Git

EN FORTELLING OM EN HISTORIEBOK

Hvorfor trenger vi Git?

- Har du noen gang villet endre noe kode, men så er du redd for å ødelegge noe du allerede har som funker?
- Har du noen gang villet regenerere en figur, men så har du endret noe av koden din sånn at det er slitsomt å finne ut av akkurat hva du gjorde for å lage den?
- Har du noen gang prøvd å kjøre en fil du skrev en stund siden, og oppdaget at den plutselig ikke funker lenger?

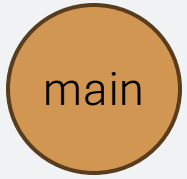
- Git løser alt dette!
- Git er et versjonskontrollsystem vi bruker for å enkelt holde kontroll over hvilke endringer vi har gjort, og enkelt å "spole tilbake" til akkurat hvordan ting så ut på et gitt tidspunkt.
- Det gjør at vi fryktløst kan kode i vei, uten å være redde for å ødelegge eller miste noe viktig mens vi holder på.

Konsepter vi skal dekke

- Den generelle strukturen til git
  - Konseptet "tree" og "branch"
  - "Remote" vs. "Local"
- Arbeidsflyt med git i praksis
- Kort oversikt over de viktigeste kommandoene, og anbefalt verktøy for å holde oversikt og lære mer.

Git er en historiebok

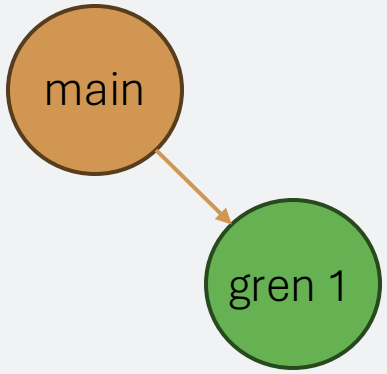
- Poenget er at alle endringer du gjør lagres for ettertiden – da blir det lett å finne tilbake til spesifikke versjoner, og det er veldig trygt å gjøre endringer uten å være redd for å ødelegge ting, du kan alltids bare rulle tilbake til forrige stabile versjon hvis alt går skeis.
- I tillegg organiserer vi utviklingen på en sånn måte at flere personer kan redigere koden, uten at vi ødelegger for hverandre.
- Derfor er historien organisert som et Tre: Stammen til treet kaller vi "main", og det er der vi har den siste stabile versjonen, og hele dens utviklingshistorikk. Når vi starter et prosjekt i git ser historikken vår sånn her ut:



- Det er bare én node (ett avsnitt i historieboka) som sier at vi har startet prosjektet.
- I de neste lysbildene skal jeg forklare den generelle strukturen i hvordan vi skriver i historieboka. De tekniske detaljene (hvilke kommandoer vi bruker osv.) kommer senere.

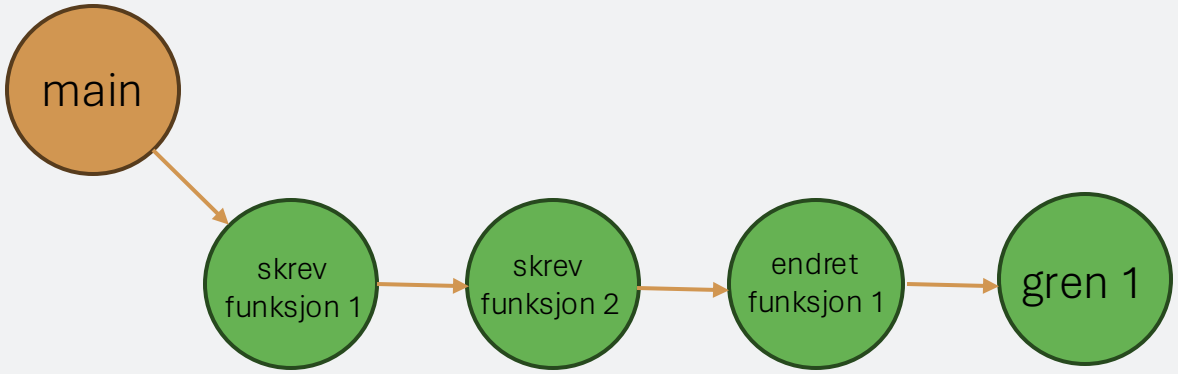
Strukturen til Git

- Nå som vi har et prosjekt, ønsker vi å gjøre endringer, da oppretter vi en gren ("branch"), sånn at historien ser sånn her ut

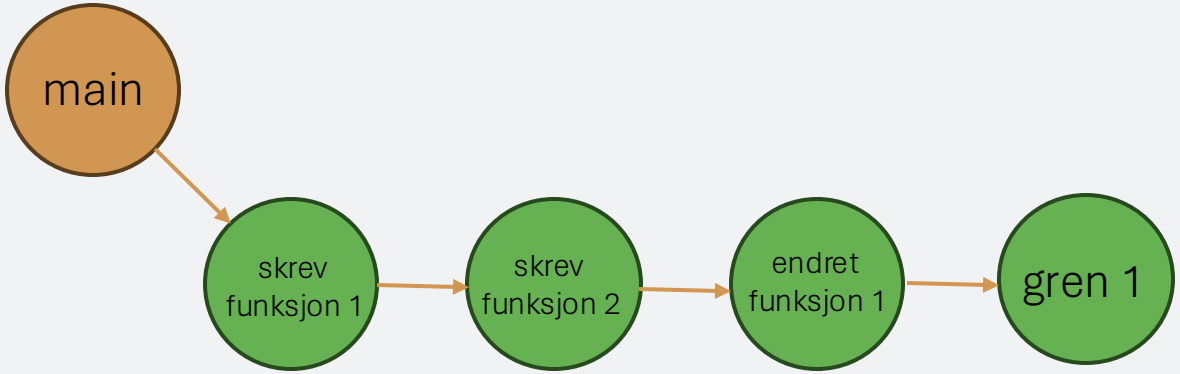


Bruk
git checkout –b gren_1
For å opprette en ny branch

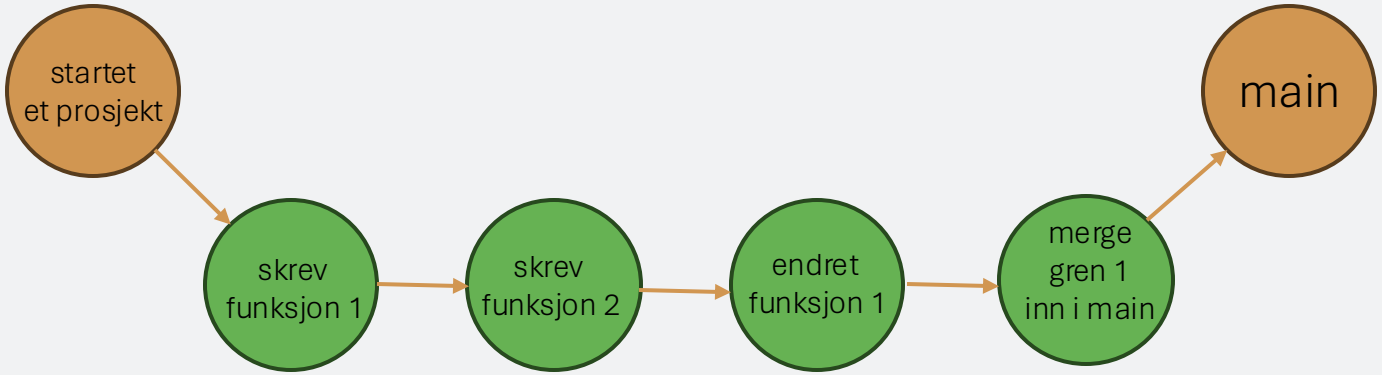
- Når vi gjør endringer så skriver vi dem inn til grenen vår



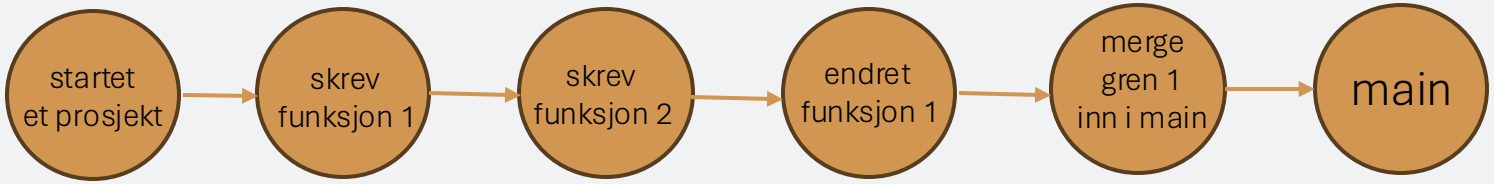
- Hver av nodene her representerer en endring som vi har lagret (mer om nøyaktig hvordan vi gjør det senere)



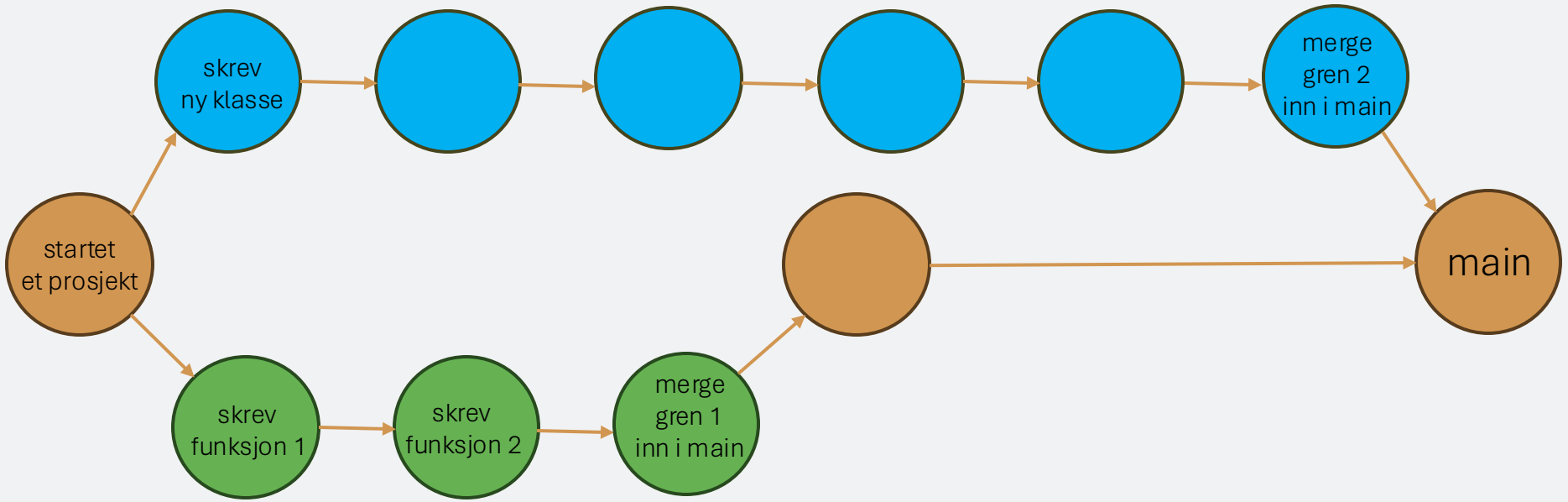
- Når vi er fornøyd med arbeidet vi har gjort, smelter vi grenen inn igjen i stammen ("merge"),



- Og som du kan se av figuren over, kan vi like gjerne tegne det sånn her:



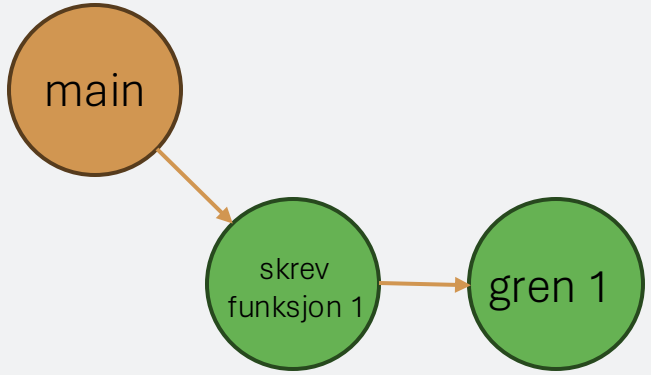
- To viktige grunner til at vi gjør dette:
  - Hvis flere personer jobber samtidig kan vi jobbe på hver vår gren, sånn at vi ikke tråkker hverandre på tærne. Hvis vi gjør endringer som kræsjer med hverandre (på hver vår gren) kan vi finne ut av det sammen når vi skal merge inn i main.
  - Hvis vi ødelegger noe er det lett å forkaste endringene våre, og begynne på nytt fra sist ting funket. Det er også lett å spore hvor endringene som ødela ting skjedde, sånn at man kan fikse det uten å forkaste masse arbeid.
- En typisk utviklingshistorikk ser da ut som noe sånn her:



- Hvor én person har gjort endringer på den grønne grenen, mens en annen har gjort arbeid på den blå. Når den blå skal "merges" inn til main, hjelper git oss med å løse eventuelle konflikter mellom endringene gjort på den grønne og den blå grenen. Stort sett er det smertefritt.

Remote vs. lokal

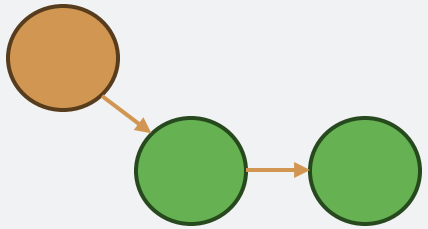
- Nå som vi har sett på trestrukturen til git, er det på tide å avklare forskjellen på det lokale treet du har på pc'en, og tre et som er lagret i skyen (f.eks. på github). Vi tar utgangspunkt i at det finnes et eksisterende prosjekt på github, med en struktur som ser ut som



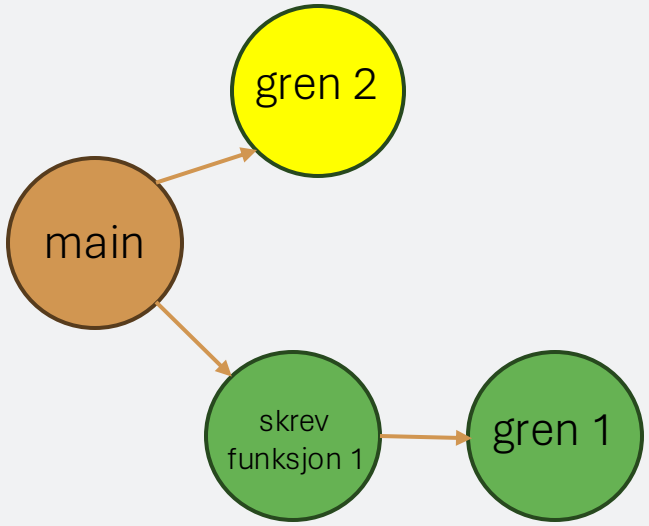
- Når vi skal jobbe med dette prosjektet, begynner vi med å klone det ("clone"), så vi får en struktur som ser ut som

- Deretter kan vi opprette vår egen gren på den lokale versjonen av treet (neste side)

GitHub

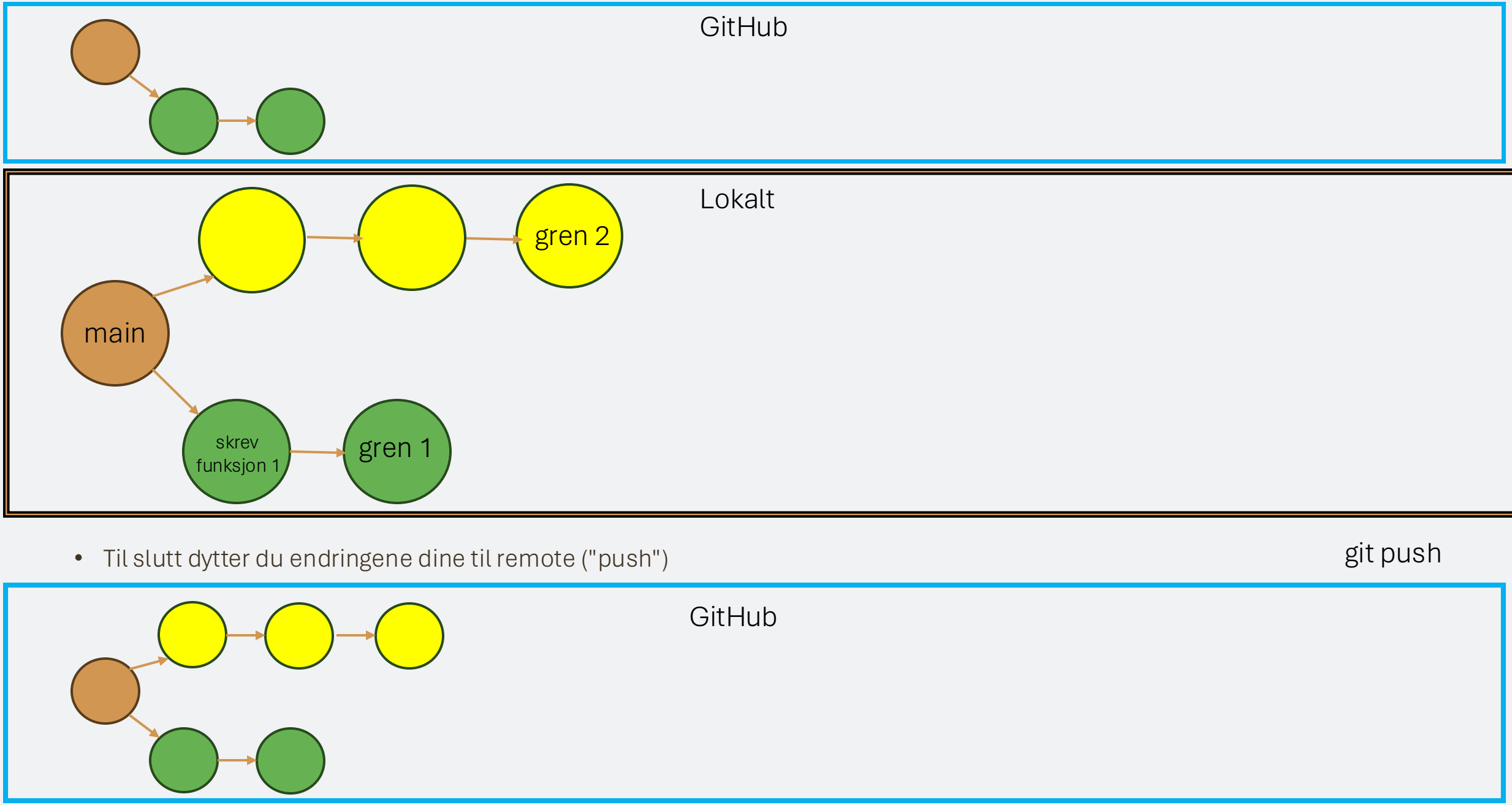


Lokalt



git checkout –b gren_2

- Så kan du gjøre endringer på den nye grenen din (neste side)



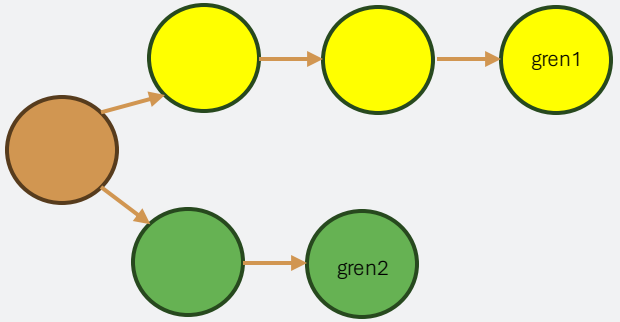
- Til slutt dytter du endringene dine til remote ("push")

git push

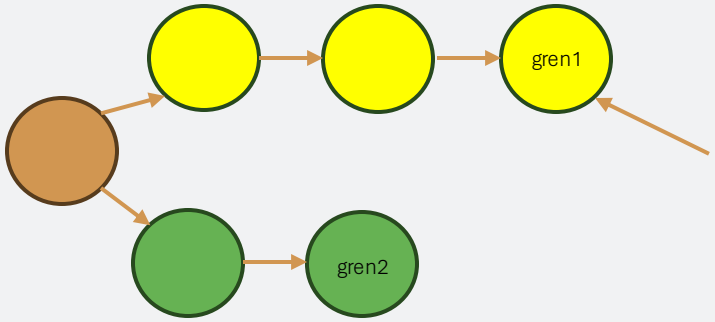
Hvis noen andre pusher før deg

- Når man er flere som jobber på et prosjekt kan du komme i en situasjon hvor noen andre har sendt endringer til remote som du har lyst til å hente til din lokale maskin, altså en situasjon som ser sånn her ut, hvor det er funksjonalitet på "gren1" som du ønsker å hente til "gren2"

GitHub

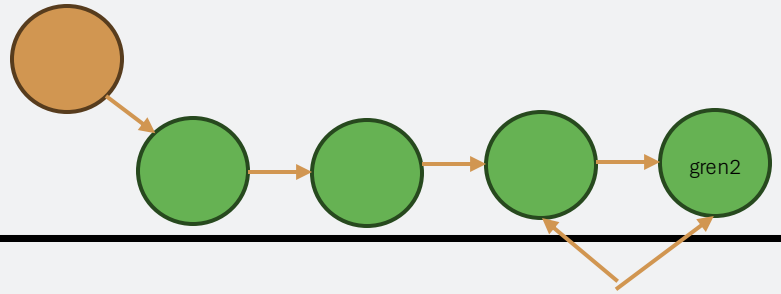


Min maskin



Endringer jeg har gjort (og pushet) som du ønsker å hente til "gren2"

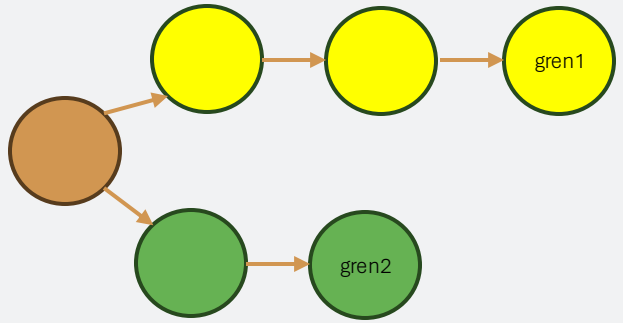
Din maskin



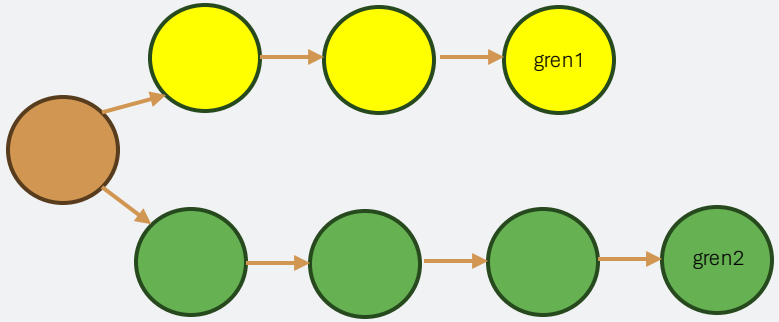
Endringer du har gjort siden sist du pushet

Hvis noen andre pusher før deg

- Det problemet løses ved at du først "pull"'er alle nye endringer fra remote, slik at versjonen du har lokalt inneholder alle nye endringer på remote. Deretter "merge"'er du "gren1" inn i din "gren2"



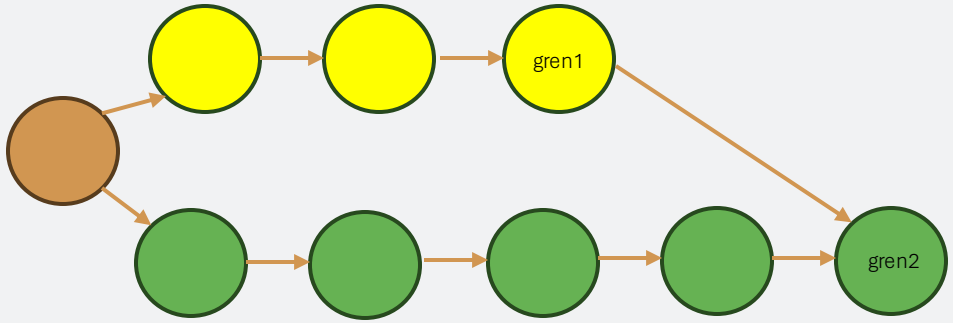
GitHub



Din maskin

Denne kommandoen henter "gren1" fra remote til vår maskin.

git switch –c gren1 origin/gren1



Din maskin

Denne kommandoen smelter endringer på "gren1" inn i "gren2". Kjør "git branch" først for å sjekke at du er på "gren2" når du kjører kommandoen.

git merge gren1
git commit –m "Hentet endringer fra gren1"

Kort om god praksis og "divergente" grener

- Hvis man utvikler kode på forskjellige grener er det god praksis å sørge for at de gjevnlig "merges".
- Hvis ikke kan man ende opp med en situasjon hvor det er gjort forskjellige endringer i samme kode, som ikke er kompatible.
- Når man da prøver å "merge" får man en "merge conflict" fordi git ikke vet hvilke endringer den skal beholde.
- Hvis man har veldig mange sånne problemer, snakker man om "divergente" grener.
  - De kan være slitsomme å ha med å gjøre.
- Derfor: Sørg for å "merge" endringer fra "main" med gjevne mellomrom, sånn at du ikke divergerer vekk.

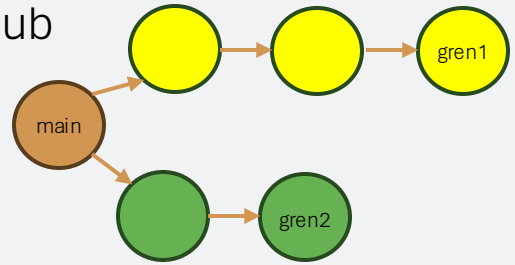
Når flere jobber på samme prosjekt: Pull requests

- En "Pull request" (PR) har et litt forvirrende navn, men er kritisk når man er flere som jobber sammen.
  - Navnet kommer av at du forespør (request) at *noen andre* skal "pull"'e endringene dine.
- Når vi er flere som jobber på samme prosjekt er arbeidsflyten typisk som følger:
  - Du gjør lokale endringer på din "branch"
  - Du pusher dine lokale endringer til remote
  - Når du er fornøyd, og vil merge til main, oppretter du en Pull request på GitHub, hvor du ber om at din branch skal merges til main.
  - Noen andre ser over det du har gjort, ber kanskje om endringer, og godkjenner pull requesten
    - Nå har "main" endret seg på remote, men *ikke på din lokale maskin*.
  - Du kjører
    - git checkout main
    - git pull
  - For å hente endringene på main til din lokale maskin.
  - Så kjører du ofte
    - git checkout min_andre_gren
    - git merge main
  - For å smelte inn endringene til andre grener du jobber på (for å unngå divergente grener).
- Se grafisk forklaring på neste side

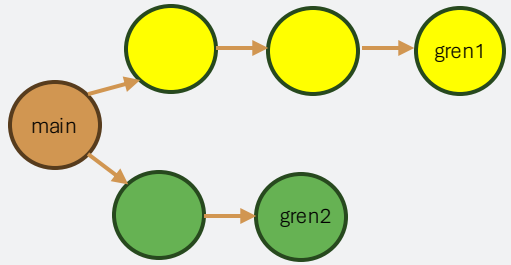
Når flere jobber på samme prosjekt: Pull requests

- La oss si at du har gjort endringer på "gren1", og ønsker å dytte dem til "main". Du har allerede kjørt "git push" så endringene dine er på GitHub. (fortsetter på neste siden)

GitHub

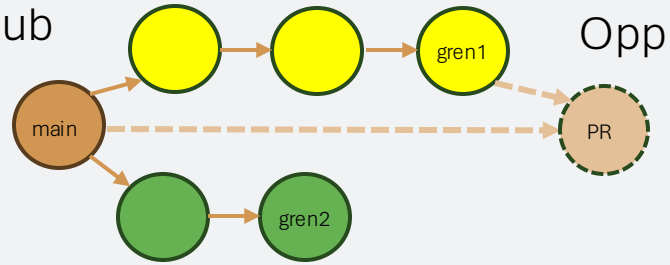


Din maskin

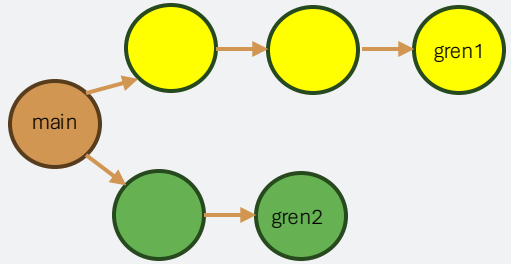


GitHub

Oppretter en PR

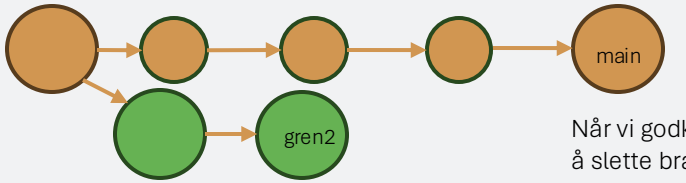


Din maskin



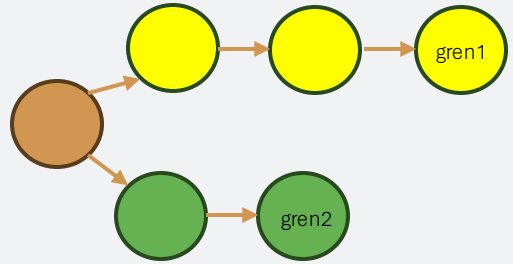
GitHub

PR godkjennes



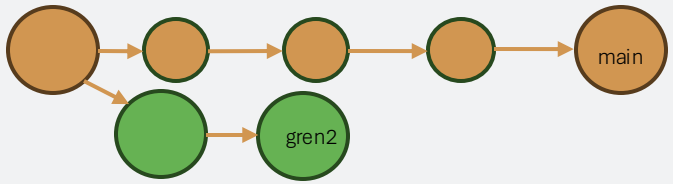
Når vi godkjenner en PR pleier vi å slette branchen fra remote.

Din maskin

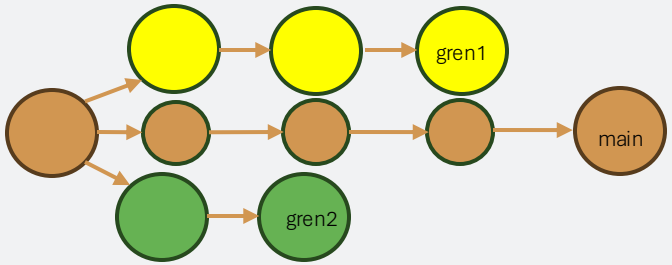


Merk: Din maskin er ikke oppdatert enda

GitHub



Din maskin



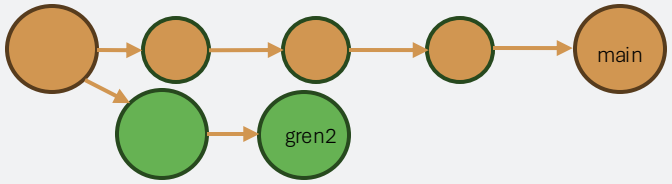
Denne kommandoen henter endringene på main fra GitHub til din maskin.

git pull

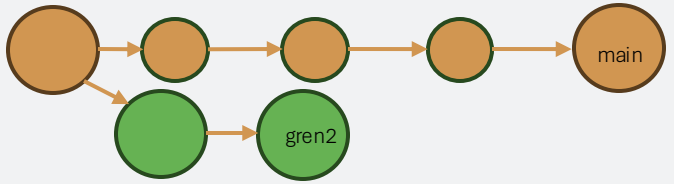
Når flere jobber på samme prosjekt: Pull requests

- Nå som endringene dine på "gren1" har blitt merget til "main" kan vi trygt slette den, og for å unngå at "gren2" divergerer, kan det være lurt å merge "main" inn til "gren2".

GitHub



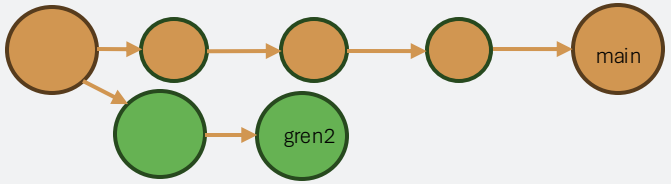
Din maskin



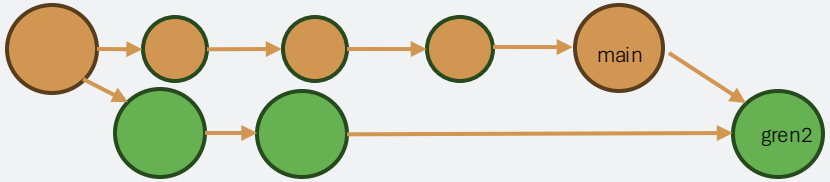
Denne kommandoen sletter "gren1".
Hvis "gren1" ikke er merget kommer git til å varsle oss.

git branch –d gren1

GitHub



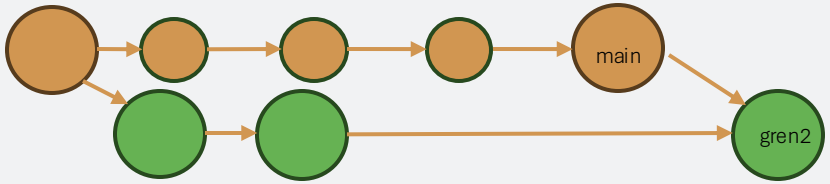
Din maskin



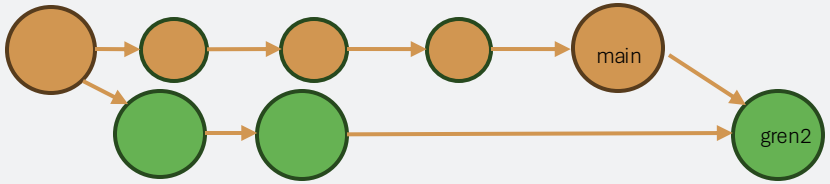
Henter endringer på "main" inn i "gren2"

git merge main

GitHub



Din maskin



Sender endringene til remote

git push

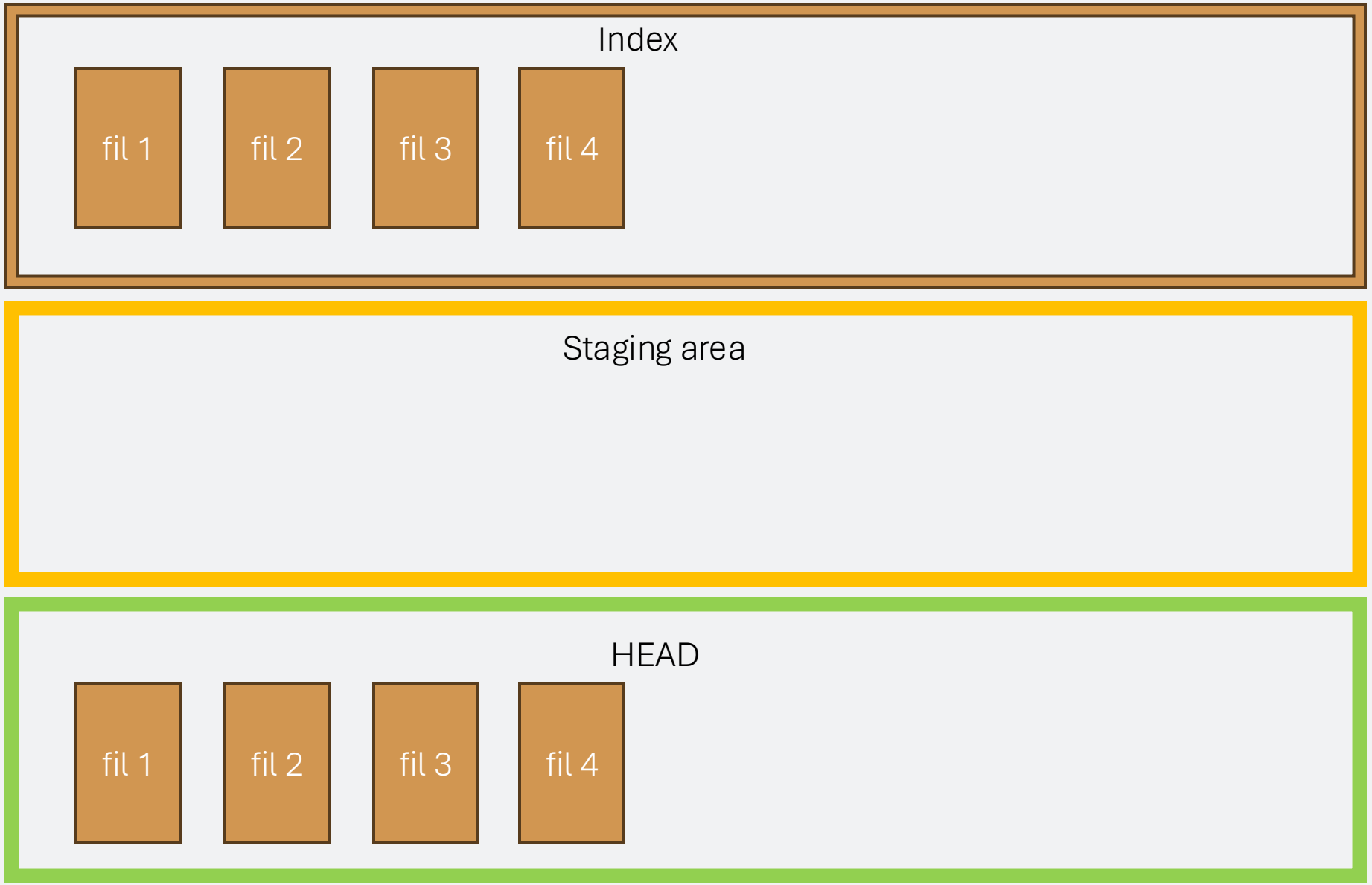
Kort oppsummering

- Vi kloner prosjektet til en lokal maskin
- Lager en ny branch
- Gjør endringer på den lokale branchen (commit)
- Pusher endringene til remote
- Når vi er fornøyd med endringene på en branch (for eksempel når vi har ferdigstillt en ny funksjonalitet) kan vi "merge"e den branchen inn i main.
- Hvis det er gjort endringer på remote som vi ønsker å hente, kan vi "pull"'e dem til vår maskin.
- Det er god praksis å merge endringer fra "main" til vår branch med gjevne mellomrom sånn at vi ikke får "divergente grener".
- Når flere jobber på samme prosjekt bruker vi "pull requests" for å ha bedre kontroll på hva som merges til main.

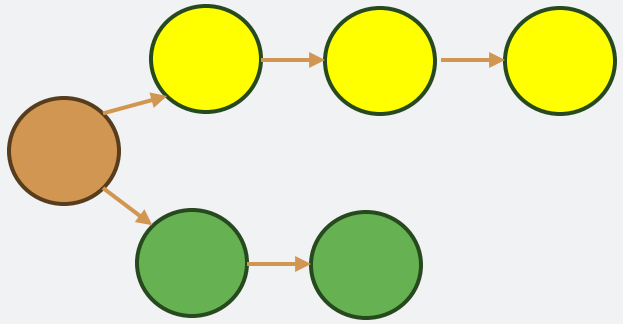
Mer om lokale endringer

- Frem til nå har vi sett på den overordnede arbeidsflyten, og etablert at Git fungerer som en historiebok.
  - Når du vil endre historieboka gjør du først lokale endringer på din pc, før du sender endringene til "remote"
- Nå skal vi se spesifikt på prosessen du går gjennom når du endrer noe lokalt og skriver dem til din lokale historiebok.
  - I de neste lysbildene har vi altså ikke noe konsept om en "remote" det vi beskriver her foregår utelukkende lokalt.
- Prosessen med å endre en branch består av tre steg
  - Skrive kode
  - Legge endringene i "staging area"
  - "commit"'e endringene til branchen.

- Vi kan dele det du har på pc'en inn i tre deler: Indexen, Staging area, og "det du ser" (HEAD på git-språk).
- Grafisk kan vi vise det sånn her:

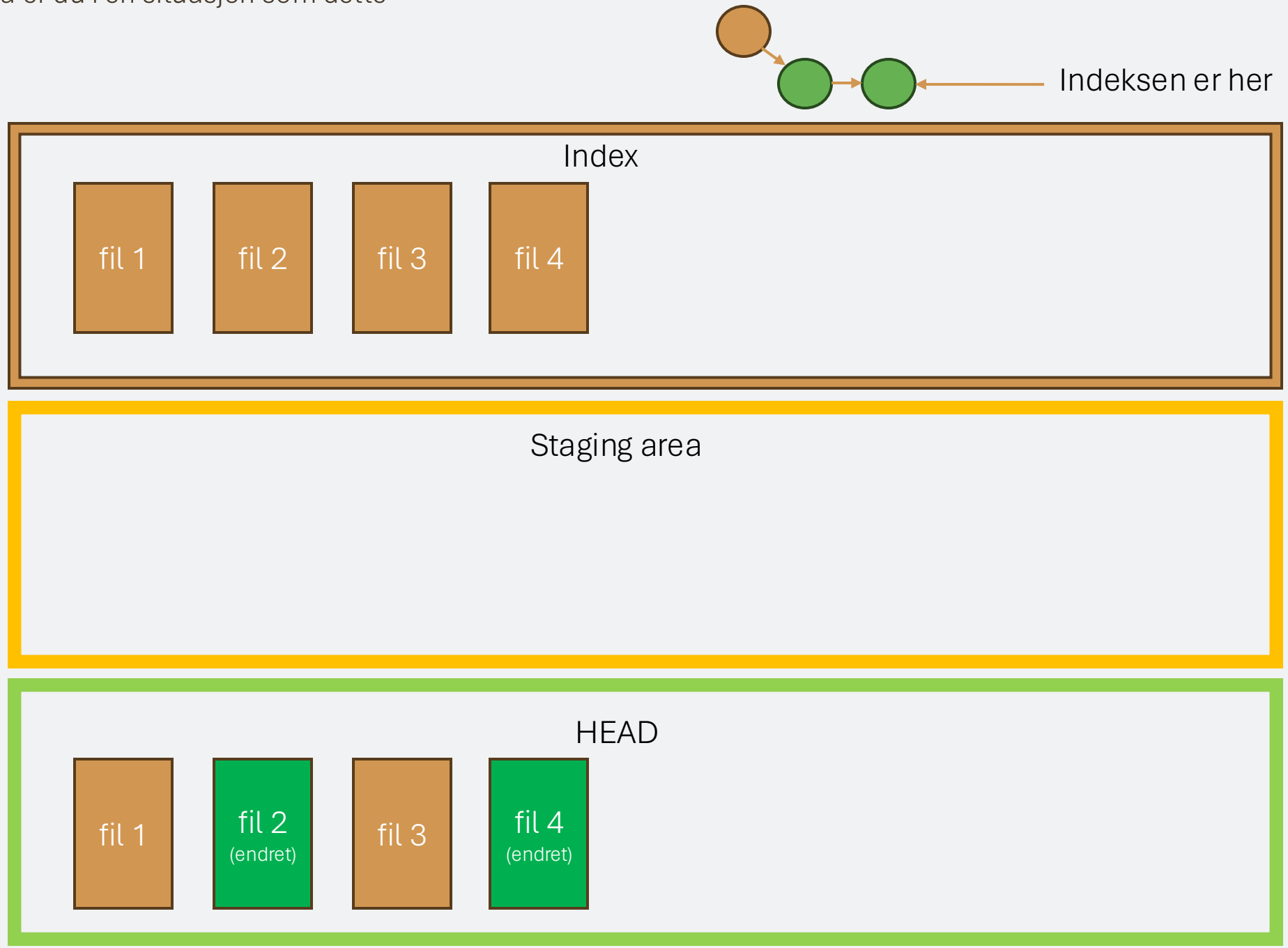


- Når du åpner en fil på pc'en, eller ellers navigerer lokalt på maskinen din, ser du det som er i HEAD blokken.
- Nodene vi har tegnet tidligere, i figurer som denne

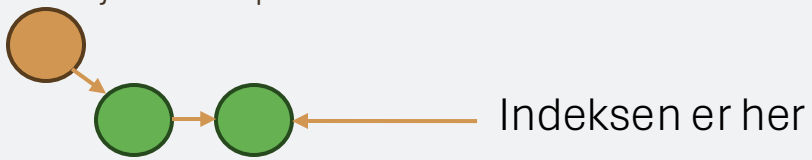


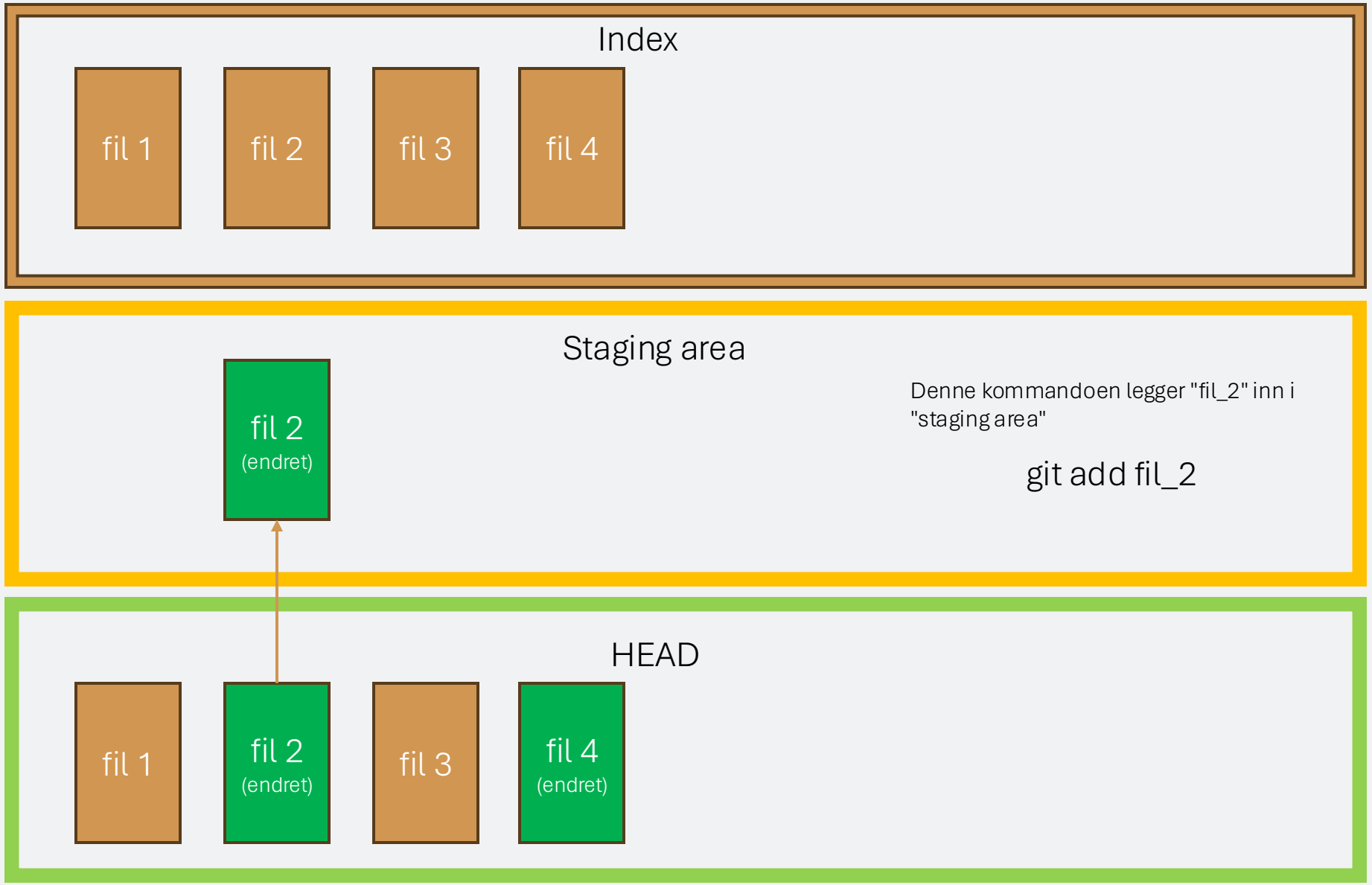
- representerer filene som er i "index" blokken. "Indexen" er altså historieboka vår, mens "HEAD" er skrivebordet vårt.
- "Staging area" kan vi tenke på som en egen stabel på skrivebordet, hvor vi samler opp endringer som skal legges inn i historieboka.
- La oss nå si at du gjør noen endringer på et par filer (markert med grønt, neste lysbilde)

- Da er du i en situasjon som dette

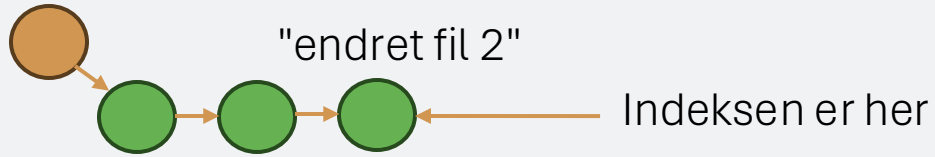


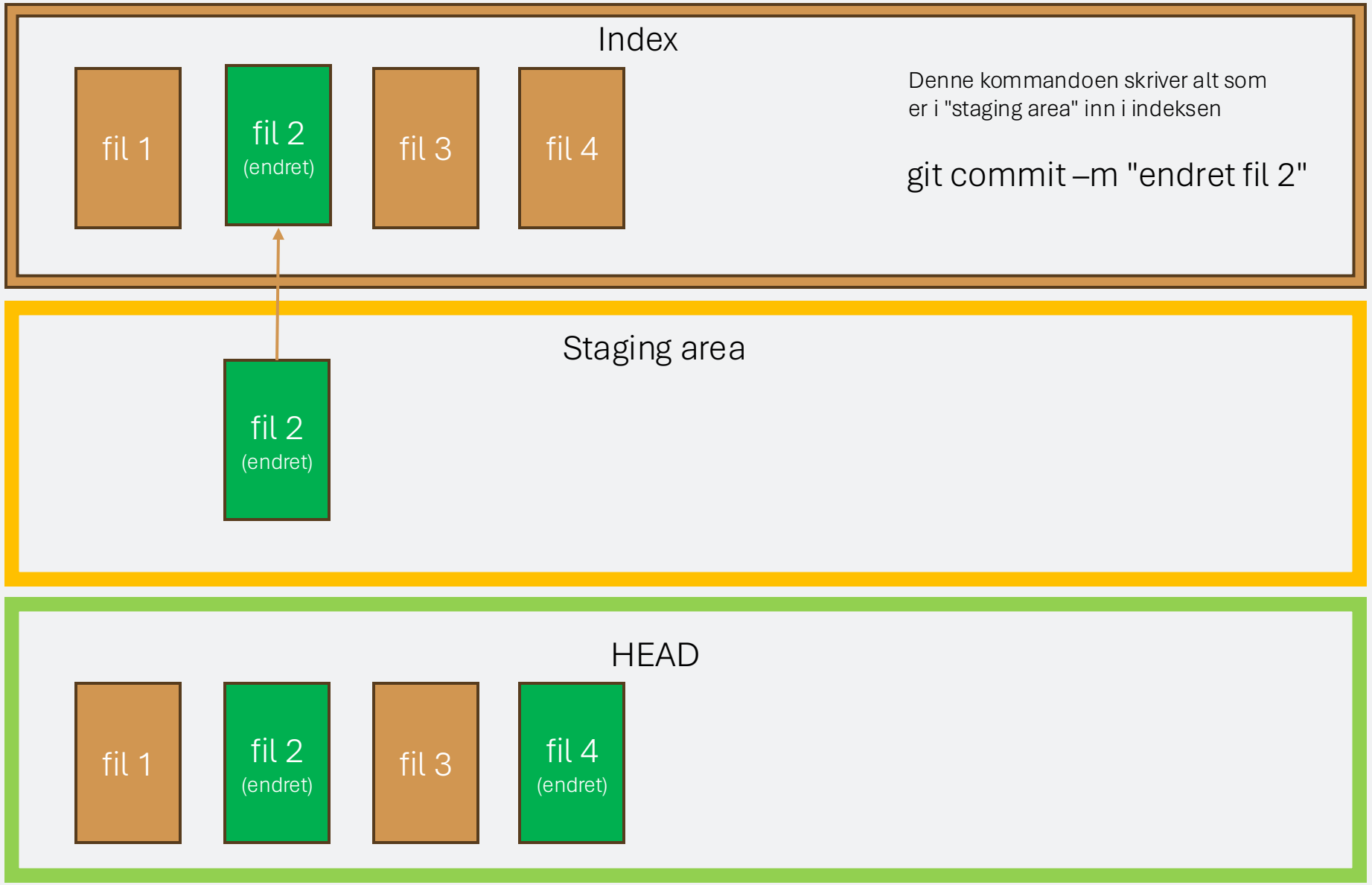
- Nå kan vi tenke oss at du vil lagre endringene i fil 2, men vil fortsette å jobbe litt på fil 4 før du skriver den inn i historieboka (indeksen).
- Da legger du fil 2 inn i "Staging area"





- "Staging area" er ganske enkelt et sted hvor du kan se over alt en siste gang før du skriver det inn i indexen.
- Når du er fornøyd med status på filene i "Staging area" skriver du dem til indexen, og legger ved en liten melding om hva du har endret





Merk: Når du kjører *git commit* dyttes alle endringene i "staging area" til indeksen

Tips: Bruk *git status* for å se hva som er endret, hva som er i "staging area" osv. før du kjører *git commit*

Oversikt over de viktigeste kommandoene

KOMMANDO	BRUK	LYSBILDE
• git branch	• Gir en liste over alle lokale brancher	• n/a
• git checkout –b <branch_navn>	• Oppretter en ny branch, og flytter til branchen	• 5 / 10
• git checkout <branch_navn>	• Flytter HEAD til en annen eksisterende branch	• 15
• git add <filnavn>	• Legger til en fil i staging area	• 23
• git status	• Gir en kompakt oversikt over endrede filer	• 24
• git commit	• Sender filene i "Staging area" til din lokale indeks	• 13 / 24
• git commit –m "en liten melding"	• Kortversjon for å legge inn en melding uten å bruke vim	• 13 / 24
• git push	• Sender endringene i din lokale indeks til remote indeks	• 11 / 17

• git merge <annen_branch>	• Henter endringer fra <annen_branch> inn i branchen vi er på.
	  • Typisk bruk er at vi vil hente oppdatteringer fra main, og skriver
	  • git merge main
		• 6 / 13 / 17

Noen andre kommandoer som kan være fine

KOMMANDO	BRUK
• git switch -c <gren> origin/<gren>	• Henter branchen <gren> fra remote til din lokale maskin. (Lysbilde 13)
• git stash	• Legger endrede filer til side for senere. Tenk på det som at "stash" er skrivebordsskuffen. Når du vil ta en titt på en annen branch må du legge arbeidet du holder på med i skuffen for å rydde plass på skrivebordet før du kjører • git checkout <annen_branch> for å åpne den andre branchen
• git stash pop	• Henter ut det som ligger i skrivebordsskuffen, og legger det i HEAD.

Siste anbefalinger

- Sjekk ut appen "GitKraken": https://www.gitkraken.com
- Det er en app som er veldig fin for å hjelpe til med å få oversikt over hvordan git fungerer – den gir veldig god visualisering av prosjektflyten. Måten ting er illustrert her er inspirert av måten gitkraken viser prosjektet.
- Det koster litt, men er verdt hver krone for noen som skal lære git.

- **NB:** Hvis du søker på et problem, og noen sier at du skal bruke "git rebase", eller på andre måter "endre historien" ("changing history"), kan det være lurt å spørre om hjelp før du gjør det.
  - Så lenge vi har en intakt git-historikk er det nesten umulig å miste arbeid vi har gjort.