☐ Tell sekunder

LAST NED PDF

Introduksjon

I denne oppgaven skal vi lage vårt eget spill!

Vi lært mye allerede! Her er en oppsummering:

Verktøy til Elm-utvikling:

- Bruke Try Elm til å lage programmer
- Bruke Elm Repl til å lage funksjoner
- Bruke Elm Repl til å teste funksjoner
- Bruke Elm Reactor til å lage programmer

Programmeringsspråket Elm:

- Vite hva en funksjon er
- Vite hva en modul er

Teknologi for å vise innhold til en nettleser:

- HTML
- SVG

Snakke med omverdenen:

- Se hva brukeren skriver i input-felter

Det er slettes ikke verst! Det er lov å være stolt!

For å kunne lage skikkelige spill, mangler vi to byggesteiner:

- Hvordan simulerer vi *tid*?
- Hvordan ser jeg hva brukeren trykker på av taster mens tiden går?

Det er hva vi skal gå gjennom nå!

Steg 0: Hva er forskjellig fra før?

For å innføre tid ber vi Elm om å si ifra hvert sekund. Vi abonnerer på når det har gått tid.

Norsk	Engelsk
Abonnere	Subscribe
Abonnement	Subscription

Eksempel på strukturen på et tidligere program:

```
main =
  beginnerProgram
  { model = ""
  , view = view
  , update = update
  }
```

Tidligere har nettsiden kun endret seg hvis brukeren har gjort noe. Da trenger vi bare å si noe om:

1. hvilken data vi skal lagre (`model`),
2. hvordan dataen skal vises (`view`), og
3. hvordan brukerens handlinger skal oppdatere tilstanden i programmet vårt (`update`).

Vi var altså kun avhengig av hva brukeren gjorde! Hvis brukeren ikke gjorde noe, skjedde det ingenting. Dette stemmer ikke lenger!

Slik vil det nye programmet vårt være bygget opp:

```
main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

Her er det tre forskjeller:

1. Vi bruker `Html.program` i stedet for `Html.beginnerProgram`, som lar oss lage mer komplekse programmer
2. Vi bruker `init` i stedet for `model` til å beskrive hvordan ting skal være når vi starter opp
3. Vi definerer abonnementer, `subscriptions`!

Hva kan vi abonnere på?

Vi kan abonnere på mange forskjellige ting:

1. Når det har gått X sekunder (`Time`)
2. Når brukeren trykker på **Pil opp** (`Keyboard`)
3. Når noen andre sender oss en beskjed (`WebSocket`)

Steg 1: En sekundteller

Her er en enkel sekundteller:

```
import Html exposing (Html, text)
import Time exposing (Time, second)


main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }


-- MODEL

type alias Model =
  { count : Int
  }

start =
  { count = 0
  }

init : (Model, Cmd Msg)
init =
  (start, Cmd.none)


-- UPDATE

type Msg
  = Tick Time


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Tick newTime ->
      (bump model, Cmd.none)


bump model =
  { model | count = model.count + 1 }


-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every second Tick


-- VIEW

view : Model -> Html Msg
view model =
  text (toString model.count)
```

- Kopier inn koden i **Try Elm** eller kjør den lokalt med `elm reactor`.

Ser noe du ikke har sett før? Nå skal vi gå gjennom bit for bit.

Hvor langt har vi talt?

Hva skjedde med modellen?

```
-- MODEL

type alias Model =
  { count : Int
  }

start =
  { count = 0
  }

init : (Model, Cmd Msg)
init =
  (start, Cmd.none)
```

Den eneste informasjonen vi er interessert i, er hvor langt vi har talt. Derfor består modellen vår kun av `count`.

Når vi starter programmet, starter tellingen på `0`. `zero` sier at vi har talt til `0`.

Hva er så disse `Cmd`-greiene? `Cmd` lar oss sende informasjon til resten av verden. Det bruker vi ikke ennå! Derfor er `Cmd` litt i veien for oss, men når vi er over på `Html.program`, må vi ha den med. Hvis du ser rundt i koden finner du `Cmd.none`, som betyr "gjør ingenting".

- [ ] Endre `count` i funksjonen `start`. Hva gjør programmet nå?
- [ ] Hva skjer om `count` blir veldig stor?
- [ ] Kan `count` være mindre enn null?

Oppdatere telleren

```
-- UPDATE

type Msg
  = Tick Time


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Tick newTime ->
      (bump model, Cmd.none)


bump model =
  { model | count = model.count + 1 }
```

Beskjeden vår (`Msg`) kan nå bare være én ting: **det har gått tid siden sist**.

Hvert sekund, kjører vi modellen vår gjennom funksjonen `bump`. Den "dytter" ("bump" på engelsk) modellen vår ett sekund fram i tid.

- [ ] Kan du få timeren vår til å telle baklengs?
- [ ] Kan du få timeren vår til å telle ned fra 10 til 0? Obs! Her må du også endre `start`!

- [ ] Kan du få timeren vår til å telle fortere?
- [ ] Utfordring: Timeren får ikke telle til høyere enn 10. Når den kommer til 10, skal den begynne fra start igjen! Tips: bruk et if-uttrykk.

Gå fort, gå sakte

```
-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every second Tick
```

Her abonnerer vi på sekunder. Verdien `second` kommer fra modulen `Time`.

- [ ] Finn `second` i modulen `Time`.
- [ ] Hva skjer om du abonnerer på timer i stedet for sekunder?
- [ ] Hva skjer om du abonnerer på millisekunder?

Øverst i `Time-modulen` finner vi:

```
type alias Time =
    Float
```

Dette betyr at tid bare er et tall. Disse kan vi plusse og dele!

- [ ] Klarer du å abonnere på annenhvert sekund? Hva må du da sende inn i stedet for `second`?
- [ ] Klarer du å abonnere på tidels sekunder?

Hint: hva er `second * 2`?

La det blinke!

```
-- VIEW

view : Model -> Html Msg
view model =
  text (toString model.count)
```

Hmm, den så litt kjedelig ut. La oss lage litt blinking i stedet!

Obs! Har du epilepsi? Hopp over denne oppgaven!

Først skal vi gjøre bakgrunnen rød:

```
view : Model -> Html Msg
view model =
  div
    [ style
        [ ("backgroundColor", "red")
        ]
    ]
    [ text (toString model.count)
    ]
```

Husk å importere de nye funksjonene!

- [ ] I hvilke moduler finner vi `div` og `style`?

Men nå ble det jo alltid rødt! Dårlig blinking. En måte å få til blinking er å sette oddetall til rød og partall til grønn.

Noen partall: 0, 2, 4, 6, 8

Noen oddetall: 1, 3, 5, 7, 9

Ser du at annenhvert tall er partall, og annenhvert oddetall? Da kan vi få til partall!

For å sjekke om et tall er partall kan du se hva du får til rest når du deler på

2. Funksjonen rem gir resten ved divisjon. rem 5 4 gir 1, fordi tallet 5 gir en hel firer, og **1** ekstra.

Her prøver jeg i `elm repl`:

```
> rem 0 2
0 : Int
> rem 1 2
1 : Int
> rem 2 2
0 : Int
> rem 3 2
1 : Int
> rem 4 2
0 : Int
> rem 5 2
1 : Int
> rem 6 2
0 : Int
> rem 7 2
1 : Int
> rem 8 2
0 : Int
> rem 9 2
1 : Int
```

- [ ] Ser du mønsteret? Hva skjer når vi deler oddetall på 2? Hva skjer når vi deler partall på 2?
- [ ] Skriv funksjonen isEven. Denne skal fungere slik:

```
> isEven 0
True : Bool
> isEven 1
False : Bool
> isEven 2
True : Bool
> isEven 3
False : Bool
> isEven 4
True : Bool
> isEven 5
False : Bool
> isEven 6
True : Bool
> isEven 7
False : Bool
> isEven 8
True : Bool
> isEven 9
False : Bool
```

- [ ] Bruk funksjonen `isEven` til å bestemme om du skal ha `("backgroundColor", "red")` eller `("backgroundColor", "green")`!

... Stopp her for å prøve selv!

Her lager jeg med vilje litt avstand ...

Ikke scroll ned om du vil prøve selv ...

... men hvis du vil se et forslag, så kommer det nå!

`isEven` sammenlikner resten vi får når vi deler på 2 med 0:

```
isEven x =
  rem x 2 == 0
```

`getColor` bruker `isEven` til å si om en modell skal vises som rød eller grønn:

```
getColor model
  = if isEven model.count
    then "red"
    else "green"
```

... før vi plugger dette inn i viewet vårt:

```
view : Model -> Html Msg
view model =
  div
    [ style
        [ ("backgroundColor", getColor model)
        , ("height", "100%")
        ]
    ]
    [ text (toString model.count)
    ]
```

- [ ] Kan du få det til å blinke *og* gå fort? Obs! Mange farger!