शाखाहरूको संक्षिप्त परिचय (Branches in a Nutshell)

Git शाखाकरणलाई राम्रोसँग बुझ्नका लागि, हामीलाई Git ले डाटालाई कसरी भण्डारण गर्छ भनेर हेर्न आवश्यक छ।

जसरी "Git के हो?" मा उल्लेख गरिएको छ, Git ले डाटालाई *changesets* वा फरकहरूका रूपमा भण्डारण गर्दैन, बरु यसलाई *snapshots* को श्रृंखलाका रूपमा भण्डारण (store) गर्छ।

जब तपाईं एक *commit* गर्नुहुन्छ, Git ले एक *commit object* भण्डारण गर्छ जसमा तपाईंले *stage* गरेको सामग्रीको स्न्यापशटमा एक *pointer* हुन्छ। यस *object* मा लेखकको नाम र इमेल ठेगाना, तपाईंले टाइप गरेको सन्देश, र यस *commit* लाई प्रत्यक्ष रूपमा अघिल्लो *commit* हरू (यसको अभिभावक वा अभिभावकहरू) को *pointer* पनि हुन्छ:

- *Initial commit* का लागि शून्य अभिभावक (Parent)।
- सामान्य *commit* का लागि एउटा अभिभावक (Parent)।
- दुई वा बढी शाखाहरूको *merge* बाट उत्पन्न *commit* का लागि धेरै अभिभावक (Parent)।

यसलाई दृश्यात्मक रूपमा बुझ्नका लागि, कल्पना गर्नुहोस् कि तपाईंको एउटा *directory* छ जसमा तीनवटा फाइलहरू छन्, र तपाईंले तिनलाई *stage* गरी *commit* गर्नुभएको छ।

Stage गर्दा, प्रत्येक फाइलको लागि एक *checksum* (SHA-1 hash) गणना हुन्छ, र त्यस संस्करणलाई Git भण्डारमा भण्डारण गरिन्छ (Git ले तिनलाई *blobs* भनेर सम्बोधन गर्छ) र यो *checksum* लाई *staging area* मा थपिन्छ:

```
$ git add README test.rb LICENSE
$ git commit -m 'Initial commit'
```

जब तपाईं git commit चलाउनुहुन्छ, Git ले प्रत्येक *subdirectory* (यस अवस्थामा, केवल मूल परियोजना *directory*) को *checksum* गणना गर्छ र यसलाई Git भण्डारमा (Repository) *tree object* का रूपमा भण्डारण गर्छ।

त्यसपछि Git ले एउटा *commit object* सिर्जना गर्छ जसमा *metadata* र मूल परियोजना *tree* मा एक *pointer* हुन्छ ताकि आवश्यक पर्दा उक्त स्न्यापशट पुनः सिर्जना गर्न सकियोस्।

अब तपाईंको Git भण्डारमा पाँचवटा *objects* छन्:

1. तीनवटा *blobs* (प्रत्येकले तीन फाइलहरूको सामग्री प्रतिनिधित्व गर्छ)।
2. एक *tree* (जसले *directory* को सामग्री र कुन फाइल नाम कुन *blob* मा भण्डारण गरिएको हो भन्ने विवरण दिन्छ)।
3. एक *commit* (जसमा मूल *tree* को *pointer* र सबै *commit metadata* हुन्छ)।

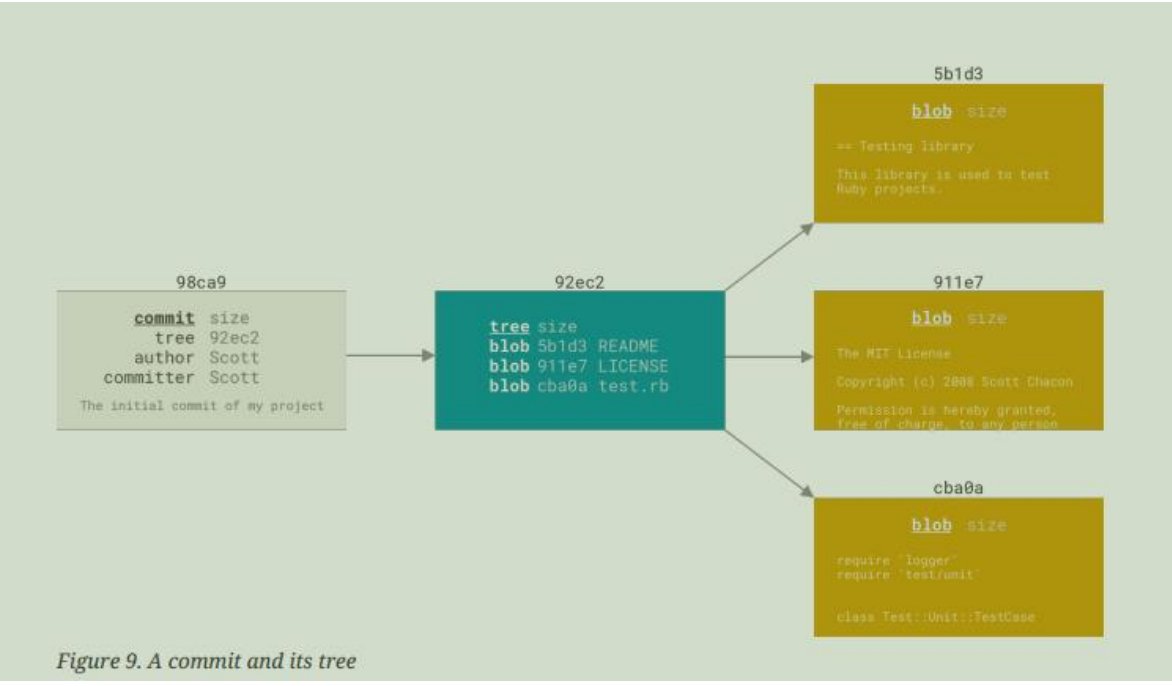

Figure 9. A commit and its tree

यसलाई अब नेपालीमा सरल तरिकामा व्याख्या गरौं:

1. **Commit Object (98ca9)**:
    - यो एउटा **commit object** हो, जसले तलका कुराहरू समावेश गर्दछ:
        - **Tree Object (92ec2)** मा एक pointer, जसले परियोजनाको डिरेक्टरी संरचना र फाइल सामग्रीको प्रतिनिधित्व गर्दछ।
        - Metadata, जस्तै लेखकको नाम, commit गर्ने व्यक्तिको नाम, र commit मेसेज।
        - **Parent Pointer**: यदि पहिले commit भएको छ भने, त्यसतर्फ इशारा गर्छ। (यस अवस्थामा यो सुरुवाती commit हो, त्यसैले कुनै parent छैन।)
2. **Tree Object (92ec2)**:
    - **Tree Object** ले डिरेक्टरीमा भएका फाइलहरूको सूची राख्छ।
    - यसले प्रत्येक फाइल नाम (जस्तै, README, LICENSE, test.rb) लाई **Blob Object** सँग जडान गर्छ, जसले फाइलको सामग्री राख्छ।
3. **Blob Objects (5b1d3, 911e7, cba0a)**:
    - प्रत्येक **Blob Object** ले एक फाइलको वास्तविक सामग्री राख्छ।
    - उदाहरणका लागि:
        - Blob 5b1d3 ले README फाइलको सामग्री समावेश गर्छ।
        - Blob 911e7 ले LICENSE फाइलको सामग्री समावेश गर्छ।
        - Blob cba0a ले test.rb फाइलको सामग्री समावेश गर्छ।
4. **यदि तपाईं अर्को commit गर्नुहुन्छ भने**:
    - नयाँ commit ले आफ्नो **Tree Object** मा इशारा गर्नेछ, जसले अपडेट गरिएको परियोजनाको snapshot देखाउँछ।
    - नयाँ commit ले अघिल्लो commit (98ca9) तर्फ पनि इशारा गर्नेछ, जसले commits को चेन बनाउँछ।

यो संरचनाले Git लाई सजिलै:

- समयसँगै भएका परिवर्तनहरू ट्र्याक गर्न।
- कुनै पनि commit मा परियोजनाको सही अवस्था पुनः बनाउन सक्षम बनाउँछ।

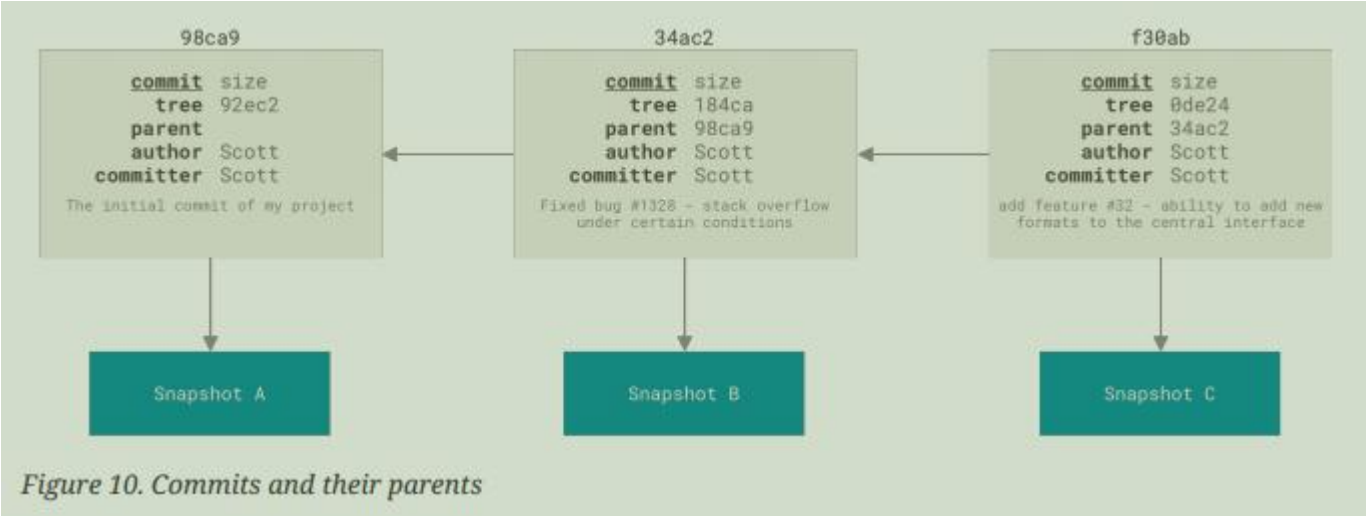

Figure 10. Commits and their parents

चित्रले Git commits को श्रृंखला र उनीहरूका parents (अघिल्लो commits) लाई देखाउँछ। यसलाई सरल भाषामा नेपालीमा व्याख्या गरौं:

1. **Commit 98ca9 (Snapshot A)**:
    - यो पहिलो commit हो।
    - यसमा कुनै **parent commit** छैन किनभने यो सुरुवाती commit हो।
    - यसले **Tree Object (92ec2)** मा इशारा गर्छ, जसले snapshot A को डाटा (परियोजनाको फाइलहरूको संरचना र सामग्री) समावेश गर्दछ।
2. **Commit 34ac2 (Snapshot B)**:
    - यो दोस्रो commit हो।
    - यसले आफ्नो **parent commit (98ca9)** मा इशारा गर्छ, जसले यस commit अघि परियोजनाको अवस्था ट्याक गर्दछ।
    - यसले **Tree Object (184ca)** मा पनि इशारा गर्छ, जसले Snapshot B (परियोजनाको नयाँ अवस्थामा) को डाटा समावेश गर्दछ।
    - Commit मेसेजले समस्या समाधान (stack overflow) उल्लेख गरेको छ।
3. **Commit f30ab (Snapshot C)**:
    - यो तेस्रो commit हो।
    - यसले आफ्नो **parent commit (34ac2)** मा इशारा गर्छ।
    - यसले **Tree Object (0de24)** मा इशारा गर्छ, जसले Snapshot C (परियोजनाको अझ अपडेट गरिएको अवस्था) को डाटा समावेश गर्दछ।
    - Commit मेसेजले नयाँ सुविधा (new formats to the interface) थपिएको उल्लेख गरेको छ।

संक्षेपमा:

- प्रत्येक commit ले आफ्नो parent commit मा इशारा गरेर commits को चेन बनाउँछ।
- हरेक commit ले snapshot (परियोजनाको अवस्थामा भएको परिवर्तन) को प्रतिनिधित्व गर्छ।
- Git ले यी snapshots र commits लाई efficient तरिकाले ट्याक गर्छ, जसले तपाईंलाई कुनै पनि बिन्दुमा फर्किन अनुमति दिन्छ।

Git मा Branch के हो? (नेपालीमा)

Git मा **Branch** भनेको कुनै commit मा रहेको हल्का तौलको movable pointer हो।

मुख्य कुरा:

1. **Default Branch:**
    - Git मा default branch लाई "master" भनिन्छ।
    - जब तपाईं commits बनाउन थाल्नुहुन्छ, तपाईंलाई "master" branch दिइन्छ, जसले तपाईंले अन्तिममा गरेको commit तर्फ इशारा गर्दछ।
2. **Commit गर्दा के हुन्छ?**
    - जब तपाईं नयाँ commit गर्नुहुन्छ, "master" branch को pointer स्वचालित रूपमा नयाँ commit तर्फ अगाडि सर्दछ।

3. **"Master" को विशेषता:**
    - "master" branch Git मा कुनै विशेष branch होइन।
    - यो अरू branches जस्तै समान छ।
    - **किन प्राय: "master" हुन्छ?**
        - किनकी git init कमाण्डले यो default रूपमा बनाउँछ।
        - धेरैजसो व्यक्तिले यसलाई परिवर्तन गर्न झन्झट मान्दैनन्।

सरल शब्दमा:

Git मा branch ले commits ट्र्याक गर्न सहयोग गर्दछ। "master" branch Git को default शाखा हो, तर तपाईं यसलाई मनपर्ने नामले परिवर्तन गर्न सक्नुहुन्छ।

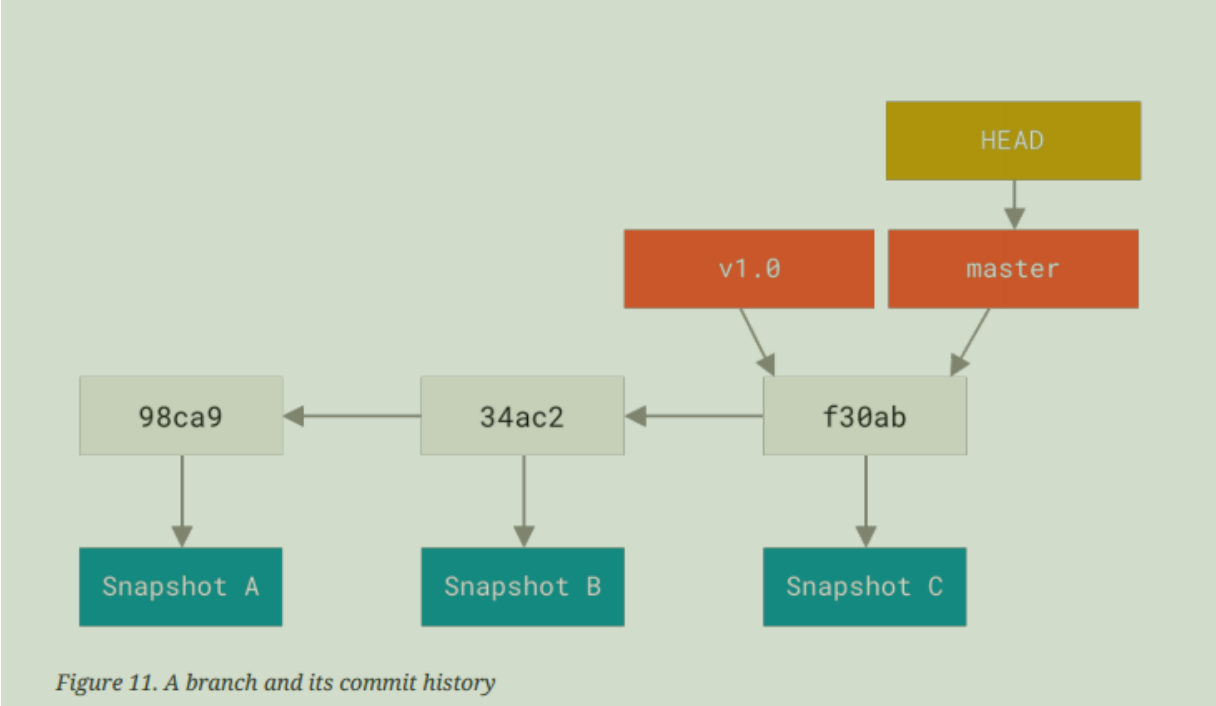

Figure 11. A branch and its commit history

दिएको छवि Git को **Branch** र Commit History को सम्बन्ध देखाउँछ। यसलाई संक्षिप्तमा वर्णन गर्दा:

1. **Snapshots (Commits):**
    - 98ca9, 34ac2, र f30ab विभिन्न commits (snapshots) हुन्।
    - यी snapshots हरूमा तपाईंको project को अवस्थाको विवरण छ।
2. **Branch Pointer:**
    - **master branch** commit f30ab मा इशारा गरिरहेको छ।
    - Branch भनेको कुनै निश्चित commit मा इशारा गर्ने pointer हो।
3. **Tag:**
    - v1.0 ले commit f30ab लाई indicate गरेको छ। Tags प्राय: release versions जनाउनका लागि प्रयोग गरिन्छ।
4. **HEAD:**
    - **HEAD** भनेको Git मा सक्रिय branch (या commit) को संकेत हो।
    - यहाँ HEAD master branch मा छ, जसले बताउँछ कि हालको सक्रिय branch "master" हो।

के हुन्छ commit गर्दा?

- नयाँ commit गर्दा master pointer र HEAD दुबै अगाडिको commit तर्फ सर्छन्।

- यसले branches लाई project को इतिहास ट्याक गर्न सजिलो बनाउँछ।

Creating a New Branch - - - -

नयाँ शाखा सिर्जना गर्दा के हुन्छ भने, यो तपाईंलाई घुम्नको लागि नयाँ pointer सिर्जना गर्दछ। मानौं तपाईं testing नामक नयाँ शाखा सिर्जना गर्न चाहनुहुन्छ। तपाईं यसलाई यसरी गर्नुहुन्छ:

```
$ git branch testing
```

यसले तपाईं अहिले रहेको commit मा नयाँ pointer सिर्जना गर्दछ।

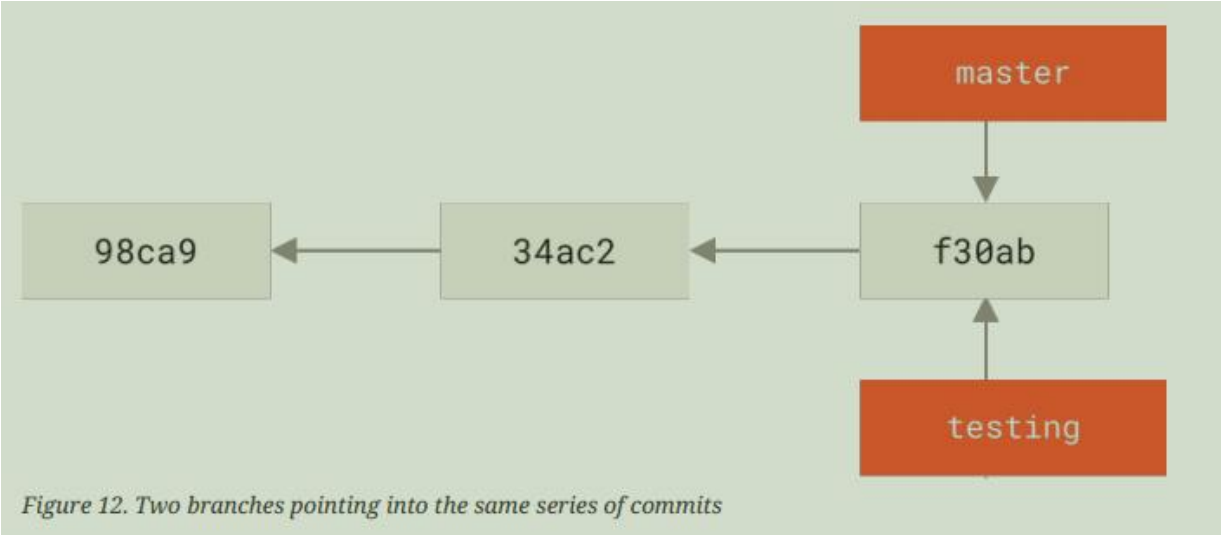

Figure 12. Two branches pointing into the same series of commits

How does Git know what branch you're currently on?

Git ले तपाईं अहिले कुन शाखामा हुनुहुन्छ भनेर कसरी थाहा पाउँछ भने, यसले एउटा विशेष pointer राख्छ जसलाई HEAD भनिन्छ। ध्यान दिनुहोस् कि यो अन्य VCSs (जस्तै Subversion वा CVS) मा भएका HEAD को अवधारणासँग धेरै फरक छ। Git मा, यो तपाईं अहिले जुन स्थानीय शाखामा हुनुहुन्छ, त्यहाँको pointer हो। यस अवस्थामा, तपाईं अझै master शाखामा हुनुहुन्छ। git branch आदेशले केवल नयाँ शाखा सिर्जना गर्यो – यसले तपाईंलाई त्यसमा स्विच गराएन।

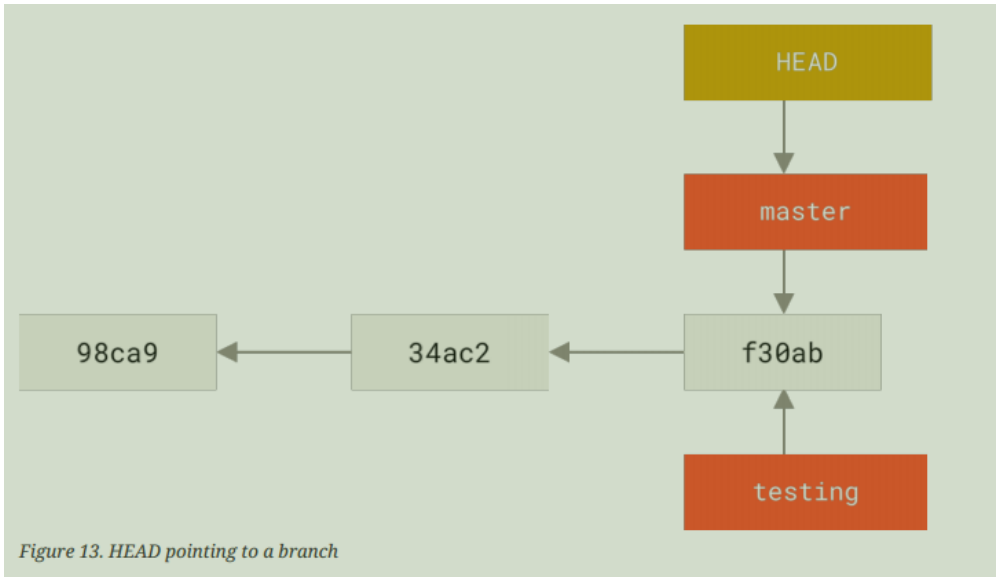

Figure 13. HEAD pointing to a branch

तपाईं यसलाई सजिलै देख्न सक्नुहुन्छ एउटा सामान्य git log आदेश चलाएर, जसले तपाईंलाई शाखा `pointer` हरू कहाँ इशारा गरिरहेको छ भन्ने देखाउँछ। यो विकल्पलाई `--decorate` भनिन्छ।

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to
the central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

तपाईं देख्न सक्नुहुन्छ कि `master` र `testing` शाखाहरू सिधै `f30ab` commit को नजिक छन्।

Switching Branches

शाखामा स्विच गर्नु भनेको तपाईंको कामलाई अर्को शाखामा लैजानु हो। जब तपाईं एउटा शाखामा काम गर्दै हुनुहुन्छ र अर्को शाखामा जान चाहनुहुन्छ भने, तपाईंलाई git checkout आदेश प्रयोग गर्नु पर्छ। अब, हामी नयाँ testing शाखामा स्विच गर्नेछौं:

```
$ git checkout testing
```

यो आदेश चलाउँदा, Git तपाईंको HEAD (जो तपाईंको वर्तमान स्थितिलाई ट्र्याक गर्दैछ) लाई testing शाखामा पठाउँछ। यसले तपाईंलाई नयाँ शाखामा काम गर्न अनुमति दिन्छ। अब तपाईं testing शाखामा हुनुहुन्छ र त्यहाँ भएका सबै परिवर्तनहरूमा काम गर्न सक्नुहुन्छ।

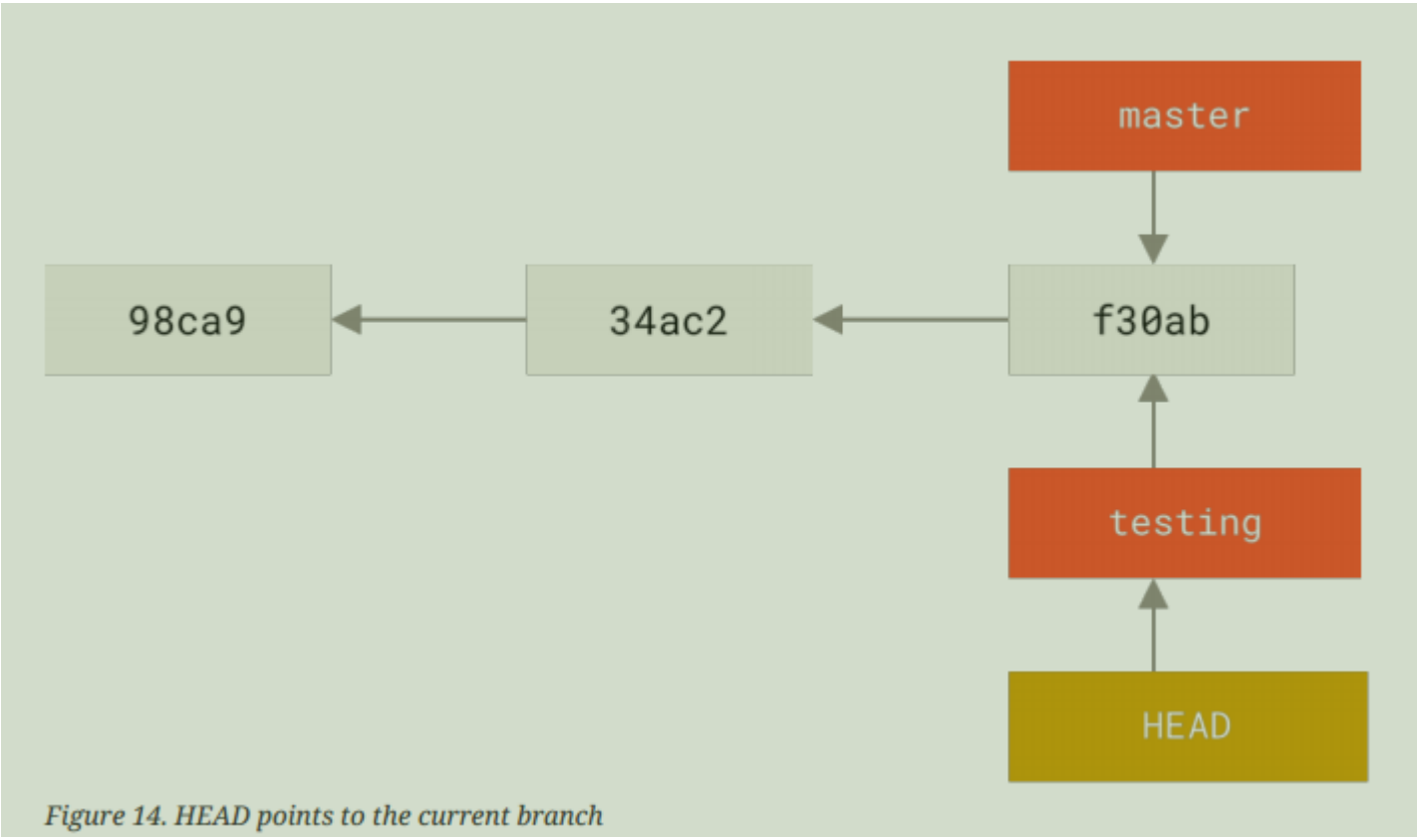

Figure 14. HEAD points to the current branch

What is the significance of that?

यो महत्वपूर्ण छ किनकि जब तपाईं शाखामा स्विच गर्नुहुन्छ, Git ले तपाईंको कार्यलाई सोही शाखामा राख्नेछ, र तपाईंले त्यस शाखामा भएका नयाँ परिवर्तनहरूलाई commit गर्न सक्नुहुन्छ। यसलाई अझ राम्रोसँग बुझ्नको लागि, हामी अर्को commit गरौं:

```
$ vim test.rb
```

यो आदेशले test.rb फाइल खोल्छ, जहाँ तपाईं परिवर्तन गर्न सक्नुहुन्छ। त्यसपछि, तपाईंले गरेको परिवर्तनलाई commit गर्न:

```
$ git commit -a -m 'Make a change'
```

यहाँ, `-a` विकल्पले `Git` लाई तपाईंका सबै ट्र्याक गरिएका फाइलहरूमा गरिएको परिवर्तनहरू `commit` गर्न भन्यो। `-m` को साथमा तपाईंले `commit` को लागि सन्देश दिनुहुन्छ, जसले बताउँछ कि तपाईंले के परिवर्तन गर्नुभयो। यसले `Git` लाई तपाईंको परिवर्तनलाई याद राख्न र ट्र्याक गर्न मद्दत गर्छ, ताकि पछि तपाईं यी परिवर्तनहरूको इतिहास देख्न सक्नुहुन्छ।

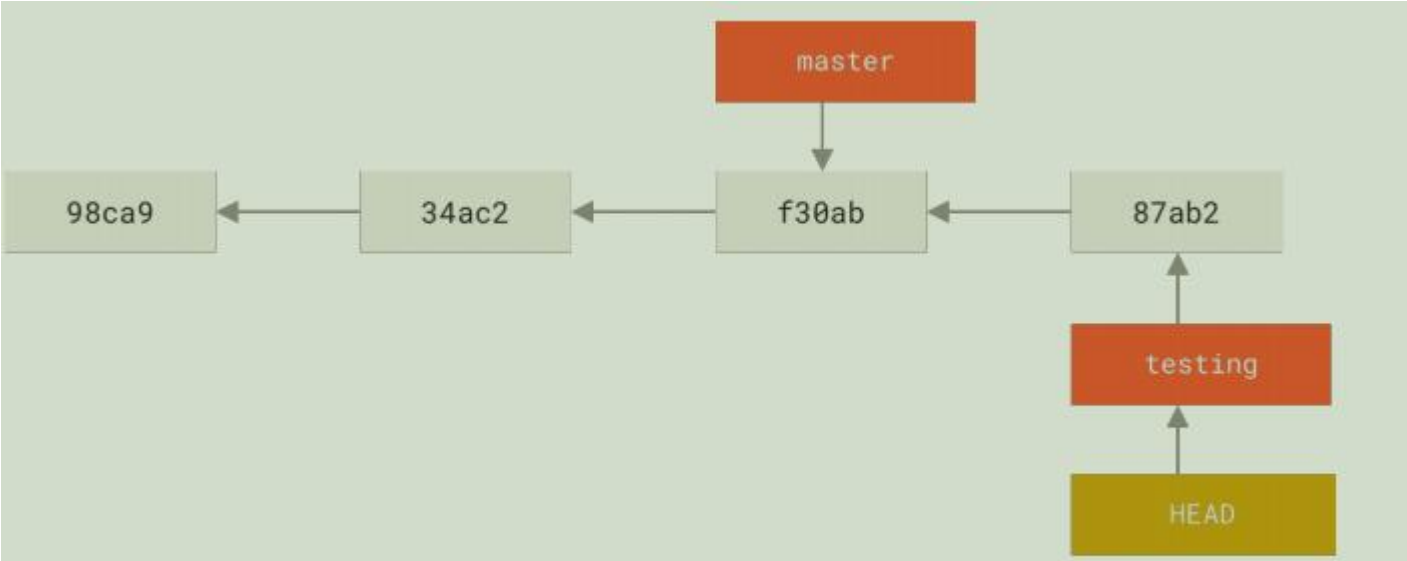

Figure 15. The HEAD branch moves forward when a commit is made

यो interesting छ, किनकि अब तपाईंको **`testing`** शाखा अगाडि बढिसकेको छ, तर तपाईंको master शाखा अझै पनि सोही **`commit`** मा रहेको छ जुन तपाईंले git **`checkout`** चलाएर शाखा स्विच गर्नु अघि थियो।
यसको अर्थ यो हो कि **`Git`** ले तपाईंको दुईवटा शाखाहरूको अवस्थालाई अलग राखेको छ। तपाईंको **`testing`** शाखामा भएको परिवर्तनहरू नयाँ **`commit`** मा थपिएका छन्, जबकि master शाखा अझै पुरानो commit मा रहेको छ। यसले तपाईंलाई विभिन्न शाखामा फरक-फरक काम गर्न अनुमति दिन्छ, बिना अन्य शाखामा भएका परिवर्तनहरूलाई असर पुर्याए।

Let's switch back to the master branch:

अब, हामी फेरि master शाखामा फर्कौं र त्यहाँ भएका परिवर्तनहरूमा काम गरौं:

```
$ git checkout master
```

यो आदेशले तपाईंलाई master शाखामा फर्काउँछ र त्यहाँ भएका सबै परिवर्तनहरूमा काम गर्न सक्नुहुन्छ। Git ले तपाईंलाई जतिसुकै शाखामा स्विच गर्न अनुमति दिन्छ, र तपाईंको सबै कामको ट्र्याक राख्छ।

यदि तपाईं अहिले git log चलाउनु भयो भने, तपाईंलाई "testing" शाखा कहाँ गएको हो भनेर अचम्म लाग्न सक्छ, किनकि यसले तपाईंको अपेक्षाअनुसार **"testing"** शाखा देखाउने छैन।

तर, चिन्ता नगर्नुहोस्! शाखा हराएको छैन; **Git** ले तपाईंलाई थाहा छैन कि तपाईं त्यस शाखामा रुचि राख्नुहुन्छ, त्यसैले यसले तपाईंलाई के देखाउनु पर्ने हो भन्ने अनुमान गर्छ। अरूसँग भन्नु पर्दा, **git log** ले डिफल्ट रूपमा केवल तपाईंले हालमा स्विच गरेको शाखामा भएका commit इतिहासलाई मात्र देखाउँछ।

यदि तपाईंको चाहेको शाखाको commit इतिहास देखाउन चाहनुहुन्छ भने, तपाईंले स्पष्ट रूपमा शाखाको नाम दिनु पर्छ, जस्तै:

```
$ git log testing
```

साथै, यदि तपाईं सबै शाखाहरूको commit इतिहास देखाउन चाहनुहुन्छ भने, तपाईंले **--all** विकल्प थप्नुपर्नेछ:

```
$ git log --all
```

यसले सबै शाखाहरूका **commit** इतिहास देखाउँछ, जसले तपाईंलाई विभिन्न शाखाहरूमा भएका परिवर्तनहरू एक साथ देख्न मद्दत गर्दछ।

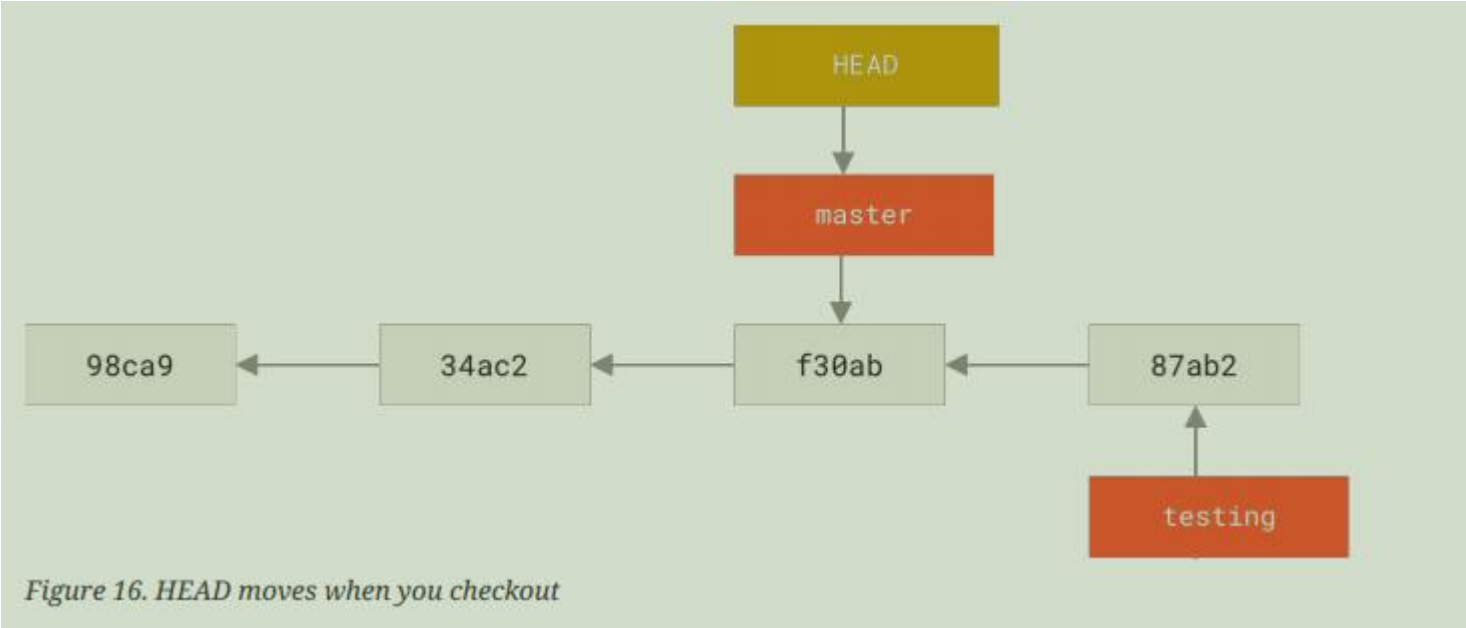

Figure 16. HEAD moves when you checkout

यो command ले दुईवटा काम गर्यो। ---

First, यसले HEAD pointer लाई फेरि master शाखालाई देखाउने गरी सार्यो।

Second, यसले तपाईंको working directory का फाइलहरूलाई master शाखाले देखाएको snapshot मा फर्काइदियो।

यसको अर्थ, अबदेखि तपाईंले गर्ने परिवर्तनहरू पुरानो version बाट फरक दिशामा जानेछन्।

Why is this important?

यसले testing शाखामा तपाईंले गरिसकेको कामलाई rewind गरेर तपाईंलाई फरक दिशामा काम गर्ने मौका दिन्छ। यो विशेषता Git लाई धेरै लचिलो बनाउँछ, किनकि तपाईं एउटै प्रोजेक्टको विभिन्न version हरूमा सजिलै काम गर्न सक्नुहुन्छ।

यो प्रक्रिया यस्तो देखिन्छ: तपाईं testing शाखाबाट master शाखामा फर्कनुभयो, त्यसपछि master बाट नयाँ दिशामा प्रोजेक्टलाई अगाडि बढाउन सक्नुहुन्छ।

Switching branches changes files in your working directory

Git मा शाखा (branch) switch गर्दा तपाईंको working directory मा भएका फाइलहरू परिवर्तन हुन्छन् भन्ने कुरा जान्नु महत्त्वपूर्ण छ।

What does this mean?

यदि तपाईं पुरानो शाखामा switch गर्नुहुन्छ भने, तपाईंको working directory का फाइलहरू त्यस शाखामा अन्तिम पटक commit गरिएको अवस्थामा फर्किन्छन्।

यसको अर्थ, त्यो शाखामा काम गर्दा भएका फाइलका परिवर्तनहरू मात्र देखिनेछन्।

तर, **if Git cannot do it cleanly,** जस्तै तपाईंको हालको फाइलहरूमा परिवर्तनहरू भए र ती पुरानो शाखाको snapshot सँग मेल खाँदैनन् भने, Git ले तपाईंलाई switch गर्न दिदैंन।

Why is this helpful?

यसले तपाईंको project को consistency जोगाउँछ र कुनै पनि data loss हुनबाट बचाउँछ। Git ले तपाईंलाई सुरक्षित रूपमा शाखाहरूमा काम गर्न मद्दत गर्छ।

Let's make a few changes and commit again: - - - - - - - - - - - - - - - -

अब केही परिवर्तन गरौं र commit गरौं:

```
$ vim test.rb
$ git commit -a -m 'Make other changes'
```

What just happened?

अब तपाईंको project को इतिहास दुई फरक दिशामा गएर diverge भएको छ।

तपाईंले पहिले एक नयाँ शाखा (branch) बनाएर केही काम गर्नुभयो, त्यसपछि मुख्य शाखा (main branch) मा फर्केर अर्को काम गर्नुभयो।

Why is this interesting?

यी दुबै परिवर्तनहरू अलग-अलग शाखाहरूमा सुरक्षित छन्।

- तपाईं चाहनुभएको बेला यी शाखाहरू बीच switch गर्न सक्नुहुन्छ।
- तपाईं तयार भएपछि यी शाखाहरूलाई merge गरेर एउटै बनाउनु पनि सक्नुहुन्छ।

Why is this amazing?

यो सबै तपाईंले branch, checkout, र commit जस्ता सरल command हरू प्रयोग गरेर गर्न सक्नुहुन्छ। यसले Git लाई परियोजनामा काम गर्ने उत्कृष्ट र प्रभावकारी उपकरण बनाउँछ।

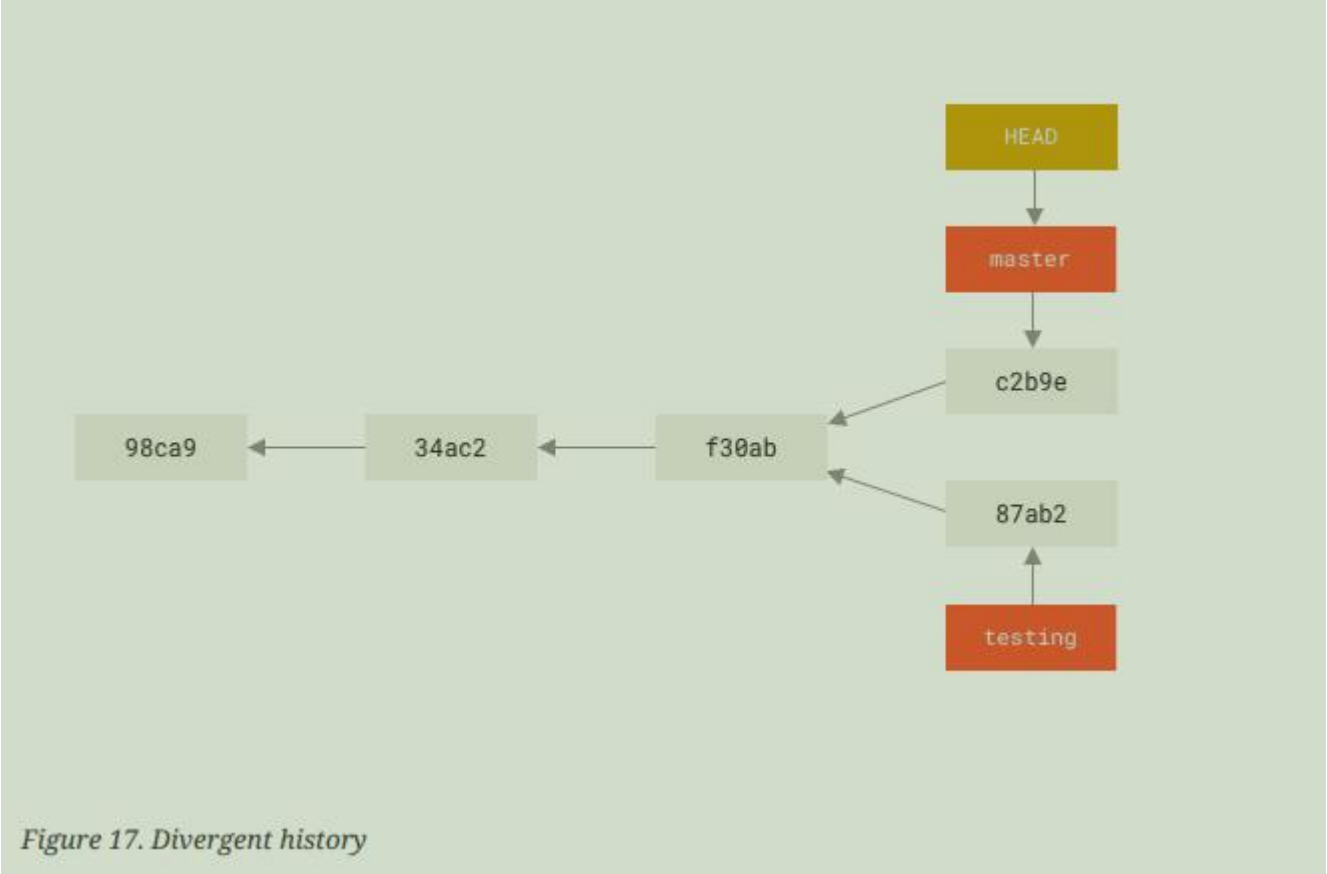

Figure 17. Divergent history

Visualizing Branch History with Git Log - - - - - - - - -

तपाईं आफ्नो commit history र शाखाहरू (branches) को स्थिति सजिलै हेर्न सक्नुहुन्छ git log command को प्रयोग गरेर।

यदि तपाईं यो command चलाउनुहुन्छ:

```
$ git log --oneline --decorate --graph --all
```

What will it show?

यसले तपाईंको commits को history देखाउँछ, शाखा pointers कहाँ छन् र तपाईंको इतिहास (history) कसरी "diverged" भएको छ भनेर।

उदाहरणका लागि:

```
* c2b9e (HEAD, master) Make other changes
| * 87ab2 (testing) Make a change
|/
* f30ab Add feature #32 - ability to add new formats to the central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 Initial commit of my project
```

How does it work?

Git मा branch वास्तवमा एउटा साधारण फाइल मात्र हो, जसमा 40-character को SHA-1 checksum हुन्छ।

SHA-1 checksum commit लाई संकेत (point) गर्छ।

Why is this useful?

Branch बनाउन र हटाउन (destroy गर्न) Git मा धेरै सस्तो छ।

नयाँ branch बनाउनको लागि, केवल 41 bytes (40 characters र एक newline) फाइलमा लेख्नुपर्छ।

Why should you know this?

यो जान्दा तपाईंलाई Git को efficiency र branching mechanism कति fast र effective छ भन्ने कुरा बुझ्न सजिलो हुन्छ।

Git vs. Older VCS Tools: Why Git Branching is Better - - - - - - - - - - - -

Git branching प्रणाली पुरानो VCS (Version Control Systems) उपकरणहरूको branching प्रणालीभन्दा धेरै efficient र तेज छ। यहाँ केही महत्वपूर्ण अन्तरहरू छन्:

Older VCS Tools

पुरानो VCS उपकरणहरूमा शाखा बनाउँदा, projects का सबै files लाई अर्को directory मा **copy** गर्नुपर्छ।

- **Time-Consuming:** यो प्रक्रिया केही seconds देखि minutes लाग्न सक्छ, विशेष गरी ठूलो project को लागि।
- **Storage-Intensive:** Project files को duplicate ले धेरै ठाउँ ओगट्छ।

Git

Git मा branching पूर्ण रूपमा फरक छ।

- **Instantaneous Process:** Branch बनाउने प्रक्रिया तुरुन्त हुन्छ। कुनै पनि size को project मा branching मा time लाग्दैन।
- **Efficient Storage:** Git ले केवल pointer बनाएर शाखा (branch) को जानकारी राख्छ, जसले memory बचत गर्छ।

Advantages of Git's Branching System

1. **Fast Merging:** Commit गर्दा Git ले parents को जानकारी राख्ने भएकोले proper merge base पत्ता लगाउन सजिलो हुन्छ।
2. **Encourages Branching:** Branching तेज र सजिलो भएकोले developers ले धेरै branches बनाउने र उपयोग गर्ने प्रेरणा पाउँछन्।
3. **Better Collaboration:** Teams ले आफ्नै branches मा काम गरेर अर्को शाखा सँग merging गर्दा conflict समाधान गर्न सजिलो हुन्छ।

Why Should You Care?

Git को branching प्रणालीले productivity बढाउँछ, development workflow लाई streamline गर्छ, र collaboration लाई अझै सहज बनाउँछ।

यसले तपाईंको students लाई पुराना systems र modern systems बीचको भिन्नता बुझ्न र Git branching प्रणालीको फाइदालाई appreciate गर्न सहयोग गर्नेछ।

Why Should You Create and Switch Branches Together? - - -

Development workflow मा branch बनाउने र तुरुन्तै switch गर्ने आदत राख्नु धेरै फाइदा जनक हुन्छ। किनकि यो process तपाईंको कामलाई अझै streamline गर्न सहयोग गर्छ। अब हेर्नु, Git ले कसरी यो कामलाई सजिलो बनाउँछ।

Creating and Switching in a Single Step

सामान्यतः, नयाँ branch बनाएपछि त्यसमा switch गर्नु पर्ने हुन्छ। यो process Git मा एकै पटक निम्न command प्रयोग गरेर गर्न सकिन्छ:

```
$ git checkout -b <newbranchname>
```

उदाहरणका लागि, feature-branch बनाउन र switch गर्न:

```
$ git checkout -b feature-branch
```

Using git switch (Git Version 2.23 and Later)

Git को newer versions (2.23+) मा, git switch ले branching process लाई अझै user-friendly बनाएको छ।

Switch to an Existing Branch

पहिले नै बनेको branch मा जान:

```
$ git switch testing-branch
```

Create and Switch to a New Branch

नयाँ branch बनाउन र switch गर्न:

```
$ git switch -c new-branch
```

`-c` भनेको "`create`" हो। तपाईं चाहनुहुन्छ भने full flag `--create` पनि प्रयोग गर्न सक्नुहुन्छ।

Return to the Previous Branch

पछिल्लो पटक checkout गरिएको branch मा फर्कन:

```
$ git switch -
```

यो तपाईंलाई तुरुन्तै पछिल्लो branch मा switch गराउँछ।

---------- END ----------