IN1010 Game of Life – 2023

Introduksjon

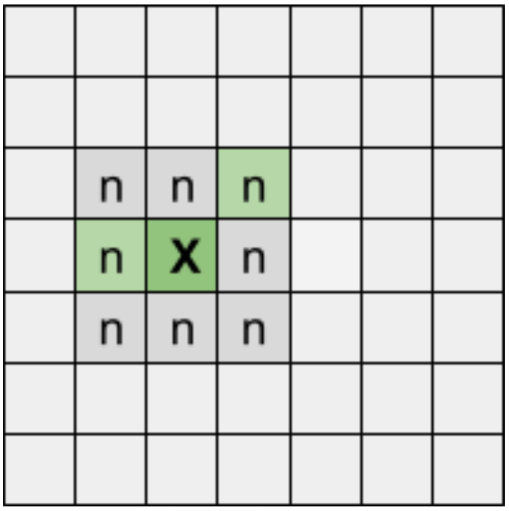
I denne innleveringen skal du lage et program som simulerer cellers liv og død. Dette skal du gjøre ved hjelp av en modell kalt *Conway's Game of Life.* (Les mer om *Game of Life* her.) Dette var obligatorisk oppgave nr 8 i IN1000 sist høst, men nå skal den programmeres på nytt i Java.

Kort fortalt skal programmet holde styr på et rutenett av en gitt størrelse der hvert felt på brettet inneholder en celle. En celle kan være levende eller død. Simuleringen utspiller seg gjennom flere generasjoner ved hjelp av jevnlige oppdateringer der celler dør eller lever avhengig av sine omgivelser.

Programmet skal la brukeren observere simuleringen generasjon for generasjon ved å tegne opp spillebrettet i terminalen sammen med tilleggsinformasjon om hvilken generasjon vi ser på samt hvor mange celler som for øyeblikket lever.

Spillets regler

En ny generasjon skapes ved at alle cellene på brettet endrer status avhengig av sine naboceller. Som naboceller regnes alle berørte celler, både levende og døde.



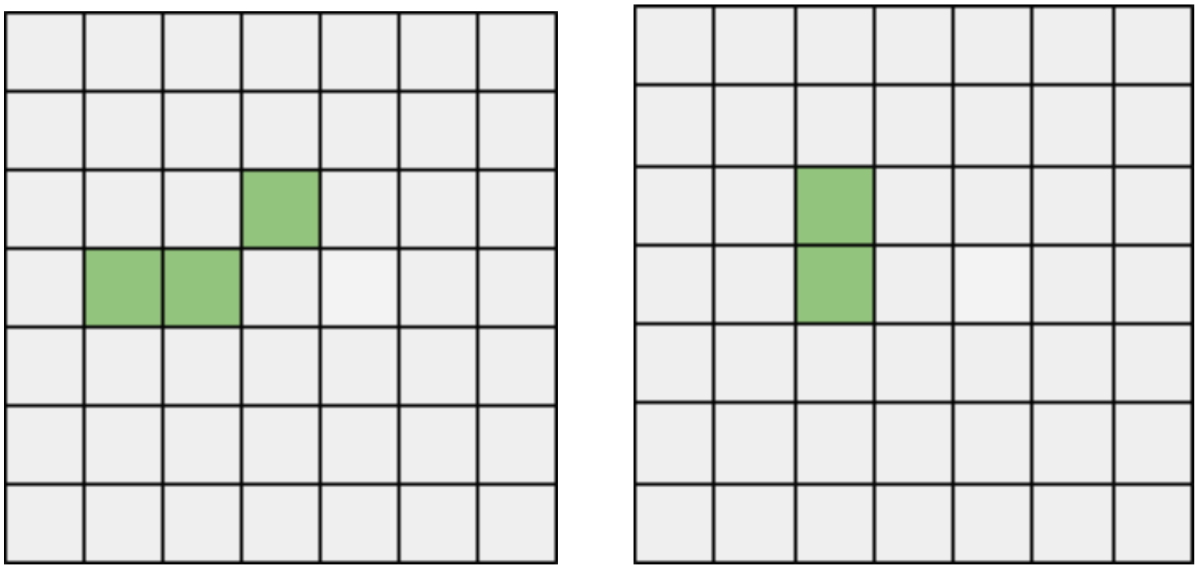
Illustrasjon 1: En celle (X) og dens naboceller (n, grønne celler er levende). X har altså 8 naboceller og 2 av dem er levende.

En celles nye status bestemmes av følgende regler:

- Dersom cellens nåværende status er *levende*:
  - Ved færre enn to levende naboceller dør cellen (*underpopulasjon*).
  - Ved to eller tre levende naboceller vil cellen leve videre.
  - Hvis cellen har mer enn tre levende naboceller, vil den dø (*overpopulasjon*).

- Dersom cellen er *død*:
  - Cellens status blir levende (*reproduksjon*) dersom den har nøyaktig tre levende naboer.
  - Ellers forblir den død.

Merk at oppdateringen av alle cellenes status skjer *samtidig!* Det betyr at vi må bestemme ny status på alle celler avhengig av nåværende status *før* oppdateringen faktisk skjer. (Dette håndteres med variabelen `antLevendeNaboer` i `Rutenett` som vi kommer til senere.)

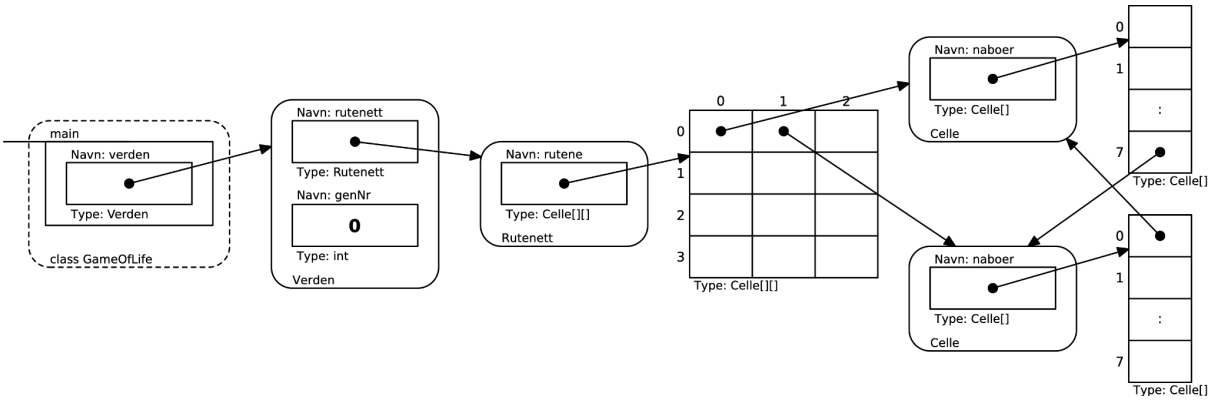


Illustrasjon 2: To generasjoner av celler i et 7x7-størrelse spillebrett. To celle døde, én overlevde og én død celle er blitt levende.

Struktur

Programmet består fire klasser som alle skal leveres i hver sin kodefil. Det følger også med testfiler for klassene `Celle`, `Rutenett`, og `Verden`. Her kan man kalle på ulike testmetoder som skriver «Alt riktig» eller en beskjed om hvilken av metodene som ikke gjør det den skal. Når man har gjort en deloppgave, så trenger man kun å fjerne kommentartegnet for den aktuelle deloppgavetesten, så kan man kompilere, kjøre og se at alt virker. Man skal ikke levere testfilene.

Nedenfor kan du se hvordan datastrukturen kan se ut under kjøring. Klassen `GameOfLife` som inneholder «hovedprogrammet» (`main`-metoden), skal referere en instans av klassen `Verden`, som igjen refererer til en instans av klassen `Rutenett`. I denne instansen er det en to-dimensjonal array med referanser til instanser av klassen `Celle`, hvor hver celle kommer til å ha en array med referanser til sine naboer (som er andre instanser av klassen `Celle`).



Illustrasjon 3. Datastrukturen under kjøring

Celle

Filnavn: *Celle.java*

Klassen beskriver en celle under simuleringen. En celle skal ha en variabel som beskriver status (levende/død).

NB! Statusvariabelen skal være av typen **`boolean`**.

Filen *TestCelle.java* inneholder ferdige metoder som kan brukes til å teste metodene.

1. Skriv en **konstruktør** for klassen som oppretter cellen med status *død* som utgangspunkt. Skriv også instansvariablene `naboer` som settes lik en array med åtte elementer, `antNaboer` som angir nåværende antall naboer og som initieres til 0 og til sist `antLevendeNaboer` som også initieres til 0.

   Merk at en celle bare vet *hvilke* andre celler som er naboer, ikke hvor de ligger. Den vet heller ikke hvor den selv ligger i rutenettet. Det er `Rutenett`-objektet som kjenner plasseringen av hver celle ved å se på hvilken rad og hvilken kolonne i den todimensjonale arrayen som refererer til cellen.

   Testmetode: **`testCelle`**

2. Skriv metodene **`settDoed`** og **`settLevende`** som ikke tar noen parametere, men som setter statusen til cellen til henholdsvis *død* og *levende*.

   Testmetode: **`testSettDoedLevende`**

3. Skriv metoden **`erLevende`** som returnerer cellens status: **`true`** hvis cellen er levende og **`false`** ellers.

   Testmetode: **`testErLevende`**

4. Skriv metoden **`hentStatusTegn`** som returnerer en **`char`**-verdi som er en tegnrepresentasjon av cellens status til bruk i tegning av brettet. Dersom cellen er levende, skal det returneres en **`'O'`**, mens hvis den er død, returneres et punktum.

   Testmetode: **`testHentStatusTegn`**

5. Skriv metoden **`leggTilNabo`** som har en instans av klassen `Celle` som parameter (nabo), og legger nabo til i arrayen `naboer` og øker `antNaboer`.

   Testmetode: **`testLeggTilNabo`**

6. I tillegg trenger vi følgende to metoder når vi skal oppdatere statusen for en celle:

   - **`tellLevendeNaboer`** som går gjennom `naboer` og teller antall levende naboer. Svaret skal legges i instansvariabelen `antLevendeNaboer`.

   Testmetode: **`testTellLevendeNaboer`**

   - **`oppdaterStatus`** som ut fra spillets regler (se tidligere avsnitt) endrer statusen (dvs levende eller død) til en celle basert på antall levende naboer.

   Testmetode: **`testOppdaterStatus`**

Rutenett

Filnavn: *Rutenett.java*

Denne klassen beskriver et todimensjonalt brett som inneholder celler. Rutenett skal holde styr på hvilke celler som skal endre status og oppdatere disse for hver generasjon.

Filen *TestRutenett.java* inneholder ferdige metoder som kan brukes til å teste klassen.

1. Skriv **konstruktøren** for klassen Rutenett. Konstruktøren tar imot dimensjonene på rutenettet og lagrer disse i instansvariablene `antRader` og `AntKolonner`.

   Test-metode: **`testKonstruktoerUtenRutenett`**

2. I konstruktøren ønsker vi også å lage en ny instansvariabel `rutene` i form av en todimensjonal array. (Elementene i arrayen skal siden fylles med referanser til instanser av klassen `Celle`.)

   Husk: En todimensjonal array av `Celle`-r deklareres slik:

   ```
   Celle[][] rutene;
   ```

   og opprettes slik når det er `r` rader og `k` kolonner:

   ```
   rutene = new Celle[r][k];
   ```

3. Skriv metoden **`lagCelle`** som oppretter en instans av klassen `Celle` og legger den inn på en plass i den todimensjonale array-en `rutene` ut fra de to parametrene `rad` og `kol`. Hver celle i rutenettet skal ha ⅓ sjanse for å være levende når den legges inn i rutenettet. Her kan man bruke metoden `settLevende` fra klassen `Celle`.

   Hint: Testen `Math.random()<=0.3333` vil gi **`true`** hver tredje gang i snitt.

4. Metoden **`fyllMedTilfeldigeCeller`** går gjennom hele rutenettet og sørger for at det blir fylt med celler. Bruk metoden `lagCelle` til å opprette cellene. Dette rutenettet utgjør utgangspunktet, eller «0-te generasjon» for cellesimuleringen vår.

   Test-metode: **`testFyllMedTilfeldigeCeller`**

5. Skriv metoden **`hentCelle`** som tar imot en celles koordinater (rad og kolonne) i rutenettet som parametre, og returnerer cellen med den gitte posisjonen. Hvis en ulovlig rad- eller kolonneindeks er gitt (for lav eller for høy indeks), så skal metoden returnere **`null`**.

   Test-metode: **`testHentCelle`**

6. For å vise frem rutenettet skal du skrive metoden **`tegnRutenett`**. Denne metoden skal bruke en dobbelt for-løkke for å skrive ut hvert element i rutenettet. Tips til formatering av utskrift:

   ● Du kan se et eksempel på et rutenett helt til slutt i denne oppgaveteksten, men din løsning behøver ikke se akkurat slik ut. Du kan tegne rutenettet akkurat slik du vil.

   ● For å unngå linjeskift etter hver utskrift kan du bruke `System.out.print(...)`.

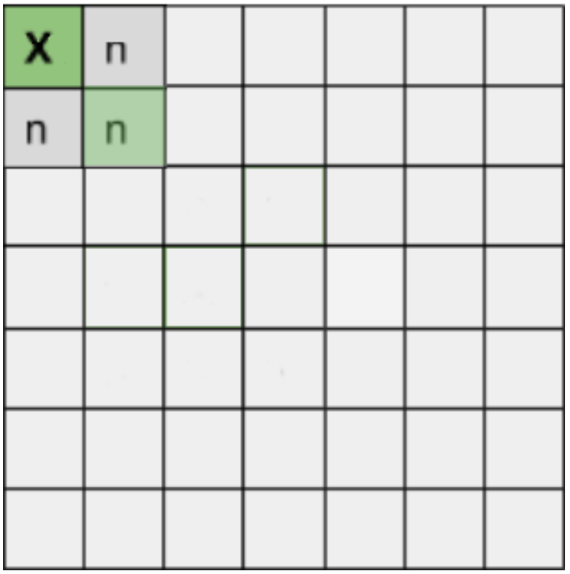
   ● Det kan være lurt å «tømme» terminalvinduet mellom hver utskrift. Dette kan du for eksempel gjøre ved å skrive ut et titalls blanke linjer før du skriver ut brettet.

   Test-metode: **`testTegnRutenett`**

7. For å bestemme hvilke celler som skal være *levende* og *døde* i neste generasjon, trenger vi å vite statusen til hver celles nabo. Rutenett-klassen skal derfor inneholde

en metode **`settNaboer`**. Metoden skal ta imot en celles koordinater (rad og kolonne) som parametre og sette referanser til alle instanser av klassen *Celle* som er nabo for den gitte Celle-instansen. Hver celle kan ha maksimalt 8 naboer, men celler langs kanten eller i hjørnene har færre.

Hint: Husk at du kan bruke `hentCelle` til å hente celler uten å få feilmeldinger for ulovlige indekser. Det vil si, om du for eksempel kaller `hentCelle` og sender -1 som argument for rad eller noe slikt, skjer ikke noe verre enn at du får **`null`** tilbake (og kan enkelt sjekke for det). Se illustrasjon 1 og 4 som viser to ulike celler med naboer.



Illustrasjon 4: Cellen X har tre naboer, to døde og én levende.

Pass på å sjekke at alle naboene til en Celle er riktige. Det kan være lurt å printe ut (rad,kol) for alle naboene til en gitt Celle og sjekke at indeksene er riktige. På illustrasjonen over har celle X med koordinater (0,0) tre naboer med plass (0,1), (1,0) og (1,1). Pass på kanter og hjørner, og at man ikke legger til den gitte cellen som nabo til seg selv.

Test-metode: **`testSettNaboer`**

8. Skriv en metode **`kobleAlleCeller`** som ved hjelp av en dobbel for-løkke skal kalle på `settNaboer` for hver plass (rad, kol) i rutenettet.

   Test-metode: **`testKobleAlleCeller`**

9. I tillegg trenger vi en metode **`antallLevende`** som skal returnere antall levende celler i rutenettet. Dette kan du enklest gjøre ved å gå gjennom rutenettet og øke en teller for hver levende celle du finner.

   Test-metode: **`testAntallLevende`**

Verden

Filnavn: *Verden.java*

I denne klassen skal vi oppdatere rutenettet og skrive ut hver generasjon av simuleringen.

1. Skriv **konstruktøren** for klassen `Verden`. Den skal ta inn antall rader og kolonner som parametere, og opprette en instans av klassen `Rutenett` som lagres i instansvariabelen `rutenett`. Klassen skal holde styr på antall generasjoner og trenger derfor å ha en instansvariabel `genNr` som skal initieres til 0. I tillegg så skal man fylle rutenettet med tilfeldige celler og koble cellene sammen.

   Tips: Bruk metoder fra `Rutenett`-klassen.

   Test-metode: **`testKonstruktoer`**

2. Skriv metoden **`tegn`** som skal tegne rutenettet i tillegg til å skrive ut generasjonsnummeret og antall levende celler.

   Test-metode: **`testTegn`**

3. Skriv metoden **`oppdatering`** som skal gjøre tre ting:

   - Gå gjennom alle celler i rutenettet og telle og oppdatere antallet levende naboer for hver celle.
   - Gå gjennom alle celler i rutenettet på nytt og oppdatere status (dvs om den er levende eller død) på hver celle.
   - Husk å oppdatere telleren for antall generasjoner.

   Test-metode: **`testOppdatering`**

Hovedprogrammet **`GameOfLife`**

Filnavn: *GameOfLife.java*

Skriv klassen **`GameOfLife`** med et hovedprogram (en `main`-metode). Det skal opprette en `Verden` med 8 rader og 12 kolonner og tegne 0-te generasjon. Så skal det følge utviklingen gjennom 3 generasjoner og skrive ut rutenettet for hver generasjon.

Mulige utvidelser (frivillig)

I stedet for alltid å ha 8 rader og 12 kolonner, kan du endre programmet til å spørre brukeren om størrelsen på rutenettet når programmet starter.

Du kan alternativt la antallet rader og kolonner være programparametre som hentes fra `main`-metodens parameter (ofte kalt `arg`).

I stedet for å gå gjennom et fast antall generasjoner, kan du endre programmet slik at det etter hver generasjon spør brukeren om det skal fortsette.

Eksempel på utskrift av spillebrettet

```
Generasjon nr 0:
+---+---+---+---+---+---+---+---+---+---+---+---+
| O | . | . | . | O | O | . | . | . | O | . | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| O | . | O | O | . | O | . | . | O | . | . | O |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | . | . | O | . | . | O | O | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | . | O | . | O | O | O | . | O | . | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| O | . | O | . | . | O | . | O | . | O | O | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | . | . | . | O | . | O | . | O | . | O | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | . | O | . | . | . | . | . | . | . | . | O |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | O | O | . | O | . | O | O | O | . | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
Det er 38 levende celler.
Generasjon nr 1:
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | . | O | O | O | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | O | O | . | O | . | O | O | . | . | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | . | O | . | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | . | O | . | O | . | . | . | O | O | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | O | O | . | . | . | . | . | . | O | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | . | O | . | O | O | O | O | . | O | O |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | O | . | O | O | O | . | . | . | O | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
| . | O | O | O | . | . | . | . | O | . | . | . |
+---+---+---+---+---+---+---+---+---+---+---+---+
Det er 39 levende celler.
```