

function beregn() {
    var fraIndex = document.getElementById("lstFra").value;
    var tilIndex = document.getElementById("lstTil").value;

    var avstand = avstandListe[fraIndex][tilIndex];

    document.getElementById("utskrift").innerHTML = "Avstanden fra " + ↪
        sted[fraIndex] + " til " + sted[tilIndex] + " er " + avstand + ↪
        " kilometer";
}
</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

4. Test nettsiden og kontroller at du kan beregne avstandene.

Fra: Arendal ▾
Til: Kristiansand ▾

Avstanden fra Arendal til Kristiansand er 71 kilometer

Det nye i denne koden er først og fremst at vi oppretter elementene i en nedtrekksliste ut fra en array. Det finnes mange måter å lage elementer på, men her har vi valgt å gjøre det mest mulig likt det du kjenner fra tidligere. For hvert element i **sted**-arrayen oppretter vi et **<option>**-element som får teksten tilsvarende stedsnavnet og **value** tilsvarende indeksen. Merk deg at vi må lage to helt like **<option>**-elementer, da samme element ikke kan plasseres to steder.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Assosiative arrayer

Assosiative arrayer er arrayer som bruker navn i stedet for nummer/indeksering på de ulike elementene. Ettersom det ikke er gitt hvilke indeksverdier vi da skal benytte, må vi angi dem selv. Ved å bruke assosiative arrayer kan vi lage en samling med ulike verdier, gjerne også med ulike datatyper. Vanligvis kaller vi indeksen i en assosiativ array for en *nøkkel*.

Lage en assosiativ array

Du kan lage en ny assosiativ array på følgende måte:

```
var personinfo = {fornavn: "Harald", etternavn: "Svikkelund", alder: 78}
```



Legg spesielt merke til at assosiative arrayer opprettes ved hjelp av {} og ikke [] rundt verdiene.

Alternativt kan vi sette verdiene en etter en:

```
var personinfo = {}
personinfo["fornavn"] = "Harald";
personinfo["etternavn"] = "Svikkelund";
personinfo["alder"] = 78;
```

Du kan tenke på en assosiativ array som et slags informasjonskort.

personinfo	
fornavn	Harald
etternavn	Svikkelund
alder	78

Vi kan få tak i innholdet i et element ved å bruke klammeparenteser på samme måte som i en vanlig array, men angivelsen av hvilken verdi vi ønsker arbeide med, er da en nøkkel og ikke en tallbasert indeks.

```
var tekst = personinfo["etternavn"]; //Svikkelund
```

Med assosiative arrayer kan vi også bruke denne noe mer elegante skrivemåten:

```
var tekst = personinfo.etternavn; //Svikkelund
```



Helt formelt er ikke assosiative arrayer i JavaScript arrayer i det hele tatt, men såkalte objekter. Vi kommer mer tilbake til dette senere i boka, men foreløpig kan du tenke på det som arrayer med tekstlige indekser.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Iterere gjennom en assosiativ array

Ønsker vi å gå gjennom (iterere gjennom) en assosiativ array, kan vi bruke en spesiell for-løkke som kalles `for..in`:

```
for (var nøkkel in personinfo) {
    tekst += "Nøkkelen " + nøkkel + " har verdien " + ↗
        personinfo[nøkkel] + "<br />";
}
```

I parentesen i for-løkka lager vi en ny variabel som heter `nøkkel`. Denne variabelen vil inneholde navnet på hvert element i arrayen for hver runde gjennom løkka. Inne i løkka skriver vi ut nøkkelen og tilhørende innhold i elementet. Programmet hopper automatisk ut av løkka etter den siste verdien. Resultatet blir:

```
Nøkkelen fornavn har verdien Harald
Nøkkelen etternavn har verdien Svikkelund
Nøkkelen alder har verdien 78
```

Løkker av typen `for..in` kan også benyttes som et alternativ til ordinære for-løkker for vanlige arrayer dersom indeks skal gå fra 0 til slutt. Dette er imidlertid ikke vanlig å gjøre.

```
var tallListe = [45, 67, 34, 54, 23, 42];
var sum = 0;

for(var i in tallListe) {
    sum += tallListe[i];
}
```

Sjekke om en nøkkel finnes

For å sjekke om en spesifikk nøkkel finnes i en assosiativ array, kan vi benytte en såkalt `if..in`:

```
if ("etternavn" in personinfo) {
    document.getElementById("utskrift").innerHTML = personinfo["etternavn"];
}
```

Dersom testen slår til, betyr det altså at vi er sikre på at den assosiative arrayen har et element med denne nøkkelen, og kan dermed hente ut tilhørende verdi.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Rader og felt

Ved å la hvert element i en vanlig array være en assosiativ array får vi en form for tabell med rader og feltnavn – på samme måte som en tabell i en database:

```
var personregister = [ ];
personregister[0] = {etternavn: "Bør", yrke: "Lege"};
personregister[1] = {etternavn: "Hansen", yrke: "Politi"};
personregister[2] = {etternavn: "Lia", yrke: "Ingeniør"};
personregister[3] = {etternavn: "Jensen", yrke: "Flyger"};
```

Hvis vi tar utgangspunkt i eksempelet ovenfor, vil feltnavnene være *etternavn* og *yrke*. Vi kan da hente ut yrket til den tredje personen på følgende måte:

```
var yrke = personregister[2]["yrke"]; //Ingeniør
```

Tekststrenger og arrayer

Tekststrenger kan sees på som en form for arrayer der hvert element er ett tegn, og vi kan lett konvertere mellom en verdi av typen string og en array. Ønsker vi for eksempel å sortere en tekststreg, kan vi konvertere den til en array og så bruke **sort**-metoden:

```
var navnString = "Halvard Per Ole Sigurd Allan";
var navnListe = navnString.split(" ");
navnListe.sort();
var navnSortert = navnListe.join(" ");
document.getElementById("utskrift").innerHTML = navnSortert;
// Allan Halvard Ole Per Sigurd
```

Her bruker vi en metode kalt **split** som finnes i alle tekststrenger. Denne konverterer tekststrengen til en array med ord. Vi angir mellomrom som skilletegn det skal deles på. Når vi har sortert arrayen, konverterer vi den tilbake til en tekststreg ved hjelp av **join**-metoden. Her angir vi også mellomrom som tegnet vi setter mellom ordene.

Du kan også plukke ut ett bestemt tegn i en tekststreg ved å henvise til tegnets indeks. I tillegg har tekststrenger, slik som arrayer, en **length**-egenskap:

```
var tekst = "Hei";
var andreTegn = tekst.charAt(1); //Henter ut bokstaven e
var lengde = tekst.length; //Henter ut 3
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Spørreprogram



Vi skal her lage en enkel utgave av en quiz. For å unngå for komplisert programkode så er denne quizen ganske begrenset i funksjonalitet og brukeropplevelse.

Spørsmålene lagres som assosiative arrayer i en annen array som holder spørsmålene samlet. Brukeren svarer på et spørsmål ved å trykke på alternativet. Er det riktig, skal alternativet farges grønt; er det feil, skal rød farge benyttes.

1. Lag et nytt HTML-dokument du kaller *sporsmaal.html* ut fra *mal.html*.
2. Legg inn brukergrensesnittet bestående av en paragraf for spørsmålet, tre alternativer i form av **-elementer, en knapp for neste spørsmål og et **-element for poengberegning:

```
<body>
  <p id="sporsmaalstekst"></p>
  <ul>
    <li id="alt1"></li>
    <li id="alt2"></li>
    <li id="alt3"></li>
  </ul>
  <p id="utskrift"></p>
  <button id="btnNeste">Neste spørsmål</button>
  <p>Poeng: <span id="poeng"></span></p>
</body>
```

3. Legg in tre stiler: to klasser for hhv. riktig og feil svar, samt at alle **-elementer skal ha en hånd som musepeker:

```
<style>
  .riktig {color:green;}
  .feil {color:red;}
  li {cursor:pointer;}
</style>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

4. Legg inn programkoden som skal styre quizen:

```

<script>

window.onload = oppstart;

var sporsmaal = [];
sporsmaal[0] = {tekst: "Hva heter hovedstaden i Norge", alt1: "Stockholm", alt2: "Oslo", alt3: "Paris", riktig: 2};
sporsmaal[1] = {tekst: "Hva er 2+7", alt1: "4", alt2: "5", alt3: "9", riktig: 3};
sporsmaal[2] = {tekst: "Hva er en Fossekall", alt1: "Fugl", alt2: "Hest", alt3: "Fisk", riktig: 1};

var poeng = 0;
var sporsmaailIndex = -1;
var harSvart = false;

function oppstart() {
    document.getElementById("alt1").onclick = svar;
    document.getElementById("alt2").onclick = svar;
    document.getElementById("alt3").onclick = svar;
    document.getElementById("btnNeste").onclick = neste;

    document.getElementById("poeng").innerHTML = poeng;

    neste();
}

function svar(evt) {
    if (harSvart === false) {

        var svar;

        if(evt.target === document.getElementById("alt1")) {
            svar = 1;
        }
        else if(evt.target === document.getElementById("alt2")) {
            svar = 2;
        }
        else if(evt.target === document.getElementById("alt3")) {
            svar = 3;
        }

        if(svar === sporsmaal[sporsmaailIndex].riktig) {
            poeng++;
            evt.target.className = "riktig";
        }
        else {
            evt.target.className = "feil";
        }

        harSvart = true;
        document.getElementById("poeng").innerHTML = poeng;
    }
}

```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```

function neste() {
    sporsmaalIndex++;
    document.getElementById("sporsmaalstekst").innerHTML = sporsmaal[sporsmaalIndex].tekst + "?";
    document.getElementById("alt1").innerHTML = sporsmaal[sporsmaalIndex].alt1;
    document.getElementById("alt1").className = "";
    document.getElementById("alt2").innerHTML = sporsmaal[sporsmaalIndex].alt2;
    document.getElementById("alt2").className = "";
    document.getElementById("alt3").innerHTML = sporsmaal[sporsmaalIndex].alt3;
    document.getElementById("alt3").className = "";

    harSvart = false;

    if (sporsmaalIndex >= sporsmaal.length - 1) {
        document.getElementById("btnNeste").disabled = true;
    }
}
</script>

```

På tross av ganske enkel funksjonalitet består dette eksempelet av svært mye kode. Vi skal derfor forsøke å ta for oss koden ganske grundig.

Som globale variabler lager vi først en array kalt **sporsmaal**, som igjen inneholder assosiative arrayer med feltene **tekst**, **alt1**, **alt2**, **alt3** og **riktig**. Det siste feltet forteller hvilket alternativ som er det rette, og er enten 1, 2 eller 3. Det lages også en variabel med navn **poeng** som kun skal telle opp antall riktige.

Variabelen **sporsmaalIndex** skal holde styr på hvilket spørsmål i arrayen vi er på. Ettersom vi ennå ikke har startet, settes denne til -1. Variabelen **harSvart** skal hindre at brukeren svarer flere ganger på samme spørsmål. Denne settes til **false** hver gang et nytt spørsmål vises, og så til **true** når brukeren svarer.

Under oppstarten av nettsiden forteller vi at alle de tre alternativene (****-elementer) skal ha samme hendelsesfunksjon med navn **svar** som hendelse ved klikk. Vi oppdaterer også poengvisningen. Etter å ha koblet en hendelsesfunksjon ved navn **neste** til knappen tvinger vi denne hendelsen til å utføres også før knappen blir trykket, slik at første spørsmål vises. Å tvinge frem utførelsen gjøres ved å skrive **neste()**. Dette er et såkalt *funksjonskall*, som vi skal omtale mer i neste kapittel.

I hendelsen **neste** øker vi **sporsmaalIndex** med 1, samt setter teksten og de tre alternativene. Dersom vi er på siste spørsmål, gjør vi også knappen for neste spørsmål ultigjengelig.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Ettersom hendelsesfunksjonen **svar** er koblet til alle de tre ``-elementene, benytter vi egenskapen **target** hos hendelsesparametren **evt** til å finne ut hvilket element som faktisk forårsaket hendelsen. Ved å gjøre tre tester kan vi sette en variabel kalt **svar** til hhv. 1, 2 eller 3 ettersom hvilket element som ble trykket.

Deretter sammenligner vi det brukeren svarte, med det riktige svaret for det aktive spørsmålet. Er det rett, setter vi ``-element som ble trykket sitt klassenavn til **riktig**; ble det feil, setter vi det til **feil**.

All koden i hendelsen **svar** skal kun utføres dersom brukeren ikke alt har svart. Derfor gjør vi en test på om variabelen **harSvart** er **false**, før vi utfører koden. Som en del av kodeutførelsen settes den deretter til **true**.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

7 Funksjoner

I dette kapitlet vil du lære

- hva en funksjon er
- hvordan man skriver egne funksjoner
- om programflyt og funksjoner
- noen retningslinjer for å lage gode funksjoner
- om rekursive funksjoner
- om innebygde funksjoner

La oss starte dette kapitlet med en forenklet forklaring på hva en *funksjon* egentlig er. Tenk deg at du kommer over en smart måte for å finne det minste av to tall. Dette ønsker du å gjøre mange steder i programkoden din, så du kopierer kodelinjene til alle disse stedene. Etter hvert finner du imidlertid ut at det er en liten feil i denne programkoden. Dermed får du nå jobben med å påse at endringene du gjør ett sted, også gjøres alle steder der kopien av denne programkoden står.



minsteAvToTall

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Det hadde vært bedre å kunne gi disse kodelinjene et kallenavn, slik at du i din programkode kunne referert til kodelinjene med dette navnet i stedet for å kopiere dem. På denne måten ville det kun vært ett sted å endre kodelinjene, og endringene ville blitt reflektert alle de stedene du refererte til dette navnet. Dette er i bunn og grunn hva en funksjon er.

I programkoden kan vi *definere* en funksjon med et gitt navn og fortelle hvilke instruksjoner denne funksjonen skal bestå av. Deretter kan vi be maskinen om å utføre funksjonen hver gang vi ønsker å benytte denne programkoden. Dette omtales som et *funktionskall*, eller å *kalle opp* funksjonen.

Du kan også tenke på en funksjon som en gruppering av enkle instruksjoner til å forme en ny og mer avansert instruksjon/kommando. Riktignok er ikke dette noen absolutt definisjon, for funksjoner kan gjøre så mye mer. Det er allikevel en tanke du kan holde på inntil du har fått mer forståelse for hva funksjoner er, og hvordan de benyttes.

En annen stor fordel med funksjoner er at de gir oss mulighet til å dele opp programmet i mindre deler. Hver funksjon tar seg av noen få oppgaver i programmet. Dermed kan vi lage programmet del for del – nesten på samme måte som å sette sammen byggeklosser. Dette gjør at vi beholder oversikten og ikke så lett blir overveldet hvis vi skal lage et større program.

Funksjoner kalles av den grunn også *delprogrammer*. I stedet for å bestå av mange forvirrende instruksjoner kan programkoden ved hjelp av funksjoner heller bestå av noen få beskrivende funktionskall. Lurer man på hva en funksjon egentlig gjør, kan man da gå og se på *funktionsdefinisjonen*.

Som du vil se, har vi allerede benyttet funksjoner i forbindelse med hendelser. Dette kapitelet skal imidlertid fokusere på å lage funksjoner som kan benyttes uten noen direkte kobling til hendelser.

TIPS

Funksjoner er ikke alltid det helt korrekte navnet i en objektorientert sammenheng. Vi skulle egentlig gjort et skille mellom *funktions* og *metoder* i programkoden. Metoder har allikevel så mye med begrepet *klasse* å gjøre at vi holder oss til betegnelsen funksjoner foreløpig, inntil vi får lært litt mer om objektorientert programmering.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Enkle funksjoner

Etter å ha gitt en deg en idé om hva en funksjon er, skal vi se på et helt enkelt eksempel. Vi definerer her en funksjon med navn **skrivUtHilsen**, som skriver ut meldingen *Hei!* i en meldingsboks.

```
function skrivUtHilsen() {  
    alert("Hei!");  
}  
  
function oppstart() {  
}
```

Kjører vi koden slik, vil det imidlertid ikke skje noe som helst. Det er fordi funksjonen kun er definert og ikke blir utført.

Pass på at du ikke definerer en funksjon inne i (altså mellom { og }) en annen eksisterende funksjon. Dette er både mulig og lovlig å gjøre i JavaScript, men vil kunne gi en del problemer som er vanskelige å løse på det nivået vi nå programmerer.



Funksjonsnavn følger samme navngivningsregler som variabelnavn, dvs. liten forbokstav, og eventuelt med camelCase-stil dersom det er flere ord.



Vi ber systemet om å utføre funksjonen (kalle opp funksjonen) ved å skrive funksjonsnavnet etterfulgt av to parenteser, på samme måte som da vi definerte funksjonen.

```
function skrivUtHilsen() {  
    alert("Hei!");  
}  
  
function oppstart() {  
    skrivUtHilsen();  
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Selv om de foreløpig virker ganske unødvendige, er det viktig å huske på parentesene i et funksjonskall. Det er nemlig parentesene som gjør at systemet gjenkjenner dette som en funksjon og ikke en variabel.

Vi kan kalle funksjonen hvor mange ganger vi måtte ønske, og den blir da utført én gang for hvert kall.

```
function skrivUtHilsen() {  
    alert("Hei!");  
}  
  
function oppstart() {  
    skrivUtHilsen();  
    skrivUtHilsen();  
    skrivUtHilsen();  
}
```



Vi kan også kalle opp en funksjon tidligere i kodefila enn der vi skriver selve funksjonsdefinisjonen. Grunnen til dette er at alle funksjonsdefinisjoner blir behandlet først, uansett rekkefølge i koden.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Parametere

Til nå har vi kun sett på funksjoner som gjør eksakt det samme hver gang de blir kalt. Vi kan imidlertid også lage funksjoner som gjør jobben på litt ulike måter, avhengig av verdien på såkalte *parametere* til funksjonen.

En parameter er en verdi som funksjonen får tilsendt idet den blir kalt.

Funksjonsdefinisjonen blir da å regne som en slags mal, der verdien i parameteren avgjør hvordan det ferdige produktet skal se ut.

La oss illustrere parametere ved å utvide funksjonen **skrivUtHilsen**, som vi alt har lagd. Vi ønsker fortsatt å skrive ut en hilsning, men funksjonen skal nå ikke inneholde et fastsatt hilsningsord. Hva som skal benyttes som hilsningsord, skal i stedet bli sendt med funksjonskallet ved hjelp av en parameter. Eksempler på hilsningsord kan da være *hei*, *hallo*, *heisann* osv.

Vi må dermed definere funksjonen med en parameter som vi kaller **hilsningsord**. Som du ser, benyttes parameteren som om den var en vanlig lokal variabel inne i kodeblokken til funksjonen:

```
function skrivUtHilsen(hilsningsord) {
    alert(hilsningsord + "!");
}
```

Når vi så skal benytte funksjonen, kan vi gjøre det på følgende måte:

```
function oppstart() {
    skrivUtHilsen("Hallo");
    skrivUtHilsen("Hei");
}
```

Fordelen nå er at vi enkelt kan bytte hilsningsord i akkurat dette kallet på funksjonen, uten å skrive om selve funksjonen. Den faktiske verdien vi sender med til funksjonen, kalles et argument.

Det er verdt å merke seg forskjellen på begrepene *parameter* og *argument*. En parameter er den formelle definisjonen, mens et argument er den faktiske verdien i funksjonskallet.

En funksjon kan også ta flere parametere, slik at det er flere ulike elementer vi kan endre på. Vi kan for eksempel utvide funksjonen til også å inkludere en parameter for navn på en person:

```
function skrivUtHilsen(hilsningsord, navn) {
    alert(hilsningsord + " " + navn + "!");
}
```

Funksjonen må nå kalles med samme antall parametere og rekkefølge:

```
function oppstart() {
    skrivUtHilsen("Hallo", "Ole");
    skrivUtHilsen("Hei", "Tom");
}
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Eksempel - Sirkelfunksjon

Vi skal i dette eksempelet lage en funksjon som tegner sirkler inne i sirkler. Som parametere har vi x- og y-koordinatene for senterpunktet til sirklene, radius på den ytterste sirkelen, antall sirkler og fargene som skal benyttes.

I praksis lager vi nå en "utvidelse" av hva som allerede finnes i JavaScript for å tegne med. Vi lager en ny funksjon som er basert på andre eksisterende funksjoner.

1. Lag et nytt HTML-dokument du kaller *sirkeltogner.html* ut fra *mal.html*.
2. Legg inn et canvas som vi skal tegne på:

```
<body>
    <canvas id="tegneflate" width="400" height="400" />
</body>
```

3. La canavset få en kantlinje rundt seg:

```
<style>
    #tegneflate {border-style:solid;}
</style>
```

4. Lag funksjonen som tegner en rekke sirkler utenpå hverandre. Funksjonen tar som parametere x, y, radius på ytterste sirkel, antall sirkler, en array med fargenavn som skal benyttes, og contexten som det skal tegnes på:

```
function sirkler(x, y, radius, antall, farger, ctx) {
    var radiusTrekk = radius / antall;
    var teller = 0;
    while (teller < antall) {
        ctx.strokeStyle = farger[teller % farger.length];
        ctx.beginPath();
        ctx.arc(x, y, radius, 0, 2 * Math.PI);
        ctx.stroke();
        radius = radius - radiusTrekk;
        teller++;
    }
}
```

5. Legg inn kode i **oppstart**-funksjonen som benytter funksjonen **sirkler** til å tegne noen testsirkler:

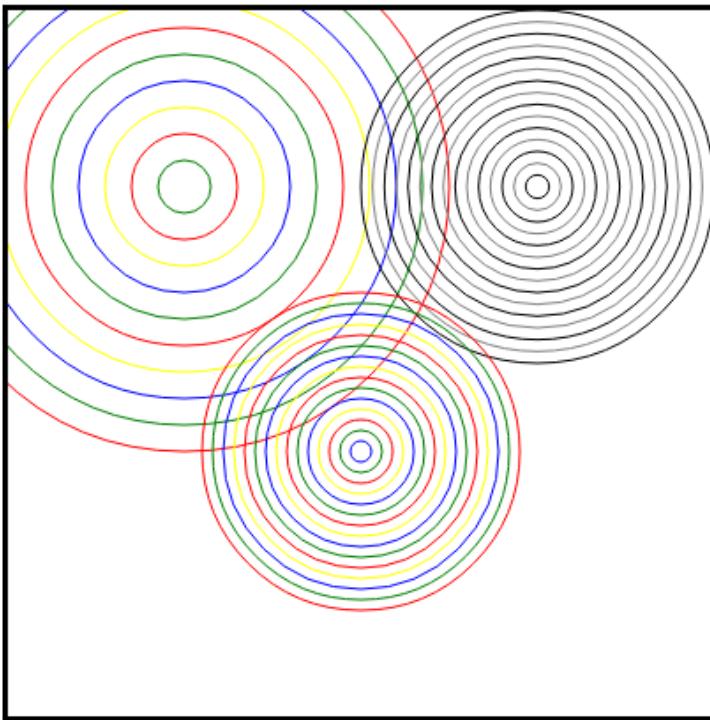
```
window.onload = oppstart;

function oppstart() {
    var ctx = document.getElementById("tegneflate").getContext("2d");
    var farger1 = ['red', 'green', 'blue', 'yellow'];
    var farger2 = ['black', 'gray'];

    sirkler(100, 100, 150, 10, farger1, ctx);
    sirkler(300, 100, 100, 15, farger2, ctx);
    sirkler(200, 250, 90, 15, farger1, ctx);
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

- Test nettsiden, og kontroller at sirklene blir tegnet.



Inne i funksjonen **sirkler** lager vi først noen variabler. Variabelen **radiusTrekk** er verdien vi minsker radiusen med for hver sirkel. Vi lager også en teller. For hver gang programmet går gjennom løkka, settes først en linjefarge.

Å finne linjefargen gjøres ved å plukke ut fargen fra parameteren **farger**. Indeksen det skal plukkes fra, finner vi ved å ta **teller** modulo lengden på arrayen. Er telleren mindre enn lengden, benyttes den som den er. Blir den større enn lengden, trekkes lengden fra så mange ganger det går. Dermed får vi f.eks. ved en lengde 4 på farge-arrayen indeks som er 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3 osv. Dette er et smart triks for å "rullere" rundt i en array.

Deretter tegnes sirkelen med gitt farge og radius. Variabelen **radius** minskes så med **radiusTrekk**, slik at den er klar til å tegne en litt mindre sirkel i neste iterasjon av løkka.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Funksjoner som returnerer verdier

Det finnes i all hovedsak to ulike typer funksjoner. I den første gruppen finner vi dem som utfører en *handling*, slik som hendelsesfunksjonene og funksjonene vi har sett på til nå i kapitlet. Den andre typen, som vi skal se på nå, er funksjoner som *returnerer* en verdi. Funksjoner kan også være av begge typer.

La oss definere funksjonen for minimum av to tall som vi skisserte i introduksjonen til dette kapitlet. Hadde funksjonen ikke skullet returnere noen verdi, kunne den sett slik ut:

```
function minimum(tall1, tall2) {
    if (tall1 < tall2) {
        alert(tall1);
    }
    else {
        alert(tall2);
    }
}
```

Og i bruk kunne funksjonskallet sett slik ut:

```
function oppstart() {
    minimum(3, 7); // Viser 3 i en meldingsboks
}
```

Problemet nå er at vi kanskje ønsker å bruke funksjonen i mange ulike sammenhenger. Det er da ikke sikkert at vi ønsker å vise resultatet i en meldingsboks hver gang. En bedre løsning vil derfor være at funksjonen *returnerer* resultatet slik at vi kan bruke det slik vi vil.

Vi gjør derfor en endring på funksjonsdefinisjonen:

```
function minimum(tall1, tall2) {
    if (tall1 < tall2) {
        return tall1;
    }
    else {
        return tall2;
    }
}
```

Legg merke til nøkkelordet **return** inne i kodeblokken til funksjonen. Det som står etter **return**, er den verdien funksjonen skal returnere.

I bruk vil vi nå få en verdi tilbake fra funksjonen, som vi for eksempel kan lagre i en variabel:

```
function oppstart() {
    var svar = minimum(3, 7); // Tallet 3 lagres i svar
    alert("Svaret ble: " + svar);
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

TIPS

Det er ofte lurt å tenke seg at funksjonskallet blir byttet ut med verdien som funksjonen returnerer.

Vi kan til og med bruke returverdien fra et funksjonskall som argument til et annet. I eksempelet nedenfor finner vi dermed den minste verdien av tre tall. Først finner systemet minimum av 2 og 9, for deretter å finne minimum av 5 og denne verdien.

```
function oppstart() {
    var svar = minimum(5, minimum(2, 9)); // Tallet 2 lagres i variabelen svar
}
```

Her er et annet eksempel på en funksjon med parametere og returverdi som beregner skatt ut fra lønn og skattekoeffisient:

```
function beregnSkatt(loenn, prosent) {
    var skatt = loenn * prosent / 100;
    return skatt;
}
```

Vi kommer inn med to parametere, **loenn** og **prosent**, som brukes for å beregne skatten. Resultatet returneres ved hjelp av kodeordet **return**.

Vi kan så bruke funksjonen på denne måten:

```
function oppstart() {
    var minSkatt = beregnSkatt(250000, 28);
    alert(minSkatt); // Skriver ut 70000
}
```

I utgangspunktet kan ikke en funksjon returnere mer enn én verdi, men denne ene verdien kan for eksempel være en array (som jo består av flere verdier/elementer).

TIPS

MERK

En funksjon vil alltid avbrytes når maskinen støter på en **return**-kommando og returverdi dermed er bestemt. Dersom du ikke benytter kontrollstrukturer, må **return**-kommandoen være siste instruksjon i funksjonen. Benyttes kontrollstrukturer, kan det derimot være mulig å plassere flere **return**-kommandoer i en funksjon, hvorav maksimalt én blir utført.

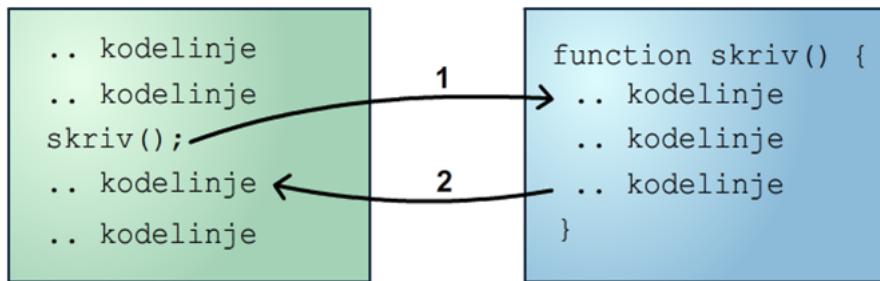
I noen tilfeller kan vi også ha nytte av en tom returverdi. Ved å kun skrive **return** uten verdi et sted i funksjonskoden betyr dette at funksjonen skal avbrytes på dette punktet. Dette trikset benyttes ofte ved unntakstilfeller i store funksjoner som ikke er forventet å produsere noen verdi tilbake, og kan sammenlignes med nøkkelordet **break** i løkker.

TIPS

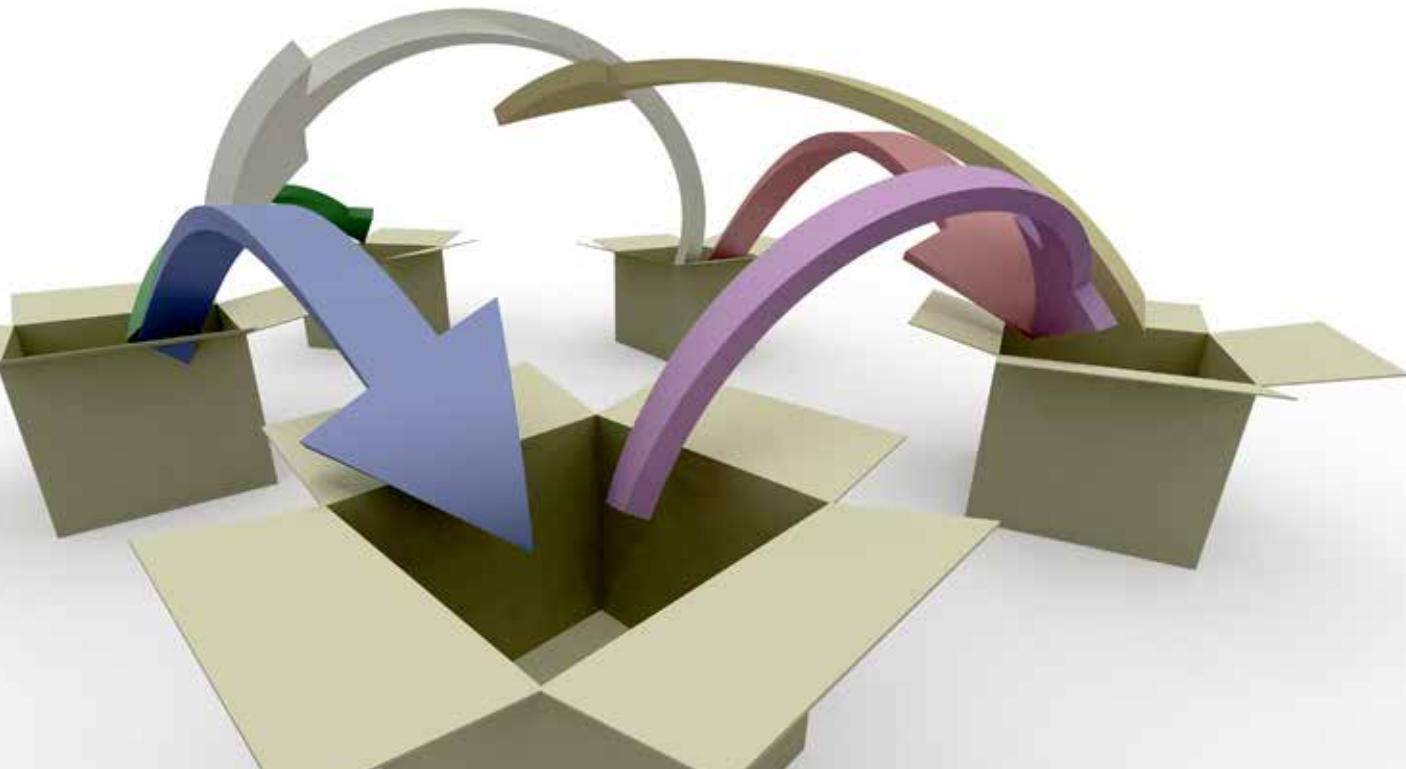
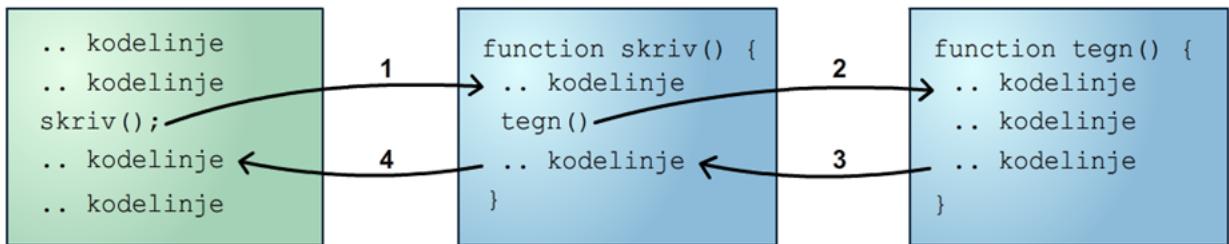
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Programflyt

Vanligvis tenker vi oss at et program blir kjørt linje for linje i den rekkefølgen de står i programmet. Dette blir annerledes når vi bruker funksjoner. Etter at vi har kalt opp en funksjon, vil programmet hoppe til den første kodelinjen inne i funksjonen og kjøre videre derfra. Når programmet er ferdig med siste linje i funksjonen eller møter på kodeordet **return**, hopper programmet tilbake og kjører kodelinjen etter der vi kalte opp funksjonen:



En funksjon kan også kalle opp en annen funksjon osv.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Globale og lokale variabler

Vi nevnte alt i kapitelet om variabler forskjellen på globale og lokale variabler. Nå som du har fått en større forståelse for funksjoner, repeterer vi imidlertid dette kort.

Variabler som er definert utenfor en funksjon, vil også gjelde inne i alle funksjoner i koden. Det vil si at hvis vi forandrer på verdien til en variabel inne i funksjonen, vil den nye verdien også gjelde utenfor. Variabler som lages utenfor funksjoner, kalles ofte *globale variabler*. I eksempelet nedenfor lager vi variablene `minVerdi` utenfor funksjonen. Inne i funksjonen `leggTilEn` øker vi verdien på `minVerdi` med 1, og vi benytter variablene til utskrift i funksjonen `oppstart`:

```
var minVerdi = 5;

function leggTilEn() {
    minVerdi = minVerdi + 1;
}

function oppstart() {
    leggTilEn();
    alert(minVerdi); // 6
}
```

Hvis vi derimot definerer variabler inne i en funksjon, vil de bare gjelde der. Slike variabler kalles ofte *lokale variabler*. I eksempelet nedenfor lager vi variablene `minTekst` inne i funksjonen. Hvis vi prøver å referere til `minTekst` utenfor funksjonen eller i andre funksjoner, får vi en feilmelding:

```
function settTekst() {
    var minTekst = "Hallo!";
}

function oppstart() {
    settTekst();
    alert(minTekst); // Feilmelding
}
```

Dersom du definerer både en lokal og en global variabel med samme navn, vil ikke dette generere noen feilmelding. Internt i funksjonen vil det imidlertid være den lokale variablene som du henviser til. Dette er en kilde til feil som er vanskelig å finne.



Lokale variabler som er lagd inne i en funksjon, blir slettet med en gang systemet er ferdig med å kjøre funksjonen. Dette gjør at vi kan kalle opp en funksjon flere ganger uten at vi lager duplikater av variabler. For å ta vare på verdier fra gang til gang som funksjonen blir kalt, må variablene derfor være globale.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Umiddelbart kan det virke som en god fremgangsmåte alltid å lage variabler globale. Skal vi skrive god programkode, skal imidlertid funksjoner helst få inn alle verdier gjennom paramete, og gi ut verdier gjenom returverdier. Internt kan de arbeide med lokale variabler, med andre ord være mest mulig isolerte. Globale variabler skal derfor kun benyttes når andre løsninger er mindre hensiktsmessige. Mer om dette når vi snart skal se på retningslinjer for funksjoner.

Rekursive funksjoner

Det er mulig å la en funksjon kalle opp seg selv. Dette kalles en *rekursiv funksjon*. Vi kan for eksempel skrive:

```
function rekursivFunk(tall) {
    document.getElementById("utskrift").innerHTML += tall + "<br />";
    tall--;
    if(tall > 0) {
        rekursivFunk(tall);
    }
}

function oppstart() {
    rekursivFunk(10); // Tallene 10 til 0 skrives ut
}
```

Her lager vi en funksjon hvor vi kommer inn med et tall som parameter. Først skrives tallet ut, og så trekkes 1 fra tallet. Så sjekker vi at tallet fortsatt er større enn 0. Hvis testen slår til, kaller funksjonen opp seg selv, hvis ikke, returnerer programmet til der funksjonen ble kalt opp. Når vi kaller opp funksjonen med verdien 10, blir tallene 10 til 0 skrevet ut.

Som du ser, imiterer vi her oppførselen til en løkke ved hjelp av en if-test og en funksjon. De fleste problemer, slik som dette, vil være enklere å løse med ordinære løkker, men enkelte typer problemer vil være svært mye enklere løst med rekursive funksjoner.

TIPS

For at programmet skal vite hvor det skal hoppe tilbake til etter at vi har kalt opp en funksjon, blir posisjonen i programmet lagret før programmet hopper til funksjonen. Denne lista med programposisjoner kalles en *stack*.

Hvis vi lager en rekursiv funksjon som aldri slutter å kalle på seg selv, vil systemet bli overfylt med funksjonskall, og vi får samme effekt som en uendelig løkke. Dette omtales som en *stack overflow*.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Funksjon som verdi

En spesiell egenskap ved funksjoner i JavaScript er at de også kan oppstre som en verdi på samme måte som et heltall eller en tekststreng. Vi kan f.eks. lagre referanser til funksjoner i en variabel:

```
function summer(tall1, tall2) {
    return tall1 + tall2;
}

function oppstart() {
    var mattefunksjon = summer;

    var verdi = mattefunksjon(3, 6) // 9

}
```

I eksempelet over er effekten kun at vi lagrer funksjonen under et midlertidig navn. Denne teknikken benyttes flittig når vi kobler hendelser til elementer. I en variabel/egenskap lagres en referanse til funksjonen som skal benyttes, slik som:

```
document.getElementById("btnUtfør").onclick = utfør;
```

Når vi benytter funksjoner som en verdi, har vi ikke med parenteser eller argumenter. Vi benytter kun selve navnet. Det er fordi vi ikke ønsker å kalle funksjonen, men referere til den.



Det er også mulig å lage en array av funksjoner:

```
function summer(tall1, tall2) {
    return tall1 + tall2;
}

function multipliser(tall1, tall2) {
    return tall1 * tall2;
}

function oppstart() {
    var mattefunksjoner = [summer, multipliser];

    for(var i = 0; i < mattefunksjoner.length; i++) {
        alert(mattefunksjoner[i](3, 5)); // 8 i første melding, 15 i neste
    }
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Vi kan også overføre en funksjon som et argument til en annen funksjon.
Funksjonen **skrivUtRes** tar som første parameter en funksjon som parameter.
I funksjonen **oppstart** kaller vi opp denne funksjonen med **summer** og
multipliser som parameter:

```
function summer(tall1, tall2) {  
    return tall1 + tall2;  
}  
  
function multipliser(tall1, tall2) {  
    return tall1 * tall2;  
}  
  
function skrivUtRes(matteFunksjon, tall1, tall2) {  
    var res = matteFunksjon(tall1, tall2);  
    document.getElementById("utskrift").innerHTML += <br />  
        "Resultatet ble: " + res + "<br />";  
}  
  
function oppstart() {  
    skrivUtRes(summer, 3, 7);  
    skrivUtRes(multipliser, 2, 6);  
}
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

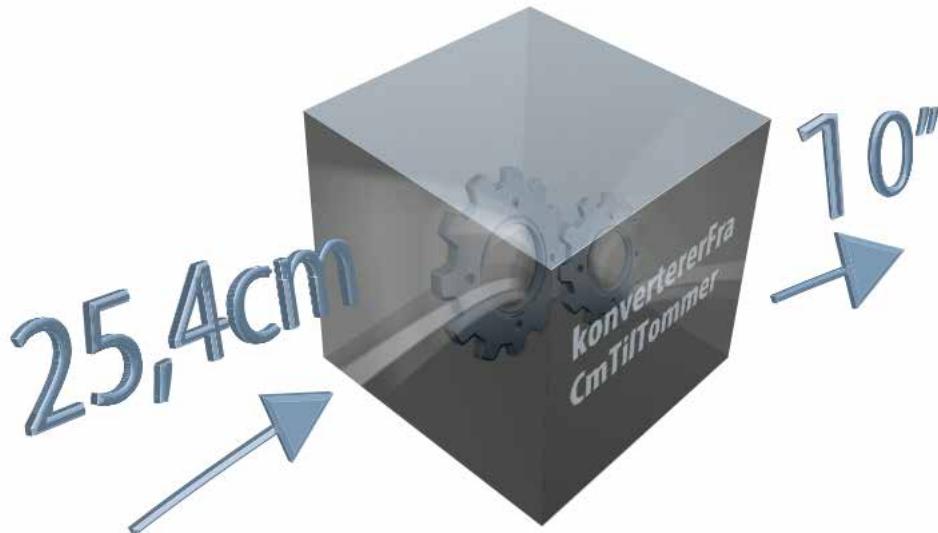
Retningslinjer for funksjoner

Nå som du har fått en liten oversikt over hva funksjoner er og kan brukes til, er det kanskje på tide å se litt på noen retningslinjer for å skrive gode funksjoner. Vi kan imidlertid starte med å oppsummere hovedgrunnene for å samle programkode i funksjoner.

Gjennom dette kapitlet har vi sett flere eksempler på at man kan spare mye skriving ved å lage funksjoner, ettersom programkoden skrives kun én gang og benyttes flere steder. Dette medfører igjen at det blir lettere å lese koden, da man leser funksjonskallene i stedet for alle linjene med programkode som ligger bak.

Funksjoner gir imidlertid også fordeler med hensyn på å redusere sannsynligheten for feil i programkoden, først og fremst fordi at om vi er sikre på at funksjonsdefinisjonen er korrekt, vil funksjonen gi riktig resultat alle steder der den kalles. Man unngår også problemet med at man glemmer å endre én kopi av koden, ettersom det med en funksjon kun er ett sted å endre.

Ofte pleier man å omtale funksjoner som en *svart boks*. Med dette mener man at når funksjonen er ferdig definert, slipper vi å bekymre oss for hvordan den gjør jobben. Det eneste vi trenger å bekymre oss for er hva funksjonen gjør, og hvordan den benyttes. Man sender med andre ord verdier inn som argumenter og får en verdi eller handling ut.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Ofte er det ikke engang vi selv som har lagd funksjonen, og vi vet dermed bare hva den gjør, men ikke hvordan. Dette gjelder for eksempel alle de innebygde funksjonene i JavaScript.

Konseptet med en svart boks krever imidlertid at vi tenker mye på designet av funksjonen. Det er for det første svært viktig å ha et navn på funksjonen som mest mulig eksakt forteller hva den gjør. Navnet *konverter* vil for eksempel være mye mer intetsigende enn navnet *konverterFraCmTilTommer*. Det samme gjelder parameterne, som bør fortelle eksakt hva argumentene som skal sendes med, benyttes til.

Man bør også tenke på at funksjonen på en eller annen måte skal takle alle mulige verdier i parameterne. Det er viktig her å ta høyde for eventuelle ugyldige verdier, slik som for eksempel en negativ verdi for alder.

Funksjoner bør med andre ord være mest mulig generelle og ikke spesialtilpasset akkurat det problemet du jobber med nå. Lager man for eksempel en funksjon som konverterer fra centimeter til tommer, bør den kunne konvertere alle tenkelige verdier for centimeter, ikke bare noen få man hadde bruk for i dette prosjektet. På den måten kan vi enkelt ta med oss funksjonen videre til andre prosjekter som en svart boks, uten å måtte endre den.

For å få generelle funksjoner bør alle verdier funksjonen trenger, komme inn gjennom parametere. Tilsvarende bør verdier som skal ut av funksjonen, gå ut gjennom returverdien. Man bør altså ikke benytte globale variabler inne i funksjonen, ettersom det ikke er sikkert at disse finnes i andre prosjekter der funksjonen kan bli benyttet senere. Vi bør for helt generelle funksjoner også unngå henvisninger til elementer i selve nettsiden (f. eks. skjemaelementer og utskrift).



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Innebygde funksjoner

I kapitlet om arrayer introduserte vi en rekke funksjoner som vi kan benytte oss av for å arbeide med arrayene. Slike sett med innebygde funksjoner finnes også for blant annet matematikk og behandling av tekststrenger.



I noen tilfeller utføres funksjonene direkte på en variabel som har en verdi. Typiske eksempler kan være:

```
var tall = 2.4324435345;  
document.getElementById("utskrift").innerHTML = tall.toFixed(2) // 2.43
```

I andre tilfeller er funksjonene knyttet opp mot et innebygget objekt. Vi har så vidt sett dette i bruk sammen med tilfeldighet og objektet **Math**:

```
var tilfeldigTall = Math.random();
```

Når funksjoner er knyttet opp mot innebygde objekter eller verdier i variabler, så omtales de mest formelt riktig som *metoder*.

TIPS

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Matematikkfunksjoner

Ofte har vi bruk for litt mer avanserte matematikkfunksjoner enn pluss, minus, gange og dele. Vi kan da bruke funksjoner fra det innebygde objektet **Math**.

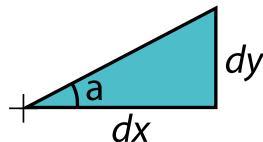
Hvis vi for eksempel skal finne kvadratroten av 16, kan vi skrive:

```
var roten = Math.sqrt(16); // Gir 4
```

sqrt er altså en funksjon/metode som beregner kvadratroten av et tall og returnerer svaret.

Andre nyttige matematikkfunksjoner er:

- **Math.pow(tall1, tall2)** – returnerer *tall1* opphøyd i *tall2*.
- **Math.round(tall)** – avrunder *tall*.
- **Math.floor(tall)** – runder *tall* nedover.
- **Math.ceil(tall)** – runder *tall* oppover.
- **Math.abs(tall)** – returnerer absoluttverdien av *tall*.
- **Math.max(tall1, tall2)** – returnerer det største tallet av *tall1* og *tall2*.
- **Math.min(tall1, tall2)** – returnerer det minste tallet av *tall1* og *tall2*.
- **Math.sin(a)** og **Math.cos(a)** – returnerer sinus- og cosinus-verdien til en vinkel. Vinkelen *a* oppgis i radianer.
- **Math.atan2(dy, dx)** – returnerer vinkelen i radianer som dannes av katetene *dx* og *dy* i en rettvinklet trekant (se figur).



I tillegg til funksjoner inneholder **Math** også en del ferdigdefinerte verdier/konstanter, slik som **PI** og **E**:

```
var omkrets = Math.PI * 4 * 4;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tilfeldige tall

For å lage tilfeldige tall kan vi bruke funksjonen `random()`. Funksjonen lager et tilfeldig tall fra og med 0 til 1 ($0 \leq \text{tall} < 1$). Tallet vil altså aldri bli eksakt 1.

```
var tall = Math.random(); // Gir f.eks. 0.6453
```

Vi kan utvide og forskyve intervallet med litt enkel matematikk. Skal vi f.eks. ha et intervall som er 20 i størrelse i stedet for 1, ganger vi med 20, og får tall fra og med 0 til 20:

```
var tall = Math.random() * 20; // Gir f.eks 17.8463
```

Ønsker vi tall fra og med 10 til 30, kan vi plusse på 10 til intervallet:

```
var tall = Math.random() * 20 + 10; // Gir f.eks 25.4823
```

For å få hele tall benytter vi funksjonen `floor` som også er en del av `Math`. Denne "hugger vekk" desimalene ved positive tall. Ettersom den øvre grensen ikke er inkludert (maksimalt kunne vi få 29.999999), får vi 10, 11, 12, 13, ..., 26, 27, 28, 29 som mulige verdier. For å få intervallet til å også inkludere 30, må vi derfor benytte 21 som "bredde" på intervallet:

```
var tall = Math.floor(Math.random() * 21) + 10; // Gir f.eks 27
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

String-funksjoner

Vi kan bruke funksjoner tilhørende string-verdier for å manipulere tekststrenger ytterligere. For eksempel kan vi hente ut deler av en tekststreng ved hjelp av funksjonen som heter **substr**:

```
var minString = "Hei på deg";
var uttrekk = minString.substr(7, 3); // deg
```

Her henter vi ut en del fra **minString** som starter fra og med tegn nr. 7 og 3 tegn i lengde. Resultatet blir deg. Vær oppmerksom på at første tegn i tekststrenger har indeks 0. Dropper vi siste argument, vil den klippe fra startposisjonen og frem til slutten av strengen.

Her er en oversikt over noen av funksjonene i **String**-klassen:

- **slice(startindeks, sluttindeks)** – fungerer likt som **substr**, men kan i tillegg ta negative verdier. F.eks. vil -6 som *startindeks* og ingen *sluttindeks* gi deg de 6 siste tegnene.
- **charAt(index)** – returnerer tegnet på angitt *index*.
- **indexOf(tekst, startindeks)** – leter etter stringen *tekst* fra *startindeks* og utover. Hvis teksten blir funnet, returneres posisjonen, hvis ikke, returneres -1. *startindeks* kan sløyfes dersom du ønsker å starte søket på første posisjon.
- **lastIndexOf(tekst, startindeks)** – samme funksjon som **indexOf**, bortsett fra at vi leter fra høyre mot venstre.
- **split(skilletegn)** – splitter opp stringen på angitt *skilletegn* og returnerer en array med delstringene.
- **replace(søkestreng, erstatning)** – erstatter alle forekomster av *søkestreng* med angitt *erstatning*.
- **toLowerCase()** – returnerer en string med alle tegnene konvertert til små bokstaver.
- **toUpperCase()** – samme som **toLowerCase()**, men til store bokstaver.
- **charCodeAt(index)** – returnerer tegnkoden til tegnet med angitt *index*. Alle tegn har en tegnkode (heltall) som brukes internt i datamaskinen.
- **fromCharCode(tegnkode)** – returnerer tegnet som ligger bak tegnkoden. Denne funksjonen benyttes på **String**-objektet, da den ikke krever at du har en string fra før:

```
var tegn = String.fromCharCode(64); // gir @
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Kryptering/Dekryptering



I dette eksempelet skal vi benytte funksjoner vi selv lager til å kryptere og dekryptere en tekst. Dette gjøres ved hjelp av en hemmelig nøkkel, som ganske enkelt er et tall.

Hensikten med dette eksempelet er å få trening i å bryte ned problemer i enklere deler, som hver kan skrives som en funksjon. I tillegg vil du gjennom eksempelet bli bedre kjent med å behandle tekststrenger. Før vi går i gang med eksempelet, må vi imidlertid forklare litt om hvordan krypteringen skal foregå.

I datamaskiner er alle tegn knyttet opp til en tallverdi. Mer korrekt vil det kanskje være å si at datamaskiner kun arbeider med tall, mens vi ser disse tallene som tegn på skjermen. Et utdrag av dette såkalte tegnsettet finner du i følgende tabell:

Verdi	Tegn								
33	!	54	6	75	K	96	`	117	u
34	"	55	7	76	L	97	a	118	v
35	#	56	8	77	M	98	b	119	w
36	\$	57	9	78	N	99	c	120	x
37	%	58	:	79	O	100	d	121	y
38	&	59	;	80	P	101	e	122	z
39	'	60	<	81	Q	102	f	123	{
40	(61	=	82	R	103	g	124	
41)	62	>	83	S	104	h	125	}
42	*	63	?	84	T	105	i	126	~
43	+	64	@	85	U	106	j		
44	,	65	A	86	V	107	k		
45	-	66	B	87	W	108	l		
46	.	67	C	88	X	109	m		
47	/	68	D	89	Y	110	n		
48	0	69	E	90	Z	111	o		
49	1	70	F	91	[112	p		
50	2	71	G	92	\	113	q		
51	3	72	H	93]	114	r		
52	4	73	I	94	^	115	s		
53	5	74	J	95	_	116	t		

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Vår enkle kryptering skal fungere slik at vi tar tallverdiene til hvert tegn og øker disse med en gitt nøkkelverdi. Dermed får vi et tilsvarende antall nye tallverdier som vi kan vise tilhørende tegn for. For å dekryptere meldingen gjør vi ganske enkelt denne prosessen motsatt vei.

Nøkkel = 6

Opprinnelig melding:						Kryptert melding:					
N	G	R	R	U	'	H	A	L	L	O	!
78	71	82	82	85	39	72	65	76	76	79	33

1. Lag et nytt HTML-dokument du kaller *kryptering.html* ut fra *mal.html*.
2. Lag følgende brukergrensesnitt bestående av to tekstbokser, to knapper og en paragraf for utskrift:

```
<body>
    <p>
        Melding: <input type="text" id="txtMelding" /><br />
        Nøkkel: <input type="text" id="txtNoekkel" /><br />
        <button id="btnKrypter">Krypter</button>
        <button id="btnDekrypter">Dekrypter</button>
    </p>
    <p id="utskrift"></p>
</body>
```

3. La oss starte med å skrive en funksjon som kan omforme ett tegn, ettersom problemet med å omforme en hel melding egentlig består av å gjøre denne operasjonen mange ganger:

```
function omformTegn(tegn, noekkel) {
    var tegnverdi = tegn.charCodeAt(0);
    tegnverdi = tegnverdi + noekkel;
    return String.fromCharCode(tegnverdi);
}
```

4. Ved å benytte funksjonen **omformTegn** kan vi nå lage en ny funksjon som omformer hele meldingen. Fortsett å skrive koden nedenfor:

```
function omformMelding(melding, noekkel) {
    var omformet = "";
    var tegn = melding.split("");
    for (var i = 0; i < tegn.length; i++) {
        omformet += omformTegn(tegn[i], noekkel);
    }
    return omformet;
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

5. Legg deretter til en klikk-hendelse på knappen **btnKrypter**, som benytter funksjonen **omformMelding**:

```
function oppstart() {
    document.getElementById("btnKrypter").onclick = krypter;
}

function krypter() {
    var melding = document.getElementById("txtMelding").value;
    var noekkel = parseInt(document.getElementById("txtNoekkel").value);
    document.getElementById("utskrift").innerHTML = omformMelding(melding, noekkel);
}
```

6. Til slutt legger vi til en tilsvarende klikk-hendelse på knappen **btnDekrypter**, men som gjør operasjonen motsatt vei:

```
window.onload = oppstart;

function oppstart() {
    document.getElementById("btnKrypter").onclick = krypter;
    document.getElementById("btnDekrypter").onclick = dekrypter;
}

function dekrypter() {
    var melding = document.getElementById("txtMelding").value;
    var noekkel = parseInt(document.getElementById("txtNoekkel").value);
    noekkel = -noekkel;
    document.getElementById("utskrift").innerHTML = omformMelding(melding, noekkel);
}
```

7. Test nettsiden, og forsøk å kryptere meldingen "ABC" med nøkkelen 23. Forsøk deretter å dekryptere svaret du fikk, med samme nøkkel og kontroller at du får ut den opprinnelige meldingen.

Melding:	<input type="text" value="ABC"/>
Nøkkel:	<input type="text" value="23"/>
Krypter	Dekrypter

XYZ

Melding:	<input type="text" value="XYZ"/>
Nøkkel:	<input type="text" value="23"/>
Krypter	Dekrypter

ABC

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Vi begynner denne forklaringen av eksempelet med å se på funksjonen **omformTegn**. Det første denne funksjonen gjør, er å hente ut tegnkoden til tegnet som kommer i parameteren og lagrer denne i variabelen **tegnverdi**.

Vi øker deretter verdien i variabelen **tegnverdi** med verdien i parameteren **noekkel**, og til slutt omformer vi den nye tegnkoden tilbake til en bokstav igjen. Ser vi videre på funksjonen **omformMelding**, er det første vi gjør, å opprette en variabel ved navn **omformet**. I denne variabelen skal vi legge tegnene fra parameteren **melding** etter hvert som de blir omformet.

Deretter benytter vi **split** på parameteren **melding**. Dette er en innebygd funksjon i alle tekststrenger som omformer innholdet til en array. Siden vi angir en tom tekststreg som deletegn, splitter den opp tekstustringen i alle tegn meldingen består av. Vi får med andre ord ut en array der hvert tegn i den opprinnelige tekstustringen er ett element.

Vi itererer så gjennom denne arrayen tegn for tegn og omformer hvert tegn ved hjelp av funksjonen **omformTegn**. Returverdiene legges hele tiden til på slutten av tekstustringen **omformet**, og når det ikke er flere tegn igjen å omforme, returnerer vi denne tekstustringen ut av funksjonen.

Klikk-hendelsen til knappen **btnKrypter** henter ut meldingen som skal krypteres fra den ene tekstboksen, og nøkkelen som konverteres til en heltallsverdi(integer) fra den andre. Disse verdiene legges i to variabler, og paragrafen **utskrift** sitt innhold settes til svaret på funksjonen **omformMelding** med disse to variablene som argumenter.



Knappen **btnDekrypter** sin klikk-hendelse skiller seg ikke så mye fra den for **btnKrypter**, annet enn at vi gjør en ekstra operasjon på variabelen **noekkel**. Vi snur ganske enkelt fortegnet slik at vi kommer tilbake til den opprinnelige verdien for hvert tegn vi omformer med nøkkelen.

Det viktigste du bør huske på fra dette eksempelet, er hvordan man kan bryte ned et avansert problem i mindre deler. Vi lagde før eksempel først en funksjon som omformer ett tegn, og benyttet deretter denne funksjonen for å omforme hele meldingen.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Palindrom



Som et eksempel på rekursive funksjoner og rekursjon skal vi lage en funksjon som finner ut om en tekststreng er et palindrom eller ikke. Et palindrom er en tekst som blir identisk om vi snur den bakvendt. Vi ser bort fra store/små bokstaver når vi skal angi om en tekst er et palindrom eller ikke. Noen eksempler er:

DVD

teppet

Agnes i senga

I dette eksempelet skal vi la et bilde vise enten en grønn V eller en rødt X som ikon.

1. Lag et nytt HTML-dokument du kaller *palindrom.html* ut fra *mal.html*.
2. Legg inn en tekstboks, en knapp og et bilde:

```
<body>
  <p>
    Tekst: <input type="text" id="txtTekst" />
    <button id="btnSjekk">Sjekk</button><br />
    <img src="" id="respons"/>
  </p>
</body>
```

3. Legg to bildefiler kalt *ja.png* og *nei.png* i samme mappe som HTML-fila.
4. Sett en bredde på bildet:

```
<style>
  #respons {width:50px;}
</style>
```

5. Lag den rekursive funksjonen:

```
function palindrom(tekst) {
  tekst = tekst.toLowerCase();
  if( tekst.length < 2 ) {
    return true;
  }
  else if (tekst.slice(0, 1) !== tekst.slice(-1)) {
    return false;
  }
  else {
    return palindrom(tekst.slice(1, -1));
  }
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

6. Koble sammen brukergrensesnittet og den rekursive funksjonen:

```
window.onload = oppstart;

function oppstart() {
    document.getElementById("btnSjekk").onclick = sjekk;
    document.getElementById("respons").style.visibility = "hidden";
}

function sjekk() {
    var tekst = document.getElementById("txtTekst").value;
    if (palindrom(tekst) === true) {
        document.getElementById("respons").src = "ja.png";
    }
    else {
        document.getElementById("respons").src = "nei.png";
    }
    document.getElementById("respons").style.visibility = "visible";
}
```

7. Test nettsiden, og kontroller at palindromsjekken fungerer.

Tekst: <input type="text" value="abba"/>	<input type="button" value="Sjekk"/>
	

Tekst: <input type="text" value="abbaa"/>	<input type="button" value="Sjekk"/>
	

I den rekursive funksjonen **palindrom** gjør vi først om teksten til små bokstaver. Deretter sjekker vi om teksten er mindre enn to tegn, i så fall vet vi at denne teksten er et palindrom, og vi returnerer **true**. Dersom første og siste tegn ikke er like, vet vi at teksten ikke er et palindrom, og vi returnerer **false**.

Dersom teksten er lengre enn to tegn og første og siste tegn var like, så sjekker vi om teksten unntatt første og siste tegn er et palindrom. Vi har med andre ord gjort problemet enklere.



Ettersom vi skal vise et bilde som svar på sjekken, velger vi å sette bildet til å være skjult under oppstarten. Når vi utfører sjekken, endrer vi til riktig filnavn, og deretter setter vi bildet til å være synlig.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

8 Finne og rette feil

I dette kapitlet vil du lære

- om feiltyper
- om feilmeldinger
- hvordan du kan forhindre feil
- å bruke ulike verktøy for å rette feil
- om de mest vanlige feilene som kan oppstå

Dersom du har forsøkt deg på eksemplene i boka og egne programmeringsprosjekter, kan du ikke unngått å ha gjort feil som gjør at programkoden enten ikke vil starte, stopper underveis eller ikke gjør det du hadde tenkt. Dette skjer til tross for at vi allerede i det første kapitlet hadde en introduksjon til problematikken rundt feil i programmer, og har gitt deg tips for å forhindre at du gjør de vanligste feilene.

Grunnen til at vi fokuserer såpass mye på feil i denne boka, er selvfølgelig ikke at vi har dårlig tro på deg som nybegynner i programmering, men snarere at det rett og slett er menneskelig å gjøre feil. Selv de mest erfarne programmerere kan ikke skrive mange linjene med programkode før de gjør en eller annen feil. Det som imidlertid skiller en nybegynner fra en erfaren programmerer, er hvor fort personen klarer å identifisere og rette feilen.

Vi skal i dette kapitlet lære deg en del teknikker som man ofte bruker for å finne og rette feil i programkode.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Feiltyper

Når vi tidligere i boka så vidt omtalte feil, skilte vi mellom to typer, nemlig *syntaktiske* og *logiske* feil. Egentlig skiller vi mellom tre typer feil, der det vi litt enkelt omtalte som syntaktiske feil, egentlig kan deles opp i *syntaktiske* og *semantiske* feil.

Syntaktiske feil

Begrepet *syntaktiske* feil omfatter typografiske feil, eller med andre ord stavefeil. Det kan for eksempel være at vi har skrevet *wihle* i stedet for *while* eller *funcion* i stedet for *function*. Dette er den enkleste formen for feil å finne og rette, ettersom vi vil bli gitt en forholdsvis enkel og grei feilmelding.

Semantiske feil

De semantiske feilene er mer som grammatikken i et språk. Disse feilene oppstår dersom vi setter sammen de ulike elementene av programmeringsspråket på en slik måte at systemet ikke forstår hva det skal gjøre. I motsetning til syntaktiske feil er de semantiske feilene av en slik art at systemet kjenner igjen elementene, men ikke sammensetningen.



Vi kan ha satt sammen elementene slik at de ikke løser problemet de skulle løse, men likevel gir mening til systemet som utfører koden. Dette er da ikke semantiske feil, men snarere logiske feil.

Typiske semantiske feil er at vi glemmer å avslutte en kodeblokk med en }, eller at vi har en betingelse på else-delen av en if-test. Også disse feilene er forholdsvis enkle å rette, ettersom systemet vil klage og gi oss, om noe kryptisk, feilmeldinger på hva det ikke forstår, og hva det forventer å finne.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Logiske feil

I motsetning til de to andre typene feil som oppstår under tolking av programkoden, vil de logiske feilene fint kunne utføres av systemet. De vil derimot vise seg mens programmet kjører, ved at programmet feiler eller ikke gjør det vi hadde tenkt. Som vi skjønner ut fra navnet, er ikke dette feil i selve programkoden, men snarere i logikken bak. Typiske feil kan være at vi har en løkke som aldri stopper, eller feil operatorer i en utregning. Logiske feil som får programkoden til å feile, kalles ofte for *kjørefasefeil*.

Logiske feil er den desidert vanskeligste typen feil å finne, ettersom vi i mange tilfeller ikke får hjelp med hvilken linje feilen har oppstått på, eller hva som er årsaken til feilen.

Det absolutt verste med de logiske feilene er at de ofte kun viser seg i helt spesielle situasjoner. Slike feil forfølger ofte programvaren lenge etter at den er ferdig og solgt til forbrukeren. Et typisk eksempel på dette er nye operativsystemer, som på tross av grundig testing hos leverandøren før lansering ganske snart krever oppdateringer for å fungere slik de var tenkt hos sluttbrukeren.

Nedenfor finner du en oversikt over de vanligste typer logiske feil.

Programmet gir feil resultater

Denne typen feil er veldig vanskelig å finne, ettersom de vanligvis bare gjelder visse verdier. Slike feil kan være svært kostbare hvis de først blir funnet etter at programmet er blitt lansert. Tenk deg for eksempel en nettbutikk som ved en feil gir 90 % rabatt på alle varer til enkelte kunder.

Programmet henger eller krasjer

Denne typen feil kommer som regel av at du har lagd en løkke som går i det uendelige, eller en funksjon som kaller opp seg selv uten å stoppe. Slike feil medfører at programkoden *henger seg*.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Grensesnittfeil

Dette er feil som gjør at brukeren ikke kan benytte programmet på riktig måte, for eksempel at en knapp blir deaktivert for lenge, eller at lister man skal velge noe fra, ikke fylles ut med verdier.

I mange tilfeller kan slike feil være lett å finne ettersom man kan "se" problemet. Det er imidlertid en fare for at feilene kun opptrer ved visse sekvenser av handlinger, og at det dermed blir vanskelig å finne feilene i en testfase der man ofte følger "normalt bruksmønster".

Programmet er for ressurskrevende

Programmet kan ofte fungere slik det skal rent funksjonsmessig, men på grunn av logiske feil i koden går det mye tregere enn det som hadde vært nødvendig. Å lokalisere årsaken til flaskehalsen er imidlertid ikke like enkelt. Ofte er feilen også relatert til skalering, slik at én bruker eller lite data ikke gir noen merkbar økning i ressursbruken. Etter hvert som antall brukere eller mengden data øker, øker imidlertid også behovet for ressurser mer og mer. Under utvikling tester man ofte bare systemet i liten skala, og skaleringsproblemene viser seg ofte kun etter at systemet er tatt i bruk.

Sikkerhetskritiske feil

En siste type feil er de som går på sikkerheten til produktet. Programmet vil fungere utmerket ved vanlig bruk, men det finnes svakheter i produktet som gjør at man

aktivt kan gå inn for å misbruke det. Typiske eksempler her vil være en nettbutikk der man kan omgå betalingen under bestilling av varer, eller en nettbank der man kan komme seg inn på kontoene til andre kunder.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Rette og forhindre feil

Det beste er selvsagt at vi ikke lager noen feil i første omgang. Hvis du følger rådene nedenfor, vil du vanligvis slippe mye frustrasjon og spare tid på feilretting. Enkelte av rådene gjelder mest nybegynnere i programmering, og etter hvert som du blir mer trygg, kan du avvike fra disse.

- **Skriv aldri mer enn noen få linjer med kode før du tester programmet.** Hvis du får feil, vet du at feilen mest sannsynlig ligger i de siste kodelinjene du skrev. En forutsetning for å teste programmet er at koden kjøres. Hvis du for eksempel skriver kode i en funksjon, må du passe på at funksjonen kalles opp slik at du får testet den.
- **Hvis du finner feil, må du rette feilen før du programmerer videre.** Ikke ta lett på feilene og anta at du kan rette dem senere. Når du har mange linjer med kode og ikke vet hvor feilen er, blir det vanskeligere å rette.
- **Hvis du får beskjed om at det er flere feil i programmet, så test programmet på nytt etter at du har rettet den første feilen.** I mange tilfeller kan de andre feilene være følgefeil til den første.
- **Lag alltid en sluttparentes etter at du har lagd en startparentes.** Dette gjør at du unngår parentesfeil.
- **Bruk alltid innrykk etter starten på en funksjon eller kodeblokk.** En slik *indentering* av program gjør det lettere å lese koden og enklere å unngå parentesfeil:

```
var teller = 0;
while (teller < 8) {
    alert(teller);
    teller++;
}
```

- **Benytt alltid krøllparenteser.** Dette gjelder selv om kontrollstrukturen består av kun én instruksjon, og krøllparenteser i teorien kan droppes.
- **Bruk operatorer også for boolske betingelser.** Selv om boolske variabler kan stå alene som en betingelse, er det greit for nybegynnere å tydeliggjøre at vi sjekker om de er true/false. Derfor er det mer hensiktsmessig for nybegynnere å skrive

```
if (riktig === true)      og      if (riktig === false)

enn

if (riktig)  og  if (!riktig)
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

- **Pass på at du navngir riktig.** Du bør følge standard navngivingsregler slik at det blir lettere å lese koden og unngå feil:
 - Bruk liten forbokstav på variabelnavn og funksjonsnavn. Bruk camelCase for å skille ord, for eksempel *antallHusPaaHeia*.
 - Ikke bruk navn som er reservert JavaScript (blir ofte fargelagt i koden).
 - Elementer i brukergrensesnittet/HTML bør ha prefikser som sier noe om hvilken type objekt vi har med å gjøre. For eksempel kan navnet på et tekstfelt være *txtMelding*, eller navnet på en knapp kan være *btnSjekk*.
 - Og kanskje viktigst av alt: Bruk navn som forklarer elementer. Navn som *temp*, *a*, *l2* osv. gjør programkoden vanskelig å lese og få oversikt over.
- **Del opp koden i funksjoner.** Hvis du har mer enn 15–20 linjer med kode i samme kodeblokk, blir det vanskelig å få oversikt over hva koden gjør. Del i stedet koden opp i flere funksjoner. Da blir koden mer oversiktlig og ryddig, og eventuelle feil begrenses til funksjonene. Når du deler opp i funksjoner, bør du passe på at hver funksjon inneholder kode som naturlig hører sammen. Regelen om bruk av funksjoner kan fravikes hvis de fleste kodelinjene gjør det samme, for eksempel å sette verdier på mange elementer i en tabell.
- **Skill funksjonene fra hverandre med kommentarer.** Dette fører til at vi får bedre oversikt over koden:

```
// minFunksjon1 -----
function minFunksjon1() {
    alert("hallo fra minFunksjon1");
}

// minFunksjon2 -----
function minFunksjon2() {
    alert("hallo fra minFunksjon2");
}
```

- **Skill ulike deler av programmet fra hverandre med kommentarer.**

```
///////////
//Innlesningsdel som blablabla...
/////////
kode

///////////
//Beregningsdel som blablabla...
/////////
kode
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

- **Kommenter koden.** Gode kommentarer vil være til stor hjelp når man skal feilsøke, ettersom de viser hva man tenkte da man skrev programkoden. I tillegg oppdager man ofte feil mens man lager kommentarene, ettersom man da må gå gjennom og forklare koden.
- **Sjekk formler og verdier.** Hvis en beregning for eksempel kan føre til at vi forsøker å dele noe på 0, må vi ta høyde for dette i koden. Som regel lager vi if-tester for å sjekke om vi får en ”ulovlig” verdi, slik at vi ikke bruker den videre i programmet.
- **Kontroller løkker.** En løkke som aldri stopper, er en vanlig årsak til at programmer krasjer eller henger. Hver gang du skriver en løkke, bør du ta deg god tid til å forsikre deg om at løkka fungerer riktig under alle forhold.
- **Unngå å kopiere kode.** En av de vanligste feilene er at vi kopierer blokker med kode som skal være nesten like hverandre, men glemmer å gjøre alle endringer i kopien. For å unngå slike feil er den beste løsningen å lage en funksjon. Alternativt kan det faktisk være bedre å skrive av koden enn å kopiere den. Kopier av kode er også notoriske feilkilder når vi senere skal rette litt på koden, men glemmer å rette alle kopiene.
- **Duplikate definisjoner av variabler og funksjoner.** I JavaScript kan du deklarere variabler og funksjoner flere ganger med samme navn. Siste deklarasjon og innhold vil være gjeldende. Dette kan skape mye forvirring da man ser på en definisjon, mens det er en annen som faktisk gjelder. Forsøk derfor å finne et system på oppsettet som gjør at duplike navn er enklere å oppdage fordi de er plassert på tilnærmet samme sted.
- **Kode som ikke blir kjørt.** Av og til forventer du at programmet skal gjøre noe, uten at noe skjer. De vanligste årsakene til dette er:
 - En if-test eller en test i en løkke slår ikke til.
 - Du har glemt å kalle opp funksjonen der koden ligger.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Feilsøkingssverktøy



Det finnes mange verktøy og metoder som gjør feilsøking og -retting enklere. Vi kommer her til å vise disse verktøyene slik de fungerer i Google Chrome.

Kommentere vekk kode

En nyttig teknikk vi har for å finne feil, er å forandre deler av koden til kommentarer slik at denne delen ikke kjøres. Når feilen forsvinner, vet vi at feilen er i koden som er kommentert bort. Ved å gradvis fjerne kommenteringene kan du se når feilen igjen dukker opp.

Kommentarer kan også være nyttige for å teste om programmet hadde fungert bra, dersom visse deler vi vet har feil, ikke hadde skapt problemene de gjør.

Skrive ut verdier

Vi kan finne feil ved å skrive ut verdier underveis i programkoden. Dette kan gjøres ved hjelp av meldingsbokser eller ved å endre innholdet i elementer på nettsiden. JavaScript har imidlertid også en egen funksjon for slike utskrifter i sammenheng med feilretting.

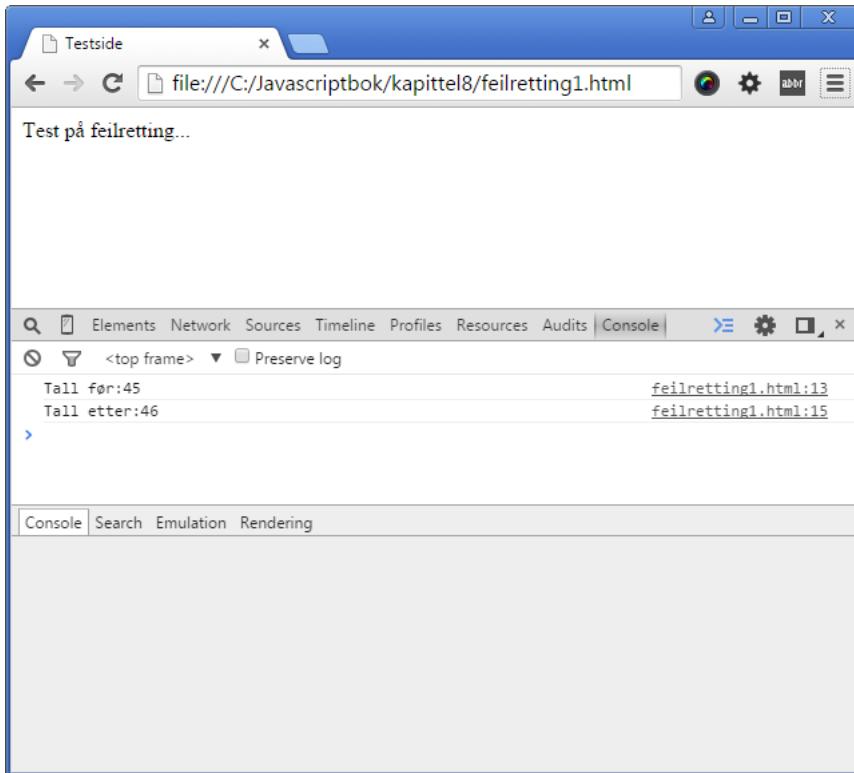
Ved å benytte kommandoen `console.log` sammen med verdier eller tekststrenger vil disse utskriftene dukke opp på et eget panel i nettleseren.

```
var tall = 45;  
console.log("Tall før:" + tall);  
tall++;  
console.log("Tall etter:" + tall);
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

I Google Chrome finner du panelet med utskriftene ved å velge **Fleire verktøy > JavaScript-konsoll**, alternativt **Ctrl + Shift + J**. Utskriftene vises sammen med linjenummeret der de ble utført.



En av fordelene med å benytte `console.log` er at meldingene ikke synes eller påvirker nettsiden for en vanlig bruker. Derfor kan utskriftene ligge der under hele utviklingsfasen i tilfelle det skulle dukke opp flere feil.



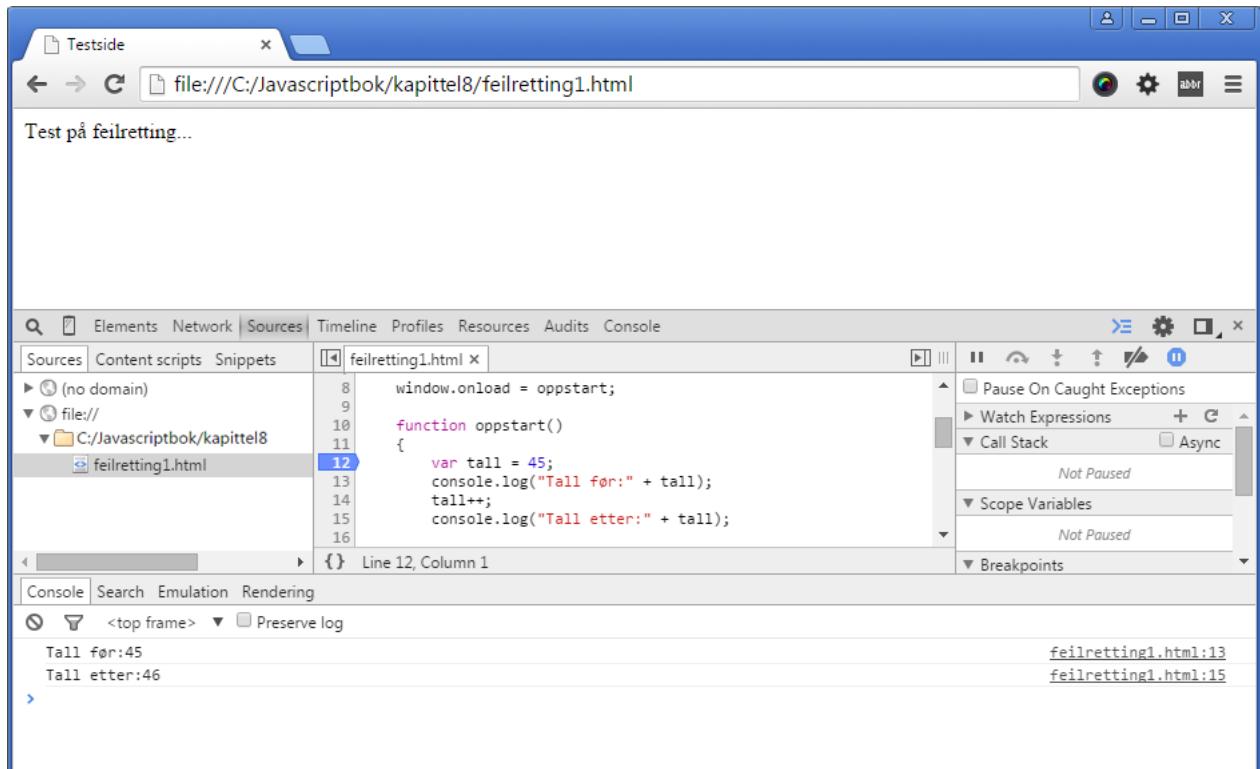
Debuggeren

I små og enkle programmer holder det vanligvis med å bruke `console.log` samt kommentere ut kode for å finne feil. Hvis du jobber med et større prosjekt, kan det være lurt å bruke et mer avansert feilrettingsverktøy. Et slikt verktøy kalles en *debugger*.

Debuggeren gir deg mulighet til å stoppe programmet på bestemte steder, samt kjøre koden linje for linje. Du får også mulighet til å se, og forandre på, verdien til variabler ved hver instruksjon. I tillegg vil programkjøringen stoppe automatisk der det oppstår feil.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

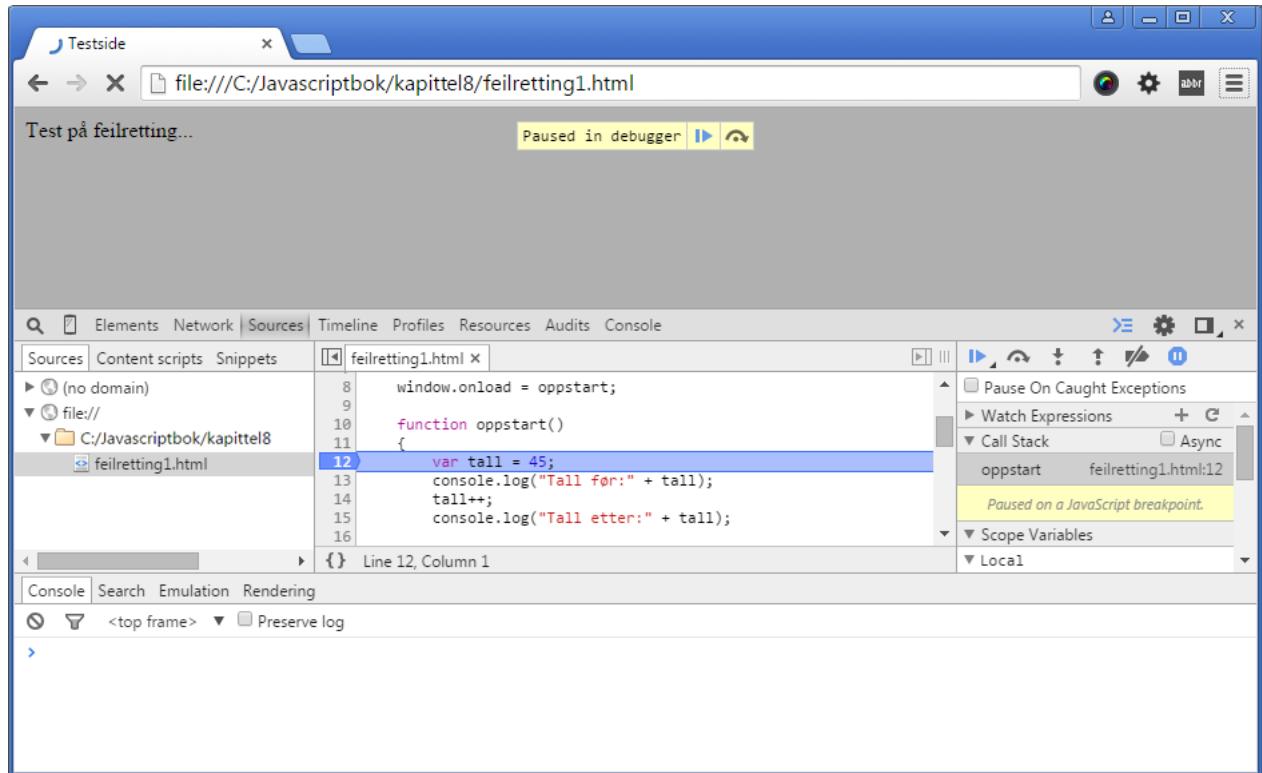
For å bruke debuggeren lager du først ett eller flere stoppunker (*breakpoints*) der du ønsker å stoppe programmet. For å gjøre dette i Chrome må du først åpne kodevisning gjennom valget **Flere verktøy > utviklerverktøy** (alternativt **Ctrl + Shift + I** eller **F12**) og velge fila der JavaScript-koden er plassert. Deretter klikker du i margen på linjen du ønsker at koden skal stoppe på. Stoppunktet markeres med en blå indikator.



Linja du setter stoppunkt på, bør inneholde en instruksjon. Tomme linjer og kontrollstrukturer egner seg dårlig.

For å starte debuggeren laster du nettsiden på nytt (**F5** eller refresh-knappen). Utføringen av koden vil nå stoppe på punktet du har markert. Systemet venter nå på instruksjoner fra deg for å gå videre.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



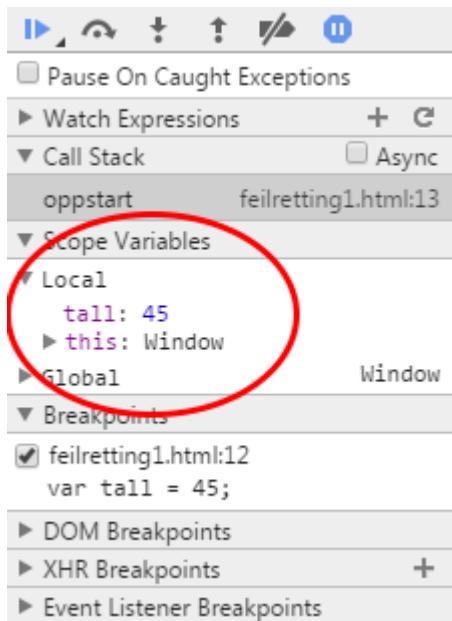
Du kan nå velge mellom flere ulike handlinger:

- 1 Fortsett** (F8) - Resume script execution - Denne handlingen kjører koden som vanlig videre til neste stoppunkt.
- 2 Hopp over** (F10) - Step over next function call - Denne handlingene går en og en instruksjon videre, men ser på funksjonskall som "én instruksjon".
- 3 Gå inn** (F11) - Step into next function call - Ved funksjonskall vil denne handlingen gå én og én instruksjon fremover inne i funksjonsdefinisjonen. For andre instruksjoner, slik som tilordning av verdi til en variabel, vil den fungere likt som hopp over.
- 4 Gå ut** (Shift + F11) - Step out of current function - Har vi benyttet handlingen *gå inn*, kan *gå ut* la resten av funksjonsdefinisjonen utføres og så stoppe på neste linje etter funksjonskallet.
- 5 Fjerne stoppunkerter** - Deactivate breakpoints - Fjerner alle stoppunkerter i koden.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

I tillegg til å kunne styre utførelsen av programkoden kan du også enkelt lese ut verdiene variabler har på aktuelle tidspunkt. Dette kan være et alternativ til å skrive ut verdiene via `console.log`.



Vi kan til og med endre verdier på variabler ved å dobbeltklikke på verdien og så skrive inn en ny verdi. Den nye verdien vil gjelde fra neste instruksjon som utføres. Dette kan være en bra strategi dersom vi ønsker å feilsøke spesialtilfeller eller fremprovosere feil.



Utføre kode fra nettleseren

Debuggeren i Google Chrome lar oss til og med utføre egne instruksjoner i nettsiden som vises. Dermed kan vi mens en nettside kjører, f.eks. gjøre en endring på en verdi eller starte et funksjonskall.

I panelet *Console* kan vi skrive inn instruksjonene vi ønsker utført etter den blå "prompten", og trykke på *return*. Disse vil da bli utført der hvor kodeutførelsen er på det aktuelle tidspunktet. På denne måten kan vi teste endringer i koden eller fremprovosere handlinger, uten å måtte endre den opprinnelige programkoden som er lagret i fila.

```

Elements Network Sources Timeline Profiles Resources Audits Console
<top frame> ▾ Preserve log
> var melding = "Hei";
<- undefined
> alert(melding);
> |

```

The screenshot shows the DevTools console tab. It displays a series of commands entered at the prompt: 'var melding = "Hei";', which results in 'undefined', and then 'alert(melding);'. The cursor is currently at the end of the second line. The tabs at the top include Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console, with Console being the active tab.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Vanlige feilmeldinger

Her er en liten oversikt over feil som ofte oppstår, og tilhørende feilmeldinger. I Chrome vises feilmeldingene i *Console*-panelet.

- **Henvise til en variabel eller funksjon med feil navn.** Dette oppstår ofte når du blander sammen små og store bokstaver eller skriver litt for fort på tastaturet. Du har for eksempel deklarert en variabel med navnet `tall` og henviser til den i koden med navnet `tlal`.

```
var tall = 45;  
alert(tlal);
```

Uncaught ReferenceError: tlal is not defined

Tilsvarende vil et kall til funksjonen `beregn` skrevet feil gi samme feilmelding

```
b ergen(4);
```

Uncaught ReferenceError: bergen is not defined

Funksjoner som er en del av et objekt (såkalte metoder), vil imidlertid gi en noe mer kryptisk feilmelding. Her er `getElementById` skrevet feil.

```
document.getElementById("utskrift").innerHTML = "Test";
```

Uncaught TypeError: undefined is not a function

Merk deg at i enkelte tilfeller vil systemet opprette en ny variabel eller egenskap med det angitte navnet i stedet for å gi deg en feilmelding. Dette gjelder f.eks. ved tilordning av verdier. Dette er ofte vanskelige feil å finne:

```
tlal = 67;  
document.getElementById("utskrift").INNERHtml = "Test";
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

- **Bruke en variabel eller funksjon som ikke er deklarert.** Dette er i bunn og grunn den samme feilen som å skrive feil navn på variabler og funksjoner, og du får de samme feilmeldingene. Du får også denne feilen hvis variablene eller funksjonene er deklarert i et annet virkningsområde.
- **Referanser til elementer i nettsiden som ikke finnes.** Om du refererer til et element med en ID som ikke finnes eller er feilskrevet, vil dette elementet bli omtalt som *null*, og du får en feilmelding om at du ikke kan gjøre operasjoner på et null-objekt. I dette eksempelet henviser vi til **utksrift** i stedet for **utskrift**

```
document.getElementById("utksrift").innerHTML = "Test";
```

Uncaught TypeError: Cannot set property 'innerHTML' of null

- **Parentesfeil.** Manglende parenteser eller for mange parenteser gir vanligvis feilmeldinger som antyder parentesfeil, for eksempel:

Uncaught SyntaxError: Unexpected token {

Uncaught SyntaxError: Unexpected end of input

Uncaught SyntaxError: Unexpected token }

For å luke ut parentesfeil må du gå gjennom koden og pare alle parentesene med hverandre til du finner feilen. Husk på at feil rapporteres for linja der det ble oppdaget at det er en feil. Når det gjelder parentesfeil, er det sjeldent på denne linja der parentesen manglet.



Helt ukritisk å sette inn parentesen på linjenummeret som det refereres til, kan i mange tilfeller fikse den syntaktiske feilen, men lager i stedet en logisk feil.

- **Variabler uten startverdi.** Du må passe på at variablene har en verdi satt før de brukes i beregninger. En vanlig feil er å deklarere en variabel uten å sette startverdi (ikke initialisert), og så sette den lik seg selv:

```
var tall;
tall = tall + 1; //Feil!
alert(tall);
```

Slike operasjoner vil ikke resultere i en direkte feilmelding i konsollet, men i den spesielle verdien *NaN* (Not a Number) for matematiske operasjoner eller *undefined* for string-konkatenering.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

- **Konverteringsfeil.** Hvis du prøver å utføre operasjoner på to verdier som ikke lar seg forene, vil du få `Nan` eller `undefined` som ny verdi

```
var tall = 45;
tall = tall * ",";
alert(tall);
```

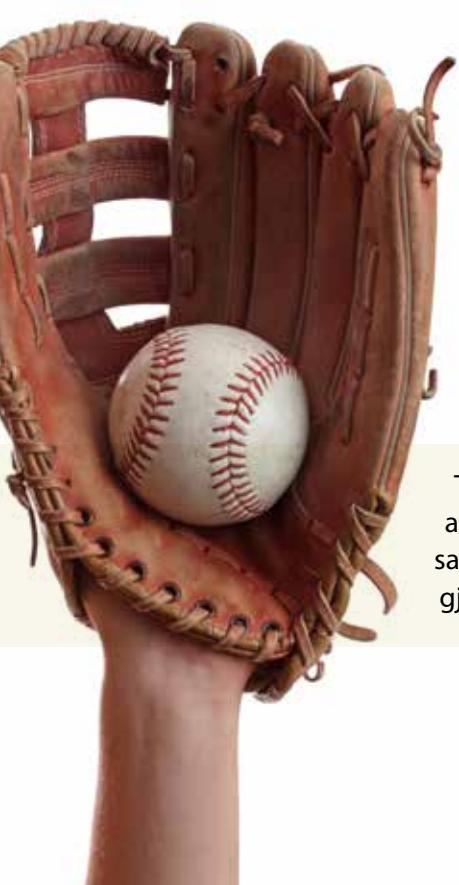
Det kan være lurt å selv introdusere feil i kode, for så å lære seg hvilken type feil som gir hvilken type feilmelding.

TIPS

Fange opp kjørefasefeil

Vi kan fange opp en del kjørefasefeil ved hjelp av en kontrollstruktur som heter `try-catch`. Hvis en feil oppstår i `try`-blokka, hopper programmet direkte til `catch`-blokka:

```
try {
    // Hvis feil oppstår her hopper programmet til catch
}
catch(err) {
    alert(err.message);
}
```



`catch` er nærmest som en funksjon å regne, hvor vi kommer inn med et error-objekt som parameter. Dette objektet inneholder informasjon om feilen.

Vi bruker ofte `try-catch` når vi vet at det kan oppstå feil på grunn av "eksterne problemer", selv om vi har alt riktig i koden. Dette gjelder for eksempel hvis programmet skal laste ned en fil fra nettet. Hvis brukeren ikke har nettilgang eller fila ikke eksisterer, vil det da oppstå en kjørefasefeil som vi kan fange opp med `try-catch`.

Try-catch bør benyttes når vi ikke har oversikt over alle mulige feilkilder, og det som kan gå galt, er mindre sannsynlig. I motsatt fall bør vi gjøre tester før vi forsøker gjøre handlingen.

TIPS

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

9 Hendelser

I dette kapitlet vil du lære

- om hvordan hendelser håndteres i JavaScript
- om musehendelser
- om drag and drop
- om tastaturhendelser
- om tidsstyrte hendelser

Dagens operativsystemer og programmer har som oftest et grafisk brukergrensesnitt som reagerer på at brukeren for eksempel klikker på trykknapper og menyer. Vi sier da at programmet er *hendelsesorientert*, ettersom programflyten i stor grad blir bestemt av hendelser brukeren utfører i programmet.

Hendelser kan også bli utløst av operativsystemet eller programmet i seg selv. For eksempel kan vi få programkoden til å utløse en hendelse når den er klar til å kjøre eller i fastsatte tidsintervaller.

Vi har allerede sett hvordan vi kan registrere hendelser knyttet til musa og til oppstarten/lastinga av en nettside. I dette kapitlet skal vi se nærmere på både musehendelser og andre typer hendelser.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Sender-lytter-modell

JavaScript bruker en *sender-lytter-modell* for å håndtere hendelser. Det vil si at elementet som utløser hendelsen, for eksempel en knapp, kaller opp en eller flere funksjoner som er registrert som *lytterfunksjoner* til knappen. For å angi dette i koden må vi, som vi har sett i denne boka, utføre to steg:

Steg 1 - Lage en lytterfunksjon

Vi lager først en *lytterfunksjon*. Dette er en funksjon som kalles opp når hendelsen oppstår, for eksempel når brukeren klikker på en knapp.

```
function klikk(evt) {  
    alert("Du klikket på knappen");  
}
```

Legg merke til at vi kommer inn med en parameter vi her kaller **evt**. Dette kalles et *hendelsesparameter* eller *hendelsesobjekt* og inneholder informasjon om hendelsen. I JavaScript er det valgfritt om vi har med denne parameteren eller ikke.

Steg 2 - Registrere lytterfunksjonen

Elementet som utløser hendelsen, må vite hvilken lytterfunksjon som skal kalles opp. Vi gjør dette ved å registrere lytterfunksjonen på elementet sin hendelse. Nedenfor registrerer vi lytterfunksjonen til knappen sin **onclick**-hendelse:

```
document.getElementById("btnSjekk").onclick = klikk;
```

En lytterfunksjon kalles også ofte bare en lytter.



Det benyttes ikke to parenteser når vi angir en funksjonsreferanse. Funksjonen skal ikke kalles, da vi kun ønsker referere til den.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Elementer i hendelseshåndteringen

For at sender-lytter-modellen skal fungere, kreves det at vi har en sender og en lytter. I tillegg kan vi ha et hendelsesobjekt med informasjon om hendelsen ut over det faktum at det har hendt:

sender, lytter (lytterfunksjon) og hendelsesobjekt.



Sender

Dette er element som sender beskjed til (kaller opp) lytterne når en hendelse skjer. Som regel er senderen det samme elementet som utløser hendelsen, for eksempel en knapp:

```
document.getElementById("btnSjekk").onclick = klikk;
```

Lytter

Lytteren er funksjonen som kalles opp av senderen når hendelsen skjer. Lytteren må være registrert i senderobjektet for en spesiell hendelse, for eksempel museklikk:

```
document.getElementById("btnSjekk").onclick = klikk;
```

Hendelsesobjekter

Sammen med funksjonskallet til lytteren sendes det med et hendelsesobjekt som inneholder informasjon om hendelsen. Hvis vi for eksempel klikker på en knapp, blir det lagd et hendelsesobjekt med informasjon om museklikket. Knappen overfører hendelsesobjektet som en parameter når den kaller opp lytterfunksjonene:

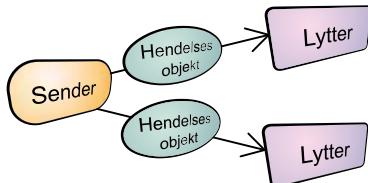
```
function klikk(evt)
{
    alert("Du klikket på knappen");
}
```

Hendelsesobjektene har forskjellige egenskaper alt etter hvilken kategori av hendelser vi ønsker å håndtere. For musehendelser er f.eks. posisjon og museknappene viktige. For tastaturhendelser er tast- og tegnkoder viktige.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Flere lyttere til én sender

Det er mulig å registrere flere lyttere til én sender:



Det vanligste er å registrere flere lyttere som tar seg av forskjellige typer hendelser, for eksempel:

```
document.getElementById("btnSjekk").onmousedown = musNed;
document.getElementById("btnSjekk").onmouseup = musOpp;
```

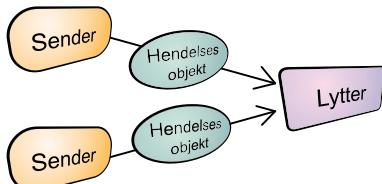
Vi kan også registrere flere lyttere for den samme hendelsen på én sender. Dette krever imidlertid en litt annen måte å koble lytterne på, ettersom vi ikke skal erstatte lytteren som alt er registrert.

```
document.getElementById("btnSjekk").addEventListener('click', klikkEn);
document.getElementById("btnSjekk").addEventListener('click', klikkTo);
```

I dette tilfellet vil både lytterfunksjonen **klikkEn** og lytterfunksjonen **klikkTo** bli utført ved klick på knappen. En av grunnene til å dele opp lytterfunksjonen i to ulike funksjoner kan være at deler av koden skal gjenbrukes også på andre sendere (se neste avsnitt).

Flere sendere til én lytter

Vi kan registrere den samme lytteren til flere ulike sendere:



```
document.getElementById("btnKnappEn").onclick = klikk;
document.getElementById("btnKnappTo").onclick = klikk;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

For å finne ut hvilken sender som kalte opp lytteren, kan vi bruke egenskapen **target** i hendelsesobjektet. Denne egenskapen inneholder en referanse til det elementet som forårsaket hendelsen. I eksempelet nedenfor finner vi ut hvilken knapp som ble klikket på, ved hjelp av en if-test:

```
function klikk(evt) {
    if (evt.target === document.getElementById("btnKnappEn")) {
        alert("Du vant kr 5000,-");
    }
    if (evt.target === document.getElementById("btnKnappTo")) {
        alert("Du tapte");
    }
}
```

Det er også vanlig å skrive kode som manipulerer eller benytter elementet som **evt.target** faktisk refererer til:

```
function klikk(evt) {
    alert("Elementet du trykket på har teksten " + evt.target.innerHTML);
}
```



Denne teknikken er svært nyttig dersom vi har mange ulike elementer i nettsiden som skal gjøre nesten det samme f.eks. rutene i et bondesjakkspill.

Virkning av hendelser

Som standard vil en hendelse også gjelde barn av elementet. Dersom vi for eksempel knytter en klikk-hendelse til en `<div>` som igjen har elementer i seg, vil også disse elementene bli klikkbare.

Egenskapen **target** vil i slike tilfeller referere til elementet som faktisk ble trykket på (altså barna) og ikke elementet der lytteren var registrert. Ønsker vi å sjekke hvilket element lytteren var registrert på, kan vi i stedet benytte **currentTarget**, som i dette tilfelle ville gitt oss `<div>`-taggen.

Avregistrere en lytter

Hvis vi ikke lenger har behov for en lytter, bør vi avregistrere den. Vi gjør dette ved å sette referansen for hendelsen lik den spesielle verdien **null** (altså ingen verdi).

```
document.getElementById("btnSjekk").onclick = null;
```

Dersom du har benyttet **addEventListener** for å registrere lytteren, fjernes den ved å utføre en helt identisk **removeEventListener** for samme hendelse og samme funksjon.

```
document.getElementById("btnSjekk").removeEventListener('click', klikkEn);
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Musehendelser

Vi ønsker ofte at programkoden skal kunne reagere på forskjellige musehendelser.
Vi skal her se nærmere på hvilke muligheter vi har.

Museklikkhendelser

Vi kan fange opp følgende hendelser:

- **onclick** - brukeren klikker med musa.
- **ondblclick** - brukeren dobbeltklikker med musa.
- **onmousedown** - brukeren trykker museknapp ned.
- **onmouseup** - brukeren slipper museknapp opp.

Musebevegelser

Vi kan fange opp følgende hendelser:

- **onmousemove** - brukeren flytter musa over elementet. Denne hendelsen utføres hver gang musepekeren flytter seg til et nytt punkt, så potensielt et stort antall ganger på musepekerens vei over et element.
- **onmouseenter** - brukeren flytter musa fra et sted utenfor elementet til et sted over elementet.
- **onmouseleave** - brukeren flytter musa fra et sted over elementet til et sted utenfor elementet.
- **onmouseover** og **onmouseout** - fungerer på samme måte som **onmouseenter** og **onmouseleave**, bortsett fra at de utføres på nytt for barn til elementer. Et **onmouseover** vil derfor bli utført både når musepekeren kommer inn over selve elementet og for hvert barn vi sveiper over og når vi går tilbake til hovedelementet igjen. Et **onmouseout** vil utføres når vi går fra hovedelementet og over på et barn til elementet, og når vi flytter musa tilbake fra et barn.
- **onwheel** - fanger opp bevegelser til scrollhjulet på musa.

Dersom du ønsker å fange opp musehendelser uavhengig av et bestemt element i dokumentet, kan du registrere hendelsen rett på selve dokumentet:

```
document.onmousemove = musFlyttet;
```

TIPS

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Informasjon i hendelsesobjektet

- **clientX** og **clientY** - koordinatene til musa i nettleserens koordinatsystem.
Øverst til venstre på nettsiden er 0,0
- **screenX** og **screenY** - koordinatene til musa i skjermens koordinatsystem.
Øverst til venstre på skjermen er 0,0
- **button** - gir deg hvilken museknapp som ble trykket. 0 er venstre, 1 er midten/musehjul, og 2 er høyre. Er musa konfigurerert for venstrehendte i brukerens system endres verdiene tilsvarende.
- **altKey**, **ctrlKey** og **shiftKey** - Gir deg en **true**- / **false**-verdi på om angitte tast var trykt ned på tastaturet mens musehendelsen ble utført.
- **detail** - gir deg en tallverdi på hvor mange ganger i en sammenhengende serie som museknappen ble trykket. Altså 2 for dobbeltklikk og 3 for trippelklikk.
- **deltaX** - den horisontale scrollingen på musehjulet
- **deltaY** - den vertikale scrollingen på musehjulet

Drag and drop

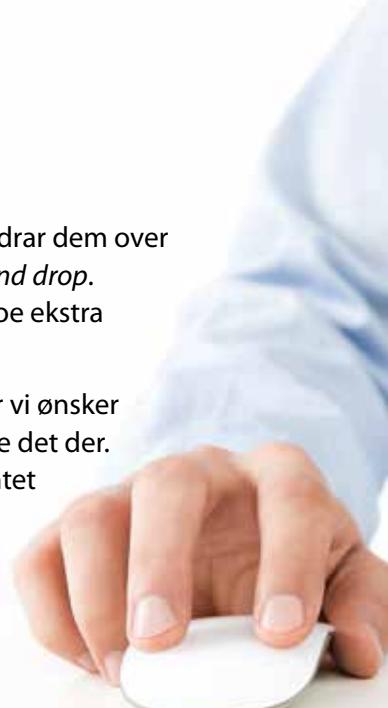
Et vanlig hendelsesforløp er at vi tar tak i objekter med musa og drar dem over til et annet område og slipper det der. Dette omtales som *drag and drop*. Denne prosessen styres også av musehendelser sammen med noe ekstra funksjonalitet.

La oss ta utgangspunkt i at du har to elementer på nettsiden, der vi ønsker å kunne dra elementet **tekst** over elementet **boks**, og så slippe det der. Når vi slipper det, så skal elementet **tekst** bli et barn av elementet **boks**, altså bli plassert "inne i" boksen.

```
<body>
  <p id="boks"></p>
  <p id="tekst">Hei</p>
</body>
```

For å få en størrelse på boksen som er tom, setter vi dette med CSS:

```
<style>
  #boks {
    width: 500px;
    height: 500px;
    background-color: gray;
  }
</style>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Deretter er det svært viktig at vi indikerer at elementet **tekst** skal være mulig å ta tak i ved hjelp av attributten **draggable**. Denne attributten kan settes på så godt som alle elementer.

```
<body>
    <p id="boks"></p>
    <p id="tekst" draggable="true">Hei</p>
</body>
```



Om vi ønsker, kan vi også sette egenskapen **draggable** gjennom JavaScript:

```
document.getElementById("tekst").draggable = true;
```

Vi må så koble hendelser til elementene. Elementet som skal dras, må få hendelsen **ondragstart** koblet til seg, mens elementet det skal være mulig å slippe på, må få hendelsene **ondragover** og **ondrop** koblet til seg.

```
function oppstart() {
    document.getElementById("tekst").ondragstart = dra;
    document.getElementById("boks").ondrop = slipp;
    document.getElementById("boks").ondragover = tillat;
}
```

I hendelsen **ondragstart** kan vi sende informasjon med i drag and drop-operasjonen. Vanligst er det å sende med ID-en til elementet som blir dratt, slik at vi kan hente ut denne ID-en (og dermed hvilket element det er) når hendelsen er slutt. I dette lille eksempelet har vi kun ett element, og derfor kunne vi utelatt dette da vi alltid vet hvilket element det er snakk om. Vi tar det allikevel med for å få samme fremgangsmåte uansett antall elementer.

```
function dra(evt) {
    evt.dataTransfer.setData("text", evt.target.id);
}
```

Neste steg vil være å fortelle at elementet **boks** faktisk aksepterer at elementet kan slippes. Dette gjør også at musepekerens symbol endres. Metoden **preventDefault** benyttes for å oppnå dette ved at vi avviker fra ordinær handling som er å nekte elementer via drag and drop.

```
function tillat(evt) {
    evt.preventDefault();
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

I dette eksempelet skal vi tillate "alt". Her kunne vi imidlertid lagt inn en if-test som sjekket hvilket element det var, eller sjekket visse egenskaper ved elementet, slik som at teksten måtte være *Hei*:

```
function tillat(evt) {
    var elementId = evt.dataTransfer.getData("text");
    if (document.getElementById(elementId).innerHTML === "Hei") {
        evt.preventDefault();
    }
}
```

Til slutt må vi utføre en handling når objektet slippes. I dette tilfellet ønsker vi å plukke ut ID-en til elementet som slippes ut fra hendelsesobjektet, og så knytte dette elementet opp som et barn til boksen. Også her må vi benytte den spesielle **preventDefault** for å overstyre normal handling.

```
function slipp(evt) {
    evt.preventDefault();
    var elementId = evt.dataTransfer.getData("text");
    evt.target.appendChild(document.getElementById(elementId));
}
```

Tastaturhendelser

Vi kan registrere hendelser forbundet med tastaturet ved hjelp av **onkeydown**, **onkeypress** og **onkeyup**.

Det er ikke helt lett å forstå forskjellen på disse tre hendelsene. For vanlige taster vil alle hendelsene bli utført når en tast trykkes på og slippes. Umiddelbart når tasten trykkes ned, utføres **onkeydown** og **onkeypress**. Dersom tasten holdes nede, vil imidlertid **onkeypress** gjenta seg flere ganger, på samme måte som det å holde nede tasten **A** mens du er i en tekstbehandler, vil gi deg en rekke A-tegn.

 Repetisjonshastigheten på en **onkeypress** styres av systeminnstillingene og varierer fra maskin til maskin. Derfor er denne dårlig egnet til f.eks. å lage spill.

Til slutt vil **onkeyup** utføres når du slipper tasten igjen. Merk deg at **onkeypress** kun vil utføres for taster på tastaturet som representerer tegn. Taster slik som ctrl, F11 og backspace fanges ikke opp av denne, og dermed kun av **onkeydown**.

For at et element skal kunne registrere tastetrykk, må det ha *fokus*. Det vil si at elementet er aktivisert slik at det er mottakelig for tastaturhendelser. For eksempel vil en input-tekstboks ha fokus når markøren står og blinker i tekstfeltet.

Ønsker vi å fange opp tastaturhendelser for hele nettsiden, uavhengig av hvilket element som har fokus, kan vi koble hendelsen direkte på **document**.

```
document.onkeydown = tastTrykket;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Informasjon i hendelsesobjektet

For tastaturhendelser av typen **KeyboardEvent** vil hendelsesobjektet blant annet inneholde:

- **keyCode** - gir deg et nummer som representerer tasten på tastaturet som ble trykket. Om du ikke tester selv, kan du finne en grei liste her:
<http://www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes>
- **charCode** - gir deg et nummer som representerer tegnet som blir trykket på tastaturet. Dette er Unicode-verdien til tegnet.
- **altKey**, **ctrlKey** og **shiftKey** - gir deg en **true/false** verdi på om angitte tast var nedtrykt på tastaturet sammen med den andre tasten som hendelsen indikerer.

Husk på forskjellen på **keyCode** og **charCode**. For eksempel vil tasten **M** gi deg tallet 77 som **keyCode** både for stor og liten bokstav, mens **charCode** vil gi 77 for stor *M* og 109 for liten *m*.



I stedet for å ha en programkode full av tallkoder for de ulike tastene kan det være lurt å lage seg et en assosiativ array der vi har listet opp tastene vi benytter.

```
window.onload = oppstart;

var taster = { A : 65, PAGE_UP: 33 };

function oppstart() {
    document.onkeydown = tastTrykket;
}

function tastTrykket(evt) {
    if(evt.keyCode === taster.A) {
        alert("A trykket");
    }
}
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Sjekke om en tast er nedtrykket

I utgangspunktet kan vi bare registrere når en tast trykkes ned (`onkeydown`), og når den slippes opp (`onkeyup`). Av og til ønsker vi imidlertid å sjekke om en tast er nedtrykt på et bestemt tidspunkt.

Vi kan løse dette med noen if-tester. I koden nedenfor lager vi en variabel av typen boolean som enten er `true` eller `false` ut fra om venstre pil tast er nedtrykt.

I tillegg har vi en klikk-hendelse på dokumentet som viser en melding ja/nei etter som om pil tasten er nedtrykt når nettsiden klikkes på.

```
<script>

window.onload = oppstart;

var pilVenstreTrykket = false;

function oppstart() {
    document.onkeydown = tastTrykket;
    document.onkeyup = tastSluppet;

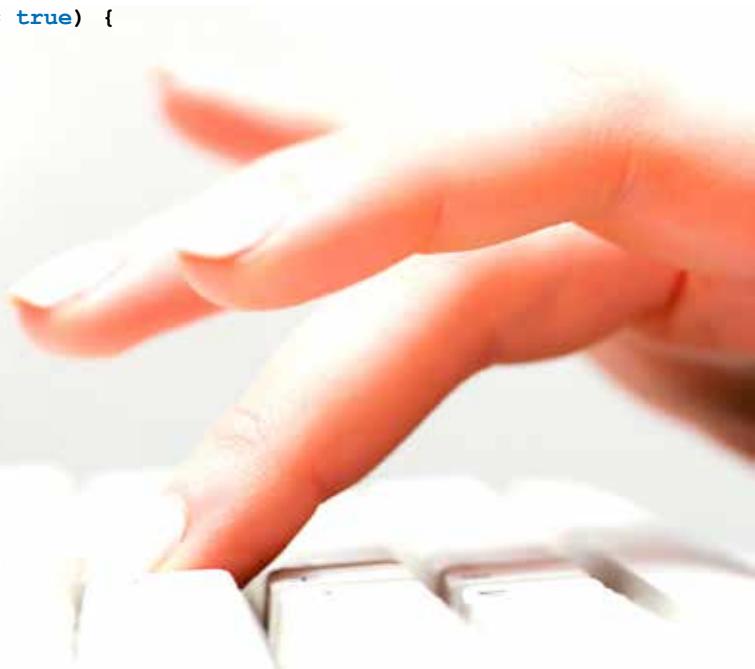
    document.onclick = klikket;
}

function tastTrykket(evt) {
    if (evt.keyCode === 37) {
        pilVenstreTrykket = true;
    }
}

function tastSluppet(evt) {
    if (evt.keyCode === 37) {
        pilVenstreTrykket = false;
    }
}

function klikket(evt) {
    if (pilVenstreTrykket === true) {
        alert("ja");
    } else {
        alert("nei");
    }
}

</script>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tidsstyrte hendelser

Tidsstyrte hendelser er hendelser som oppstår i regelmessige intervaller. Vi kan tenke oss at hendelsene "tikker og går" på samme måte som en klokke. Vi bruker vanligvis slike hendelser for å få noe til å forandre seg i løpet av tid – slik at vi for eksempel kan lage animasjon i koden eller spill.

I JavaScript følger ikke kobling av slike hendelser helt mønsteret på andre hendelser, men det er i stedet innført to spesielle funksjoner som heter `setInterval` og `setTimeout`.

Funksjonen `setInterval` gjør at en funksjon vi selv angir, gjentar seg ved et fast intervall, mens `setTimeout` gjør at en funksjon utføres én gang etter et visst antall millisekunder. Disse to funksjonene er alltid knyttet mot objektet `window`, selv om det ikke er nødvendig å skrive det eksplisitt. Funksjonene vi benytter som "lytterfunksjoner", er helt ordinære funksjoner, og det blir ikke overført noe hendelsesobjekt.

```
<script>

window.onload = oppstart;

function oppstart() {
    setInterval(hilsen, 2000);
}

function hilsen() {
    alert("Hei igjen! Går det fortsatt bra?");
}

</script>
```

Husk at det er 1000 millisekunder i ett sekund.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Stoppe tidsstyrte hendelser

Dersom vi ønsker å stoppe en tidsstyrkt hendelse, kan vi gjøre dette gjennom funksjonene `clearInterval` og `clearTimeout`. Disse vil da henholdsvis stoppe et pågående repeterende intervall eller stoppe en timeout før den utføres.

For begge disse funksjonene må vi ha en referanse til selve registreringen. Denne får vi ved å ta vare på verdien (identifikatoren) som funksjonene `setInterval` og `setTimeout` returnerer i en variabel, og så benytte denne.

I koden under har vi utvidet eksempelet med en knapp der vi kan avbryte de plagsomme meldingene.

```
<script>

window.onload = oppstart;

var intervallID;

function oppstart() {
    intervallID = setInterval(hilsen, 2000);
    document.getElementById("btnStopp").onclick = stopp;
}

function hilsen() {
    alert("Hei igjen! Går det fortsatt bra?");
}

function stopp() {
    clearInterval(intervallID);
}

</script>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Tegneprogram



I dette eksempelet skal vi la brukeren tegne på et canvas. Fra og med museknappen trykkes ned til den slippes, skal det tegnes en strek med en valgt farge. Fargen velges ved å trykke på R, G, B eller S på tastaturet for henholdsvis rød, grønn, blå og sort. Vi skal i dette eksempelet også ta høyde for at canvaset ikke nødvendigvis står i posisjonen 0,0 slik vi har gjort i tidligere eksempler med canvas.

1. Lag et nytt HTML-dokument du kaller *tegneprogram.html* ut fra *mal.html*.
2. Legg inn et canvas med ID **tegneflate**, og en span som skal benyttes til å vise valgte farge:

```
<body>
    <canvas id="tegneflate" width="500" height="500"></canvas><br />
    Farge: <span id="farge"></span> (R=Rød, G=Grønn, B=Blå, S=Sort)
</body>
```

3. Legg inn to stiler som forteller at canvaset skal ha en ramme rundt, og at fargeindikatoren skal vises som en boks med standardfargen sort:

```
<style>
    #tegneflate {
        border-style: solid;
    }

    #farge {
        width: 10px;
        height: 10px;
        display: inline-block;
        background-color: black;
    }
</style>
```

4. Legg inn programkoden som styrer funksjonaliteten:

```
<script>

window.onload = oppstart;

var ctx;
var canvasTop = 0;
var canvasLeft = 0;
var taster = { R : 82, G: 71, B: 66, S: 83};

function oppstart() {
    ctx = document.getElementById("tegneflate").getContext("2d");
    document.getElementById("tegneflate").onmousedown = musNed;
    document.getElementById("tegneflate").onmouseup = musOpp;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```

document.getElementById("tegneflate").onselectstart =
    function () { return false; }
document.onkeyup = tastSluppet;

var rect =
    document.getElementById("tegneflate").getBoundingClientRect();
canvasTop = rect.top;
canvasLeft = rect.left;
}

function musNed(evt) {
    document.getElementById("tegneflate").onmousemove = musFlytt;
    ctx.beginPath();
    ctx.moveTo(evt.clientX - canvasLeft, evt.clientY - canvasTop);
}

function musOpp(evt) {
    document.getElementById("tegneflate").onmousemove = null;
}

function musFlytt(evt) {
    ctx.lineTo(evt.clientX - canvasLeft, evt.clientY - canvasTop);
    ctx.stroke();
}

function tastSluppet(evt) {
    if (evt.keyCode === taster.R) {
        ctx.strokeStyle = "red";
    }
    else if (evt.keyCode === taster.G) {
        ctx.strokeStyle = "green";
    }
    else if (evt.keyCode === taster.B) {
        ctx.strokeStyle = "blue";
    }
    else if (evt.keyCode === taster.S) {
        ctx.strokeStyle = "black";
    }
    document.getElementById("farge").style.backgroundColor =
        ctx.strokeStyle;
}
</script>
```

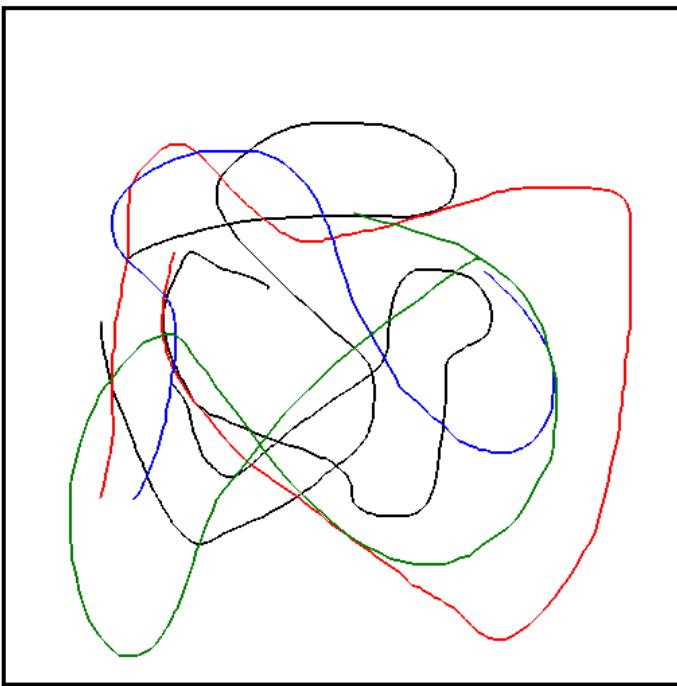
5. Test nettsiden og kontroller at du får tegnet en tegning ved bruk av de ulike fargene.

Under oppstarten av dette eksempelet setter vi hendelser for at museknappen trykkes ned og slippes opp. For å unngå at vi får en "skrivermarkør" når vi trykker og drar musen over canvaset, registrerer vi en "dummy-funksjon" på hendelsen

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

onselectstart. Denne såkalte annonyme funksjonen returnerer alltid **false**, og dette er det som skal til for å hindre at vi starter en markering.

Videre henter vi ut **top**- og **left**-posisjonen til canvaset og lagrer dette i globale variabler. Disse verdiene skal senere benyttes for å "kalibrere" museposisjonen på canvaset ut fra posisjonen til selve canvaset. Til slutt registerer vi også en tastaturhendelse som fanger opp når en tast blir sluppet, slik at vi kan bytte farge ved tastetrykk.



Farge: ■ (R=Rød, G=Grønn, B=Blå, S=Sort)

Når vi trykker ned en musetast, utføres hendelsesfunksjonen **musNed**. Denne i sin tur registerer en musehendelse kalt **musFlytt** for flytting av musa, samt starter tegningen av en strek ved å utføre **beginPath** og **moveTo**. Ettersom vi skulle ta høyde for at canvaset ikke stod i posisjonen 0,0, må vi omregne den globale posisjonen til musa på nettsiden, som **clientX** og **clientY** gir oss, til et lokalt punkt i canvaset. Dette gjøres ved å trekke fra canvases posisjon på nettsiden, som vi fant under oppstarten.

Etter hvert som musa forflyttes utføres hendelsesfunksjonen **musFlytt**.

Denne lager en linje fra forrige til nåværende posisjon og utfører en tegning av linjesegmentet gjennom kommandoen **stroke**. Når vi slipper museknappen, vil hendelsesfunksjonen **musOpp** fjerne **onmousemove**-hendelsen som **musNed** lagde, for å stoppe tegningen av streker.

Fargevelgingen gjøres ved å sjekke hvilken **keyCode** tastaturhendelsen genererte, og så settes en tilhørende farge. Indikatorboksen (**span**-elementet) får også satt den samme fargen gjennom en CSS-stil.

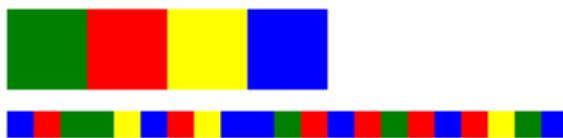
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Eksempel - Drag and drop game

Vi skal nå lage et lite spill der det vises et antall bokser med hver sin farge. Under disse boksene dukker det opp mindre firkanter (kalt *brikker*). Disse får en av fargene til boksene tilfeldig tildelt. Hvert halve sekund dukker det opp en ny brikke.

Vi kan dra en brikke og slippe på boksene. Slipper vi brikken på en boks med samme farge, forsvinner brikken, og vi får et poeng. Dersom antall brikker som ikke er behandlet, blir flere enn 20, stopper spillet. Det er altså om å gjøre å bli kvitt brikkene så fort som mulig.



1. Lag et nytt HTML-dokument du kaller *draganddropgame.html* ut fra *mal.html*.
2. Legg inn to områder og en paragraf. De to områdene skal vise hhv. boksene og brikkene. Paragrafen skal vise poengsummen:

```
<body>
    <div id="bokser">
    </div>
    <div id="brikker">
    </div>
    <p id="antall">0</p>
</body>
```

3. Legg inn stilsetting som vi skal benytte på de fremtidige elementene av klassene **boks** og **brikke**:

```
<style>
    .boks {
        width: 30px;
        height: 30px;
        display: inline-block;
    }

    .brikke {
        width: 10px;
        height: 10px;
        display: inline-block;
    }

</style>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

4. Legg inn programkoden til spillet:

```
<script>

window.onload = oppstart;

var farger = ["green", "red", "yellow", "blue"];
var brikkeTeller = 1;
var intervallID;
var antall = 0;
var slutt = false;
var maksAntall = 20;

function oppstart() {
    for (var i = 0; i < farger.length; i++) {
        var boks = document.createElement("span");
        boks.className = "boks";
        boks.style.backgroundColor = farger[i];
        boks.ondrop = slipp;
        boks.ondragover = tillat;
        document.getElementById("bokser").appendChild(boks);
    }
    intervallID = setInterval(lagElement, 500);
}

function lagElement() {
    var brikke = document.createElement("span");
    brikke.className = "brikke";
    brikke.style.backgroundColor = farger[Math.floor(Math.random() * farger.length)];
    brikke.draggable = true;
    brikke.ondragstart = dra;
    brikke.id = "brikke" + brikkeTeller;
    brikkeTeller++;
    document.getElementById("brikker").appendChild(brikke);

    if (document.getElementById("brikker").children.length > maksAntall) {
        alert("Du tapte med " + antall + " poeng");
        clearInterval(intervallID);
        slutt = true;
    }
}

function dra(evt) {
    evt.dataTransfer.setData("text", evt.target.id);
}

function slipp(evt) {
    if (slutt === true) {
        return;
    }
    evt.preventDefault();
    var elementId = evt.dataTransfer.getData("text");

    if (evt.target.style.backgroundColor ===
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```

document.getElementById(elementId).style.backgroundColor) {
    document.getElementById(elementId).ondragstart = null;

    document.getElementById("brikker").removeChild
        (document.getElementById(elementId));
    antall++;
    document.getElementById("antall").innerHTML = antall;
}

function tillat(evt) {
    evt.preventDefault();
}

</script>

```

5. Test nettsiden, og kontroller at spillet fungerer etter reglene indikert i introduksjonen til eksempelet.

Dette eksempelet er forholdsvis stort og omfattende, men består av flere mindre deler.

Først definerer vi de globale variablene vi har behov for. Variabelen **farger** er en array som rett og slett holder på de fargenavnene vi ønsker å benytte. Variabelen **brikkeTeller** inneholder nummeret på den brikken vi nå plasserer ut. Ettersom vi skal lage nye brikker mens programmet kjører, må vi gi dem ID-er. ID-en vil enkelt og greit være *brikke1*, *brikke2*, *brikke3* osv., ettersom hva **brikkeTeller** har for verdi.

Variabelen **intervalID** skal holde på en referanse til den tidsstyrte hendelsen vi skal lage, slik at vi kan stoppe den når spillet er over. Variabelen **antall** er en enkel teller som holder styr på antall poeng og **slutt** indikerer om spillet er avsluttet. Når denne er satt til **true**, skal det ikke gå an å gjøre mer i spillet.

I hendelsesfunksjonen **oppstart** benytter vi en for-løkke til å lage like mange bokser (****-elementer med klasse satt til **boks**) som det vi har farger i arrayen **farger**. Hver boks får også fargen på tilhørende indeks i arrayen. Alle boksene blir barn av **<div>**-taggen **bokser**. Til hver boks knyttes det også en **ondrop**- og en **ondragover**-hendelse.

Til slutt i oppstarten lages det en tidsstyrkt hendelse som gjentar funksjonen **lagelement** hvert 500 ms, eller altså to ganger i sekundet. Denne funksjonen lager et nytt ****-element med klasse satt til **brikke**. Brikken blir gitt en tilfeldig farge blant dem i arrayen **farger**, samt en ID bestående av teksten **brikke** og verdien til **brikkeTeller** som så økes hver gang. Brikken blir også satt til å være **draggable**, samt får en **ondragstart**-hendelse knyttet til seg. Brikken plasseres så som et barn til **<div>**-taggen **brikker**.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Dersom denne taggen har flere enn 20 barn (altså at det er synlig mer enn 20 brikker), stopper spillet med en meldingsboks, den tidsstyrte hendelsen fjernes, og variabelen **slutt** settes til **true**.

Selve drag and drop-funksjonaliteten styres av de tre hendelsesfunksjonene **dra**, **slipp** og **tillat**. Funksjonen **slipp** er her litt mer omfattende enn vanlig. Først sjekkes det om spillet er avsluttet. Er det det, utfører vi en **return** og avbryter dermed hendelsen.

Deretter sjekker vi at boksen som den aktuelle brikken ble sluppet på, har samme farge. Har den det, fjernes hendelsene fra brikken, før brikken selv fjernes. Antallet poeng økes deretter med én.

Legg merke til at denne koden er skrevet slik at vi enkelt kan endre antall farger. Det eneste som skal til, er å legge til flere fargenavn i arrayen, og spillet tar automatisk i bruk disse. Dette oppnår vi ettersom vi hele tiden refererer til lengden på arrayen i koden, i stedet for å skrive koden ut fra at det er eksakt fire farger.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

10 Lyd, video og animasjon

I dette kapitlet vil du lære

- å spille av lyd
- å kontrollere avspillingen av en video
- om programmert animasjon
- om ulike former for bevegelse og kollisjonstesting

Til nå har vi arbeidet med tekst og bilder for å skape innhold i en nettside. I dette kapitelet vil vi se nærmere på hvordan vi håndterer lyd, video og animasjon gjennom JavaScript. Behersker du disse teknikkene, vil du kunne lage enkle spill, samt mer innholdsrike presentasjoner og interaktive nettsider.

Lyd og video

Spille av lyd

Den enkleste formen for lydavspilling i JavaScript er å lage et nytt lydobjekt i koden som henter inn en lydfil, og så spille av denne. Dette vil spille av lydfilen en gang.

```
function oppstart() {  
    var lyd = new Audio("lydfil.mp3");  
    lyd.play();  
}
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

En kompliserende del av lyd og nettsider er at ulike nettlesere støtter ulike formater. Da denne boka ble skrevet, så støtten slik ut for de siste versjonene av nettleserne, men dette forbedres stadig:

	Ogg	Wav	MP3
Internet Explorer	X	X	✓
Safari	X	✓	✓
Chrome	✓	✓	✓
Firefox	✓	✓	✓
Opera	✓	✓	✓

Ut fra denne støtten kan det være lurt foreløpig å benytte MP3 som lydformat.

Det er viktig at lydfilene følger HTML-fila til nettsiden og blir referert riktig. Lyddataene blir altså ikke en del av selve HTML-fila.



HTML audio-tagg

Dersom vi ønsker å lage en synlig avspiller som brukeren kan se og interaktere med, har HTML en egen tagg kalt **<audio>** som gir oss dette:

```
<audio controls="controls" id="lydavspiller">
  <source src="lydfil.mp3" type="audio/mpeg">
  <source src="lydfil.ogg" type="audio/ogg">
  Nettleseren støtter ikke lydavspilling
</audio>
```

Legg spesielt merke til at audio-taggen kan bestå av flere ulike **source**-tagger. Dermed kan nettleseren velge å benytte det formatet den helst ønsker.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Husk at om du benytter flere ulike lydformater, så må du oppdatere alle filene om du senere endrer litt på selve lyden. Det er fort gjort å glemme å gjøre endringen i en av filene, og ikke oppdage feilen da din nettleser foretrekker en av de andre formatene.

Om vi oppretter lydavspilleren med HTML eller ved hjelp av `Audio` i JavaScript, utgjør ingen forskjell i hvordan vi styrer avspillingen. Enten henviser vi til variabelen som refererer til lydavspilleren, eller så henviser vi til elementet i nettsiden via `document.getElementById`.

```
function oppstart() {
    document.getElementById("lydavspiller").play();
}
```

HTML video-tagg

På samme måte som vi setter inn `<audio>`-tagger, støtter HTML også en `<video>`-tagg:

```
<video width="640" height="480" controls="controls" id="videoavspiller">
    <source src="videofil.mp4" type="video/mp4">
    <source src="videofil.ogg" type="video/ogg">
    Nettleseren støtter ikke videospilling
</video>
```

Vi har det samme problemet med formater når vi benytter video. Ikke alle nettlesere støtter alle formatene. På tidspunktet da denne boka ble utgitt, så situasjonen slik ut:

	Ogg	WebM	MP4
Internet Explorer	✗	✗	✓
Safari	✗	✗	✓
Chrome	✓	✓	✓
Firefox	✓	✓	✓
Opera	✓	✓	✓

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Styre lyd/videoavspilling

Både lyd og video kan styres ved hjelp av funksjonene `play` og `pause`, samt egenskapen `currentTime` (målt i sekunder) som gjør at vi kan sette avspillingen til en bestemt posisjon. Et enkelt eksempel kan f.eks. være å knytte disse tre avspillingsfunksjonene opp mot tastene A for play, B for pause og C for å flytte avspillingsposisjonen til starten.

```
function oppstart() {
    document.onkeyup = tastSluppet;
}

function tastSluppet(evt) {
    if(evt.keyCode === 65) {
        document.getElementById("lydavspiller").play();
    }
    else if(evt.keyCode === 66) {
        document.getElementById("lydavspiller").pause();
    }
    else if(evt.keyCode === 67) {
        document.getElementById("lydavspiller").currentTime = 0;
    }
}
```

I tillegg kan man få behov for egenskapen `duration`, som gir antall sekunder lyden/videoen varer. Slik kan vi f. eks. sette avspillingsposisjonen til halvveis:

```
document.getElementById("lydavspiller").currentTime= ←🚫
document.getElementById("lydavspiller").duration / 2;
```

Lydnivå kan enten hentes og settes gjennom egenskapen `volume` der f.eks. 0 er helt avskrudd lyd, 0.5 er 50 % lyd og 1.0 er maks effekt. Egenskapen `mute` kan vi sette til `true` og `false` for midlertidig å mute lyden, men beholde volumet.

I tillegg til funksjonene og egenskapene har lyd og videoavspilling flere svært kjekke hendelser som vi kan koble kode til. Vi kan bl.a. detektere at en avspilling har nådd slutten ved hjelp av hendelsen `onended`. Slik kan vi f.eks. få avspillingen til å starte på nytt hver gang den er ferdig:

```
function oppstart() {
    document.getElementById("lydavspiller").onended = slutt;
}

function slutt() {
    document.getElementById("lydavspiller").currentTime = 0;
    document.getElementById("lydavspiller").play();
}
```

Andre kjekke hendelser er `onplay`, `onpause`, `onseeked` og `onvolumechange` som utføres når brukeren (eller annen programkode) har utført en handling mot avspilleren. Hendelsen `canplay` utføres idet nok data er lastet ned for å kunne starte avspillingen.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Programmert animasjon

Vanligvis vil vi lage animasjon på en nettside ved hjelp av et animert bilde, video eller CSS. I de tilfellene der vi ønsker å lage en animasjon som er basert på input fra brukeren, eksterne data eller tilfeldighet, så må vi lage animasjonen ved hjelp av programkode. Det samme gjelder dersom vi ønsker å animere funksjonelle elementer på en nettside.



Tenk deg f.eks. at du skal lage et spill det man skyter ut en kanonkule. Brukeren skal selv kunne bestemme retning og utgangshastighet på kulen. Hvis vi skulle illustrert kanonkulen ved hjelp av et animert bilde, måtte vi lage svært mange animasjoner av kulen som tilsvarer alle de kombinasjonene av retning og utgangshastighet brukeren kan velge. Dette vil i praksis være en uoverkommelig oppgave.

Løsningen blir å la programkoden lage animasjonen. Men vær oppmerksom på at ikke all animasjon er egnet for programmering. Når et objekt har kompliserte bevegelser som er synkronisert i forhold til hverandre, blir det svært vanskelig å programmere dette, for eksempel når et menneske går eller løper.

Programmert animasjon er velegnet for å animere et objekt som flytter seg fra A til B, eller et objekt som akselererer eller spretter. Med andre ord har vi mulighet til å simulere fysisk riktige, men enkle bevegelser.

I eksemplene som følger, bruker vi **setInterval** for å gjenta en funksjon, som hver gang den utføres, forflytter et element noe. Det er i hovedsak to måter å forflytte et objekt på: enten å tegne det ut gang på gang i et canvas som en ren tegning, eller å flytte et HTML-element ved hjelp av CSS-egenskapene **top** og **left**.

Lage animasjon på et canvas

Å lage en animasjon på et canvas vil basere seg på å lagre posisjonen til elementet i variabler, samtidig som vi fjerner gammel tegning og tegner en ny tegning kontinuerlig. For hver gang vi tegner, fjerner vi først alt som er der, så tegner vi selve figuren, og til slutt endrer vi litt på posisjonsvariablene så de er klare til neste runde.

```
window.onload = oppstart;
```

```
var ctx;
var x = 50;
var y = 50;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
function oppstart() {
    ctx = document.getElementById("tegneflate").getContext("2d");
    setInterval(tegn, 100);
}

function tegn() {
    ctx.clearRect(0, 0, 200, 200);
    ctx.beginPath();
    ctx.arc(x, y, 20, 0, 2 * Math.PI);
    ctx.lineWidth = 5;
    ctx.strokeStyle = "red";
    ctx.stroke();
    x += 5;
    y += 5;
}
```

En ulempe ved å tegne på et canvas er at all grafikk forsvinner når vi fjerner tegningen ved hjelp av `clearRect`. Dersom vi har andre grafiske elementer vi ønsker beholde, kan vi imidlertid løse dette ved å legge flere `<canvas>`-elementer over hverandre.

```
<style>
    #tegneflater {
        position: relative;
    }

    #tegneflate1 {
        position: absolute;
        left: 0;
        top: 0;
        z-index: 0;
    }

    #tegneflate2 {
        position: absolute;
        left: 0;
        top: 0;
        z-index: 1;
    }
</style>

<body>
    <div id="tegneflater">
        <canvas id="tegneflate1" width="200" height="200"></canvas>
        <canvas id="tegneflate2" width="200" height="200"></canvas>
    </div>
</body>
```

Alternativt kan vi tegne ut all grafikken hver gang, også det som ikke flytter på seg.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lage animasjon ved hjelp av CSS-posisjoner

For å lage animasjon ved hjelp av CSS-posisjoner er det viktig at vi setter elementets **position**-egenskap til å være **absolute**.

```
<style>
    #boks {
        position: absolute;
        background-color: red;
        width: 50px;
        height: 50px;
    }
</style>

<body>
    <div id="boks"></div>
</body>
```

Deretter kan vi oppdatere CSS-egenskapene slik som **top**, **left** og **transform** ved jevne mellomrom, samtidig som vi endrer verdiene de er beregnet ut fra.

```
window.onload = oppstart;

var boks;
var x = 50;
var y = 50;
var grader = 0;

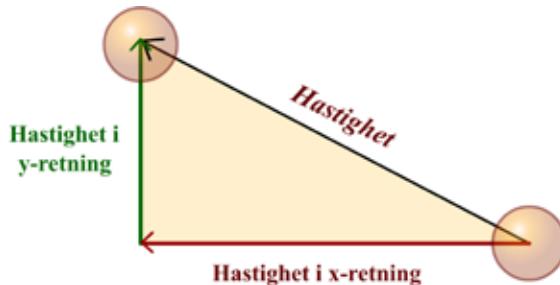
function oppstart() {
    boks = document.getElementById("boks");
    setInterval(flytt, 80);
}

function flytt() {
    boks.style.top = y + "px";
    boks.style.left = x + "px";
    boks.style.transform = "rotate(" + grader + "deg)";
    x += 5;
    y += 5;
    grader += 5;
}
```

Bevegelse og hastighet

For å få et objekt til å bevege seg må vi, som vi har sett, forandre objektets posisjon ved jevne mellomrom. Hastigheten er strekningen objektet beveger seg med per tidsintervall i horisontal og vertikal retning. I programmering er det vanligst å oppgi en forflytning/hastighet i en x-komponent og en y-komponent, som til sammen gir den visuelle hastigheten og retningen.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

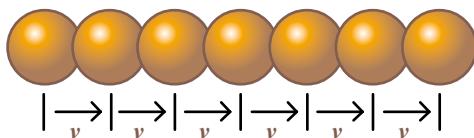


Jo større lengde vi forflytter av gangen – desto større hastighet. Vi kan også justere hvor ofte oppdateringsfunksjonen utføres, og dermed oppnå en større hastighet uten å endre strekningen objektet beveger seg per tidsintervall.

Ønsker vi en animasjon som ikke hakker, bør vi oppdatere minst 12 ganger i sekundet. Noe mer enn 24 er ikke øyet i stand til å skille. Ettersom vi også ønsker å spare systemressurser, kan 12 ganger per sekund være tilstrekkelig for tegnede animasjoner. Dette tilsvarer en oppdatering ca. hvert 80. millisekund.

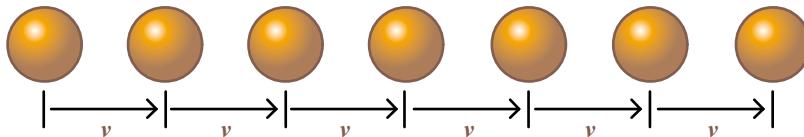
Konstant hastighet

Vi har konstant hastighet når objektet flytter seg den samme lengden per tidsintervall, uten å skifte retning. Lengden forkortes ofte til v (velocity = fart):



Konstant hastighet

Når vi forandrer lengden, forandrer vi hastighet:



Dobbeltså stor hastighet

Det er vanlig å lage variabler som inneholder forskjell/hastighet i hhv. x-retning og y-retning. Disse er det vanlig å kalle Δx og Δy . Bokstaven Δ kommer av det latinske *delta*, som kan oversettes med differanse på norsk. Dersom vi ikke forandrer disse for hver gang vi flytter, får vi en konstant fart.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
window.onload = oppstart;

var boks;
var x = 50;
var y = 50;
var dx = 4;
var dy = 4;

function oppstart() {
    boks = document.getElementById("boks");
    setInterval(flytt, 80);
}

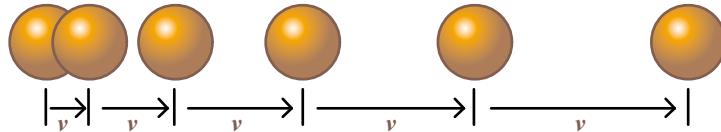
function flytt() {
    boks.style.top = y + "px";
    boks.style.left = x + "px";
    x += dx;
    y += dy;
}
```

Når vi kjører eksempelet, ser vi at objektet beveger seg 45° nedover mot høyre. Dette skjer fordi vi lar hastigheten i x- og y-retningen være like store og positive. Hvis vi hadde satt **dy** = 0, hadde vi bare hatt en bevegelse mot høyre, og hvis vi hadde satt **dx** = 0, hadde vi hatt en bevegelse rett nedover.

En negativ **dy** beveger elementet oppover, mens en negativ **dx** beveger det mot venstre.

Akselerasjon

Akselerasjon er hastighetsendring over tid:



Akselerasjon har også en størrelse og retning, på samme måte som hastigheten. Vi kan angi akselerasjon ved å lage to variabler for akselerasjon i henholdsvis x- og y-retning, **ddx** og **ddy**. Hastigheten i x- og y-retning endres da også for hver forflytning:



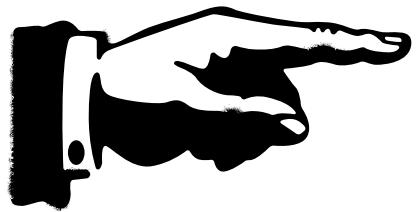
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
window.onload = oppstart;

var boks;
var x = 50;
var y = 50;
var dx = 4;
var dy = 4;
var ddx = 0.4;
var ddy = 0.1;

function oppstart() {
    boks = document.getElementById("boks");
    setInterval(flytt, 80);
}

function flytt() {
    boks.style.top = y + "px";
    boks.style.left = x + "px";
    x += dx;
    y += dy;
    dx += ddx;
    dy += ddy
}
```



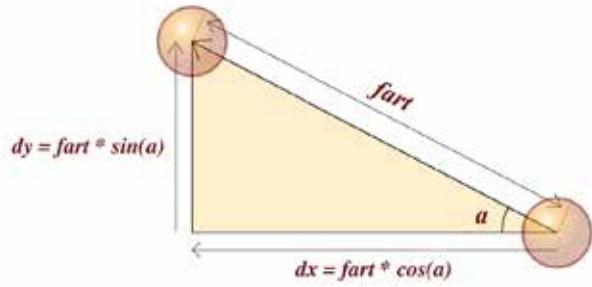
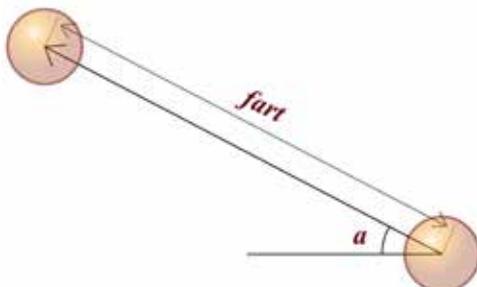
Vær oppmerksom på at **dx** og **dy** må oppdateres i samme takt som forflytningen.

Bestemme retning

I stedet for en horisontal og en vertikal hastighet, kan vi også operere med en fart og en retning. Nedenfor flytter vi kulen i en retning med vinkel **a**. Farten er lengden vi flytter.

For å flytte kulen i koordinatsystemet må vi beregne hastigheten i x- og y-retningen, da kalt **dx** og **dy**. Vi gjør dette ved å bruke de matematiske funksjonene *cos* og *sin*.

Vær oppmerksom på at *cos* og *sin* krever vinkelen angitt i *radianer*.

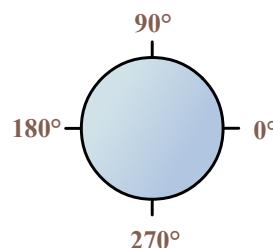
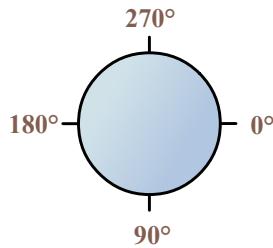


VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

I koden nedenfor angir vi fart og retning og beregner **dx** og **dy**:

```
var boks;
var x = 50;
var y = 50;
var fart = 10;
var vinkel = 30; // vinkel i grader
var a = vinkel * Math.PI / 180; //vinkel i radianer
var dx = fart * Math.cos(a);
var dy = fart * Math.sin(a);
```

Ettersom y-aksen peker nedover, vil vinkelen rett oppover være 270° . Ønsker vi å angi vinklene på "normal" måte – det vil si at vinkelen rett oppover er 90° – må vi gi dy motsatt fortegn.



`dy = fart*Math.sin(a);`

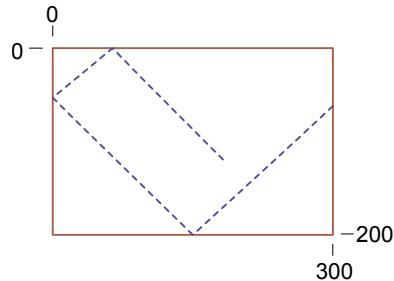
`dy = -1*fart*Math.sin(a);`

Sprett og kollisjonstesting

Vi har som regel behov for å sjekke når et objekt når en grenseverdi eller kolliderer med et annet objekt, for eksempel for å få en ball til å sprette.

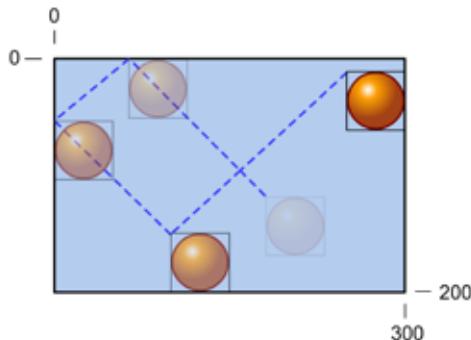
Sprett

Tenk deg at elementet skal sprette innenfor et tenkt rektangel:



Vi kan her teste om elementet kommer utenfor grenseverdiene. Er det tilfelle, reverserer vi hastigheten i den aktuelle retningen. Husk at referansepunktet for et element er øverste venstre hjørne. Vi må derfor legge til høyden for å sjekke om vi er utenfor bunnen, og legge til bredden for å sjekke om vi er utenfor høyre side:

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



```
window.onload = oppstart;

var boks;
var x = 50;
var y = 50;
var dx = 4;
var dy = 4;

function oppstart() {
    boks = document.getElementById("boks");
    setInterval(flytt, 80);
}

function flytt() {
    boks.style.top = y + "px";
    boks.style.left = x + "px";

    if(x + boks.offsetWidth > 300) {
        dx = -dx;
    }
    else if (x < 0) {
        dx = -dx;
    }

    if(y + boks.offsetHeight > 200) {
        dy = -dy;
    }
    else if (y < 0) {
        dy = -dy;
    }

    x += dx;
    y += dy;
}
```

Egenskapene `offsetWidth` og `offsetHeight` gir deg størrelsen av et element inklusive kantlinjer, men unntatt marger. I noen tilfeller kan de være et enklere alternativ enn å benytte `getBoundingClientRect`-funksjonen for å hente ut en størrelse.

TIPS

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Kollisjonstesting

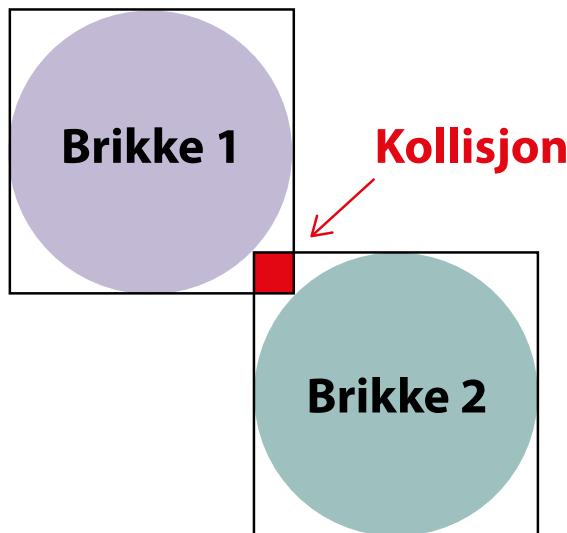
For å finne ut om to elementer overlapper, sjekker vi egentlig om de har noen felles koordinater. Dersom vi har to elementer med navn **brikke1** og **brikke2**, kan vi teste om disse overlapper ved å benytte følgende logikk:

```
var b1rect = document.getElementById("brikke1").getBoundingClientRect();
var b2rect = document.getElementById("brikke2").getBoundingClientRect();

if (b1rect.left <= b2rect.left + b2rect.width &&
    b2rect.left <= b1rect.left + b1rect.width &&
    b1rect.top <= b2rect.top + b2rect.height &&
    b2rect.top <= b1rect.top + b1rect.height) {
    // Kode som skal utføres ved kollisjon
}
```

Her plukker vi ut det såkalte bounding-rektangelet for hvert element. Dette gir oss tilgang til verdiene **left**, **top**, **bottom**, **right**, **width** og **height**. Ved å plukke ut denne informasjonen når vi trenger den, slipper vi selv å vedlikeholde den i variabler.

Deretter sjekker vi at venstre kant på brikke1 er til venstre for høyre kant på brikke2, og at høyre kant på brikke1 er til høyre for venstre kant av brikke2. Tilsvarende for topp og bunn på de to brikkene. Dette vil faktisk dekke alle situasjoner av overlapp mellom de to brikkene.



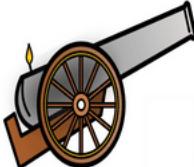
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksempel - Kanonkulespill

Vi skal i dette eksempelet lage et enkelt kanonkulespill. Med enkelt tenker vi på funksjon, for koden vil som du skal se, bli ganske omfattende. Spillet består i at vi har en kanon som kan skyte kanonkuler, og et mål som beveger seg hvert 10 sekund, som vi skal treffe. Treff gir ett poeng. Skulle kanonkulen treffe "bakken", blir det minus ett poeng.



Rotasjon: 0
Fart: 10
Poeng: -2



Vi kan styre kanonen med tastaturet, der vi justerer vinkelen med piltastene opp og ned. Farten justeres med venstre og høyre pil, og skudd utføres med mellomromstasten. Du vil sikkert se mange forbedringspotensialer med dette spillet, men vi har forsøkt å ta vekk en del funksjonalitet for å holde koden enkel.

1. Lag en HTML-fil kalt *kanonspill.html* ut fra *mal.html*.
2. Lag eller hent grafikken du trenger fra nett. Du trenger 3 grafikkfiler kalt *kanon.png*, *kule.png* og *maal.png*. Grafikken bør ha lik høyde som bredde. Plasser dem i samme mappe som HTML-fila.
3. Legg inn to lydfiler kalt *skudd.mp3* og *traff.mp3* i samme mappe som HTML-fila.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

4. Opprett de elementene vi trenger på nettsiden:

```
<body>
    <div id="info"></div>
    <div id="kanon"></div>
    <div id="maal"></div>
    <div id="kuler"></div>
</body>
```

5. Legg inn CSS-reglene som skal styre elementene på nettsiden:

```
<style>
    #kanon {
        background-image: url("kanon.png");
        background-size: 150px 150px;
        background-repeat: no-repeat;
        position: fixed;
        bottom: 0px;
        left: 10px;
        width: 150px;
        height: 150px;
        z-index: 1;
    }

    .kule {
        background-image: url("kule.png");
        background-size: 40px 40px;
        background-repeat: no-repeat;
        position: fixed;
        width: 40px;
        height: 40px;
    }

    #info {
        position: fixed;
        top: 0px;
        left: 0px;
    }

    #maal {
        background-image: url("maal.png");
        background-size: 50px 50px;
        position: fixed;
        width: 50px;
        height: 50px;
    }
</style>
```

6. Legg inn programkoden som er nødvendig for spillet:

```
<script>

window.onload = oppstart;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
var taster = {OPP: 38, NED: 40, VENSTRE: 37, HOEYRE: 39, SPACE: 32};

var kanon;
var maal;
var kuler = [];
var rotasjon = 0;
var fart = 5;
var poeng = 0;

function oppstart() {
    document.onkeydown = tastTrykket;
    kanon = document.getElementById("kanon");
    maal = document.getElementById("maal");
    setInterval(flyttKuler, 100);
    setInterval(flyttMaal, 10000);

    flyttMaal();
}

function tastTrykket(evt) {
    document.onkeydown = tastTrykket;
    if (evt.keyCode === taster.OPP) {
        rotasjon -= 5;
        rotasjon = Math.max(-90, rotasjon);
        kanon.style.transform = "rotate(" + rotasjon + "deg)";
    }
    else if (evt.keyCode === taster.NED) {
        rotasjon += 5;
        rotasjon = Math.min(0, rotasjon);
        kanon.style.transform = "rotate(" + rotasjon + "deg)";
    }
    else if (evt.keyCode === taster.VENSTRE) {
        fart -= 1;
        fart = Math.max(0, fart);
    }
    else if (evt.keyCode === taster.HOEYRE) {
        fart += 1;
        fart = Math.min(25, fart);
    }
    else if (evt.keyCode === taster.SPACE) {
        var k = document.createElement("div");
        k.className = "kule";

        var a = rotasjon * Math.PI / 180; //vinkel i radianer

        var x = kanon.getBoundingClientRect().left + ←🚫
            kanon.getBoundingClientRect().width / 2;
        var y = kanon.getBoundingClientRect().top + ←🚫
            kanon.getBoundingClientRect().height / 2;

        k.style.left = x + "px";
        k.style.top = y + "px";

        kuler[kuler.length] = {
            kule: k,
            top: y,
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```

        left: x,
        dx: fart * Math.cos(a),
        dy: fart * Math.sin(a),
        ddy: 0.1
    };

    document.getElementById("kuler").appendChild(k);

    var lyd = new Audio("skudd.mp3");
    lyd.play();
}

function flyttMaal() {
    maal.style.top = (Math.random() *
        (window.innerHeight - maal.getBoundingClientRect().height)) + "px";
    maal.style.left = (Math.random() * (window.innerWidth -
        kanon.getBoundingClientRect().width -
        maal.getBoundingClientRect().width)) +
        kanon.getBoundingClientRect().width + "px";
}

function flyttKuler() {
    for (var i = 0; i < kuler.length; i++) {
        if (kuler[i] === null) {
            continue;
        }
        kuler[i].top += kuler[i].dy;
        kuler[i].dy += kuler[i].ddy;
        kuler[i].kule.style.top = kuler[i].top + "px";

        kuler[i].left += kuler[i].dx;
        kuler[i].kule.style.left = kuler[i].left + "px";

        if (kuler[i].top > window.innerHeight || kuler[i].left >
            window.innerWidth) {
            poeng--;
            document.getElementById("kuler").removeChild(kuler[i].kule);
            kuler[i] = null;
        }
        else if (kollisjon(kuler[i].kule) === true) {
            poeng++;
            document.getElementById("kuler").removeChild(kuler[i].kule);
            kuler[i] = null;
            var lyd = new Audio("traff.mp3");
            lyd.play();
        }
    }

    document.getElementById("info").innerHTML = "Rotasjon: " +
        rotasjon + "<br />Fart: " + fart + "<br/>Poeng: "+poeng;
}

```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```

function kollisjon(kule) {
    var b1rect = maal.getBoundingClientRect();
    var b2rect = kule.getBoundingClientRect();

    if (b1rect.left <= b2rect.left + b2rect.width && ←
        b2rect.left <= b1rect.left + b1rect.width && ←
        b1rect.top <= b2rect.top + b2rect.height && ←
        b2rect.top <= b1rect.top + b1rect.height) {
        return true;
    }
    else {
        return false;
    }
}

</script>

```

7. Test spillet og kontroller at det fungerer som beskrevet i innledningen.

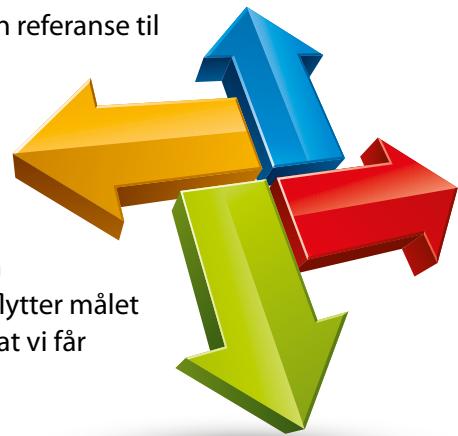
Som sagt består dette spillet av forholdsvis mye kode. Vi vil forsøke å forklare koden funksjon for funksjon, eller altså delprogram for delprogram.

De globale variablene **kanon** og **maal** vil bli benyttet som hurtigreferanser til elementene i nettsiden, slik at vi slipper å skrive hele **document.getElementById** hver gang. Variabelen **kuler** vil holde på en array der vi plasserer informasjon om alle nye kuler vi lager ved hvert skudd. Hvert element i denne arrayen er en ny assosiativ array med en referanse til selve elementet og informasjon om posisjon og fart.

I funksjonen oppstart setter vi en lytterfunksjon kalt **tastTrykket** som fanger opp tastetrykk i nettsiden. Deretter henter vi ut referansene til elementene **kanon** og **maal**. Vi registrerer også to funksjoner som skal gjentas ved gitte intervaller. Funksjonen **flyttKuler** tar hånd om alt som har med å flytte kulene vi har skutt å gjøre, mens **flyttMaal** flytter målet hvert 10. sekund. Vi kjører også **flyttMaal** med en gang, slik at vi får plassert ut målet på et tilfeldig sted.

Funksjonen **tastTrykket** håndterer fem ulike tastetrykk. Pil opp og pil ned endrer variablen **rotasjon**, samt selve rotasjonen på kanonen. Pil høyre og pil venstre endrer variablen **fart**. Alle disse fire utfallene håndterer en begrensning på minste og største verdi. Her benytter vi et triks ved hjelp av **Math.min** og **Math.max**.

Ønsker vi å begrense verdien nedover, benytter vi **Math.max** sammen med selve verdien og en nedre grense. Blir verdien mindre enn denne nedre grensen, velger **Math.max** heller grensen som ny verdi. Tilsvarende gjør vi for øvre grense med

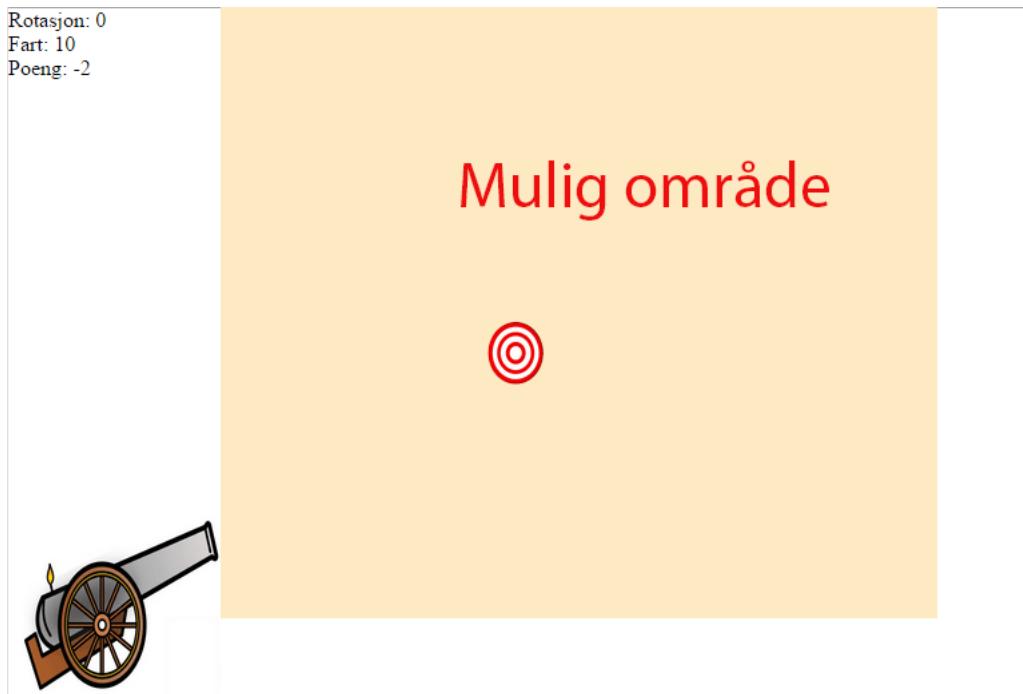


VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Math . min. Merk deg det som ved første øyekast kan virke litt ulogisk, ved at minste verdi settes ved hjelp av **max** og største verdi ved hjelp av **min**.

Vi håndterer også tasten mellomrom, som medfører et kanonskudd. Her oppretter vi først en kanonkule og beregner vinkelen **a** basert på rotasjonen og utgangspunktet for kulen gitt ved **top** og **left** i forhold til kanonen. Kulen plasseres så sist i arrayen **kuler** og legges inn som en assosiativ array. Her angis selve kulen, posisjon, en beregning av **dx** og **dy**, samt gravitasjonen **ddy**. Vi spiller også av en lyd.

Funksjonen **flyttMaal** plasserer elementet **maal** på et nytt tilfeldig sted på skjermen. Vi ønsker ikke at målet skal overlappe med kanonen, så derfor fjerner vi et tilsvarende område som kanonens bredde på venstre side. Siden posisjonen til målet angis i øvre høyre hjørne, må vi også kompensere for hhv. høyde og bredde når vi setter mulige posisjoner.



Funksjonen **flyttKuler** går gjennom alle elementer i arrayen **kuler** og flytter disse. Flytt utføres ved at vi oppdaterer **top** og **left** ved hjelp av **dx** og **dy**. Kommer kulen under bunnlinjen i nettsiden eller utenfor høyre kant, fjernes kulen, poengene minker, og plassen til kulen i arrayen tømmes.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tilsvarende sjekker vi ved hjelp av funksjonen **kollisjon** om kulen traff målet. Gjorde den det, fjernes den også, poengene økes, en lyd spilles, og plassen i arrayen tømmes. Ettersom vi får mange tomme plasser (angitt ved den spesielle verdien **null**) i arrayen, må vi for hvert element vi skal behandle sjekke om det faktisk er noen kule der. Er det ikke det, går vi direkte videre til neste element ved hjelp av nøkkelordet **continue**. Til slutt i funksjonen benytter vi også muligheten til å oppdatere informasjonsboksen med fart, rotasjon og poengsum.

Selv kollisjonstesten utføres som sagt gjennom funksjonen **kollisjon**. Denne benytter samme prinsipp som det vi har forklart om kollisjonstesting tidligere i kapitelet, og returnerer enten **true** eller **false**.

Rotasjon av kanonen, samt kulens utgangspunkt i kanonen må antageligvis justeres noe ut fra grafikken som er valgt. Koden slik den er presentert i her, går ut fra at kanonen peker rett frem i bildet, samt at kanonkulen skal skytes ut fra midten av kanonen.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

11 Eksternt innhold

I dette kapitlet vil du lære

- å hente innhold via AJAX
- å laste ned og tolke XML-filer
- å laste ned og tolke JSON-filer
- å bevare og lagre data lokalt på brukerens maskin
- om kommunikasjon med serverscript
- om lokale webservere
- å hente ut databaseinformasjon

Med eksternt innhold mener vi innhold og informasjon som hentes fra steder utenfor selve nettsiden. Både lyd, video og bilder er slike eksterne kilder, som vi alt har benyttet.

Eksternt innhold kan også være tekst. For eksempel kan vi ha en enkel tekstfil som inneholder nyheter, som så programkoden i vår nettside leser og presenterer. Dermed kan vi oppdatere denne enkle tekstfilen i stedet for kodene bak nettsiden. Dette er spesielt viktig hvis de som er ansvarlige for å oppdatere nyhetene, ikke har kunnskaper om HTML, CSS og JavaScript, eller informasjonen automatisk genereres av et annet system.

Når slikt eksternt innhold blir vist ved hjelp av programkode og kan medføre at ulike besøk til nettsiden eller handlinger fra brukeren viser ulikt innhold, kalles det *dynamisk innhold*.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lokal testserver

Mange av teknikkene som benyttes for dynamisk lasting av innhold, vil ikke fungere dersom du åpner HTML-fila i nettleseren gjennom ordinær filtilgang. Teknikkene forventer at nettsiden befinner seg på en webserver, og at den hentes ut gjennom en URL som starter med `http://` eller `https://`.

Ettersom det er mye ekstraarbeid å jobbe mot en *ekstern webserver* under utvikling, kan vi velge å installere en såkalt *lokal webserver* for testing. Et slikt produkt er WAMPServer for Windows eller MAMPServer for Mac. Det finnes mange andre tilsvarende produkter, men vi vil benytte disse i denne boka.



<http://www.wampserver.com/en/>



<https://www.mamp.info/en/>

Etter at du har installert produktet, vil du finne en mappe i installasjonskatalogen kalt `www` for WAMP eller `htdocs` for MAMP. Typisk `C:\wamp\www` for WAMP og `/Applications/MAMP/htdocs/` for MAMP, om du følger forslagene under installasjonen.

En fil kalt `test.html` plassert direkte i katalogen `www` eller `htdocs` vil bli tilgjengelig med URL-en `http://localhost/test.html` for WAMP og `http://localhost:8888/test.html` for MAMP.

Lager du en undermappe til `www` eller `htdocs` kalt `javascript`, og plasserer filen her, vil den være tilgjengelig med hhv. `http://localhost/javascript/test.html` og `http://localhost:8888/javascript/test.html`.

Det er svært viktig at du ikke dobbeltklikker på filer som du ønsker å åpne gjennom en lokal webserver. Gjør du det, vil de åpnes gjennom filtilgang (`file://`). Du må manuelt skrive inn URL-en hver gang.



Alle eksempler i dette kapitelet vil benytte en lokal webserver for testing. Det kan være lurt å forsøke å åpne noen enkle prosjekter fra tidligere gjennom den lokale webserveren og bli kjent med virkemåten, før du går videre med det som er nytt i dette kapitelet.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lese inn data fra en ekstern tekstfil

For å kunne hente inn innhold fra en ekstern fil etter at nettsiden er ferdig lastet, benytter vi som oftest en teknikk som kalles **AJAX (Asynkron JavaScript og XML)**.

Tar vi utgangspunkt i at du har en tekstfil med litt informasjon som har filnavnet **innhold.dat** og er plassert i samme katalog som HTML-fila, kan du fylle et **<div>**-element med ID **innhold** med denne teksten på følgende måte:

```
window.onload = oppstart;

var xmlhttp;

function oppstart() {
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = statusforandring;
    xmlhttp.open("GET", "innhold.dat", true);
    xmlhttp.send();
}

function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        document.getElementById("innhold").innerHTML = xmlhttp.responseText;
    }
}
```

Å hente ekstern informasjon gjennom AJAX består av tre skritt. Først lages en global variabel, som ofte blir kalt **xmlhttp**. Deretter plasserer du en kodeblokk der du ønsker at lastingen skal starte. Her fylles den globale variabelen med et **XMLHttpRequest**-objekt. Til dette objektet knyttes så en funksjon til hendelsen **onreadystatechange**, som utføres hver gang status på lastingen endres.

Deretter åpnes den eksterne ressursen gjennom metoden **open**. Første parameter i **open** angir metoden *GET* eller *POST*. Hva du skal benytte, avhenger av hvordan kilden er laget, men for alle ordinære filer er *GET* grei. Deretter kommer filnavnet. Siste parameter angir at forespørselen skal være asynkron, altså at programkoden ikke skal bli stående og vente på resultatet av forespørselen. Til slutt utføres forespørselen via metoden **send**.

Siste steg er å håndtere data vi mottar i hendelsesfunksjonen. Ettersom denne utføres hver gang status endres, må vi fortelle at vi kun er interessert i å gjøre noe dersom **readyState** er *4 (request finished and response is ready)* og **status**-koden er *200*, noe som betyr at filen fantes.



På grunn av sikkerhetsmodellen i JavaScript kan vi kun lese filer som ligger lagret på samme plassering/server som HTML-fila. Vi kommer senere tilbake til hvordan vi omgår dette ved hjelp av et server-script.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Selv dataene hentes ut fra egenskapen `responseText` på objektet. Dette er en ordinær tekststreng, og vi kan gjøre hva vi vil med denne videre.

I en tekstfil vil `\n` markere alle linjeskift og `\t` markere alle tabulatorer. Dette skiller seg fra HTML. Skriver vi ut innholdet i en tekstfil med flere linjer til en nettside, må vi derfor erstatte alle forekomster av `\n` med `
` ved hjelp av string-funksjonen `replace("\n", "
")`.

TIPS

Nedlasting av fila kan ta litt tid slik at teksten ikke vil være tilgjengelig med en gang. For at ikke programmet skal feile, må vi være helt sikre på at teksten er ferdig nedlastet før vi prøver å gjøre noe med den. Dette er grunnen til at det registreres en hendelse som "varsler" oss når fila er ferdig lastet.

Det kan være hensiktsmessig å vise en melding eller et ikon til brukeren som indikerer at data lastes idet vi starter lastingen, og så fjerne denne igjen når hendelsen varsler oss om at data er ferdig lastet. Å teste lokalt går som oftest raskt, men når nettsiden siden publiseres eksternt, kan det ta lengre tid.

TIPS

Lese inn strukturerte data

Ofte er vi ikke interessert i å behandle innholdet i en fil som en "bolt" med informasjon, men ønsker å behandle enkeltdeler av informasjonen. Det kan f.eks. være at hver linje i en fil inneholder et filnavn som skal vises som et bilde i en egen ``-tagg i nettsiden.

For å få til dette må data ha en struktur, ettersom maskiner er svært dårlig på selv å kunne kjenne igjen og plukke ut ulike deler av informasjon basert på en forståelse. En slik struktur kan vi finne på selv.

Tenk deg f.eks. at vi har et idrettsstevne med tre ulike grener som det blir gitt poeng i. Vi kan da selv velge at hver linje i en fil inneholder informasjon om én deltager, og at linjen inneholder deltagnernummer, navn og poengsummer skilt med et skilletegn. F.eks. kan vi velge % som skilletegn og få følgende informasjon i filen:

1%Ole Olsen%234%44%78

3%Kari Karisen%65%88%87

4%Per Persen%21%23%200

Som skilletegn er det lurt å velge et tegn eller en sekvens av tegn vi er sikre på at aldri vil forekomme som en del av selve informasjonen.

MERK

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lesing av strukturerte data foregår på samme måte som vi allerede har sett for tekstfiler, men data må deles opp etter at de er lest inn, i stedet for å behandles som en bokl. Dette gjøres som oftest ved hjelp av metoden `split` som er en del av alle stringer.

```
function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var data = xmlhttp.responseText;
        var linjer = data.split("\n");

        var utskrift = "";

        for (var i = 0; i < linjer.length; i++) {
            var datafelt = linjer[i].split("%");
            var poengTotalt = parseInt(datafelt[2]) + ←
                parseInt(datafelt[3]) + parseInt(datafelt[4]);
            utskrift = utskrift + "Deltaker " + datafelt[0] + " (" +
                datafelt[1] + ") fikk til sammen " + poengTotalt + ←
                " poeng<br />";

        }
        document.getElementById("innhold").innerHTML = utskrift;
    }
}
```

Med testfila som ble vist over, vil denne koden produsere følgende utskrift:

```
Deltaker 1 (Ole Olsen) fikk tilsammen 356 poeng
Deltaker 3 (Kari Karisen) fikk tilsammen 240 poeng
Deltaker 4 (Per Persen) fikk tilsammen 244 poeng
```

Stort sett all prosessering av slike strukturerte filer består først i å dele opp data i en array av linjer, ved hjelp av `split("\n")`. Deretter prosesserer vi én og én linje. For hver linje henter vi ut de ulike delene denne linja består av ved hjelp av nok en `split`, men denne gang med skilletegnet(/ene) som argument. Når vi har fått splittet opp linja, kan vi gjøre det vi ønsket å gjøre med de ulike datafeltene som nå er elementer i en array.

Når vi leser fra en fil, vil all data anses å være tekst. Derfor er det viktig å konvertere til riktig datatype, f.eks. ved hjelp av `parseInt`.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

XML

XML er en måte å strukturere tekstbaserte data på slik at de på en enkel måte kan forstås av både datamaskiner og mennesker.

Mange selskaper og organisasjoner publiserer XML-filer med for eksempel nyheter, børsdata eller værdata. Vi kan lage nettsider som laster ned og presenterer slike data. Vi kan også lage egne XML-filer med data som vi ønsker å presentere i nettsiden.

I eksempelet nedenfor har vi laget en XML-fil som inneholder informasjon om noen varer. Det er vanlig at en slik fil har filendelsen *xml*:

```
<vareutvalg>
  <vare>
    <navn>Anorakk</navn>
    <pris>498</pris>
  </vare>
  <vare>
    <navn>Skjorte</navn>
    <pris>359</pris>
  </vare>
  <vare>
    <navn>Kjole</navn>
    <pris>879</pris>
  </vare>
</vareutvalg>
```

Legg merke til at dataene er organisert i et hierarki med tagger (kalles også noder). *Topp-taggen*, eller *rot-taggen*, heter **vareutvalg**. Inne i **vareutvalg** har vi tre **vare**-tagger. Inne i hver **vare**-tagg har vi en **navn**-tagg og **pris**-tagg som inneholder data.

Vi kan også lagre data inne i selve taggen som attributter. I eksempelet nedenfor inneholder taggen **bil** attributter for **regnr**, **farge** og **vekt**:

```
<bilsalg>
  <sted>Lakseby</sted>
  <dato>28.04.2011</dato>
  <bil regnr="QZ20323" farge="rød" vekt="1677">
    <selger>D Donaldo</selger>
    <kjooper>R Harelabb</kjooper>
  </bil>
  <bil regnr="BT12023" farge="hvit" vekt="2647">
    <selger>J Mikki</selger>
    <kjooper>F Anton</kjooper>
  </bil>
</bilsalg>
```

Som du kanskje alt har lagt merke til, er HTML og XML svært like. HTML kan sies å være et faktisk språk for markup av nettsider, mens XML er et sett regler for å lage slike språk med. HTML er da med andre ord et eksempel på faktisk bruk av XML. Ikke alle versjoner av HTML følger imidlertid XML-standarden fullt ut.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lese og behandle XML

For å hente innholdet i en XML-fil benytter vi AJAX, på samme måte som med tekstfiler. Vi benytter imidlertid egenskapen `responseXML` i stedet for `responseText` på response-objektet.

I eksempelet under leser vi filen `varer.xml`, som inneholder XML-strukturen med et vareutvalg som vi viste tidligere. Innholdet legger vi en variabel vi kaller `varerXML`.

```
window.onload = oppstart;
```

```
var xmlhttp;

function oppstart() {
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = statusforandring;
    xmlhttp.open("GET", "varer.xml", true);
    xmlhttp.send();
}

function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var varerXML = xmlhttp.responseXML;
    }
}
```

Etter at vi har lest inn dataene i variabelen, kan vi jobbe videre på disse ved hjelp av stort sett de samme funksjonene som vi kan benytte når vi jobber med `document` og HTML-tagger. HTML er jo som nevnt en variant av XML.

Den største forskjellen er at vi i XML sjeldent har en ID, men heller må hente ut elementer basert på taggnavn. Dette gjøres gjennom metoden `getElementsByName`. Når vi gjør det, vil vi potensielt få ut mange elementer, da flere elementer kan ha samme navn. Ettersom elementene vi henter ut, organiseres som en array, kan vi finne ønsket element ved å angi index.

```
function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var varerXML = xmlhttp.responseXML;
        var varer = varerXML.getElementsByName("vare");

        alert(varer.length) // Viser 3 ettersom dokumentet har 3 vare-tagger
    }
}
```



En vanlig skrivefeil er å gjøre om `getElementById` til `getElementByTagName`. Husk imidlertid på at det må inn en ekstra `s` i metodenavnet, slik at det blir `getElementsByName`. Dette er fordi metoden potensielt kan returnere flere verdier.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Å hente ut tekst fra en tagg/node i XML er imidlertid ikke like enkelt som `innerHTML` i HTML. Selve teksten er regnet for å være en "barnenode" av elementet, som igjen har en verdi. Med andre ord henter vi ut verdien til første barnenode til noden når vi henter ut teksten til en node.

```
function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var varerXML = xmlhttp.responseXML;
        var varer = varerXML.getElementsByTagName("vare");

        var utskrift = "<p>";
        var totalpris = 0;

        for(var i = 0; i < varer.length; i++) {
            var pris = varer[i].getElementsByTagName("pris")[0].childNodes[0].nodeValue;
            var navn = varer[i].getElementsByTagName("navn")[0].childNodes[0].nodeValue;

            utskrift = utskrift + navn + " koster " + pris + " kroner <br />";
            totalpris += parseInt(pris);
        }

        utskrift = utskrift + "</p><p>Totalprisen blir " + totalpris +
            " kroner</p>";
        document.getElementById("innhold").innerHTML = utskrift;
    }
}
```

Koden over vil produsere utskriften:

Anorakk koster 498 kroner
Skjorte koster 359 kroner
Kjole koster 879 kroner
Totalprisen blir 1736



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Dersom vi ønsker å hente ut en attributtverdi, benytter vi egenskapen **attributes**, funksjonen **getNamedItem** og egenskapen **value** på et element. For eksempel kan vi hente ut registreringsnummer til første bil i XML-eksempelet om biler på følgende måte:

```
var bilXML = xmlhttp.responseXML;
var regnr = bilXML.getElementsByTagName("bil")[0]. .attributes.getNamedItem("regnr").value;
```



Eksempel - Oppskriftskatalog

I dette eksempelet skal vi lage en enkel oppskriftskatalog som kan vise en matoppskrift med navn, ingredienser og fremgangsmåte. Hver oppskrift er lagret som en egen XML-fil med følgende format:

```
<oppskrift>
  <navn>Kladdekkaker</navn>
  <porsjoner>4</porsjoner>
  <ingredienser>
    <ingrediens>
      <antall>1</antall>
      <enhet>ts</enhet>
      <type>vann</type>
    </ingrediens>
    <ingrediens>
      <antall>2</antall>
      <enhet>liter</enhet>
      <type>mel</type>
    </ingrediens>
  </ingredienser>
  <fremgangsmaate>
    <steg>Bland sammen</steg>
    <steg>Stek ved 250 grader</steg>
    <steg>Server</steg>
  </fremgangsmaate>
</oppskrift>
```

Hvilken oppskrift som skal vises, avgjøres ved at brukeren velger oppskrift i en nedtrekksliste. Også antallet porsjoner som oppskriften skal tilpasses til, kan velges, og mengden av hver ingrediens beregnes ut fra dette.

1. Lag et nytt HTML-dokument du kaller *oppskriftskatalog.html* ut fra *mal.html*, og plasser dette på den lokale webserveren.
2. Lag to XML-dokumenter kalt *kladdekaker.xml* (som har innholdet vist over) og *banankake.xml* med tilsvarende innhold, og plasser dem sammen med HTML-fila.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Legg til elementene som trengs under utskrift, samt en tekstboks for antall og en nedtrekksliste med oppskrifter:

```
<body>
    <h1 id="tittel"></h1>
    <div><span id="antall">0</span> porsjoner</div>
    <ul id="ingredienser"></ul>
    <ol id="fremgangsmaate"></ol>
    <hr/>
    Antall porsjoner: <input type="text" id="txtAntall" />
    Oppskrift: <select id="lstOppskrift">
        <option value="kladdekaker.xml">Kladdekaker</option>
        <option value="banankake.xml">Banankake</option>
    </select>
    <button type="button" id="btnEndre">Endre</button>
</body>
```

4. Legg inn programkoden som skal til for å få funksjonaliteten som er beskrevet:

```
<script>

window.onload = oppstart;

var xmlhttp;
var filnavn;
var antall;

function oppstart() {
    document.getElementById("btnEndre").onclick = endre;
    filnavn = "kladdekaker.xml";
    antall = 4;
    document.getElementById("txtAntall").value = antall;
    hentOppskrift();
}

function endre() {
    antall = document.getElementById("txtAntall").value;
    filnavn = document.getElementById("lstOppskrift").value;
    hentOppskrift();
}

function hentOppskrift() {
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = statusforandring;
    xmlhttp.open("GET", filnavn, true);
    xmlhttp.send();
}

function visOppskrift(oppskriftXML) {
    var navn = oppskriftXML.getElementsByTagName("navn")[0].childNodes[0].nodeValue;
    document.getElementById("tittel").innerHTML = navn;
    document.getElementById("antall").innerHTML = antall;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



```
var porsjoner = oppskriftXML.getElementsByTagName("porsjoner")[0].  
    childNodes[0].nodeValue;  
var faktor = antall / porsjoner;  
  
var ingrediensListe = oppskriftXML.getElementsByTagName("ingrediens");  
document.getElementById("ingredienser").innerHTML = "";  
for (var i = 0; i < ingrediensListe.length; i++) {  
  
    var ingrediensMengde = ingrediensListe[i].  
        getElementsByTagName("mengde")[0].childNodes[0].nodeValue;  
    var ingrediensEnhet = ingrediensListe[i].  
        getElementsByTagName("enhet")[0].childNodes[0].nodeValue;  
    var ingrediensType = ingrediensListe[i].  
        getElementsByTagName("type")[0].childNodes[0].nodeValue;  
  
    var ingrediens = document.createElement("li");  
    ingrediens.innerHTML = ingrediensMengde * faktor + " " +  
        ingrediensEnhet + " " + ingrediensType;  
    document.getElementById("ingredienser").appendChild(ingrediens);  
}  
  
var stegListe = oppskriftXML.getElementsByTagName("steg");  
document.getElementById("fremgangsmaate").innerHTML = "";  
for (var i = 0; i < stegListe.length; i++) {  
    var steg = document.createElement("li");  
    steg.innerHTML = stegListe[i].childNodes[0].nodeValue;  
    document.getElementById("fremgangsmaate").appendChild(steg);  
}  
}  
  
function statusforandring() {  
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {  
        var oppskriftXML = xmlhttp.responseXML;  
        visOppskrift(oppskriftXML);  
    }  
}  
  
</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

5. Test nettsiden, og kontroller at du kan bytte oppskrifter og vise oppskriftene for forskjellige antall porsjoner. Husk at du må teste nettsiden gjennom den lokale webserveren, altså f.eks.
<http://localhost/oppeskiftskatalog.html>

Kladdekaker

45 porsjoner

- 11.25 ts vann
- 22.5 liter mel

1. Bland sammen
2. Stek ved 250 grader
3. Server

Antall porsjoner: Oppskrift:

I denne programkoden er funksjonen **visOppskrift** det sentrale. Denne funksjonen presenterer innholdet i en XML-fil som metoden **hentOppskrift** har startet lastingen av. For å kunne gjøre denne jobben må de to variablene **filnavn** og **antall** være satt. Disse settes under oppstarten av nettsiden til standardoppskriften **kladdekaker.xml** og standardantallet 4. Variablene kan også settes av endre-knappen. I begge tilfellene kjøres **hentOppskrift** etter at variablene er satt, og **visOppskrift** utføres når data er lastet.

Først henter **visOppskrift** ut navn fra XML-fila, og presenterer dette sammen med valgt antall. Deretter beregnes en faktor basert på antallet porsjoner oppskriften er beregnet på, og ønsket antall fra brukeren. Er f.eks. oppskriften beregnet på 4, mens brukeren ønsker 8, settes faktoren til 2. Tilsvarende vil en oppskrift for 4 som ønskes til 2, gi en faktor på 0.5.

Informasjon om hver ingrediens hentes så ut, og mengden ganges med faktoren for å få riktig antall porsjoner. Hver ingrediens plasseres i en uordnet liste. Til slutt listes hvert steg av fremgangsmåten opp som en ordnet liste.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

JSON

Et alternativt dataformat til XML er **JSON** (*JavaScript Object Notation*). JSON kan noe forenklet sees på som en assosiativ array lagret i en tekstfil med et gitt format. Det er vanlig at slike filer har filendelsen **json**.

Det samme eksempelet som vi viste med varer for XML, kan i JSON se slik ut:

```
[  
    {"navn": "Anorakk", "pris": "498"},  
    {"navn": "Skjorte", "pris": "359"},  
    {"navn": "Kjole", "pris": "879"}  
]
```

Lese og behandle JSON

Fordelen med JSON er at det i JavaScript automatisk oversettes til objekter under innlesning/parsing, slik at vi langt enklere kan arbeide med dataene og referere til verdier. Dette gjøres av funksjonen **JSON.parse**.

Gjør vi det samme som med vareutvalget i XML-format, vil koden se ut som følger:

```
<script>  
  
window.onload = oppstart;  
  
var xmlhttp;  
  
function oppstart() {  
    xmlhttp = new XMLHttpRequest();  
    xmlhttp.onreadystatechange = statusforandring;  
    xmlhttp.open("GET", "varer.json", true);  
    xmlhttp.send();  
}  
  
function statusforandring() {  
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {  
        var varer = JSON.parse(xmlhttp.responseText);  
  
        var utskrift = "";  
        var totalpris = 0;  
  
        for(var i = 0; i < varer.length; i++) {  
            var pris = varer[i].pris;  
            var navn = varer[i].navn;  
  
            utskrift = utskrift + navn + " koster " + pris + " kroner <br />";  
            totalpris += parseInt(pris);  
        }  
  
        utskrift = utskrift + "<p>Totalprisen blir " + totalpris + " kroner</p>";  
        document.getElementById("innhold").innerHTML = utskrift;  
    }  
}  
  
</script>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

En JSON-fil kan også ha flere nivåer, men ikke noe tilsvarende attributter i XML.

```
{  
    "sted": "Lakseby",  
    "dato": "28.04.2011",  
    "biler": [  
        {  
            "regnr": "QZ20323",  
            "farge": "rød",  
            "vekt": "1677",  
            "selger": "D Donaldo",  
            "kjøper": "R Harelabb"  
        },  
        {  
            "regnr": "BT12023",  
            "farge": "hvit",  
            "vekt": "2647",  
            "selger": "J Mikki",  
            "kjøper": "F Anton"  
        }  
    ]  
}
```

Ved flere nivåer kan vi i koden bruke en *nestet notasjon* for å få tak i verdien:

```
var bilsalg = JSON.parse(xmlhttp.responseText);  
utskrift = bilsalg.biler[0].regnr;  
document.getElementById("innhold").innerHTML = utskrift;
```

Lese og lagre data lokalt

Lagring av data fra en nettside til egen maskin og lesing fra egen maskin til en nettside er ikke så lett som vi først skulle tro. Årsaken er kanskje opplagt. Vi vil ikke risikere at en nettside vi surfer innom, kan lese data fra vår maskin eller lagre/endre data på maskinen.

Med en del begrensninger er imidlertid såkalt *lokal lagring* mulig å gjennomføre. Du har kanskje hørt om *cookies* (*informasjonskapsler*) som én måte å lagre data på klienten. Cookies er imidlertid noe tungt å jobbe med i JavaScript.

Med HTML5 kom imidlertid en ny metode for lokal lagring, kalt *web storage*. Denne minner mye om cookies, men er enklere å jobbe med.

Data som lagres gjennom web storage, er tilgjengelig fra alle nettsider på samme domene og er tilgjengelig også om nettsiden lukkes og åpnes igjen. Web storage er imidlertid en lokal lagring hos brukeren, så flere brukere/besøkende på nettsiden kan ikke dele data. Dermed er det ikke fornuftig å lage f.eks. delt highscore for et spill med web storage. Vi kommer tilbake til lagring på selve webserveren, som kan benyttes til dette, senere i kapitelet.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Å benytte web storage er som sagt svært enkelt. Ønsker vi å lagre en verdi, benytter vi følgende kode, der vi har en oppslagsnøkkel (i dette tilfellet `navn`) og en verdi.

```
localStorage.setItem("navn", "Tom Heine");
```

Når vi senere ønsker å hente ut verdien igjen, benytter vi funksjonen `getItem`. Vi må imidlertid ta høyde for at verdien vi ønsker å hente ut ikke finnes (f.eks. at lokal lagring er slettet). Derfor sjekker vi om verdien vi fikk ut, er `null`, og håndterer i såfall denne unntakssituasjonen.

```
var navn = localStorage.getItem("navn");
if(navn === null) {
    navn = "Ukjent";
}
```

Alt som lagres i web storage, blir lagret som tekststrenger. Andre typer verdier bør konverteres til og fra typen string når de går inn og ut av web storage.

Eksempel - Bakgrunnsfarge

Vi skal nå lage en svært enkel nettside der brukeren kan velge bakgrunnsfarge på nettsiden fra en nedtrekksliste. Bakgrunnsfargen vil lagres i web storage og hentes frem hver gang nettsiden lastes, slik at nettsiden "husker" brukerens valg fra gang til gang siden besøkes.

1. Lag et nytt HTML-dokument du kaller `bakgrunnsfarge.html` ut fra `mal.html`.
2. Legg inn følgende innhold i nettsiden:

```
<body>
    <h1>Fargetest</h1>
    <p>Veldig mye fin informasjon kommer her... En gang...</p>
    <p>Velg bakgrunnsfarge: <select id="lstFarge">
        <option value="white">Hvit</option>
        <option value="red">Rød</option>
        <option value="green">Grønn</option>
        <option value="yellow">Gul</option>
    </select>
    <button type="button" id="btnEndre">Endre</button>
</body>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Legg inn følgende programkode i nettsiden:

```
<script>

window.onload = oppstart;

function oppstart() {
    var farge = localStorage.getItem("farge");
    if (farge === null) {
        farge = "white";
    }
    settBakgrunnsfarge(farge);
    document.getElementById("btnEndre").onclick = endre;
}

function endre() {
    farge = document.getElementById("lstFarge").value;
    localStorage.setItem("farge", farge);
    settBakgrunnsfarge(farge);
}

function settBakgrunnsfarge(farge) {
    document.body.style.background = farge;
}

</script>
```

4. Test nettsiden, og sett en ny farge (ikke hvit som er "standard").



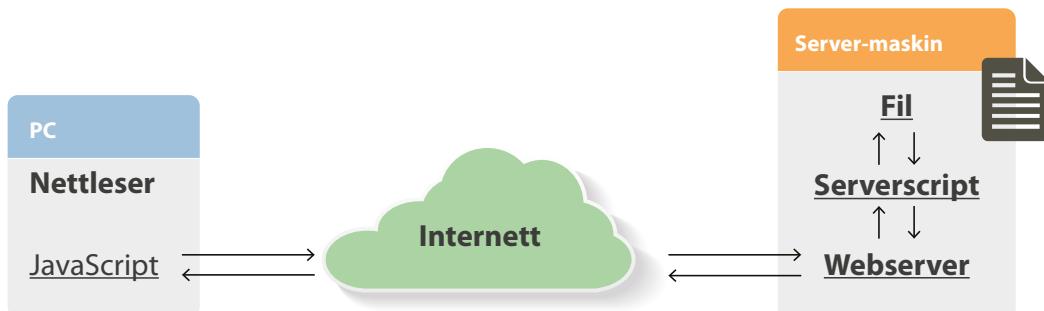
5. Lukk nettsiden/nettleseren.
6. Åpne nettsiden igjen, og kontroller at samme farge fortsatt vises.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lagre data til server

Ofte er det hensiktsmessig å kunne lagre data på webserveren der nettsiden er plassert. Dette gjelder for eksempel dersom flere brukere skal dele de samme dataene, eller vi som eiere av nettsiden skal ha tilgang til data som brukeren produserer eller benytter. Typiske eksempler kan være en delt highscoreliste, ordre i en nettbutikk eller en gjestebok.

Å ha en webserver åpen for alle når det gjelder skriving, ville imidlertid vært en stor sikkerhetsrisiko. Derfor må vi ha programvare på webserver som JavaScript-nettsiden kommuniserer med, og som står for den faktiske skrivingen.



En slik programvare kalles et *server-side-script*, i motsetning til JavaScript som er et *client-side-script*. Som oftest vil det være aktuelt å lage denne server-side-funksjonaliteten selv. JavaScript er imidlertid ikke det foretrukne språket for server-side-programmering, så vi må benytte andre språk slik som PHP, Python eller C#.

I denne boka skal vi benytte PHP som programmeringsspråk på serveren. Alle kodenfilene vi lager på serveren, må da ha filendelsen *php*. Det er dessverre ikke innenfor denne bokas rammer å se på detaljer rundt PHP og server-side-programmering, ei heller presentere alle muligheter slik programmering gir for bearbeidelse og produksjon av data. Ut over eksemplene som gis i denne boka er det anbefalt å benytte nettressurser dersom du ønsker mer kunnskap om slik programmering. Nettstedet <http://www.w3schools.com/php/> kan være et godt sted å starte.

For å starte med et enkelt eksempel først, skal vi lage et lite server-side-script som summerer to tall og viser svaret. Dette server-side-scriptet skal vi etter hvert benytte fra en nettside med JavaScript for å gjøre denne oppgaven. Summeringen kunne selvsagt også blitt gjort direkte i JavaScript, men vi velger likevel dette eksempelet for å starte med noe enkelt.

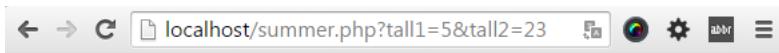
Dersom vi starter med server-side-scriptet, kan du plassere en fil kalt **summer.php** i mappa til webserveren du benytter (altså **www** eller **htdocs**). Gi denne fila følgende innhold:

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
<?php
$verdi1 = $_GET["tall1"];
$verdi2 = $_GET["tall2"];
$sum = $verdi1 + $verdi2;
echo $sum;
?>
```

Programmering i PHP går som sagt utenfor rammen til denne boka. Her er imidlertid en kort forklaring. Den første og siste linja er tagger som angir at det som ligger mellom, er PHP-kode. I koden lager vi to variabler som vi kaller **verdi1** og **verdi2**. Disse setter vi lik verdiene som kommer fra to parameterne i nettadressen. Legg merke til at variabler i PHP har et **\$**-tegn foran seg. Til slutt summerer vi de to tallene og skriver ut resultatet med kodeordet **echo**.

Å kjøre dette server-scriptet kan du gjøre ved å kalle opp siden gjennom den lokale webserveren, sammen med parameterne og tilhørende ønskede verdier:



28

Når vi kjører et PHP-script, vil programkoden bli erstattet med resultatet fra programkjøringen på webserveren, før det sendes til nettleseren. Hverken brukeren eller nettleseren vil kunne se selve koden. Forsøk selv å åpne kildekoden bak nettsiden i nettleseren, og alt du vil se, er tallet.

Vi kan nå lage en nettside med et skjema og litt JavaScript som benytter dette server-side-scriptet til summering. Det som i prinsippet skjer, er at vi leser innholdet fra *summer.php* som om det hadde vært en tekstfil med et fast innhold:

```
<script>

var xmlhttp;

window.onload = oppstart;

function oppstart() {
    document.getElementById("btnBeregn").onclick = beregn;
}

function beregn() {
    var tall1 = document.getElementById("txtTall1").value;
    var tall2 = document.getElementById("txtTall2").value;
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = statusforandring;
    xmlhttp.open("GET", "summer.php?tall1=" + tall1 + "&tall2=" + tall2, true);
    xmlhttp.send();
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```

function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var resultat = xmlhttp.responseText;
        document.getElementById("resultat").innerHTML = resultat;
    }
}

</script>

<body>
<p>Tall1: <input type="text" id="txtTall1" /><br />
Tall2: <input type="text" id="txtTall2" /></p>
<button id="btnBeregn">Beregn</button>
<p id="resultat"></p>
</body>

```

Som nevnt gjør ikke dette eksemplet noe som ikke JavaScript kunne gjort på egen hånd, men det illustrerer på en forholdsvis enkel måte hvordan vi kan kommunisere med et server-side-script fra et client-side-script. I dette tilfellet både sender vi data til og mottar data fra et server-side-script.

Skrive data

La oss ta vår lille introduksjon til server-side-scripting litt videre og lage et script som skriver til en tekstfil på serveren. Vi tar utgangspunkt i den strukturerte deltagerfila med navnet *innhold.dat*, som vi lagde tidligere i kapitlet, og som hadde følgende innhold:

```

1%Ole Olsen%234%44%78
3%Kari Karisen%65%88%87
4%Per Persen%21%23%200

```

Vi ønsker nå å lage et server-side-script vi kan benytte med følgende URL, som legger til en ny linje i denne fila:

```

http://localhost/registrerdeltager.php?deltagernummer=15&➡
navn=Knut Knutsen&res1=34&res2=54&res3=56

```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

For å få til dette må vi benytte følgende PHP-kode i fila *registrererdeltager.php* på webserveren:

```
<?php

$deltagernummer = $_GET["deltagernummer"];
$navn = $_GET["navn"];
$res1 = $_GET["res1"];
$res2 = $_GET["res2"];
$res3 = $_GET["res3"];

$linje = $deltagernummer . "%" . $navn . "%" . $res1 . "%" . $res2 . "%" . $res3 . "\n";

$fil = fopen("innhold.dat", "a");
fwrite($fil, $linje);
fclose($fil);

echo "Lagret";

?>
```

Igjen kan vi ikke forklare alle detaljer, men i hovedsak henter vi her inn verdiene fra parameterne, og legger de i variabler. Deretter lager vi en variabel kalt **\$linje**, som er det vi ønsker å skrive til fila. Merk deg at i PHP benyttes punktum, og ikke pluss, for å skjøte sammen tekststrenger. Tegnsekvensen \n gir oss linjeskift i tekstufler.

Til slutt åpnes fila for *append*. I motsetning til *write* vil ikke append overskrive eksisterende innhold i fila, men legge til data på slutten. De neste linjene utfører den faktiske skrivingen til fila og lukker deretter fila igjen. Til slutt skrives det ut en melding om at ting gikk bra, som kan være kjekk å ha under testing av scriptet.

Test gjerne å åpne følgende URL i nettleseren for å kontrollere at det fungerer å skrive data til fila:

```
http://localhost/registrererdeltager.php?deltagernummer=15&navn=Knut%20Knutsen&res1=34&res2=54&res3=566
```

Dersom du publiserer scriptet til en ekstern webserver, må du være påpasselig med at det er satt opp rettigheter slik at brukeren som webserveren opererer under, har skrivetilgang til fila.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

En enkel nettside som inneholder et skjema, og som så benytter dette PHP-scriptet, kan se ut som følger.

```
<script>

var xmlhttp;

window.onload = oppstart;

function oppstart() {
    document.getElementById("btnLagre").onclick = lagre;
}

function lagre() {
    var deltagernummer = document.getElementById("txtDeltagernummer").value;
    var navn = document.getElementById("txtNavn").value;
    var res1 = document.getElementById("txtRes1").value;
    var res2 = document.getElementById("txtRes2").value;
    var res3 = document.getElementById("txtRes3").value;
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = statusforandring;

    xmlhttp.open("GET", "registrerdeltagere.php?deltagernummer=" + ↪
        deltagernummer + "&navn=" + navn + "&res1=" + res1 + ↪
        "&res2=" + res2 + "&res3=" + res3, true);
    xmlhttp.send();
}

function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var resultat = xmlhttp.responseText; // Ikke nødvendig å ha med.

        document.getElementById("utskrift").innerHTML = "Lagret til fil";
    }
}

</script>

<body>
<p>
    Deltagernummer: <input type="text" id="txtDeltagernummer" /><br />
    Navn: <input type="text" id="txtNavn" /><br />
    Resultat 1: <input type="text" id="txtRes1" /><br />
    Resultat 2: <input type="text" id="txtRes2" /><br />
    Resultat 3: <input type="text" id="txtRes3" /><br />
</p>
<button id="btnLagre">Lagre</button>
<p id="utskrift"></p>
</body>
```

Legg merke til hvordan vi også her benytter kode som "leser" innholdet fra PHP-scriptet. Vi er imidlertid ikke interessert i hva vi faktisk leser. Mens vi leser PHP-scriptet, sørger det for å lagre data, som var hensikten. Dette blir på samme måte som om vi åpnet linken til PHP-scriptet i nettleseren for å utføre handlingen.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Eksterne XML- og JSON-kilder

Som tidligere nevnt har AJAX-teknologien en sikkerhetsbegrensning i at det kun er anledning til å lese lokale kilder. Samtidig er det svært mange interessante eksterne kilder i XML- og JSON-format.

Noen som kan nevnes, er:

Navn	Beskrivelse	URL
YR	Svært detaljerte data om værvarsel for hele verden	http://om.yr.no/verdata/xml/
Avinor	Sanntidsdata om flytrafikk på Avinors flyplasser	https://avinor.no/konsern/tjenester/flydata
Ruter	Sanntidsdata om kollektivtrafikk	http://labs.ruter.no/how-to-use-the-api.aspx
Difi	En samling av datasett fra offentlige instanser	http://data.norge.no/data

Det ville vært svært upraktisk kontinuerlig å laste ned XML- eller JSON-data som filer til webserveren manuelt. Derfor lages det ofte i stedet en såkalt *proxy*, som er et PHP-script som hver gang det blir etterspurt, henter ned data fra den eksterne kilden.

Slike proxy-script er svært enkle og følger alltid samme oppbygning. Det eneste det gjør, er å lese data fra den eksterne kilden linje for linje og så skrive disse ut som "sine egne" data.

```
<?php
header('Content-type: text/xml');
$fil = fopen("http://www.yr.no/stad/Noreg/Telemark/
Sauherad/Gvarv/varsel.xml", "r");

while($linje = fgets($fil)) {
    echo $linje;
}

?>
```

Dersom du skal hente JSON-data, endrer du første linje til

```
header('Content-type: application/json');
```



I JavaScript benytter vi da PHP-fila (f.eks. kalt *yrproxy.php*) som kilde til AJAX-kallet, slik som vi har gjort tidligere i kapitelet:

```
xmlhttp.open("GET", "yrproxy.php", true);
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Eksempel - Værdata fra YR

Vi skal her forsøke å lage en enkel nettside som plukker data fra YR sin XML-datakilde. Til dette trenger vi et PHP-proxy-script, slik det er vist tidligere i denne seksjonen. Det kan være lurt å se litt nøyne på selve XML-dokumentet og formatet på yr.no før du går videre med eksempelet, da referanseangivningen til de ulike datafeltene er mildt sagt forvirrende.

NB! Det er ingen garanti for at yr.no ikke endrer sine nettadresser eller selve XML-formatet etter at denne boka er trykket, så sjekk nøyne at ressursen fortsatt eksisterer i samme format.

1. Lag fila `yrproxy.php` med følgende innhold:

```
<?php
    header('Content-type: text/xml');

    $fil = fopen("http://www.yr.no/stad/Noreg/Telemark/
        Sauherad/Gvarv/varsel.xml", "r");

    while($linje = fgets($fil)) {
        echo $linje;
    }
?>
```

2. Opprett fila `yr.html` fra fila `mal.html`.
3. Lag følgende innhold på nettsiden:

```
<body>
    <h1 id="stedsnavn"></h1>
    <h3 id="oppdatert"></h3>
    <p id="varsel"></p>
    <img id="symbol" src="" />
</body>
```

4. Legg så inn følgende programkode:

```
<script>

window.onload = oppstart;

var xmlhttp;

function oppstart() {
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = statusforandring;
    xmlhttp.open("GET", "yrproxy.php", true);
    xmlhttp.send();
}

function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var varselXML = xmlhttp.responseXML;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
document.getElementById("stedsnavn").innerHTML = ↵
    varselXML.getElementsByTagName("forecast")[0]. ↵
    getElementsByTagName("text")[0]. ↵
    getElementsByTagName("location")[0].attributes. ↵
    getNamedItem("name").value;

document.getElementById("oppdatert").innerHTML = ↵
    varselXML.getElementsByTagName("meta")[0]. ↵
    getElementsByTagName("lastupdate")[0].childNodes[0].nodeValue;

document.getElementById("varsel").innerHTML = ↵
    varselXML.getElementsByTagName("forecast")[0]. ↵
    getElementsByTagName("text")[0]. ↵
    getElementsByTagName("location")[0]. ↵
    getElementsByTagName("time")[0]. ↵
    getElementsByTagName("body")[0].childNodes[0].nodeValue;

document.getElementById("symbol").src = ↵
    "http://symbol.yr.no/grafikk/sym/b38/" + ↵
    varselXML.getElementsByTagName("forecast")[0]. ↵
    getElementsByTagName("tabular")[0]. ↵
    getElementsByTagName("time")[0]. ↵
    getElementsByTagName("symbol")[0].attributes. ↵
    getNamedItem("var").value+".png";

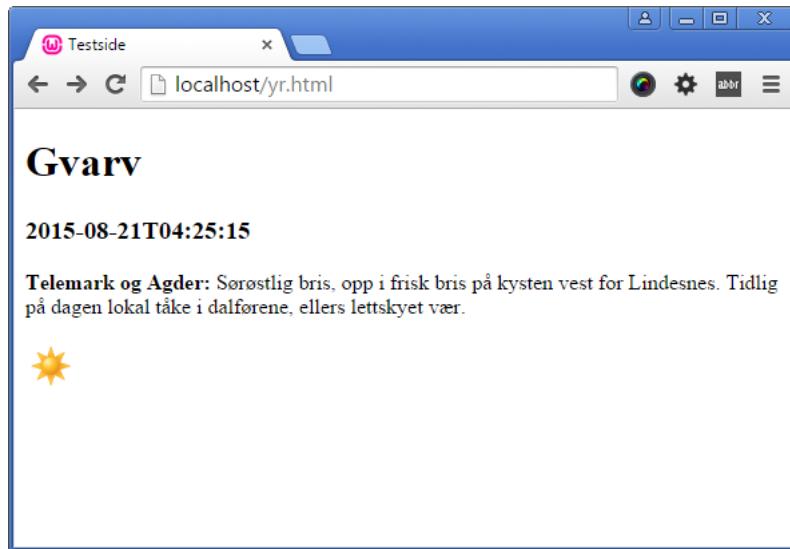
}

</script>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

5. Test nettsiden, og kontroller at du får ut værmeldingen for Gvarv. Du kan selv endre til værmelidingen for andre steder ved å endre nettadressen som er angitt i *yrproxy.php*.

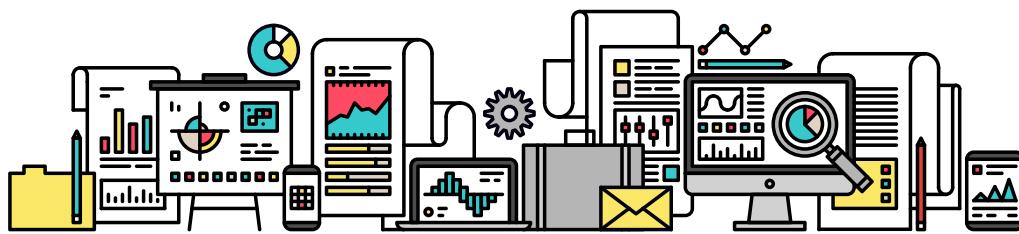


JavaScript-koden i dette eksempelet består stort sett av å hente ut verdier fra yr.no sin XML-feed gjennom *yrproxy.php* på serveren. Feltene **stedsnavn**, **oppdatert** og **varsel** settes direkte til verdier vi får fra XML. Selve utplukksangivelsene mot XML-fila er forholdsvis omfattende og i flere nivåer. Dersom du ser på XML-dataene fra yr.no, finner du imidlertid igjen elementene.

Vi ser at -taggen med ID **symbol** også får informasjon fra XML-fila, men det som ligger i XML-fila, er kun en symbolkode slik som *01d*, *01n.24*, *04* osv. Samtidig har yr.no en rekke PNG bildefiler plassert på nettadressen *http://symbol.yr.no/grafikk/sym/b38/*. Disse har filnavn som sammenfaller med symbolkodene, og filendelsen *.png*. Typisk vil bildet for *01d* dermed få nettadressen *http://symbol.yr.no/grafikk/sym/b38/01d.png*. Vi kan derfor hente alle nødvendige symboler fra yr.no sine nettsider ved å bygge sammen nettadressen til det aktuelle bildet.

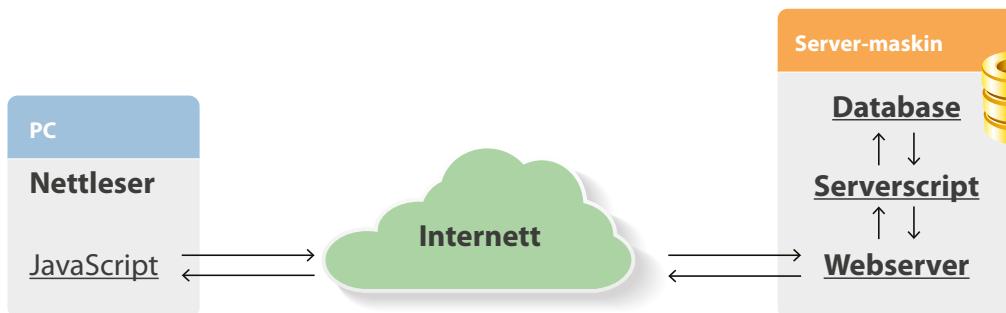
VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Databaser



Denne seksjonen tar som en forutsetning at du alt kjenner databaser og SQL.

JavaScript kan i seg selv ikke kommunisere med en database på en enkel måte uten eksterne biblioteker. PHP kan derimot enkelt koble til en database, og vi kan dermed benytte JavaScript mot et PHP-script som igjen kommuniserer mot en database. Det blir på samme måte som vi har benyttet PHP mellom JavaScript og filer på serveren tidligere.



I eksemplene i denne boka vil vi benytte databasesystemet MySQL, da dette er lett å arbeide mot fra PHP. MySQL følger med i pakkene WAMP og MAMP.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Hente ut data fra database

Når vi skal hente ut data fra en database, er det en fordel at PHP-scriptet presenterer data på en strukturert form. Vi kan da benytte linjeskift og et skilletegn, på samme måte som vi gjorde i en strukturert datafil. Eksempelet under vil produsere samme datastruktur som fila med deltagere vi så på tidligere i kapitelet. Kodden går ut fra at vi har en tabell kalt **deltager** i en database kalt **deltagerbase**.

```
<?php

$server = "localhost";
$brukernavn = "root";
$passord = "";
$databasenavn = "deltagerbase";
$sql = "SELECT deltagernummer, navn, res1, res2, res3 FROM deltager
        ORDER BY deltagernummer ASC";

//=====
$tilkobling = mysqli_connect($server, $brukernavn, $passord, $databasenavn);

$datasett = $tilkobling->query($sql);

while ( $rad = mysqli_fetch_array($datasett) ) {
    echo $rad["deltagernummer"] . "%" . $rad["navn"] . "%" .
        $rad["res1"] . "%" . $rad["res2"] . "%" . $rad["res3"] . "\n";
}

?>
```

deltager	
! deltagnumer	INT
◆ navn	VARCHAR(45)
◆ res1	INT
◆ res2	INT
◆ res3	INT
Indexes	

Dersom du lager en database slik som beskrevet, kan du teste PHP-scriptet gjennom en nettleser. Det kan være at du må endre litt på påloggingsinformasjonen for å få det til å passe til din database. Husk imidlertid på å åpne fila gjennom den lokale webserveren.



Når vi skal benytte PHP-scriptet i JavaScript, gjør vi det på samme måte som når vi leser en annen ressurs med AJAX. For JavaScript er det uvesentlig at PHP-scriptet henter sine data fra en database:

```
xmlhttp.open("GET", "hentresultaterfradatabase.php", true);
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lagre data i en database

Lagring av data i en database kan vi bygge opp på samme måte som når vi skrev data til en fil, men i stedet for å lage en linje som skrives til fila, lager vi heller en **INSERT**-setning som sendes til databasen. Ideen med parameterne er den samme.

```
<?php
```

```
$server = "localhost";
$brukernavn = "root";
$passord = "";
$databasenavn = "deltagerbase";

//=====

$stilkobling = mysqli_connect($server, $brukernavn, $passord, $databasenavn);

$deltagernummer = $_GET["deltagernummer"];
$navn = $_GET["navn"];
$res1 = $_GET["res1"];
$res2 = $_GET["res2"];
$res3 = $_GET["res3"];

$sql = "INSERT INTO deltager(deltagernummer, navn, res1, res2, res3) ←
VALUES(" . $deltagernummer . "," . $navn . "," . $res1 . "," . ←
$res2 . "," . $res3 . ")";
$stilkobling->query($sql);

echo "Lagret";

?>
```

Du kan teste scriptet gjennom å åpne en nettadresse lignende dette i nettleseren:

```
http://localhost/registrerdelgeridb.php?deltagernummer=15&←
navn=Knut Knutsen&res1=34&res2=54&res3=56
```

Fra JavaScript må vi da bygge sammen nettadressen som skal åpnes:

```
xmlhttp.open("GET", "registrerdelgeridb.php?deltagernummer=" + ←
deltagernummer + "&navn=" + navn + "&res1=" + res1 + ←
"&res2=" + res2 + "&res3=" + res3, true);
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!



Eksempel - Highscorespillet

Vi skal nå lage det som må være verdens enkleste spill, og en tilhørende highscoreliste. Spillet gir deg en tilfeldig generert poengsum når det starter. Denne poengsummen og ditt navn kan du så sende inn til en database på webserveren, og deretter leser nettsiden tilbake en highscoreliste fra webserveren som presenteres for deg.

- Opprett en database kalt *spill* med følgende tabell kalt *highscore*, og fyll tabellen med litt testdata:

highscore	
tidspunkt	TIMESTAMP
navn	VARCHAR(255)
poeng	INT(11)
Indexes	

- Lag et PHP-script med filnavnet **skrivehighscore.php**, som skal skrive til denne databasen. Merk deg at påkoblingsinformasjonen kan være annerledes hos deg avhengig av databaseoppsett:

```
<?php

$server = "localhost";
$brukernavn = "root";
$passord = "";
$databasenavn = "spill";

//=====

$stilkobling = mysqli_connect($server, $brukernavn, $passord,
$databasenavn);

$navn = $_GET["navn"];
$poeng = $_GET["poeng"];

$sql = "INSERT INTO highscore(tidspunkt, navn, poeng) VALUES(NOW(), '$navn', '$poeng')";
$stilkobling->query($sql);

echo "Lagret";
?>
```



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

3. Testkjør scriptet gjennom en nettadresse slik som denne (varierer etter plassering og type webserver). Kontroller at data blir plassert i databasen:

```
http://localhost/skrivehighscore.php?navn=Tom Heine&poeng=45
```

4. Lag et tilsvarende PHP-script med filnavn **lesehighscore.php**, som henter ut data fra databasen:

```
<?php

$server = "localhost";
$brukernavn = "root";
$passord = "";
$databasenavn = "spill";
$sql = "SELECT navn, poeng FROM highscore ORDER BY poeng DESC LIMIT 10";

//=====

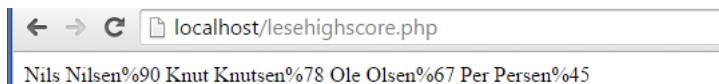
$tilkobling = mysqli_connect($server, $brukernavn, $passord, $databasenavn);

$datasett = $tilkobling->query($sql);

while ( $rad = mysqli_fetch_array($datasett) ) {
    echo $rad["navn"] . "%" . $rad["poeng"] . "\n";
}

?>
```

5. Testkjør PHP-scriptet, og kontroller at du får ut data. Linjeskift vil ikke vises, da nettleseren tolker dataene som en nettside. I kildekoden til nettsiden kan du imidlertid se dem.



6. Lag en ny nettside kalt *highscore.html* ut fra *mal.html*, og legg inn følgende innhold på nettsiden:

```
<body>
    <h1 id="poengsum"></h1>
    <p>Navn: <input type="text" id="txtNavn" /></p>
    <button type="button" id="btnRegistrer">Registrer</button>
    <p id="utskrift"></p>
</body>
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

7. Legg inn følgende programkode:

```
<script>

var xmlhttp;
var poeng;

window.onload = oppstart;

function oppstart() {
    poeng = Math.floor(Math.random()*10000);

    document.getElementById("poengsum").innerHTML = "Du fikk " + poeng + " poeng";
    document.getElementById("btnRegistrer").onclick = registrer;
}

function registrer() {
    var navn = document.getElementById("txtNavn").value;
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = sendt;

    xmlhttp.open("GET", "skrivehighscore.php?navn=" + navn +
        "&poeng=" + poeng, true);
    xmlhttp.send();
}

function sendt() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {

        xmlhttp = new XMLHttpRequest();
        xmlhttp.onreadystatechange = lest;
        xmlhttp.open("GET", "lesehighscore.php", true);
        xmlhttp.send();

    }
}

function lest() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var resultat = xmlhttp.responseText;

        var data = xmlhttp.responseText;
        var linjer = data.split("\n");

        var utskrift = "";

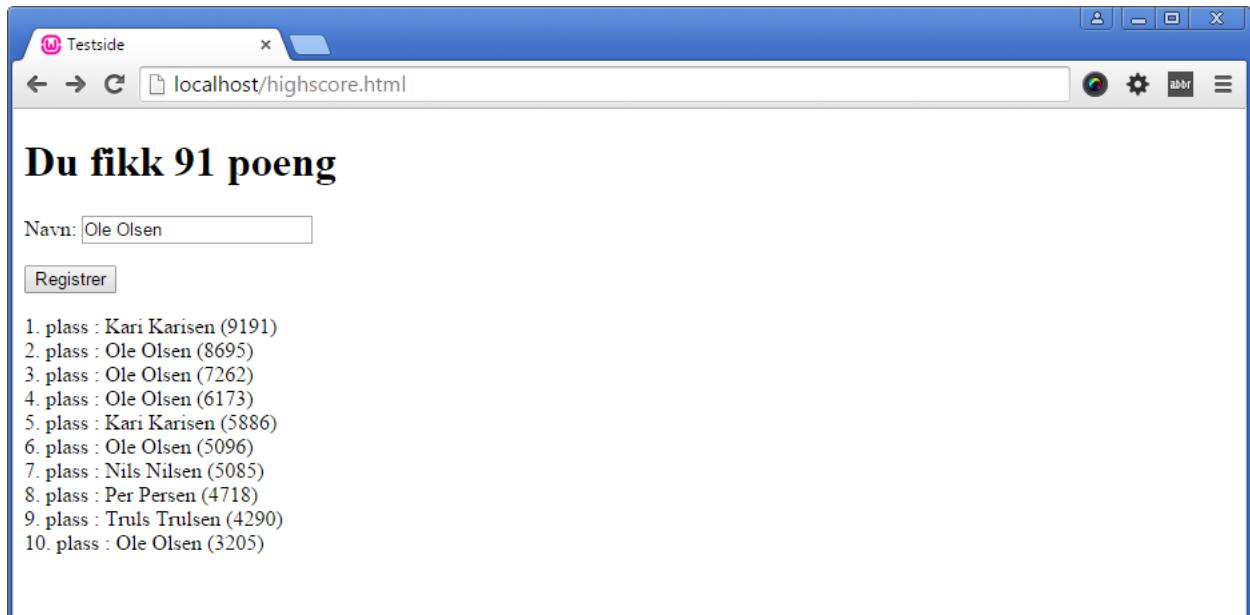
        for (var i = 0; i < linjer.length; i++) {
            var datafelt = linjer[i].split("%");
            if(datafelt.length === 2) {

                utskrift = utskrift + (i + 1) + ". plass : " +
                    datafelt[0] + " (" + datafelt[1] + ")<br />";
            }
        }
    }
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

```
document.getElementById("utskrift").innerHTML = utskrift;  
oppstart();  
  
}  
  
</script>
```

8. Test nettsiden, og kontroller at du får registrert resultatene, og at highscorelisten oppdateres.



Når nettsiden lastes, trekkes det et tilfeldig tall som brukeren får som poengsum. Denne poengsummen vises også for brukeren. Når brukeren så trykker på **btnRegistrer**, vil vi i hendelsen **registrer** forsøke å lese PHP-scriptet *skrivehighscore.php*, med **navn** og **poeng** som parameter. Dette scriptet produserer en INSERT-setning for å få plassert data i databasetabellen **highscore**. Når dette scriptet er ferdig, vil hendelsen **sendt** utføres.

PHP-scriptet produserer ikke noe data, så vi er ikke interessert i hva vi klarte å lese. Vi starter derimot en ny lese-operasjon mot PHP-scriptet *lesehighscore.php*. Dette scriptet henter ut data fra databasen på følgende format:

```
navn1%poeng1  
navn2%poeng2  
navn3%poeng3
```

Utskriften er også sortert synkende på poengsum fra PHP-scriptet. Hendelsen **lest** blir utført når leseoperasjonen er ferdig, og her splitter vi først opp resultatet i linjer, og plasserer dette i arrayen **linjer**. Hver linje splittes så opp på %-tegnet, og dersom vi får to deler (navn og poengsum), så skrives denne informasjonen ut til nettsiden.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

12 Objektorientering

I dette kapitlet vil du lære

- hva objektorientert programmering innebærer
- om klasser og objekter
- hva som ligger i begrepene arv og polymorfi
- om objektorientering i JavaScript

Objektorientert programmering er en programmeringsteknikk, eller mer korrekt et *programmeringsparadigme*, som har snudd opp ned på måten man programmerer. Konseptet stammer blant annet fra arbeidet nordmennene Kristen Nygaard og Ole-Johan Dahl gjorde gjennom å utvikle det objektorienterte programmeringsspråket Simula på 60-tallet. Det var allikevel først på 90-tallet at objektorienterte programmeringsspråk kom på moten, og i dag benytter de fleste programmeringsspråkene en objektorientert tankegang.

JavaScript har en noe annen tilnærming til hvordan objektorientert programkode skrives enn det mer tradisjonelle språk som C# og Java har. Allikevel er JavaScript et programmeringsspråk der man kan utvikle objektorientert programkode på lik linje med andre språk.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Vi vil i dette kapitlet fokusere på å vise hvordan ulike objektorienterte prinsipper gjøres i JavaScript. For mer informasjon om objektorientert programmering generelt henviser vi til andre kilder (f.eks. IT 2, basisbok). Kapitelet vil også kun introdusere hvordan objektorientering kan benyttes i JavaScript, så for mer detaljert informasjon må du gå til andre kilder.

Klasser og objekter

Det er ikke lett å forklare hva *klasser* og *objekter* er med få ord, men vi skal allikevel forsøke å gjøre en forenkling som vil gi deg det du trenger å vite for å komme i gang.

Det første vi kan gjøre, er å tenke på klasser som en definisjon av en datatype. For eksempel kan vi lage en klasse for å representere en person. Denne vil da kanskje bestå av to tekstverdier for fornavn og etternavn, samt en tallverdi for alder.

Det som er viktig å merke seg, er at en klasse vanligvis kun definerer hvilke data som skal representeres, og ikke hvilke verdier som skal benyttes. Vi kan altså tenke på en klasse som en slags mal, der vi har åpne felter for de faktiske verdiene.

Disse feltene kalles gjerne *klassevariabler*, men de kan også kalles *egenskaper*, *datamedlemmer* eller *feltvariabler* – kjært barn har mange navn.

Å fylle denne malen med verdier gjør vi idet vi oppretter *objekter* fra klassen. Dette blir da de faktiske elementene som skal lagres i minnet på maskinen. For eksempel kan vi opprette et antall objekter fra en person-klasse, hvor hvert objekt representerer én bestemt person med konkrete verdier for fornavn, etternavn og alder.

Objekter kalles også ofte instanser av klassen.



En klasse kan også ha det vi kaller *metoder*. En metode er en funksjon som defineres i klassen, og som arbeider med verdiene som objektet av klassen holder på. En person-klasse kunne for eksempel hatt metoder for å hente ut navn eller beregne fødselsår ut fra alder.

I denne boka har vi sett mange eksempler på metoder, men vi har inntil nå omtalt dem som funksjoner. Det er kun de funksjonene som ikke er koblet til en klasse, vi egentlig bør kalle funksjoner. Alt annet er det mer riktig å kalle metoder.

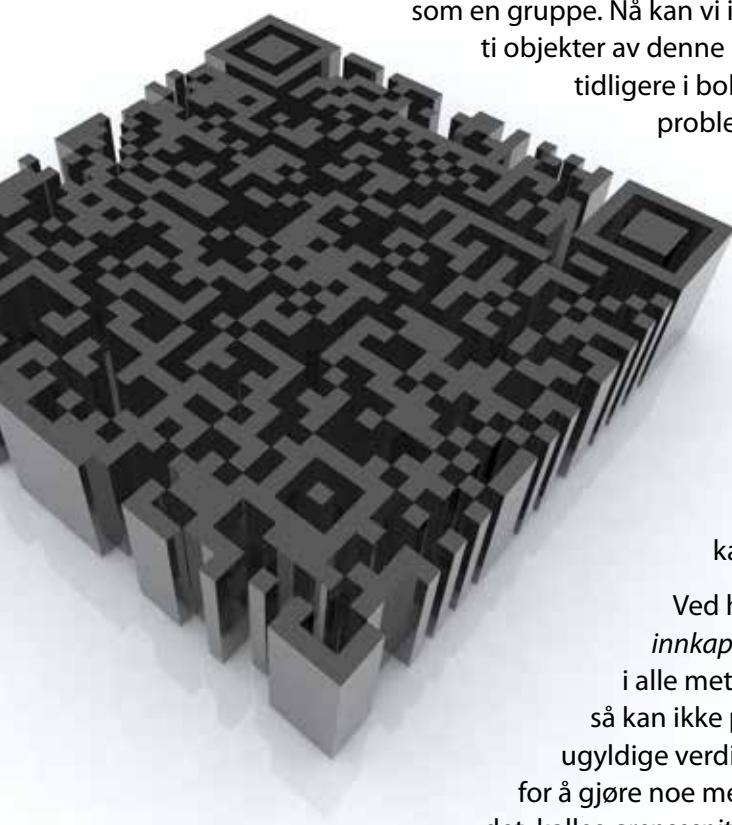


VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

En klasse har også en helt spesiell metode som kalles *konstruktør*. Det er denne metoden vi kjører når vi ønsker å opprette et nytt objekt fra klassen. Ofte har konstruktørene også parametere, som gjør at egenskapene til objektet fylles med verdier vi angir som argumenter til konstruktøren.

En av ideene med å arbeide med klasser og objekter er å oppnå *modularitet*. Det vil si at vi samler variabler og programkode som tilhører en bestemt del av programmet i en modul, eller mer korrekt en klasse. På den måten er det enklere å skille de ulike delene av programmet fra hverandre og få en mer oversiktlig struktur.

Tenk deg selv tilfellet med å behandle ti personer i programmet om vi måtte ha håndtatt ti ulike variabler for fornavn, ti ulike variabler for etternavn og ti variabler for alder, samt selv skrevet programkode for å behandle tre og tre variabler som en gruppe. Nå kan vi i stedet definere en klasse og behandle ti objekter av denne klassen. Som du kanskje husker, har vi tidligere i boka benyttet assosiative arrayer til å løse dette problemet i JavaScript, og en assosiativ array er som vi skal se, faktisk et objekt med en udefinert klassespesifikasjon.



Med tankegangen om modularitet kommer også ideen om *abstraksjon*. Dette vil si at når klassen er ferdig skrevet av oss eller andre, slipper vi å bekymre oss for hvordan verdier lagres og metoder fungerer internt. Vi bare benytter klassen og dens metoder. Det er litt som begrepet *sort boks*, som vi omtalte i kapitlet funksjoner.

Ved hjelp av abstraksjon får vi også til såkalt *innkapsling*. Dette vil si at så lenge vi tar feilsjekker i alle metoder som endrer egenskapene til objektet, så kan ikke programkode på utsiden av objektet sette ugyldige verdier på disse. De metodene som vi kan bruke for å gjøre noe med objektet eller endre på egenskapene til det, kalles *grensesnittet*.

Hver del av systemet blir altså en selvstendig innkapslet enhet, og siden vi kun kommuniserer med objekter gjennom grensesnittene, er det liten sjanse for at programkode i de ulike delene av systemet forstyrrer hverandre. Dette er ofte en stor kilde til feil, da programkode ett sted i applikasjonen endrer en variabel som programkode et annet sted i applikasjonen arbeider med.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Arv

Et annet begrep det er viktig å kjenne til innenfor objektorientert programmering, er *arv*. Ideen med arv er at vi kan ta utgangspunkt i en eksisterende klasse og så spesialisere denne ytterligere for videre bruk. Under spesialiseringen kan vi legge til egenskaper og metoder, samt overkjøre allerede eksisterende metoder med nytt innhold.

For å forstå arv kan det derfor være lurt å tenke på dette som å utvide en eksisterende klasse med mer funksjonalitet. Merk deg imidlertid at selv om det å utvide en eksisterende klasse sparer oss for mye tid kontra å skrive den nye klassen fra bunnen av, er dette kun en bieffekt. Hovedårsaken til at man benytter arv, er at klassene også får en relasjon i et såkalt arvehierarki.

Dersom vi har et arvehierarki bestående av klassen *Kjøretøy*, som utvides til klassene *Bil* og *Buss*, kan vi si at *Kjøretøy* er en fellesnevner for de to andre. Dette medfører at vi kan omtale både *Bil* og *Buss* som spesialiseringer av det mer generelle begrepet *Kjøretøy*. Denne teknikken kalles *polymorfisme*, og alle steder i programkoden der vi arbeider med *Kjøretøy*, vil også objekter av typen *Bil* og *Buss* passe inn, ettersom de kan opptre som (og er) et *Kjøretøy*.



Hvorfor objektorientering?

En av de største fordelene med objektorientert programmering er som nevnt tidligere at våre programmer blir modulbaserte, og vi får kontroll på en begrenset kommunikasjon mellom de ulike delene. Dette gjør det mulig for flere personer å utvikle ulike deler av systemet samtidig og likevel vite at de vil fungere sammen. Dette forutsetter at man på forhånd har blitt enig om grensesnittet for kommunikasjonen mellom modulene.

I tillegg til dette kan vi lett gjenbruke ulike deler av systemet i flere andre prosjekter og benytte andres klasser i våre prosjekter, takket være abstraksjonen som gjør at det blir enkelt å benytte avansert programkode. Innkapslingen vil også sikre at vi benytter klassene på en korrekt måte.

Når vi programmerer objektorientert, vil vi også fort oppdage at mange av problemene vi skal løse, minner svært mye om hverandre. Derfor har man utviklet såkalte *design patterns*, som ganske generelt forteller hvordan ulike grupper av problemer best mulig skal løses. Ved hjelp av disse "malene" kan man lett få gode objektorienterte løsninger på de fleste problemer.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Objektorientering i JavaScript

Som tidligere nevnt løses objektorientering ganske utradisjonelt i JavaScript. Vi kan lage objekter uten først å ha tydelig definert noen klasse, slik som er vanlig, og når vi først lager det som kan minne om klasser, defineres disse ofte ved hjelp av litt spesielle funksjoner.



Versjon 6 av JavaScript (ECMAScript® 2015) introduserer en del nye teknikker som gjør objektorientering i JavaScript mer likt andre språk. Det tar imidlertid litt tid før alle nettlesere støtter denne versjonen.

Ettersom objektorientering i JavaScript er ganske "løst" definert, finnes det en mengde måter å gjøre det på. Vi vil her forsøke å vise en av de enklere måtene, som samtidig samsvarer mye med slik det gjøres i andre programmeringsspråk.

Vi har allerede sett flere eksempler på objekter. Blant annet er en assosiativ array et objekt. Når vi skriver følgende kode, så lager vi et objekt med instansnavn **person1** ut fra en klasse som vi aldri definerer spesifikt. Egenskapene til denne klassen er **fornavn**, **etternavn** og **alder**:

```
var person1 = {fornavn: "Per", etternavn: "Olsen", alder: 45}
```

Vi kan også lage objekter ut fra innebygde klasser ved hjelp av nøkkelordet **new** og et kall til **konstruktøren**. Konstruktøren er en spesiell funksjon som setter "oppstartsverdier" på objektet. Vi har tidligere blant annet opprettet objekter ut fra klassen **Date**.

```
var tid = new Date();
```

Vi kan så benytte og endre egenskapene til objektet slik som

```
alert(person1.alder);  
person1.alder = 67;
```

Mange objekter har også metoder, slik som

```
alert(tid.getSeconds());  
alert(tid.toLocaleDateString());
```

Selv variabler av typen string og integer er egentlig objekter og har innebygget både egenskaper og metoder. Ved disse *primitive datatypene* benytter vi imidlertid ikke nøkkelordet **new** ved opprettelse:

```
var tekst = "hei på deg";  
alert(tekst.length);  
alert(tekst.splice(-2));
```

Andre innebygde objekter, slik som **Math**, har også egenskaper og metoder:

```
var tall = Math.random();  
var omkrets = Math.PI * 4 * 4;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Egne klasser

Egne klasser lages i JavaScript ved hjelp av funksjoner. Konstruktøren til klassen er selve funksjonen, og parameterne til funksjonen er dermed parameterne til konstruktøren. Metoder til klassen er funksjoner som defineres inne i selve klassefunksjonen på en litt spesiell måte der muligheten for at funksjoner kan være verdier til variabler/egenskaper, benyttes.

```
function Person(fornavn, etternavn, alder) {
    this.fornavn = fornavn;
    this.etternavn = etternavn;
    this.alder = alder;
    this.bliEldre = function() {
        this.alder++;
    }
}
```

Nøkkelordet **this** henviser litt forenklet forklart til egenskaper som knyttes til klassen. Vi lager altså en egenskap kalt **fornavn** som får verdien til parametret **fornavn**. Nøkkelordet **this** er avgjørende for å skille på disse to tingene som har samme navn fra hverandre.

Et eksempel på bruk av klassen vil kunne være:

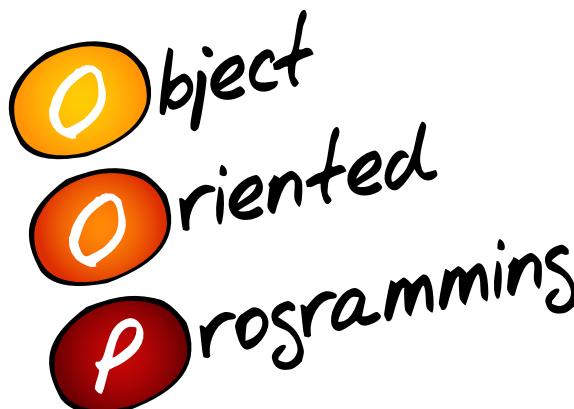
```
window.onload = oppstart;

function oppstart() {
    var person1 = new Person("Ole", "Olsen", 45);
    var person2 = new Person("Per", "Persen", 78);

    alert(person1.fornavn); // Ole
    alert(person2.alder); // 78
    person2.bliEldre();
    alert(person2.alder); // 79
}
```

De første gangene man ser objektorientert kode med klassedefinisjoner og bruk gjennom objekter, så virker teknikken utrolig komplisert og unødvendig tungvint. Det er først når man har forstått konseptet, at alle fordelene med OOP virkelig synes.

TIPS



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Arv

En forenklet variant av arv i JavaScript vil være at hver nye klasse oppretter et objekt av klassen de skal arve fra, og legger til nye egenskaper og metoder på dette. Vi kan utvide eksempelet vi nettopp så på, som omhandler personer, til å ha to klasser som spesifiserer/arver **Person**-klassen.

Den ene klassen er **Elev**, som i tillegg til å være en person også holder styr på antall dager som er skulket. Klassen har en metode som kjøres hver gang en ny skulk skal bli registrert. Klassen **Lærer** har i tillegg til opplysningene en person har, også et ansattnummer.

```
window.onload = oppstart;

function Person(fornavn, etternavn, alder) {
    this.fornavn = fornavn;
    this.etternavn = etternavn;
    this.alder = alder;
    this.bliEldre = function() {
        this.alder++;
    }
}

function Elev(fornavn, etternavn, alder) {
    var p = new Person(fornavn, etternavn, alder);
    p.dagerSkulket = 0;
    p.skulk = function() {
        p.dagerSkulket++;
    }
    return p;
}

function Laerer(fornavn, etternavn, alder, ansattnummer) {
    var p = new Person(fornavn, etternavn, alder);
    p.ansattnummer = ansattnummer;
    return p;
}

function oppstart() {
    var person1 = new Elev("Ole", "Olsen", 45);
    var person2 = new Laerer("Per", "Persen", 78, "332000");

    alert(person1.fornavn);
    alert(person2.alder);
    person2.bliEldre();
    alert(person2.alder);

    person1.skulk();
    alert(person1.dagerSkulket);
    alert(person2.ansattnummer);
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Vær klar over at det i JavaScript er mange måter å gjøre objektorientering på.

Måten vi her har utført klasser og arv på, er den vi mener er enklest å forstå for en nybegynner, og den som i oppbygning minner mest om hvordan det løses i andre programmeringsspråk.

Spesielt metoden for arv har imidlertid noen svakheter ved seg. Blant annet kan vi ikke sjekke hvilken type et objekt er ved hjelp av nøkkelordet `instanceof`. Vil du lære mer om objektorientering i JavaScript, vil vi derfor anbefale at du starter med å utforske teknikken *prototyping* som gir en bedre, men mer avansert versjon av objektorientering.



VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Hurtigreferanse

VARIABLER OG KONSTANTER	262
Lage (deklarer) en variabel	
Angi verdi på en variabel som allerede er lagd	
Angi ny verdi ut fra gammel verdi	
Modulo-operatoren	
Datatyper med primitive verdier	
Konvertere tekst til tall	
Konvertere tall til tekst	
Formatere et tall (bestemme antall desimaler)	
Lage en konstant	
TEKST	263
Sette sammen tekst	
Bruke escape-tegn for linjeskift og tabulator	
Trekke ut en tekst fra en annen tekst	
Søke etter en tekst i en annen tekst	
SKJEMAELEMENTER	264
Lage tekstboks	
Hente verdi fra tekstboks	
Lage knapp	
Lage avkrysningsbokser	
Sjekke om avkrysningsboks er markert	
Lage nedtrekksliste/listeboks	
Hente verdi fra nedtrekksliste/listeboks	
Lage radioknapper	
Sjekke om radioknapp er markert	
KONTROLLSTRUKTURER.....	266
Lage en if-test	
Lage en if-else-test	
Lage en if-else if-else-test	
Relasjonsoperatorer	
if-test med og	
if-test med eller	
Lage en while-løkke	
Lage en før-løkke	
FUNKSJONER.....	268
Lage en funksjon som returnerer en verdi	
Bruke (kalle opp) funksjonen	
Lage en funksjon som ikke returnerer en verdi	
Bruke (kalle opp) funksjonen	
ARRAYER.....	269
Lage en array	
Legge inn verdier (elementer) i en array	
Få tak i en verdi i en array	
Iterere gjennom en array	
Lage en array og samtidig legge inn verdier	
Legge til en verdi (element) i slutten av en array	
Slette elementer i en array	
Søke etter et element i en array	
Lage en to-dimensjonal array	
Få tak i en verdi i en to-dimensjonal array	
ASSOSIATIVE ARRAYER.....	270
Lage og sette verdier i en assosiativ array	
Få tak i en verdi i en assosiativ array	
Lage en array som inneholder assosiative arrayer (to-dim.)	
Iterere gjennom en assosiativ array	
Sjekke om en nøkkel finnes i en assosiativ array	

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

HENDELSER.....	271
Registrere en lytterfunksjon	
Lage en lytterfunksjon	
Fjerne en lytterfunksjon	
Musehendelser	
Informasjon i hendelsesobjektet til musehendelser	
Drag and drop	
Tastaturhendelser	
Informasjon i hendelsesobjektet til musehendelser	
Starte en tidsstyrтt hendelse	
Stoppe en tidsstyrтt hendelse	
GRAFIKK	273
Lage canvas	
Hente ut context	
Tegne en linje	
Tegne en sirkel	
Tegne et rektangel	
Endre linjefarge og tykkelse	
Tegne fyll	
ENDRE HTML-DOKUMENTET.....	274
Lage et nytt element og legge det til i dokumentet	
Fjerne et element fra dokumentet	
ENDRE CSS	275
Sette nye stiler direkte	
Endre CSS-klasse	
Fjerne CSS-klasse	
LYD OG VIDEO.....	275
Spille av lydfil	
Sette inn lydavspiller	
Sette inn videoavspiller	
Manipulere en avspiller	
Detektere avspillingsslutt	
DYNAMISK INNHOLD.....	276
Laste inn data fra fil	
Dele opp strukturert data	
Hente ut XML-data	
Hente ut verdier og attributtverdier fra XML-data	
Hente ut JSON-data	
Hente ut verdier fra JSON-data	
Skrive til web storage	
Lese fra web storage	
MATTE	278
Mattefunksjoner	
Vinkelberegnung	
Konvertere mellom grader og radianer	
Generere tilfeldige tall	

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Variabler og konstanter

Lage (deklarere) en variabel

```

Kodeordet var      Variabelnavn
                   (skal alltid ha liten forbokstav)
                   ↓
var antallBiler = 234;           "Settes lik"
                                ↓
                                Verdi
  
```

Angi verdi på en variabel som allerede er lagd

```
antallBiler = 10;
```

Angi ny verdi ut i fra gammel verdi

```

Ny verdi          Gammel verdi
↓                ↓
antallBiler = antallBiler + 1; // Øker med én

// Kortformer:
antallBiler++;      // Øker med én
antallBiler--;      // Minsker med én
antallBiler += 3;    // Øker med 3
antallBiler -= 5;    // Minsker med 5
antallBiler *= 2;    // Ganges med 2
antallBiler /= 6;    // Deles på 6
  
```

Modulo-operatoren

```

var epler = 47;
var personer = 3;
var eplerPrPers = Math.floor(epler / personer);
var tilOvers = epler % personer ;
  
```

Datatyper med primitive verdier

```

var tall1 = -3;        // Heltall
var tall2 = 5.7;       // Desimaltall
var tekst = "Hei";     // Tekst
var test = false;       // true eller false
  
```

Konvertere tekst til tall

```

var tekst = "2344.5";
var tall = parseFloat(tekst);
  
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Konvertere tall til tekst

```
var tall = 2344.5;
var tekst = tall.toString();
```

Formatere et tall (bestemme antall desimaler)

```
var tall = 29.3638;           Antall desimaler
var formatertTall = tall.toFixed(2);
alert(formatertTall); // 29.36
```

Lage en konstant

```
Kodeordet const
const DAGER_I_UKA = 7;
```

Konstantnavn (skal alltid ha STORE bokstaver)

Tekst

Sette sammen tekst

```
Konkateneringstegegn (sette sammen tekst)
var tid = "Jul";
var melding = "Ikke lenge igjen til " + tid;

alert(melding); // Ikke lenge igjen til Jul
```

Bruke escape-tegn for linjeskift og tabulator

```
// Angi linjeskift i filer:
var nyLinje = "\n";
var tekst1 = "Navn:" + nyLinje + "Roald Grønningen";

// Angi linjeskift i nettsiden:
var nyLinje = "<br />";
var tekst2 = "Navn:" + nyLinje + "Roald Grønningen";

// Angi anførselstegn:
var anf = "\"";
var tekst3 = "Han er helt " + anf + "syk" + anf;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Trekke ut en tekst fra en annen tekst

```

Indeks 0           Start-indeks   Antall tegn
↓                 ↓             ↙
var teksten = "Dramatisk!";
var uttrekk = teksten.substr(3, 5);
alert(uttrekk); //matis

```

Søke etter en tekst i en annen tekst

```

Søkeord           Start-indeks
↓               ↙
var tekst = "Her var det mye trafikk. Tror jeg går.";
var indeks = tekst.indexOf("trafikk", 0);
alert(indeks); // 16 Hvis ikke funnet blir indeks -1

```

Skjemaelementer

Lage tekstboks

```

Verdi som vises som standard i tekstboksen
↓
<input type="text" id="txtNavn" value="startverdi"></input>

```

Hente verdi fra tekstboks

```
var navn = document.getElementById("txtNavn").value
```

Lage knapp

```
<button id="btnKnapp" type="button">Klikk meg...</button>
```

Lage avkrysningsbokser

```

<input type="checkbox" name="chkPizza" />Jeg like pizza<br />
<input type="checkbox" name="chkPasta" checked="checked" />Jeg
liker pasta

```

Sjekke om avkrysningsboks er markert

```

if (document.getElementById("chkPizza").checked === true) {
    // Kode
}

```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Lage nedtrekksliste/listeboks

```
Antall elementer som er synlige av gangen  
↓  
<select id="lstKunder" size="1">  
  <option value="34212">Ole Olsen</option>  
  <option value="5514">Per Persen</option>  
  <option value="21345">Nils Nilsen</option>  
</select>
```

Hente verdi fra nedtekksliste/listeboks

```
var kundenummer = document.getElementById("lstKunder").value
```

Lage radioknapper

```
Alle elementer i en gruppe må ha samme navn  
↓  
<input type="radio" name="kjonn" id="rbtnKjonnMann"  
value="mann" />Mann<br />  
<input type="radio" name="kjonn" id="rbtnKjonnKvinne"  
value="kvinne" />Kvinne
```

Sjekke om radioknapp er markert

```
if (document.getElementById("rbtnKjonnMann").checked === true) {  
  // Kode  
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Kontrollstrukturer

Lage en if-test

```

Kodeordet if           Test   Test-operator
↓                   ↓       ↓
if (antallBiler < 20) {
    // Her skrives koden for hva som skal
    // gjøres hvis testen slår til
}

```

Lage en if-else-test

```

if (antallBiler < 20) {
    // Her skrives koden for hva som skal
    // gjøres hvis testen slår til
}
else {
    // Her skrives koden for hva som skal
    // gjøres hvis testen ikke slår til
}

```

Lage en if-else if-else-test

```

if (antallBiler < 10) {
    // Koden som skal utføres hvis testen slår til
}
else if (antallBiler < 20) {
    // Koden som skal utføres hvis testen slår til og
    // testen/ene ovenfor ikke slår til
}
else {
    // Koden som skal utføres hvis ingen test slår til
}

```

Relasjonsoperatører

`==` er lik

`!=` ikke lik (forskjellig fra)

`<` mindre enn

`>` større enn

`<=` mindre enn, eller lik

`>=` større enn, eller lik

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

if-test med og

```

        Og
        ↓
if (antallBiler > 5 && antallBiler < 20) {
    // Kode som skal utføres hvis testen slår til
}

```

if-test med eller

```

        Eller
        ↓
if (antallBiler < 10 || antallBiler > 20) {
    // Kode som skal utføres hvis testen slår til
}

```

Lage en while-løkke

```

Variabel vi tester på
↓
var teller = 0;
Kodeordet while   Test
↓
while (teller < 5) {
    // Kode vi skriver her kjøres så mange ganger som vi
    // går gjennom løkka (i dette eks. 5 ganger)
    ↓ Forandre verdi på variabelen
    teller++;
}

```

Lage en for-løkke

```

Kodeordet for
↓
for (var teller = 0; teller < 5; teller++) {
    // Kode vi skriver her kjøres så mange ganger som vi
    // går gjennom løkka (i dette eksemplet 5 ganger)
}

```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Funksjoner

Lage en funksjon som returnerer en verdi

```

Kodeordet      Funksjonsnavn (skal alltid
function       ha liten forbokstav)      Parametere (variabler som overfører
                                            verdier til funksjonen). Flere parametere
                                            skiller med komma.

↓                         ↓
function leggSammen(tall1, tall2) {
    // Kode inne i funksjonen
    ↓                         ↓ Lokal variabel (gjelder ikke utenfor funksjonen)
    var sum = tall1 + tall2;
    return sum;
}
}                         Kodeordet return      Verdi som returneres

```

Bruke (kalle opp) funksjonen

```

Variabel som settes lik det      Funksjonskall      Argumenter
funksjonen returnerer           ↓                         ↓
↓                         ↓                         ↓
var minSum = leggSammen(5, 3);   Funksjonskall      Argumenter
                                  ↓                         ↓
                                  Verdier vi overfører
                                  til funksjonen
alert(minSum); // 8

```

Lage en funksjon som ikke returnerer en verdi

```

function melding(navn) {
    alert("Hei på deg din " + navn + "!");
}

```

Bruke (kalle opp) funksjonen

```

melding("Padde"); // Hei på deg din Padde!

```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Arrayer

Lage en array

```
var minArray = [ ]; // Array definert uten verdier
```

Legge inn verdier (elementer) i en array

```
Indeks  
↓  
minArray[0] = "Brød";  
minArray[1] = "Melk";  
minArray[2] = "Juice";
```

Få tak i en verdi i en array

```
var verdi = minArray[1];  
alert(verdi); // Melk
```

Iterere gjennom en array

```
Antall elementer i arrayen  
↓  
for (var teller = 0; teller < minArray.length; teller++) {  
    // Gjør noe med hvert element, for eksempel skrive ut:  
    var verdi = minArray[teller];  
    alert(verdi);  
}
```

Lage en array og samtidig legge inn verdier

```
Indeks 0      Indeks 1      Indeks 2  
↓            ↓            ↓  
var minArray = ["Brød", "Melk", "Juice"];
```

Legge til en verdi (element) i slutten av en array

```
minArray.push("Peanøtter"); // Alternativ 1  
minArray[minArray.length] = "Peanøtter" // Alternativ 2
```

Slette elementer i en array

```
Start-indeks      Antall elementer vi vil slette (fra og med start-indeksten)  
↓            ↓  
minArray.splice(2, 1);
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Søke etter et element i en array

Får tak i indeksen til elementet vi finne. Hvis det ikke blir funnet, blir verdien -1

```
var indeks = minArray.indexOf("Melk");
```

Elementet vi vil finne

Lage en to-dimensjonal array

```
var minArray = [ ];
    ^ 0   1   2
minArray[0] = [5, 31, 7]; | 0
minArray[1] = [2, 9, 22]; | 1
minArray[2] = [50, 72, 4]; | 2
```

Få tak i en verdi i en to-dimensjonal array

Indeks i "ytterste" array Index i "innerste" array

```
var verdi = minArray[1][2];
alert(verdi); // 22
```

Assosiativer arrayer

Lage og sette verdier i en assosiativ array

```
var person = {navn: "Olsen", alder: 22, yrke: "snekker"};
```

Få tak i en verdi i en assosiativ array

```
var alder = person.alder; // Objekt-måten
```

```
var alder = person["alder"]; // Array-måten
```

Lage en array som inneholder assosiativer arrayer (to-dim.)

```
var personer = [];
personer[0] = {navn: "Olsen", alder: 22, yrke: "snekker"};
personer[1] = {navn: "Lia", alder: 43, yrke: "politi"};
personer[2] = {navn: "Hansen", alder: 34, yrke: "lærer"};

// Får tak i en verdi i arrayen
var yrke = personer[1]["yrke"];
alert(yrke); // Politi
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Iterere gjennom en assosiativ array

```
for (var noekkel in person) {
    alert("Nøkkel: " + noekkel + ",verdi: " + person[noekkel]);
}
```

Sjekke om en nøkkel finnes i en assosiativ array

```
if ("navn" in person) {
    // Kode
}
```

Hendelser

Registrere en lytterfunksjon

Element som kan utløse hendelse Navn på hendelse
 ↓ ↓
`document.getElementById("btnSjekk").onclick = sjekk;` Navn på funksjon som skal kalles opp hvis hendelsen utløses (lytterfunksjon)

Lage en lytterfunksjon

Hendelsesobjekt (inneholder info om hendelsen). Trenger ikke være med om det ikke benyttes
 ↓
`function sjekk(evt) {
 // Her skrives koden som skal kjøres når hendelsen
 // utløses
}`

Fjerne en lytterfunksjon

```
document.getElementById("btnSjekk").onclick = null;
```

Musehendelser

- `onclick` – brukeren klikker med musa.
- `ondblclick` – brukeren dobbeltklikker med musa.
- `onmousedown` – brukeren trykker museknapp ned.
- `onmouseup` – brukeren slipper museknapp opp.
- `onmousemove` – brukeren flytter musa over elementet. Denne hendelsen utføres hver gang musepekeren flytter seg til et nytt punkt, så potensielt et stort antall ganger på musepekerens vei over et objekt.
- `onmouseenter` – brukeren flytter musa fra et sted utenfor elementet til et sted over elementet.
- `onmouseleave` – brukeren flytter musa fra et sted over elementet til et sted utenfor elementet.
- `onmouseover` og `onmouseout` – fungerer som onmouseenter og onmouseleave, bortsett fra at de utføres på nytt for barn til elementer. Et onmouseover vil derfor bli utført både når musepekeren kommer inn over selve elementet og for hvert barn vi sveiper over og når vi går tilbake til hovedelementet igjen. Et onmouseout vil utføres når vi går fra hovedelementet og over på et barn til elementet, og flytter musa tilbake fra et barn.
- `onwheel` – fanger opp bevegelser til scrollhjulet på musa.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Informasjon i hendelsesobjektet til musehendelser

- **clientX og clientY** - koordinatene til musa i nettsidenens koordinatsystem. Øverst til venstre på nettsiden er 0,0.
- **screenX og screenY** - koordinatene til musa i skjermens koordinatsystem. Øvest til venstre på skjermen er 0,0.
- **button** - gir deg hvilken museknapp som ble trykket. 0 er venstre, 1 er midten/musehjul og 2 er høyre. Er musen konfigurert for venstrehendte i brukerens system, endres verdiene tilsvarende.
- **altKey, ctrlKey og shiftKey** - gir deg en true-/false-verdi på om angitte tast var nedtrykt på tastaturet mens musehendelsen ble utført.
- **detail** - gir deg en tallverdi på hvor mange ganger i en sammenhengende serie som museknappen ble trykket. Altså 2 for dobbeltklikk og 3 for trippelklikk.
- **deltaX** - den horisontale scrollingen på musehjulet.
- **deltaY** - den vertikale scrollingen på musehjulet.

Drag and drop

```
document.getElementById("draelement").ondragstart = dra;
document.getElementById("slippelement").ondrop = slipp;
document.getElementById("slippelement").ondragover = tillat;

function dra(evt) {
    evt.dataTransfer.setData("text", evt.target.id);
}

function tillat(evt) {
    evt.preventDefault();
}

function slipp(evt) {
    evt.preventDefault();
    var elementId = evt.dataTransfer.getData("text");
    // Kode
}
```

Tastaturhendelser

- **onkeydown** - når en tast trykkes ned (også systemtaster).
- **onkeyup** - når en tast slippes opp (også systemtaster).
- **onkeypress** - når en tast trykkes ned. Gjentas regelmessig så lenge tasten holdes nedtrykt (ikke systemtaster).

Informasjon i hendelsesobjektet til musehendelser

- **keyCode** - gir deg et nummer som representerer tasten på tastaturet som ble trykket.
charCode - gir deg et nummer som representerer tegnet som blir trykket på tastaturet.
Dette er Unicode-verdien til tegnet.
- **altKey, ctrlKey og shiftKey** - gir deg en true-/false-verdi på om angitte tast var nedtrykt på tastaturet sammen med den andre tasten som hendelsen indikerer.

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Starte en tidsstyrтt hendelse

```

Global variabel           Funksjon som skal utføres   Antall millisekunder mellom hver utførelse
↓                         ↓                           ↓
intervallID = setInterval(tikk, 200);

function tikk() {
    // Kode
}

```

Stoppe en tidsstyrтt hendelse

```

Global variabel med referanse til intervallet som er startet
↓
clearInterval(intervallID);

```

Grafikk

Lage canvas

```
<canvas id="tegneflate" width="200" height="200"></canvas>
```

Hente ut context

```
var ctx = document.getElementById("tegneflate").getContext("2d");
```

Tegne en linje

```

ctx.beginPath();
ctx.moveTo(50, 100); // x-verdi og y-verdi
ctx.lineTo(100, 200); // x-verdi og y-verdi
ctx.stroke();

```

HUSK: x-verdi måles fra venstre kant, y-verdi måles fra toppen.

Tegne en sirkel

```

ctx.beginPath();
ctx.arc(50, 50, 25, 0, 2 * Math.PI); // x-verdi, y-verdi, diameter,
                                         // sirkelstart og sirkelslutt
ctx.stroke();

```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Tegne et rektangel

```
ctx.beginPath();
ctx.rect(50, 100, 100, 100); // x-verdi, y-verdi, bredde og høyde
ctx.stroke();
```

Endre linjefarge og tykkelse

```
ctx.strokeStyle = "red"; // En CSS-farge
ctx.lineWidth = 4;
```

NB! Gir kun effekt til det vi tegner etter endringen

Tegne fyll

```
ctx.fillStyle = "red";
ctx.fill();
```

NB! Må gjøres før vi evt. tegner kantlinje med stroke()

Endre HTML-dokumentet

Lage et nytt element og legge det til i dokumentet

```
var nyttElement = document.createElement("p");
nyttElement.innerHTML = "Litt mer tekst...";
Foreldrelement
document.getElementById("tekst").appendChild(nyttElement);
Barn
```

Fjerne et element fra dokumentet

```
Foreldrelement
document.getElementById("tekst").removeChild(nyttElement);
Barn
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Endre CSS

Sette nye stiler direkte

```
document.getElementById("tekst").style.color = "red";
document.getElementById("tekst").style.display = "none";
document.getElementById("tekst").style.borderStyle = "solid";
```

Endre CSS-klasse

```
document.getElementById("ingress").className = "roedtekst";
```

Fjerne CSS-klasse

```
document.getElementById("ingress").className = "";
```

Lyd og video

Spille av lydfil

```
var lyd = new Audio("lydfil.mp3");
lyd.play();
```

Sette inn lydavspiller

```
<audio controls="controls" id="lydavspiller">
  <source src="lydfil.mp3" type="audio/mpeg">
  <source src="lydfil.ogg" type="audio/ogg">
  Nettleseren støtter ikke lydavspilling
</audio>
```

Sette inn videoavspiller

```
<video width="640" height="480" controls="controls"
id="videoavspiller">
  <source src="videofil.mp4" type="video/mp4">
  <source src="videofil.ogg" type="video/ogg">
  Nettleseren støtter ikke videospilling
</video>
```

Manipulere en avspiller

```
document.getElementById("lydavspiller").play();
document.getElementById("lydavspiller").pause();
document.getElementById("lydavspiller").currentTime = 0;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Detektere avspillingsslutt

```
document.getElementById("lydavspiller").onended = slutt;

function slutt() {
    // Kode
}
```

Dynamisk innhold

Husk at du må kjøre prosjektet gjennom en webserver.

Laste inn data fra fil

```
var xmlhttp;

function oppstart() {
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = statusforandring;
    xmlhttp.open("GET", "innhold.dat", true);
    xmlhttp.send();
}

function statusforandring() {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
        var data = xmlhttp.responseText;
    }
}
```

Dele opp strukturert data

```
var data = xmlhttp.responseText;
var linjer = data.split("\n");

for (var i = 0; i < linjer.length; i++) {
    var datafelt = linjer[i].split("%");

    // Gjør noe med verdiene i arrayen datafelt
}
```

Hente ut XML-data

```
var xmlData = xmlhttp.responseXML;
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Hente ut verdier og attributtverdier fra XML-data

```
var verdi = xmlData.getElementsByTagName("pris")[0].childNodes[0].nodeValue;
var verdi2 =
    xmlData.getElementsByTagName("bil")[0].attributes.getNamedItem("regnr").value;
```

Hente ut JSON-data

```
var jsonData = JSON.parse(xmlhttp.responseText);
```

Hente ut verdier fra JSON-data

```
var verdi = jsonData.datafelt;
var verdi2 = jsonData.datafelt2.dataliste[0].datafelt
```

Skrive til web storage

```
localStorage.setItem("navn", "Ole Olsen");
```

Lese fra web storage

```
var navn = localStorage.getItem("navn");
if (navn === null) {
    navn = "ukjent";
}
```

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Matte

Mattefunksjoner

```
var tall11 = Math.round(20.749); // (21) - avrunding
var tall12 = Math.floor(20.749); // (20) - avkorting nedover
var tall13 = Math.ceil(20.749); // (21) - avkorting oppover
var tall14 = Math.max(3.8, 7.5); // (7.5) - maksimumsverdi
var tall15 = Math.min(3.8, 7.5); // (3.8) - minimumsverdi
var tall16 = Math.abs(-50); // (50) - absolutt-verdi
var tall17 = Math.pow(2, 3); // (8) - 2 opphøyd i 3
var tall18 = Math.sqrt(9); // (3) - kvadratroten av 9
var tall19 = Math.PI; // (3.14...)
```

Vinkelberegning

```
var tall11 = Math.cos(vinkel i radianer); //cosinus
var tall12 = Math.sin(vinkel i radianer); //sinus
var tall13 = Math.atan2(dy, dx); //motstående vinkel i radianer
```

Konvertere mellom grader og radianer

```
var grader = radianerVerdi * 180 / Math.PI;
var radianer = graderVerdi * Math.PI / 180;
```

Generere tilfeldige tall

```
var tall11 = Math.random(); // (0 <= tall < 1) - desimaltall
var tall12 = Math.floor(Math.random() * 11) + 5; // Heltall mellom
                                                // 5 og 15
```

↓ ↓

Intervallbredde Forskyvning

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Symboler

!= 96
 && 98
 % 73
 == 96
 || 99
 <audio> 201
 <canvas> 55
 \n 64, 223
 <option> 50
 <select> 50
 \t 223

A

absolute 206
 abstraksjon 254
 addEventListener 184
 AJAX 222
 altKey 186, 189
 AND 98
 append 239
 arc 57
 argument 143
 array 118
 arv 255
 assosiativ array 132
 Asynkron JavaScript og XML 222
 attributes 228
 audio 200, 202
 avkrysningsboks 41, 48

B

beginPath 56
 betingelse 95
 boolean 66
 Brackets 14
 break 112
 breakpoints 174
 btn 41
 button 41, 186

C

camelCase 19, 62
 canplay 203
 canvas 55
 case-sensitiv 19, 63
 catch 179
 charAt 158
 charCode 189
 charCodeAt 158
 checkbox 48
 checked 48
 chk 41
 chrome 14
 className 38
 clearInterval 192
 clearRect 94, 205
 clearTimeout 192
 client-side 30
 client-side-script 236
 clientX 186
 clientY 186
 code completion 14
 concat 126
 console.log 172
 const 81
 context 55
 continue 113
 cookies 233
 ctrlKey 186, 189
 currentTarget 184
 currentTime 203

D

database 245
 datamedlemmer 253
 date 43
 datetime-local 43
 debugger 173
 definere 140
 deklarere 61
 dekrementering 72
 delprogrammer 140
 deltaX 186
 deltaY 186
 design patterns 255
 desimalskillett 66
 desimaltall 66
 detail 186
 document 188
 drag and drop 186
 draggable 187
 dreamweaver 14
 duration 203
 dynamisk innhold 220

E

egenskaper 253
 eksternt innhold 220
 ekstern webserver 221
 else 87
 else if 88
 escaped 64

F

false 66, 85
 fargekoding 21
 feil 24
 feilmelding 26
 feilrettingsverktøy 173
 feilsøking 172
 feltvariabler 253
 fill 57
 fillStyle 57
 float 66
 flyttall 66
 for 122
 for.in 133
 for-løkker 104
 fromCharCode 158
 funksjon 139
 funksjonsdefinisjon 140
 funksjonskall 140
 funksjonsreferanse 181

G

GET 222
 getBoundingClientRect 211, 212
 getElementsByTagName 226
 getItem 234
 getNamedItem 228
 global variabel 68, 149
 grensesnitt 254
 grensesnittfeil 168

H

hardkoding 62
 heltall 66
 heltallsdivisjon 73
 hendelse 32
 hendelsesobjekt 181, 182
 hendelsesorientert 180
 hendelsesorientert programmering 13
 hendelsesparameter 34, 181

I

i 122
 if..in 133
 if-test 85
 indeks 119
 indentering 169
 indexOf 126, 158
 informasjonskapsel 233
 initialisere 61
 inkrementering 72
 innerHTML 35
 innkapsling 254
 input-type 43
 instanceof 259
 instruksjon 10
 integer 66
 iterasjon 102, 106
 iterere 133

J

Java 30
 JavaScript Object Notation 232
 join 126
 JSLint 18
 JSON 232
 JSON.parse 232

K

kalle opp 140
 keyCode 189
 kjorefasfeil 167, 179
 klasse 253
 klassevariabel 253
 knapp 41
 kodeblokk 85
 kodekonvensjon 62
 kommentarer 22
 kommentering 22
 kompleks datatype 82
 konkatenerere 77
 konkateneringsoperatoren 77
 konstanter 81
 konstruktør 254, 256
 kontrollflyt 102
 kontrollstruktur 83
 konvertering 67, 80
 krøllparentes 85

L

lastIndexOf 158
 left 204
 length 121
 likhetsoperator 70
 lineTo 56
 lineWidth 56
 liste 118
 logiske feil 24, 166, 167
 logisk operator 98
 logisk uttrykk 85, 95
 logisk verdi 66
 lokal testserver 221
 lokal variabel 68, 149
 lokal webserver 221
 Ist 41
 lytter 181, 182
 lytterfunksjon 181
 løkker 101

M

MAMPServer 221
 matematisk operator 71
 Math 155
 Math.abs 156
 Math.atan2 156
 Math.ceil 156

VURDERINGSEKSEMPLAR - KOPIERING FORBUDT!

Math.cos 156
 Math.floor 73, 156, 157
 Math.max 156
 Math.min 156
 Math.pow 156
 Math.random 92, 155, 157
 Math.round 156
 Math.sin 156
 Math.sqrt 156
 max 44
 maxlen 44
 meldingsboks 31
 metoder 140, 155, 253
 min 44
 modularitet 254
 modulo-operatoren 73
 moveTo 56
 musehendelser 185
 mute 203
 MySQL 245

N

name 52
 nedtrekksliste 41, 50
 negeringsoperatoren 99
 nestede kontrollstrukturer 109
 nestede løkker 109
 nestet notasjon 233
 noder 225
 Notepad++ 14
 null 64
 number 43, 66
 nøkkel 132
 nøkkelord 62

O

objekt 253, 256
 objektorientert programkode 252
 objektvariabler 82
 offsetHeight 211
 offsetWidth 211
 onchange 46
 onclick 185
 ondblclick 185
 ondragover 187
 ondragstart 187
 ondrop 187
 onended 203
 onkeydown 46, 188
 onkeypress 188
 onkeyup 188
 onmousedown 185
 onmouseenter 185
 onmouseleave 185
 onmousemove 185
 onmouseout 185
 onmouseover 185
 onmouseup 185
 onpause 203
 onplay 203
 onreadystatechange 222
 onseeked 203
 onselectstart 195
 onvolumechange 203
 onwheel 185
 open 222
 operator 70
 OR 99

P

parameter 143
 parseFloat 80
 parseInt 80

password 43
 pause 203
 PHP 236
 PI 156
 placeholder 45
 play 203
 pop 126
 position 206
 POST 222
 prefiks 41
 presedens 73
 preventDefault 187, 188
 primitiv datatype 64, 82, 256
 programflyt 148
 programmeringslogikk 11
 programmeringsparadigme 252
 programmeringsspråk 12
 prototyping 259
 proxy 241
 pseudokode 11
 push 126

R

radianer 57, 209
 radioknapper 41, 52
 range 43
 rbtn 41
 readyState 222
 rect 57
 referanse 82
 rekursiv funksjon 150
 relasjonsoperatører 96
 removeEventListener 184
 repetisjonshastigheten 188
 replace 158
 responseText 223
 responseXML 226
 rest 73
 resume script execution 175
 return 146
 reverse 126

S

sann 66
 screenX 186
 screenY 186
 sekvensielt 83
 semantiske feil 166
 send 222
 sender 182
 sender-lytter-modell 181
 server-side 30
 server-side-script 236
 setInterval 94, 191, 204
 setTimeout 191
 shift 126
 shiftKey 186, 189
 sikkerhetskritisk feil 168
 size 51
 skilltegn 223
 skjemaelement 41
 skjema 41
 slice 158
 sort 126
 sort boks 254
 splice 126
 split 158, 224
 sprett 210
 stack 150
 stack overflow 150
 status 222
 step 44
 step into next function call 175

step out of current function 175
 step over next function call 175
 stoppunkter 174
 string 64
 stroke 56
 strokeStyle 56
 style 38
 Sublime Text 14
 substr 158
 svart boks 153
 synlighet 47
 syntaktiske feil 24
 syntax highlighting 14, 21

T

tabell 118
 target 184
 tekstsheets 41, 42
 tekststrenge 64, 134
 teller 106, 122
 ternary conditional operator 90
 teste nettsiden 32
 text 42
 this 257
 tidsstyrte hendelser 191
 tilordningsoperatoren 70
 time 43
 todimensjonal array 128
 toFixed 76, 80
 toLowerCase 158
 top 204
 toString 80
 toUpperCase 158
 transform 206
 true 66, 85
 try 179
 try-catch 179
 txt 41

U

uemelige løkker 103
 unære operatører 71
 undefined 64, 120
 unshift 126
 usann 66
 utviklingsmiljø 14

V

valgoperatoren 90
 valgsetninger 83, 85
 value 42, 45, 48, 228
 var 81
 variabel 60
 variabelnavn 62
 Virkningsområde 68
 volume 203

W

WAMPServer 221
 web storage 233
 WebStorm 14
 while-løkker 102
 window 191
 window.onload 30
 write 239

X

x 55
 XML 225
 XMLHttpRequest 222

Y

y 55



