# Aplicaciones Telemáticas

## Tema 0.2. Introducción a Git y GitHub

J. E. López Patiño, F. J. Martínez Zaldívar

DEPARTAMENTO DE COMUNICACIONES

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

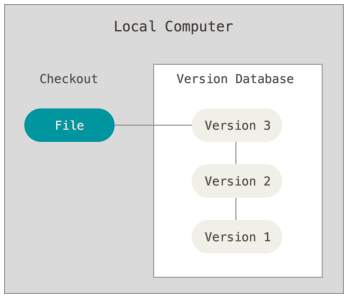
# Índice

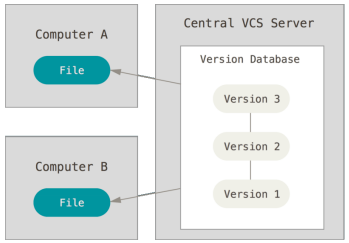
# Índice

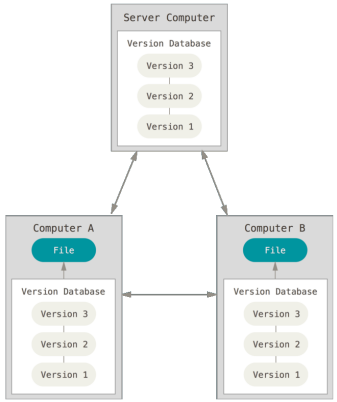
# Índice

# Contexto

- VCS: Version Control System
- Modelos: local, centralizado y distribuido



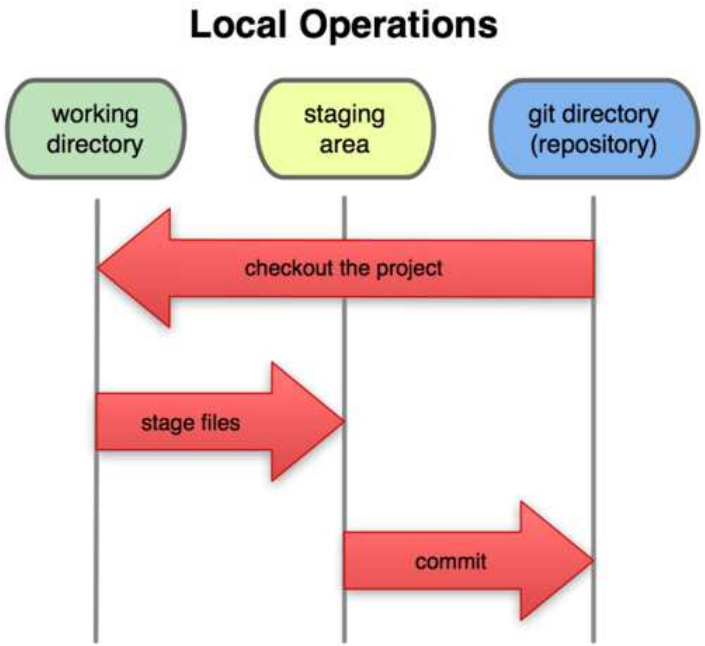

- Seguimiento de cambios en código
- Sincroniza codigo entre distintas personas
- Se puede volver a versiones antiguas
- Permite ramificaciones del código
- Git: modelo distribuido y replicado
- Referencias Git:
  - `https://git-scm.com/doc`
  - `http://marklodato.github.io/visual-git-guide/index-en.html`
  - `http://www.vogella.com/tutorials/Git/article.html`
  - ...

# Instalación

- Descarga desde URL: `https://git-scm.com/`
- Documentación: `https://git-scm.com/`
- En clase: uso de consola (en vez de GUI —Graphical User Interface—)
- Configuración entorno Git:
  - Global: una vez y para todos los respositorios
  - Individual: para cada repositorio (sobreescribe global)
- Creación/uso de repositorios

# Repositorios: locales y remotos

- Repositorio: almacenamiento de todo el historial de documentos
- Repositorio *local*:
  - *Directorio de trabajo: Working Directory (WD)*
  - *Almacenamiento intermedio: staging/index area*
  - *Repositorio: en directorio* `.git`




Una posibilidad de interacción entre áreas

- Repositorio *remoto*. Posibilidades:
  - Sin directorio de trabajo (WD) *(bare)*:
  - No tiene *index/staging area*
  - Repositorio está en un directorio (remoto)
  - Acceso de lectura y/o escritura (control) a repositorio por parte de usuarios autorizados
  - Se puede transferir información entre repositorios locales y remotos

# *Hash* de un fichero

- *Hash*: función resumen
- SHA-1: cualquier secuencia de bits la *resume* en:
  - 160 bits (40 dígitos hexadecimales)
- Comparación de ficheros: comparación de sus *hashes*
- Formación: `SHA1("blob SIZE\0CONTENT")`
  Como se podrá comprobar posteriormente `git` permite obtener el *hash* de un fichero como

```
$ git hash-object fichero
```

**IMPORTANTE**: el carácter $ denota el *prompt* del sistema, es decir, no debe introducirse como parte del comando `git`.

# Localmente: estados de un fichero y notación

- Modelo:
  - Pueden existir hasta tres versiones distintas de un fichero con seguimiento ubicadas respectivamente en WD, index y repositorio
- Estados de un fichero:
  - *Sin seguimiento VCS*: *untracked* (Notación: '?')
  - *Con seguimiento VCS y Nuevo*: primera vez que se incluye en el *index* (Notación: 'A')
  - *Con seguimiento VCS y Modified*: modificado respecto a cierta referencia (Notación: 'M')
    - Un fichero en el WD se considera modificado cuando lo está respecto a su versión en el *index*
    - Un fichero en el *index* se considera modificado cuando lo está respecto a su versión en el repositorio
  - *Con seguimiento VCS y Actualizado*: respecto a cierta referencia (Notación: ' ' )
    - Un fichero en el WD se considera actualizado cuando coincide con la versión del *index*
    - Un fichero en el *index* se consdiera actualizado cuando coincide con la versión del repositorio
- Ejemplos:

| *index* | WD | Significado |
|---|---|---|
| '?' | '?' | Sin seguimiento |
| 'A' | ' ' | Añadido por primera vez al *index*, la versión en el WD *actualizada* en el *index* |
| 'A' | 'M' | Añadido por primera vez al *index*, la versión en el WD *modificada* respecto al *index* |
| ' ' | 'M' | Versión del *index actualizada* en el repositorio; versión en WD *modificada* en el *index* |
| ' ' | ' ' | Versión del *index actualizada* en el repositorio; versión en WD *actualizada* en el *index* |
| 'M' | 'M' | Versión del *index modificada* en el repositorio; versión en WD *modificada* en el *index* |

# Índice

# Índice

# Configuración inicial de Git

- Comando en **consola de comandos**:

  `git <subcomando> <parametros>`
- Existen opciones gráficas: GUI
- Ayuda

```
// Muestra comandos usuales
$ git help

// Para todos los comandos y subcomandos:
$ git help -a

// Ayuda para comando concreto
$ git help <comando>

// Alternativamente
$ git <comando> --help

// O bien
$ git <comando> -h
```

- Configuración inicial de Git:

```
// Indicación de usuario y email
$ git config --global user.name "Paco"
$ git config --global user.email "fjmartin@dcom.upv.es"
$ git config --global core.editor "'C:\Program Files...'"
// Sin opción --global: ajustes solo validos para ese repositorio (requiere git init previo)
// Con opción --global: solo necesaria la primera vez (no requiere git init previo)

// Listado de configuración
$ git config --list
```

# Creación y eliminación de un repositorio *local*

- Por creación explícita: lo creamos *nosotros*

```
// Creación de directorio de trabajo (WD)
$ mkdir directorio

// Cambio a WD
$ cd directorio

// Inicialización de repositorio (local): creación de directorio .git
$ git init
// Repositorio en directorio oculto .git



// Alternativa: inicialización de repositorio (típicamente remoto, sin WD)
$ git init --bare
// Repositorio en el propio directorio.
// Intención: repositorio "remoto" compartido por varios usuarios
```

- Por clonación: ya existe *remotamente* y vamos a *copiarlo* —clonarlo— localmente:

```
$ git clone URL_REPOSITORIO_REMOTO  [ DIRECTORIO_LOCAL_(WD) ]
```

**Notación**: normalmente, una expresión entre `[corchetes]` denotará algo opcional

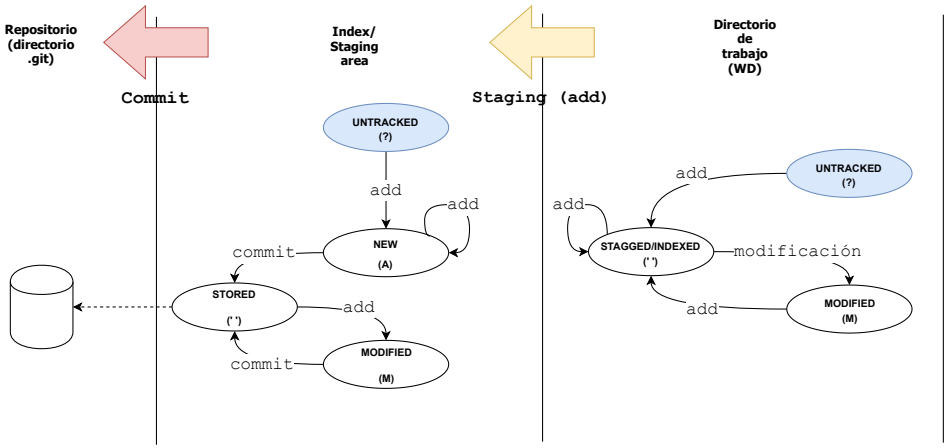
- Si se borra el directorio `.git` se **elimina todo rastro** de `git`.
  - Desaparecería el repositorio
  - Desaparecería el *index*
  - Permanecería el WD

# Índice

# Envío al index: add y status

```
//// Ficheros a index o staging area:

// Envío de un fichero al index
$ git add fichero

// Envío de todos los ficheros modificados o nuevos,
// pero no los borrados
$ git add .

// Envío de todos los ficheros modificados o
// borrados, pero no los nuevos
$ git add -u

// Envío de ficheros modificados, nuevos y borrados
$ git add -A



//// Visualización del  estado del index y del WD

// Estado
$ git status

// Estado abreviado: XY
//  X: estado fichero en staged, Y estado en WD:
// 'A': añadido,   'M': modif.,   ' ': sin modif.,
// '?': untracked, 'D': borrado
$ git status -s
```

## Estados de un fichero:

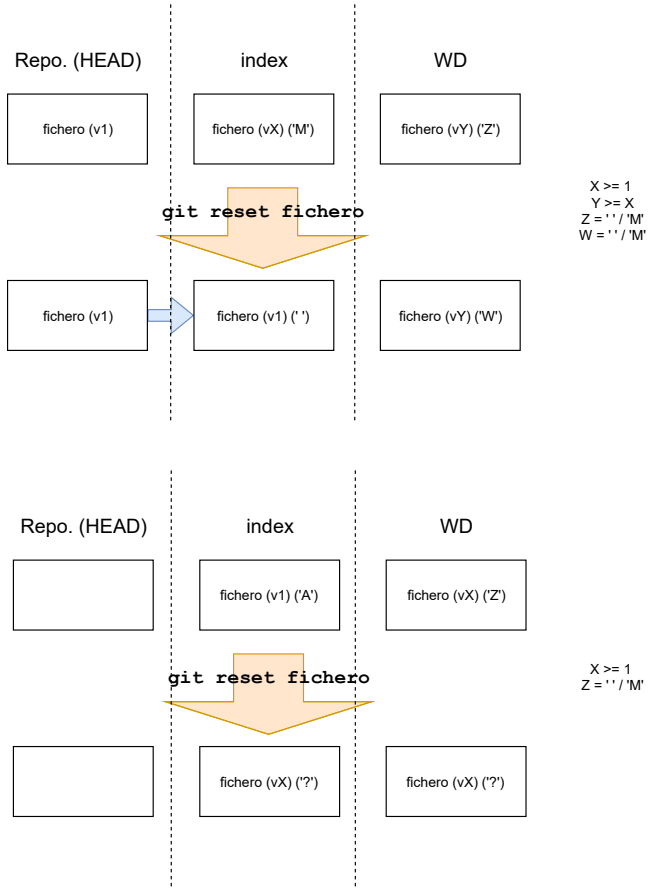


# Diferencias entre la versión del WD y del `index`: `diff`

```
// Si no se especifica <fichero> entonces se muestran las
// diferencias entre todos los ficheros del WD y del index

// Muestra diferencias entre dir. trabajo y staging:
$ git diff [<fichero>]
diff --git a/<fichero> b/<fichero>
index ca3f56e..e64b280 100644  //// Hash de los ficheros y permisos
--- a/<fichero>
+++ b/<fichero>
@@ -1,4 +1,4 @@
-Esta línea está solo en el primer fichero
+Esta línea está solo en el segundo fichero
 Esta línea está en ambos
 Y esta, también



// Lista los ficheros en el index (y su hash)
$ git ls-files -s

// Obtiene el hash de un fichero concreto del WD. Comparación de hashes para verificar variación
$ git hash-object fichero
```

# Control del index: reset

```
// Si el fichero ya existe en el repositorio, se recarga
// en el index la versión apuntada por HEAD del repositorio
// La versión en el WD queda inalterada -será (' ') o ('M')
// en función de la versión recargada en el index-

// Si el fichero no existe en el repositorio, aparece como
// sin seguimiento

$ git reset fichero
```



# Cesar seguimiento: `rm`

```
// A partir de ahora, deja sin seguimiento (untracked) fichero.
// Se requiere opcion -f si la versión WD no está actualizada
// en el index, a no ser que el index esté actualizado con
// el repositorio.
// El fichero no es borrado del WD

$ git rm --cached fichero
```

| index | WD |
|---|---|
| 'M' | ' ' |
| `git rm --cached fichero` | |
| 'D' '?' | '?' |

| index | WD |
|---|---|
| 'M' | 'M' |
| `git rm -f --cached fichero` | |
| 'D' '?' | '?' |

| index | WD |
|---|---|
| 'A' | ' ' |
| `git rm --cached fichero` | |
| '?' | '?' |

| index | WD |
|---|---|
| 'A' | 'M' |
| `git rm -f --cached fichero` | |
| '?' | '?' |

| index | WD |
|---|---|
| ' ' | ' ' |
| `git rm --cached fichero` | |
| 'D' '?' | '?' |

| index | WD |
|---|---|
| ' ' | 'M' |
| `git rm --cached fichero` | |
| 'D' '?' | '?' |

# Fichero .gitignore

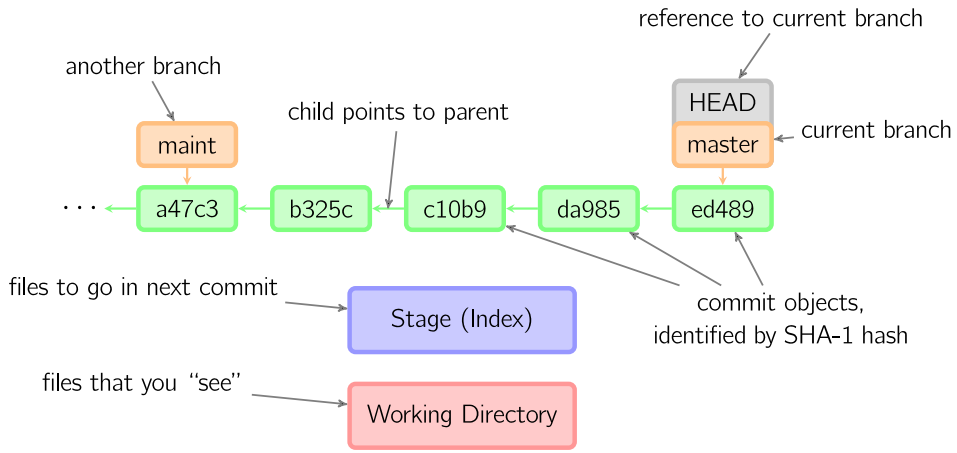
```
//// Fichero .gitignore
// Contiene "patrones" de nombres de
// ficheros a ignorar.
// En distintos subdirectorios podemos tener distintos .gitignore


// Ejemplo de fichero .gitignore
// En GitBash / Linux / Mac OS-X ...
$ cat .gitignore
// En MS-DOS
> type .gitignore

*.tmp
/directorio
fichero.html
dirx/aeiou*.tmp

// No se incluirán
// *.tmp: ficheros con sufijo .tmp
// /directorio: la carpeta directorio (aquí / hace referencia a ubicación de .gitignore)
// fichero.html: cualquier fichero ubicado en cualquier carpeta denominado fichero.html
// dirx/aeiou*.tmp: cualquier fichero con nombre aeiou-cualquier caracter o caracteres-.tmp que
//                  esté en una carpeta de nombre dirx


//// Modificación de .gitignore: borrón y cuenta nueva
$ git rm -r --cached .                    // se dejan de seguir todos los ficheros (desde directorio actual "."
                                          // y recursivamente en todos los subdirectorios "-r")
$ git add .                               // se vuelven a seguir a continuación todos los filtrados por .gitignore
$ git commit -m "modificacion .gitignore" // actualización en repositorio
```

# Concepto de *commit*

- Un *commit* representa un envío al repositorio de ficheros del index con seguimiento y con alguna modificación pendiente
- Un *commit* es identificable mediante el *hash* o función resumen SHA-1 de
  - Toda la estructura de directorios y ficheros
- Si cambia cualquier bit de cualquier fichero/directorio, cambia su *hash*
- El *hash* de un commit puede identificar al *commit*:
  - Con los 4, 5 o 6 primeros dígitos hexadecimales: suficiente

# Concepto de *rama*

- **Rama**: secuencias de *commits*. Tienen nombre, ejemplo: **master**
  - Sentido de flechas: un commit apunta a su padre (puede ser confuso)
- Fusión o *merge*: fusión de ramas
- Puntero **HEAD**: puntero a rama donde enlazar siguiente *commit* (puntero a rama actual). Posicionamiento relativo:
  - **HEAD^**: el padre de **HEAD**
  - **HEAD~n**: n padres anteriores a **HEAD**
  - **HEAD~** = **HEAD~1** = **HEAD^**
- Flechas entre commits: puntero a **anterior** *commit*

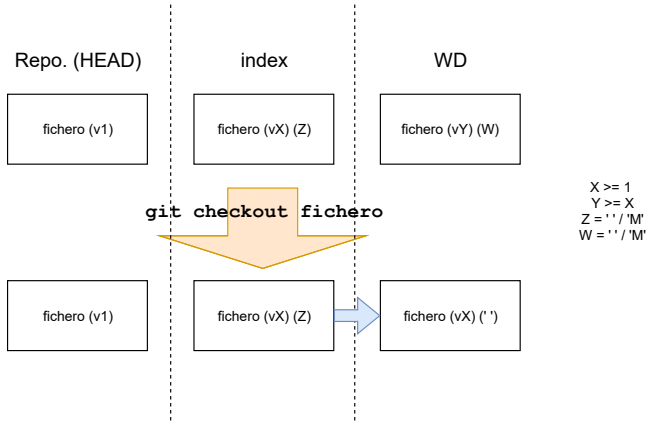


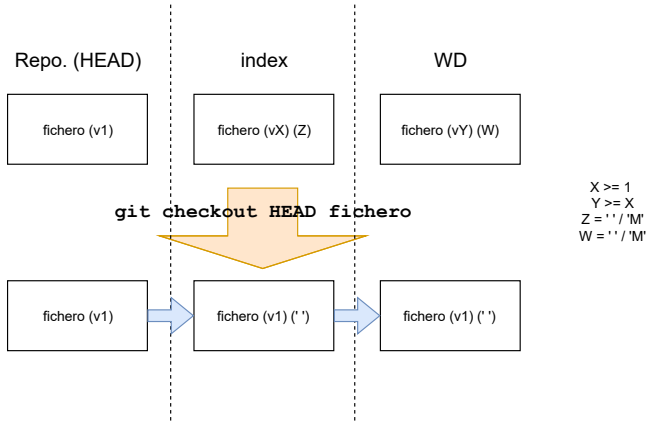
# Del `index` al repositorio: `commit` y sus *log*

```
// Envío al repositorio
$ git commit -m "Mensaje"

// Nuevo estado
$ git status

// Si no se especifica -m "Mensaje": se abre ventana de edición
// para introducir mensaje. El editor que se abre es el configurado con
$ git config core.editor  "'c:\Archivos de programa\...'"
// Si no se especifica editor: por defecto vim
// Para eliminarlo:
$ git config  --unset core.editor

// Con apertura automática de editor (sin parámetro -m "Mensaje...")
$ git commit

// Informacion de los commits
$ git log
// Mostrando graficamente (--graph) todas las ramas (--all) con
// detalles (--decorate), con una sola línea de info. (--oneline)
$ git log --all --decorate=full --graph --oneline


// Staging y commit consecutivos:
$ git commit -a

// Con mensaje en linea
$ git commit -a -m "Mensaje"

// Alternativamente
$ git commit -am "Mensaje"

// Anular un commit (¡cuidado!)
$ git commit -m "Mensaje"
// Se olvido un fichero!!!!
$ git add fichero_olvidado
// Rectificación o anulación de (sólo) el último commit
$ git commit --amend

// Lista la instantánea del repositorio apuntada por HEAD
$ git ls-tree -r HEAD
```

# Control del index y del WD: checkout

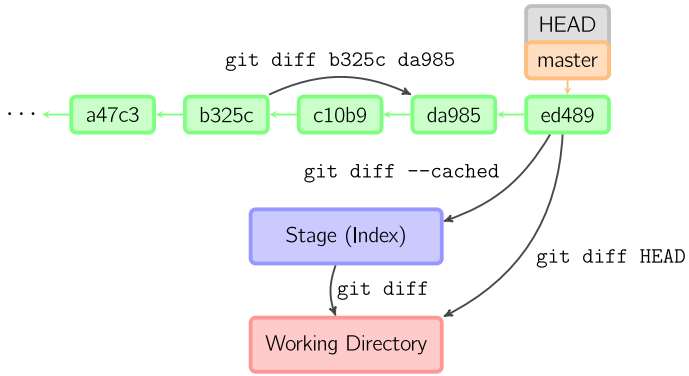
```
// Recarga el fichero del WD con la version del index
$ git checkout fichero
// o bien
$ git checkout -- fichero
// -- es para que no haya ambigüeda con la notación.


// Recarga el fichero del WD y del index con la versión
// guardada en el repositorio
$ git checkout HEAD fichero
```



# Diferencias respecto a repositorio: de nuevo `diff`

```
// Muestra diferencias entre index y el ultimo commit
$ git diff --cached [<fichero>]

// Muestra diferencias entre último commit y WD
$ git diff HEAD [<fichero>]


// Comparación entre el fichero <fichero> del dir. de trabajo
// y el del último commit
$ git diff HEAD -- [<fichero>]


// Comparación entre el último y penúltimo commit de <fichero>
$ git diff HEAD~ HEAD [<fichero>]


// Comparación entre los commits b325c... y da985...
// de fichero
$ git diff b325c da985 [<fichero>]
```

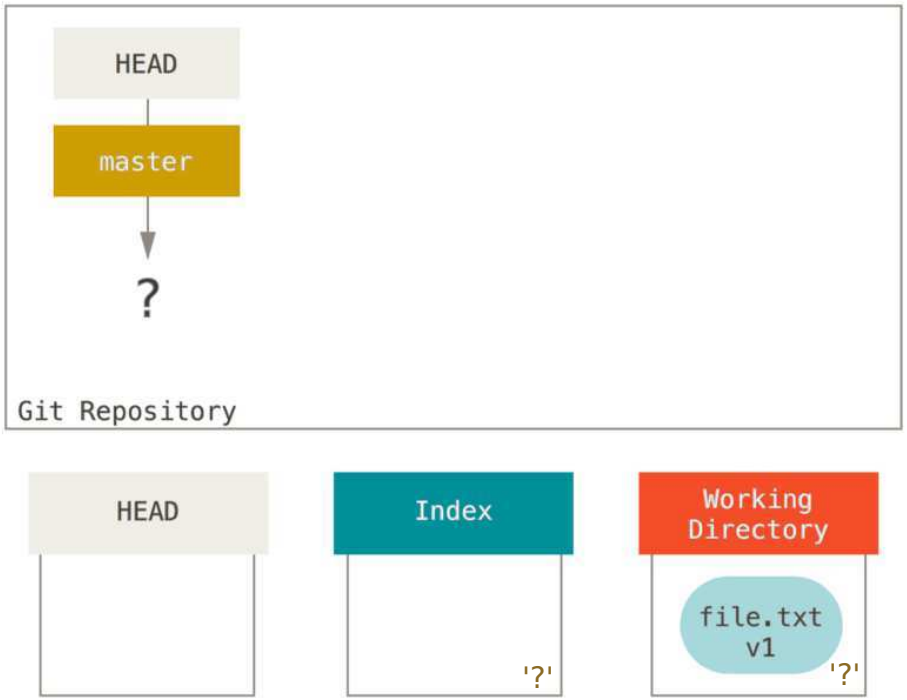


# Borrar/mover ficheros

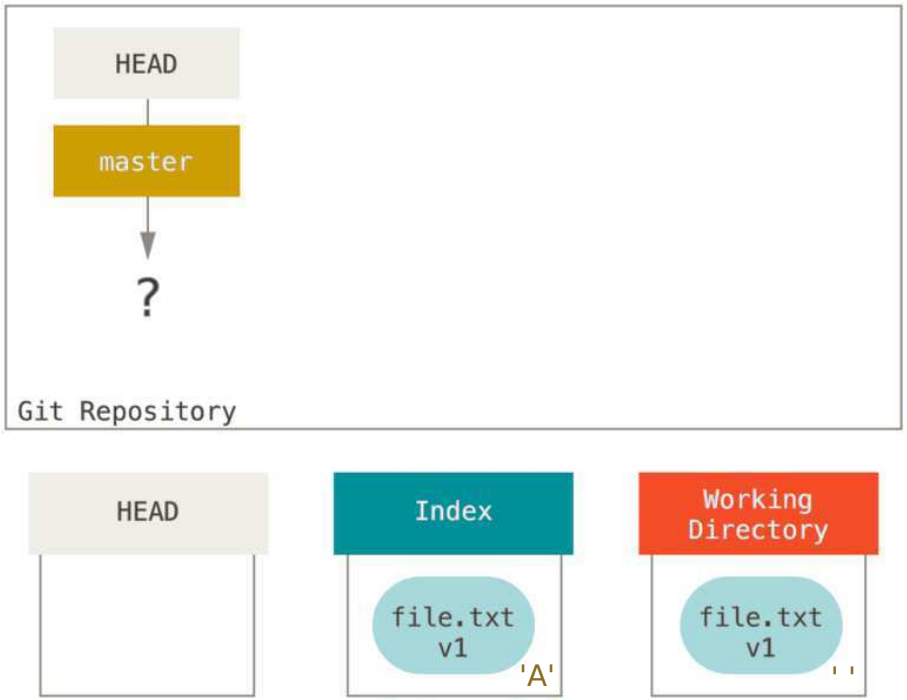
```
// Borra fichero del staging y del dir. de trabajo

// Primera forma:
$ rm fichero
$ git add fichero
$ git commit -m "..."

// Más compacto y seguro:
$ git rm fichero
$ git commit -m "..."

// Borra fichero del staging pero NO de dir. de trabajo. Lo deja untracked
$ git rm --cached fichero
$ git commit -m "..."

// Mover (renombrar) ficheros
$ git mv fichero_original fichero_final
// Equivalente a
$ mv fichero_original fichero_final
$ git rm fichero_original
$ git add fichero_final
```

# Índice

# Estadio inicial

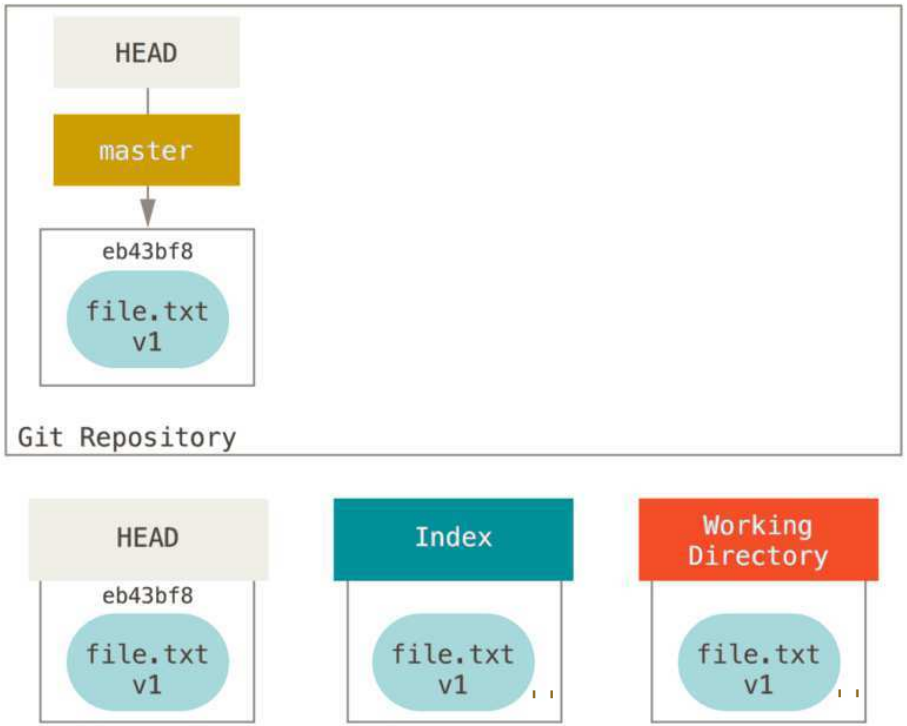


# Adición al *index*



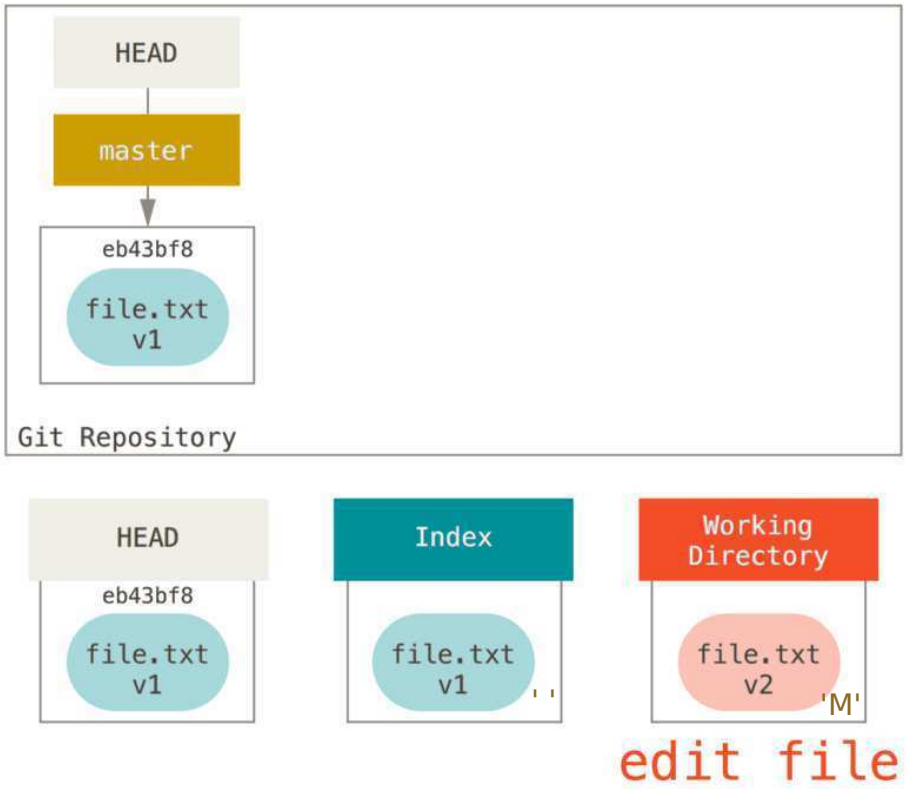

git add file.txt

# Del *index* al repositorio

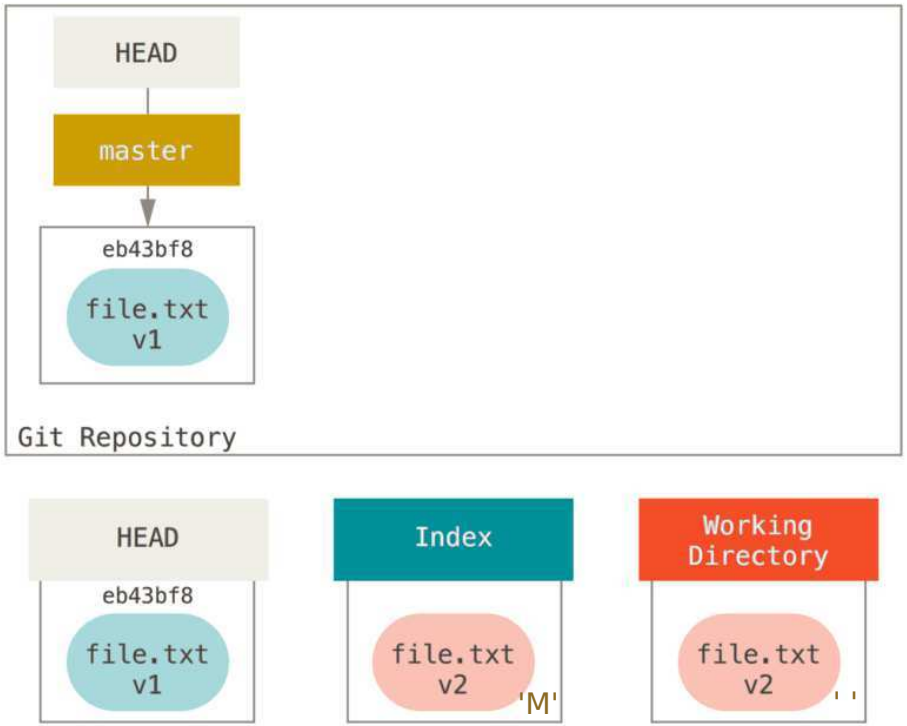



git commit -m "..."

# Modificación de fichero en espacio de trabajo

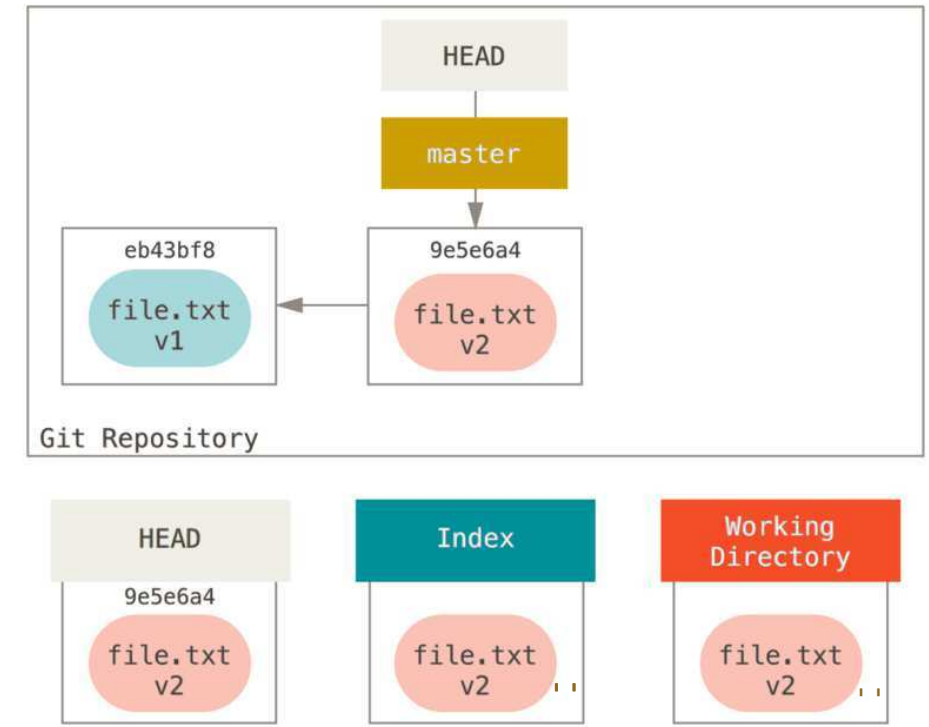


# Inclusión en el *index* de la actualización



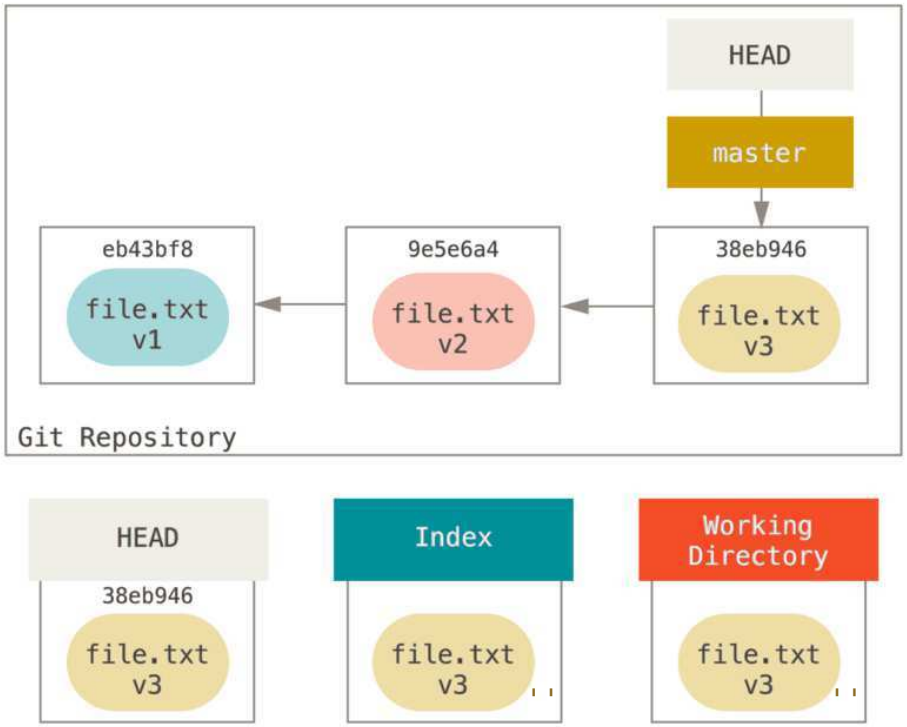

git add file.txt

# Actualización en el repositorio desde el *index*



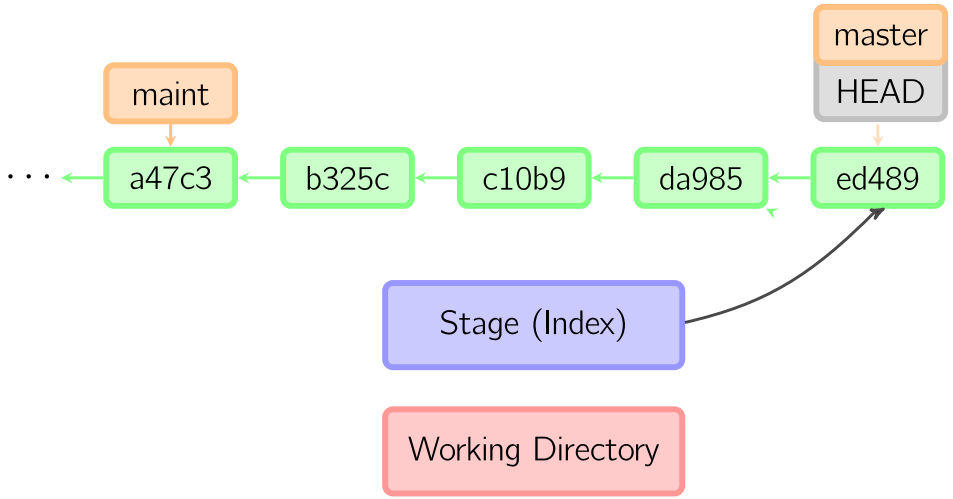

git commit -m "..."

# Actualización dir. de trabajo y directo al *index* y repositorio



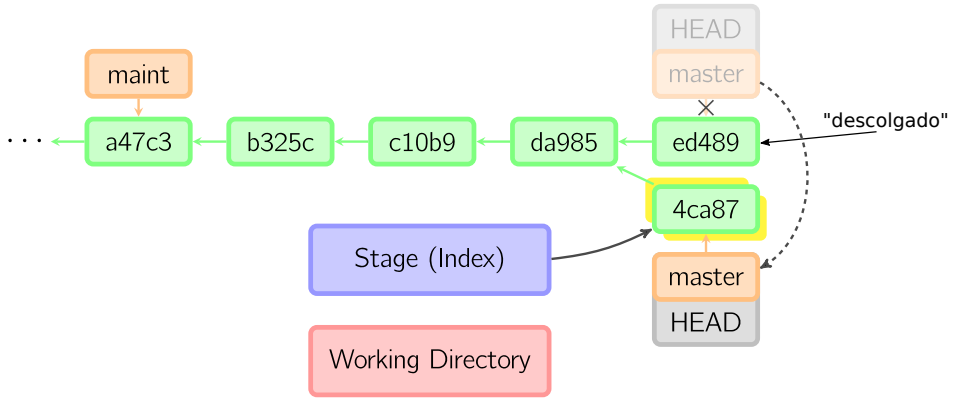

git commit -am "..."

# Rectificar último `commit`




---

`git commit --amend`



# Índice

# Algunos simuladores de Git

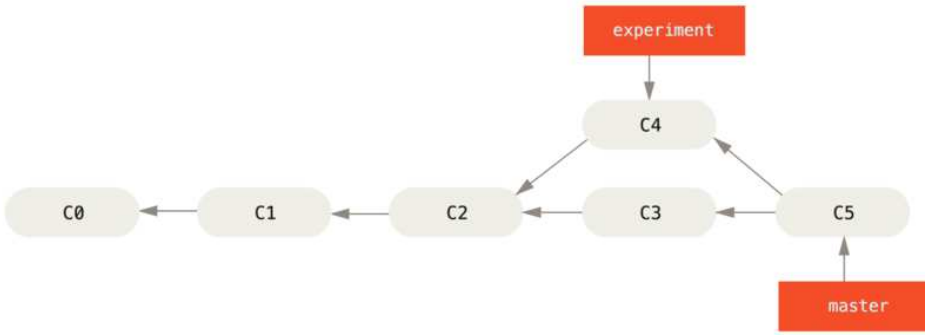
- `http://git-school.github.io/visualizing-git/`
- `https://learngitbranching.js.org/`
- No es propiamente un simulador sino un resumen gráfico interactivo sobre comandos git: `http://ndpsoftware.com/git-cheatsheet.html#loc=local_repo;`

# Índice

# Índice

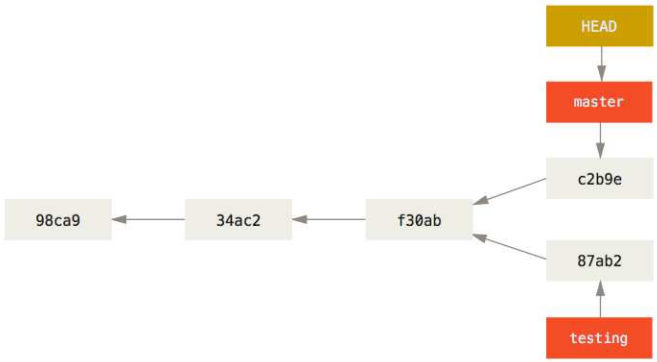
# Concepto de rama

- Secuencia de commits que marca una trayectoria (en cronología inversa) desde el último commit de dicha rama hasta el primero del repositorio
- En un repositorio puede existir más de una rama
- Todas ellas tendrán al menos un *ancestro común*

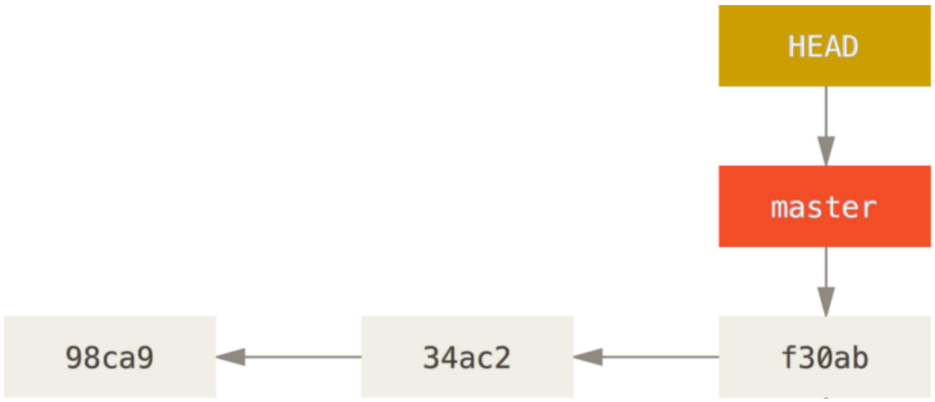


# Punteros de rama y cabecera

- Punteros:
  - Puntero de rama: apunta al último *commit* de la rama (identifica a la rama, `master`, `testing`, —figura ejemplo—)
  - Puntero de cabecera (`HEAD`): siempre apunta al padre del siguiente *commit*
  - Tras un *commit* se actualizan tanto puntero de rama como `HEAD`
  - Cambio de rama: implica cambio de puntero de rama y (depende) de puntero de cabecera `HEAD`.
- Habitualmente el puntero de rama **actual** y HEAD coinciden
  - Cambio de rama ⇒ cambio de puntero de rama ⇒ cambio de puntero HEAD (habitualmente)
  - En caso contrario: situación de *detached HEAD* (¡cuidado!)
  - Veremos en qué circunstancias puede ocurrir

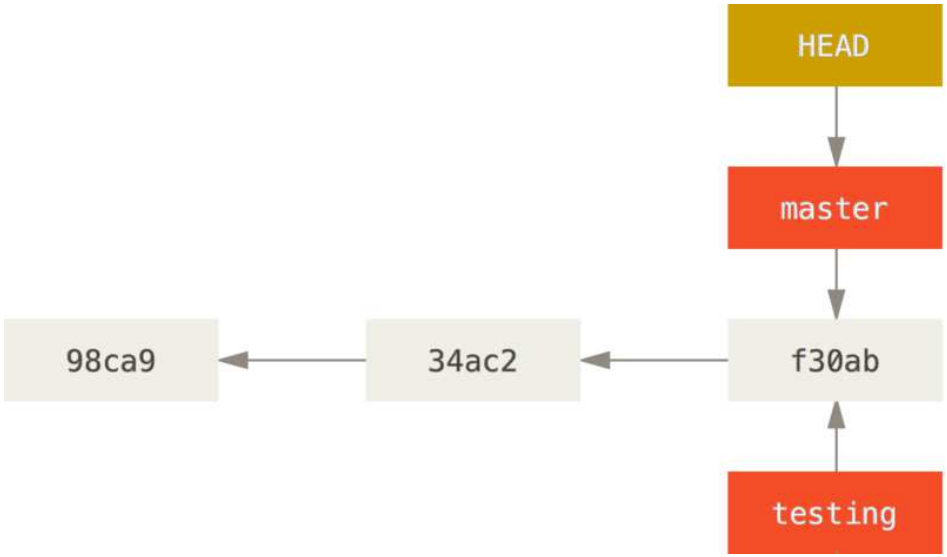


# Situación de partida



# Creación de nueva rama

```
// Creación de nueva rama
$ git branch testing
```



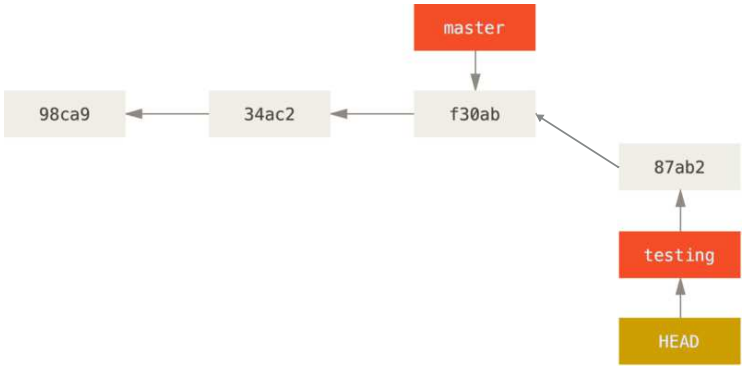
# Conmutación a nueva rama

```
// Conmutación a nueva rama
$ git checkout testing
```

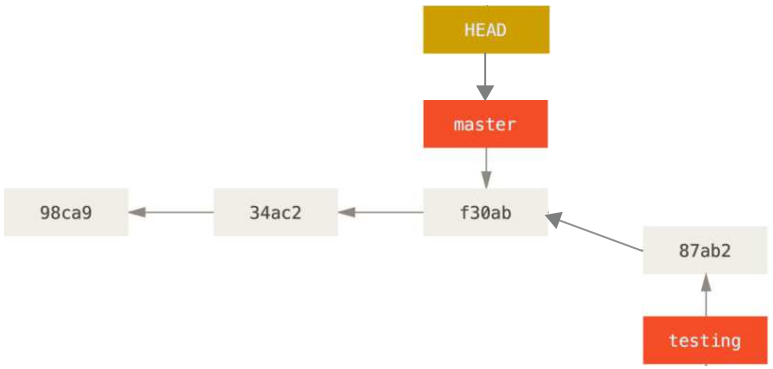

```
// Los dos comandos siguientes:
$ git branch testing
$ git checkout testing

// son equivalentes a:
$ git checkout -b testing
```

# commit desde la nueva rama

```
// Nuevo/modificación de ficheros
$ ...
// Commit desde la nueva rama
$ git commit -am "Mensaje"
```

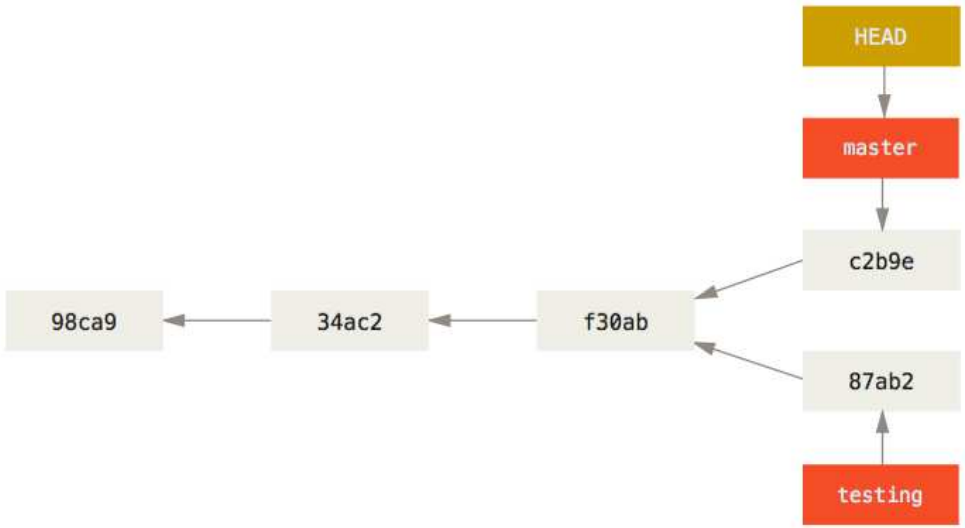



```
// Conmutación a rama master
$ git checkout master
```



# Nuevo commit desde la rama master

```
// commit desde master
$ git add .
$ git commit -m "mensaje"
// O bien
$ git commit -am "mensaje"

// log:
// git log --oneline --decorate --graph --all
```



# Índice

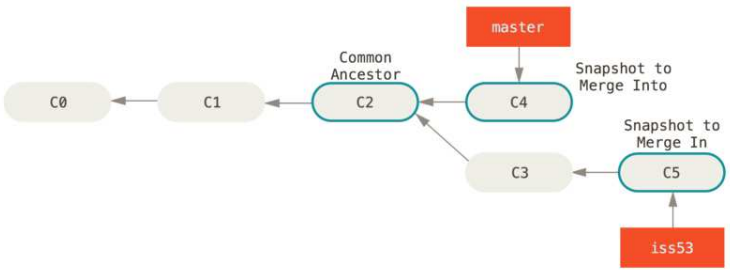
# Generación de un *merge commit*

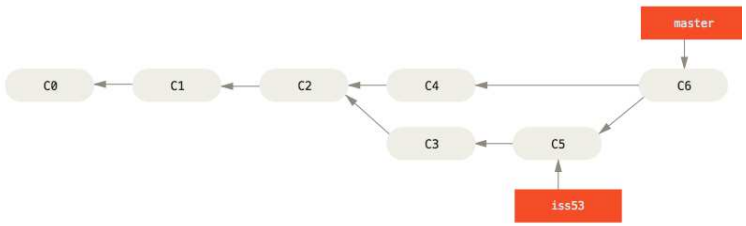
- Creación de un *merge commit*

```
// A rama master
$ git checkout master

// Muestra ramas: * rama actual
$ git branch

// Fusión
$ git merge otra_rama    // iss53 en el ejemplo gráfico

// Resolución de conflictos: manual
// Edicion de fichero(s) conflictivos
<<<<<<< HEAD
...
=======
...
>>>>>>> rama

// Resolución de conflicto con herramienta (mergetool)
$ git mergetool

// Seguir indicaciones
$ git status
```

```
// A rama master
$ git checkout master

// Muestra ramas: * rama actual
$ git branch

// Opcion escogida a priori (solo) en los conflictos:

// La rama actual (nuestra: ours)
$ git merge -Xours otra_rama

// La otra rama (de ellos: theirs)
$ git merge -Xtheirs otra_rama

// Seguir indicaciones
$ git status
```

```
// Notación:
//  master^1: padre de máster por la rama master
//  master^2: padre de máster por la otra rama
//            fusionada (iss53)
```

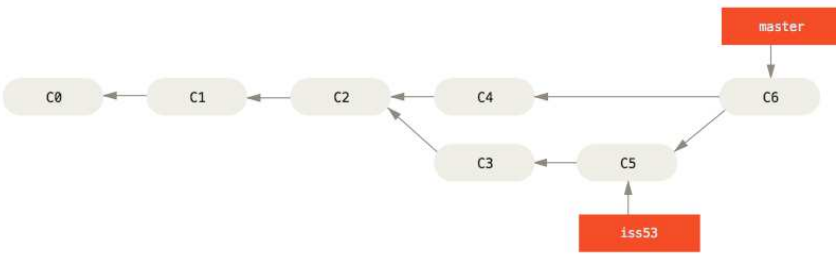


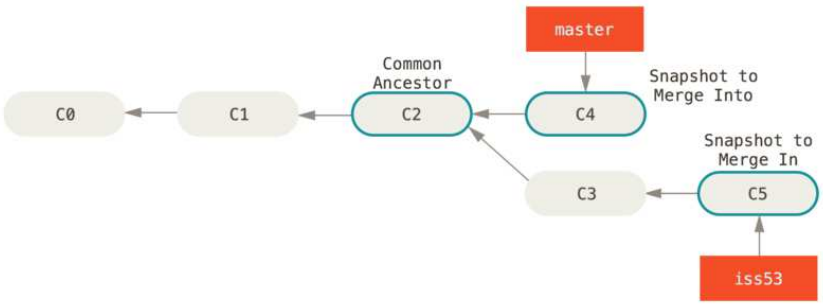



# Anulación de un *merge commit*

- Anulación del *merge commit*

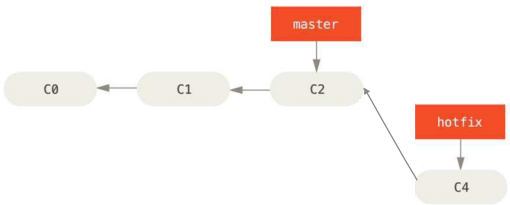
```
// Aborta la fusión: ¡¡¡ Sólo si no ha finalizado completamente !!!
$ git merge --abort

// Si ya se ha fusionado:
$ git reset --hard HEAD^
```

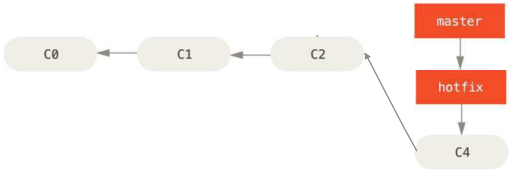


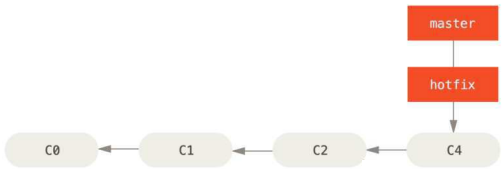



# Fusión *fast-forward*

- Fusión *fast-forward*: puntero de la rama de referencia es ancestro de la otra.

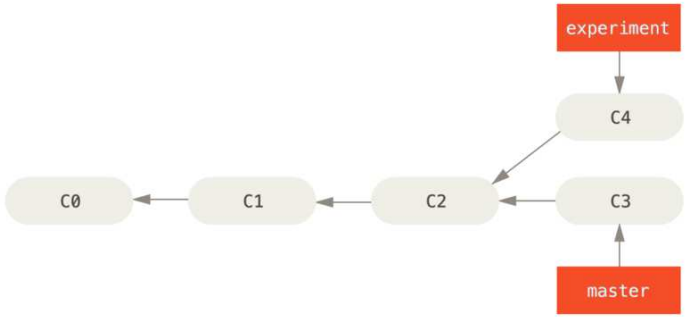


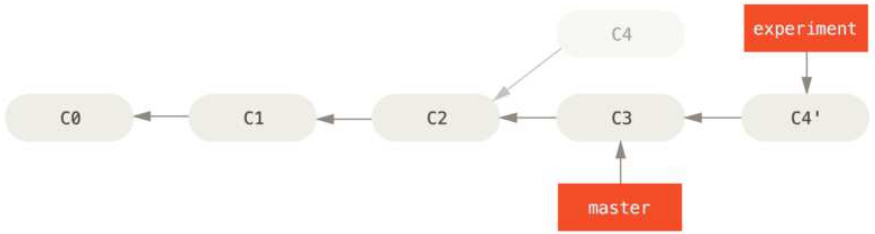
# Rebase

- Intención: *linealizar una fusión de dos ramas*
- Punto de partida:



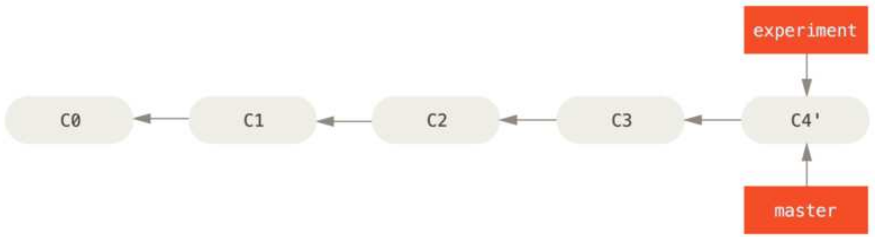

- Rebase:

```
$ git checkout experiment
$ git rebase master
```




- *Fast-forward*:

```
$ git checkout master
$ git merge experiment
```




- No es tan obvio: puede implicar solucionar un montón de conflictos
- Con la opción `-i`: *rebase interactivo* (elección y orden de commits)

# Revert

- *Revierte* la acción cometida en un *commit* concreto y la aplica a continuación en un nuevo commit
- Si el commit es `HEAD`, las consecuencias son obvias, pues se consigue el mismo estado que en `HEAD^`
- Si el commit no es `HEAD`, el procedimiento puede ser tedioso y puede implicar resolver numerosos conflictos

```
$ git commit -am "Mensaje"
// Supongamos que estamos en un commit abcdef...

// Seguimos editando...
$ git commit -am "Otro mensaje"
// Supongamos que estamos en un commit xyzklm...

$ git revert HEAD
// Conseguimos el mismo estado en el que estuvimos con el commit abcdef..., pero
// añadiendo un commit nuevo (cuyo hash es distinto a abcdef...)
```

# Cherry-pick

- Elección discreta de commits que se añadirán a la rama actual:

```
$ git cherry-pick 234556 9acf34
```

- Muy probablemente implicará resolver conflictos

# Stash

- Guardado provisional de estado en rama sin salvar en repositorio, para posteriormente volver. Aparición automática de rama `refs/stash`

```
// Por ejemplo, estamos en rama master
$ git checkout master

// Edicion de un fichero
// No se salva el estado de la rama actual en el repositorio
// pero sí en una "pila"
$ git stash

// Por ejemplo se conmuta momentaneamente a otra rama
$ git checkout rama

// Vuelta a rama
$ git checkout master
// Visualización de ficheros: no están tal y como estaban antes de stash.

// Lista de "stashes"
$ git stash list

// Aplicación del último stash o de uno en concreto [stash@]
$ git stash apply
// o de uno en concreto [stash@]
$ git stash apply stash@{n}

// Eliminación de la rama refs/stash
$ git stash drop
```

# Índice

# Información

```
// Lista ramas
$ git branch

// Lista ramas con detalles
$ git branch -v

// Lista las ramas fusionadas
$ git branch --merged

// Lista las ramas no fusionadas
$ git branch --no-merged

// Borra ramas (fusionadas)
$ git branch -d rama

// Borra ramas (fusionadas y no fusionadas)
$ git branch -D rama
```

# Índice

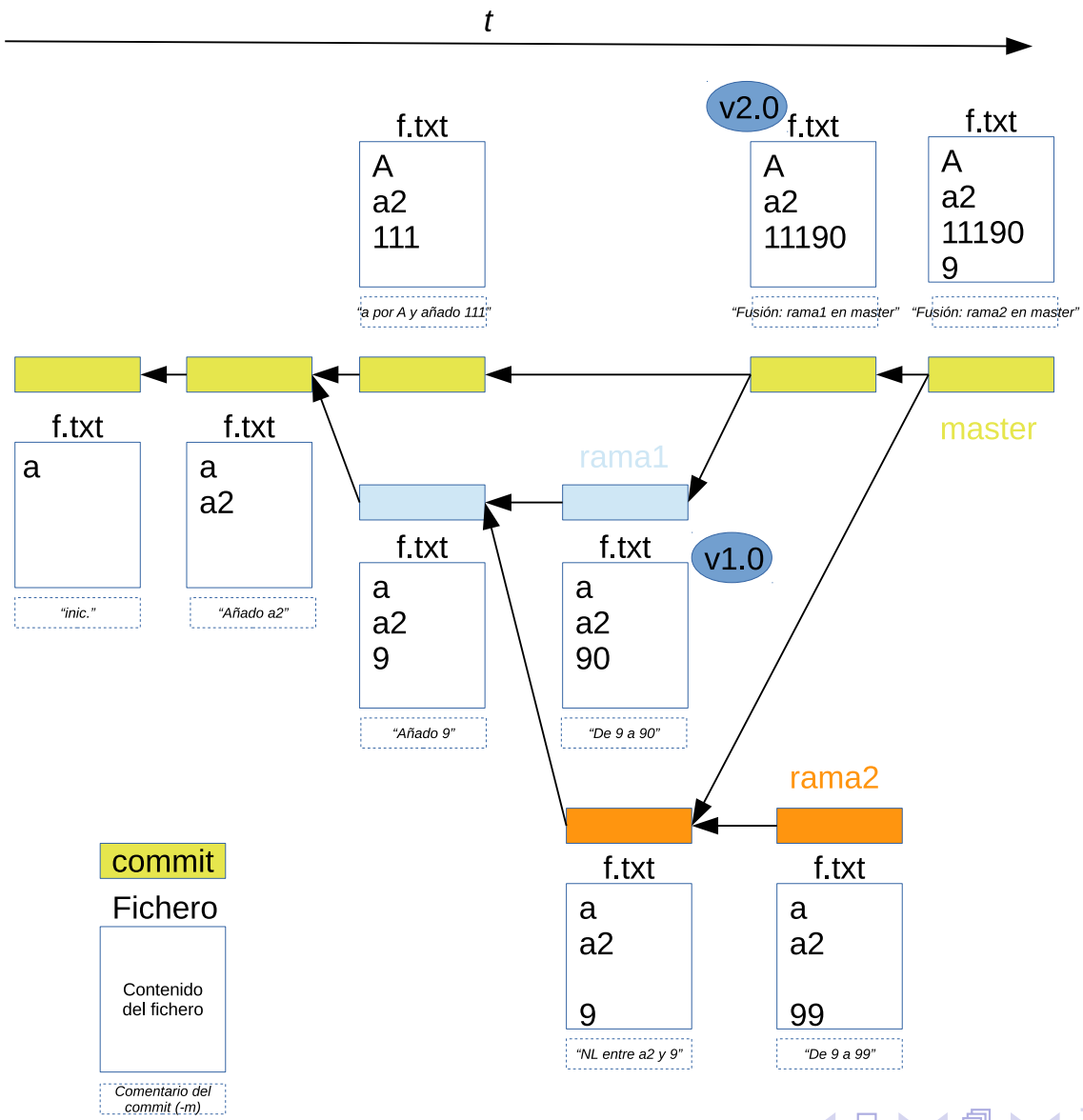
# Etiquetas *(tags)*

- Puntos específicos del repositorio
- Dos tipos: anotadas y ligeras

```
// Creación de etiqueta anotada
$ git tag -a v2.0 -m "Versión 2.0 del código"
// Creación de etiqueta ligera
$ git tag v3.0

// Lista todas las etiquetas
$ git tag

// Detalla una etiqueta
$ git show v2.0

// Creación de etiqueta a posteriori: commit 9fcdd04
$ git tag -a v4.0 9fcdd04

// Las etiquetas HAY QUE SUBIRLAS al repositorio EXPLÍCITAMENTE
$ git push origin v4.0
// O bien todas:
$ git push origin --tags

// Checkout: en detached HEAD
$ git checkout v3.0

// Si se desea hacer commits sobre el commit de la etiqueta
$ git checkout v3.0
$ git checkout -b nueva_rama_v3.0
// O bien
$ git checkout -b nueva_rama_v3.0 v3.0

// Descarga directa de tag de repositorio remoto
$ git pull origin v4.0
```

# Índice

# Ejemplo

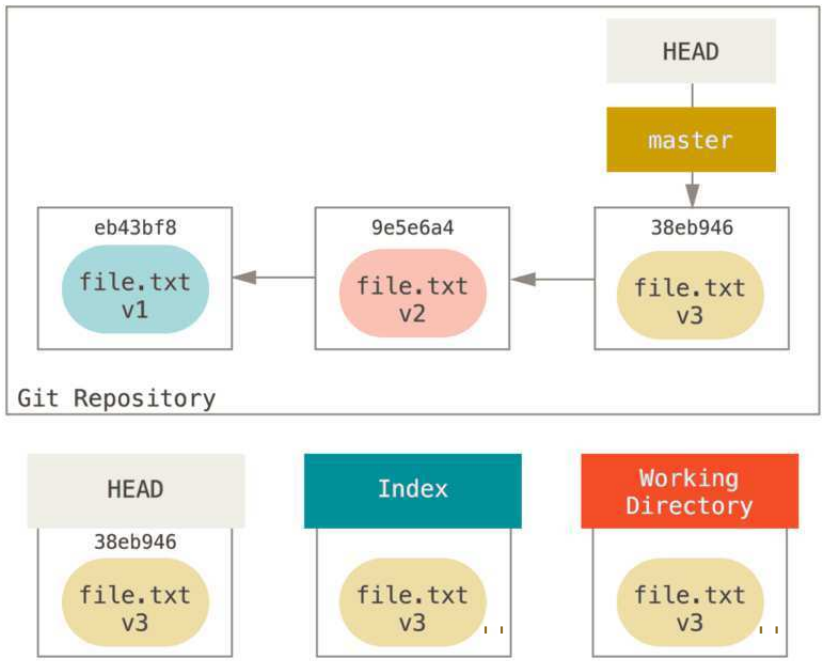

t
f.txt
A
a2
111
"a por A y añado 111"
v2.0
f.txt
A
a2
11190
"Fusión: rama1 en master"
f.txt
A
a2
11190
9
"Fusión: rama2 en master"
master
f.txt
a
"inic."
f.txt
a
a2
"Añado a2"
rama1
f.txt
a
a2
9
"Añado 9"
f.txt
a
a2
90
"De 9 a 90"
v1.0
rama2
f.txt
a
a2
9
"NL entre a2 y 9"
f.txt
a
a2
99
"De 9 a 99"
commit
Fichero
Contenido del fichero
Comentario del commit (-m)

# Índice

# Índice

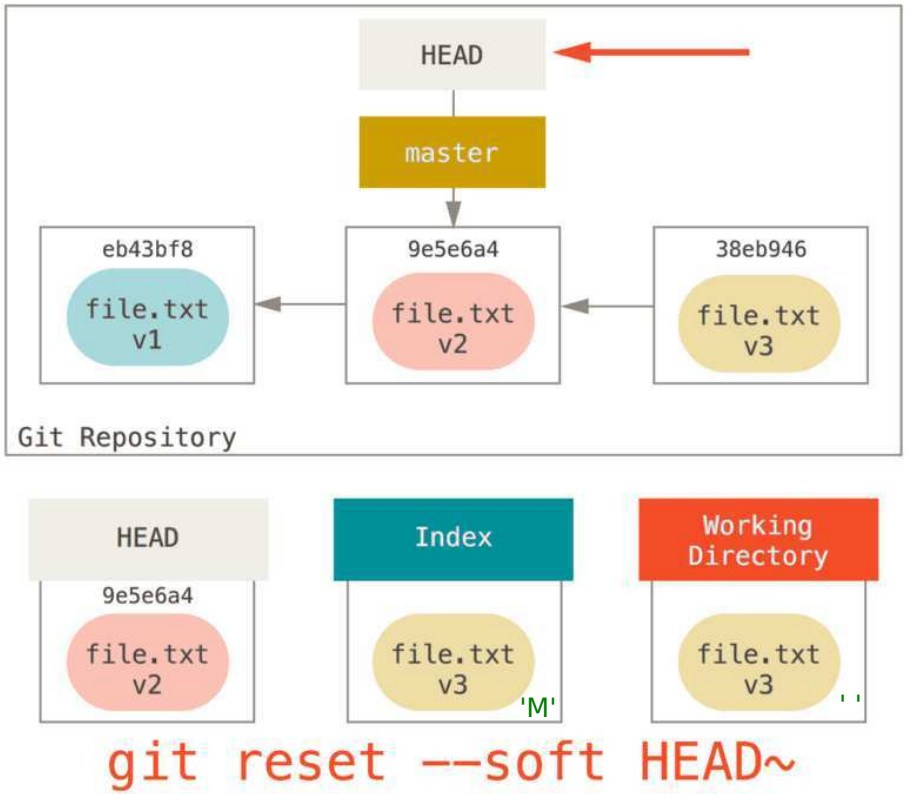
# Planteamiento

- El comportamiento de `reset` depende del tipo de parámetros
- Si **NO** intervienen ficheros concretos: manipula puntero `HEAD` y puntero de rama al unísono
  - En tal caso, tres modos: `--soft`, `--mixed` y `--hard`
  - Todos los ficheros del commit/index/directorio de trabajo están involucrados
- Punto de partida:

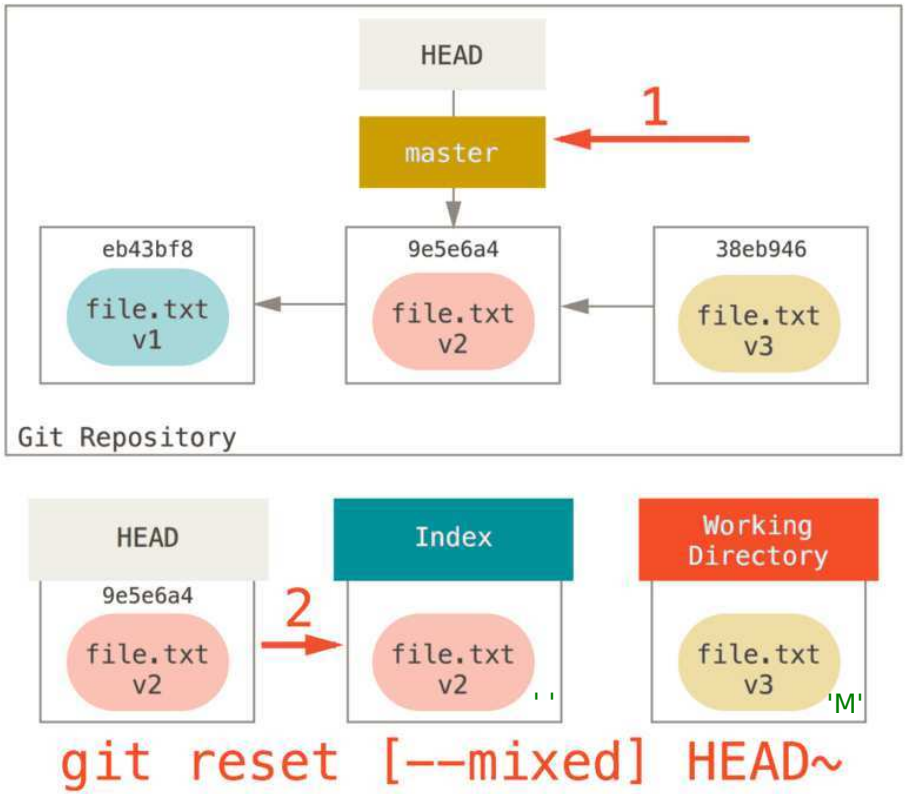



git commit -am "..."

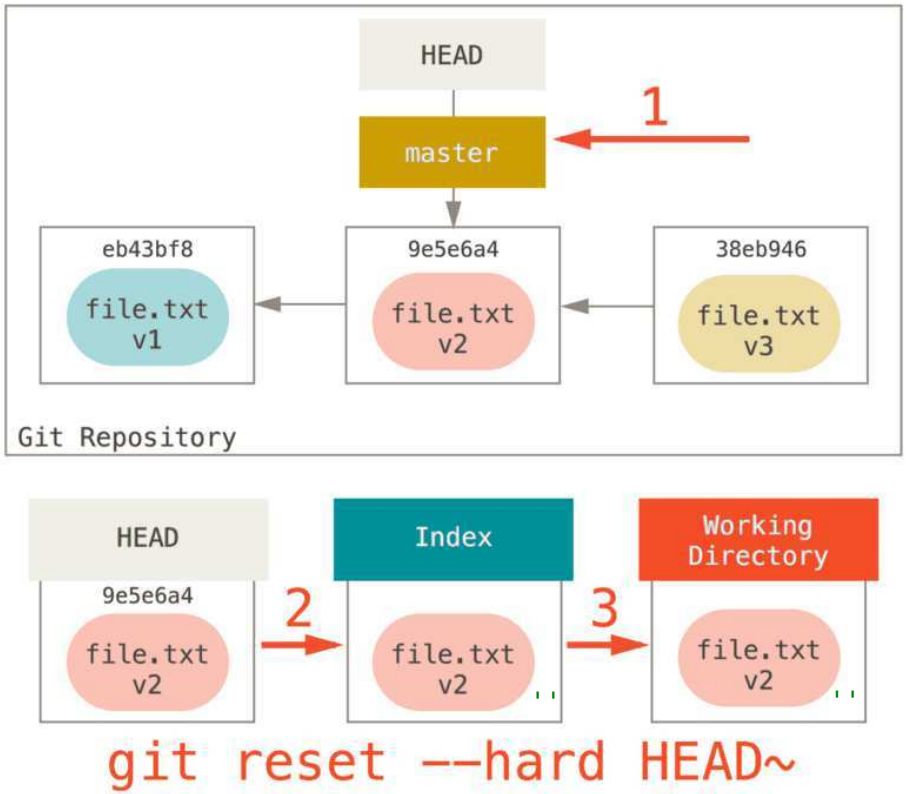
# reset *soft*


HEAD
master
eb43bf8
file.txt
v1
9e5e6a4
file.txt
v2
38eb946
file.txt
v3
Git Repository
HEAD
9e5e6a4
file.txt
v2
Index
file.txt
v3
'M'
Working
Directory
file.txt
v3
' '
git reset --soft HEAD~

# reset *mixed*

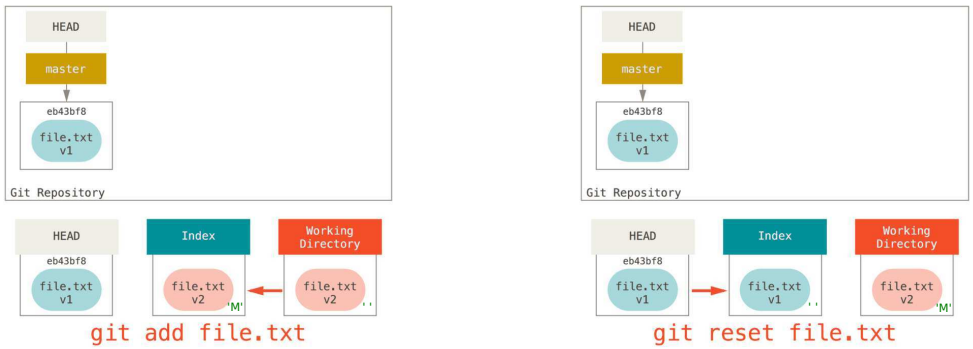

HEAD
master
1
eb43bf8
file.txt
v1
9e5e6a4
file.txt
v2
38eb946
file.txt
v3
Git Repository
HEAD
9e5e6a4
file.txt
v2
2
Index
file.txt
v2
Working
Directory
file.txt
v3
'M'
git reset [--mixed] HEAD~

# reset *hard*

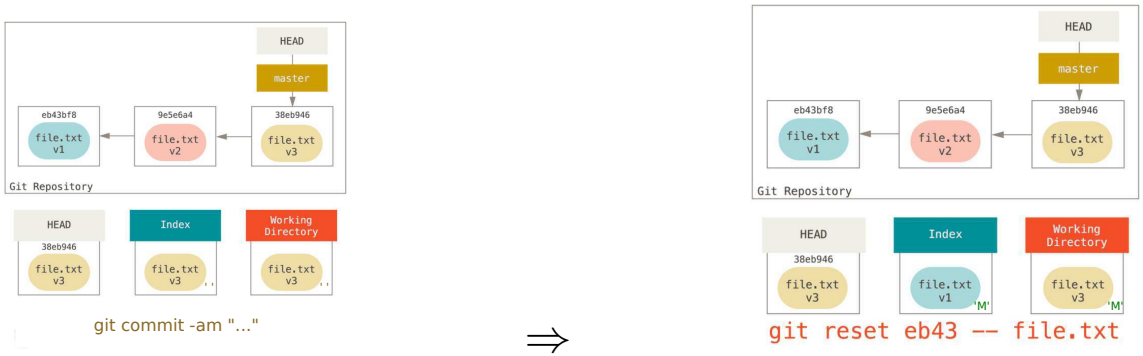

HEAD
master
1
eb43bf8
file.txt
v1
9e5e6a4
file.txt
v2
38eb946
file.txt
v3
Git Repository
HEAD
9e5e6a4
file.txt
v2
2
Index
file.txt
v2
3
Working
Directory
file.txt
v2
git reset --hard HEAD~

# Reset (con fichero(s)): opuesto a add

- Si hay fichero(s) como parámetro al comando: **NO** hay movimiento de `HEAD` ni de puntero de rama (ni opciones `--hard` ni `--soft`, al no mover HEAD).
- Sólo está involucrado el o los ficheros indicados.
- Con esta combinación, `reset` sería lo opuesto a `add`




- Con un commit SHA:



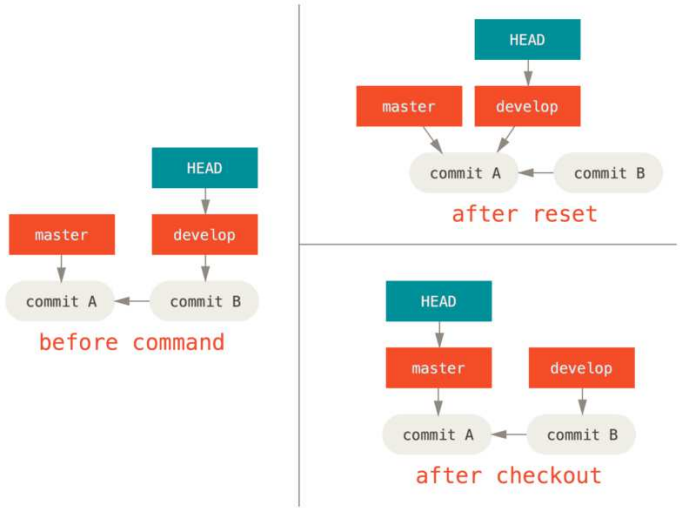

Notación `-- fichero`: eliminar ambigüedad con algún otro tipo de parámetro: confusión entre nombres de ficheros y nombres de rama, por ejemplo.

# Índice

# checkout y reset

- Diferencias entre `git checkout [rama]` y `git reset --hard [rama]`
  - `checkout` verifica que el WD actual tiene todos los cambios guardados antes de conmutar al WD correspondiente al especificado en el *checkout* (`reset`, no hace esta verificación)
  - `checkout` solo actualiza `HEAD` para apuntar a la nueva rama; `reset` mueve `HEAD` y el puntero de la rama

```
// Supongamos que estamos en la rama develop:
//   diferencia entre
$ git reset master
//   y
$ git checkout master
```



# checkout y reset con ficheros

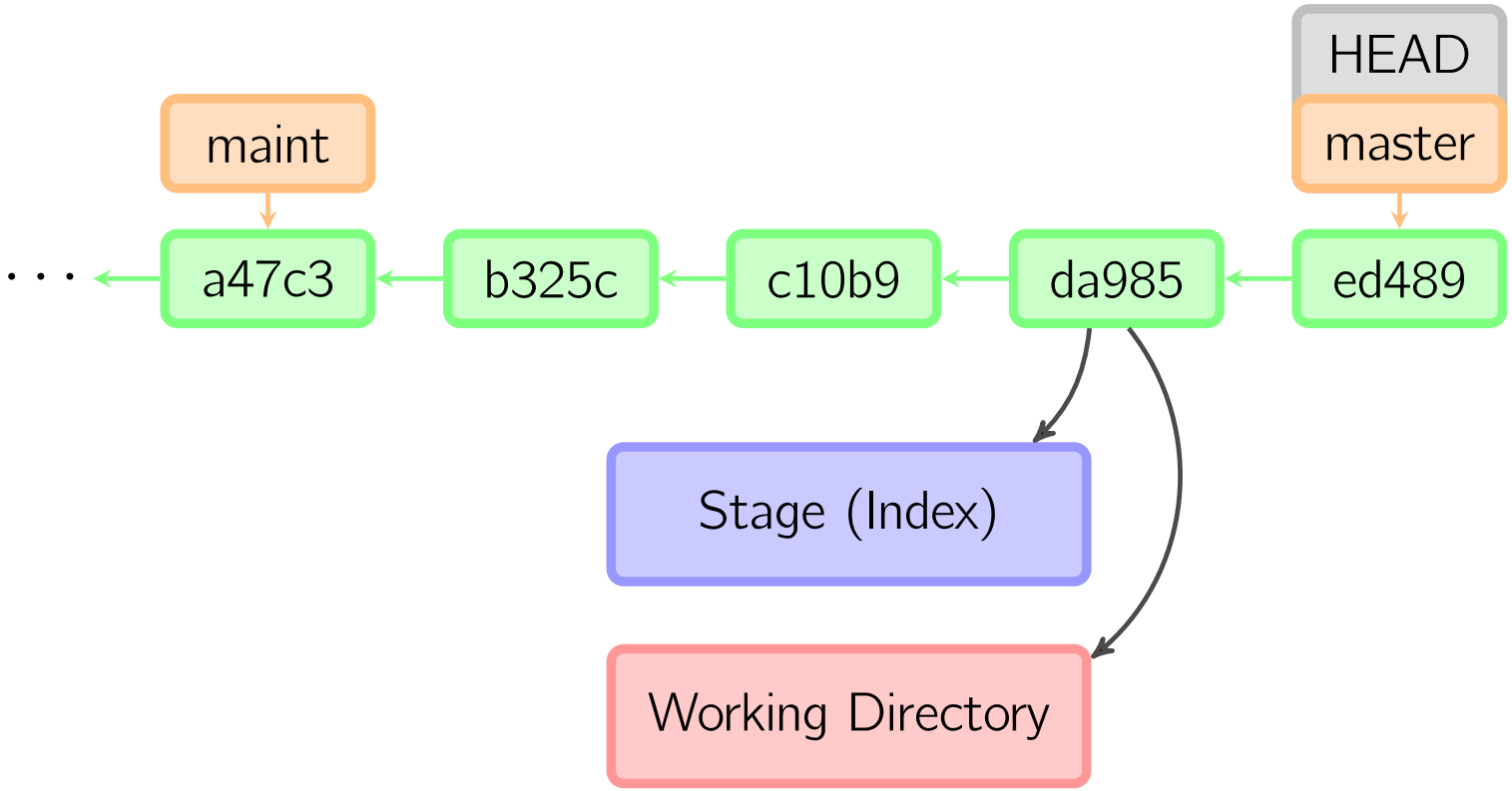
- Con ficheros:
  - Al igual que `reset`, `checkout` **NO** mueve **HEAD** ni la rama. No avisa de posible pérdida de información.
  - Carga tanto en el index como en el directorio de trabajo el fichero desde el commit deseado
- Tabla resumen:

| | Actualización | Index | Dir. trabajo | Seguro* |
|---|---|---|---|---|
| **Especificación con commits** | | | | |
| `reset --soft [commit]` | Rama+HEAD | No | No | Sí |
| `reset [--mixed] [commit]` | Rama+HEAD | Sí | No | Sí |
| `reset --hard [commit]` | Rama+HEAD | Sí | Sí | **NO** |
| `checkout <commit>` | HEAD | Sí | Sí | Sí |
| **Especificación con ficheros** | | | | |
| `reset [commit] <ficheros>` | — | Sí | No | Sí |
| `checkout [commit] <ficheros>` | — | Sí (si se especifica commit) | Sí | **NO** |

*Seguro: **"NO" significa que reescribe ficheros en WD sin preguntar**

# Ejemplo: checkout

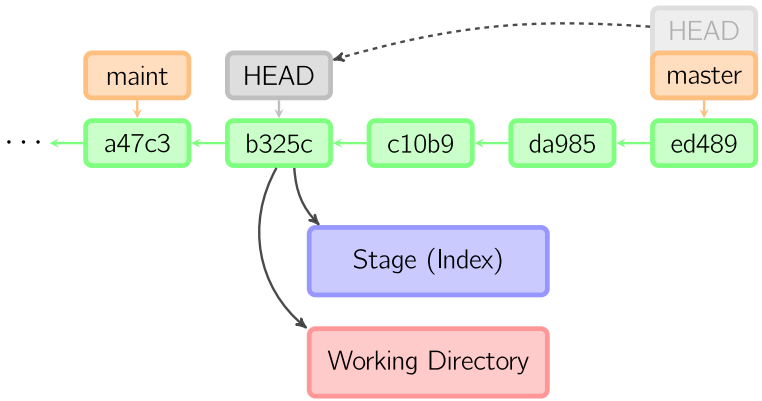
`git checkout HEAD~ files`



# checkout con *detached HEAD*

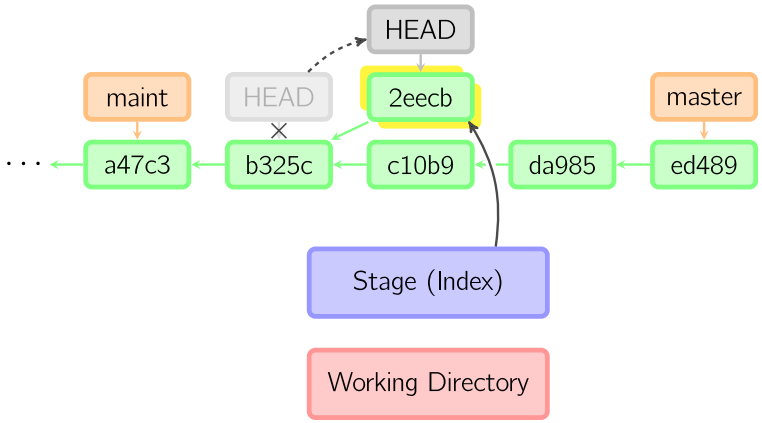
```
$ git checkout master~3
```

git checkout master~3



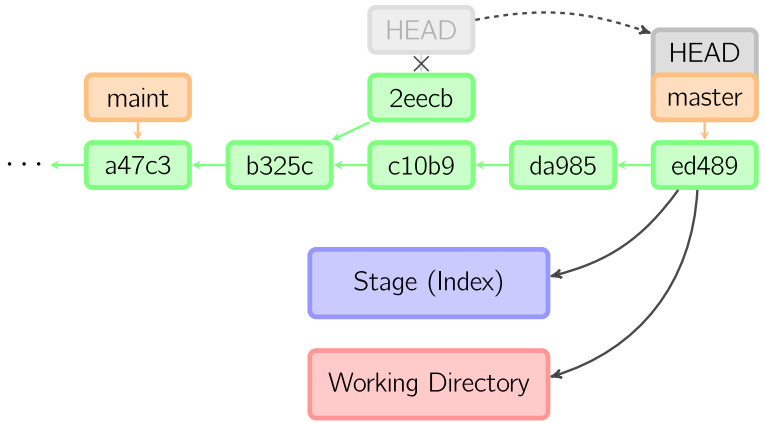

```
$ git commit
```

git commit




```
$ git checkout master
```

git checkout master




Referencia a *commit* 2eecb perdida, pues no tiene nombre de rama

Solución:

- Antes de abandonar e ir a una rama, ponerle nombre:

```
$ git checkout -b new
```

- Emplear `git reflog` o `git log -p`

# Índice

# Índice

# Conceptos

- Repositorios remotos *similares* a los repositorios locales
- Acceso (lectura y/o escritura: permisos) posible entre diferentes usuarios. Varios protocolos de acceso a un repositorio remoto:
  - `local`
  - `http/https`
  - `ssh`
  - `git`

  Depende de cómo esté configurado el acceso a dicho repositorio

# Creación de un repositorio local

- Creación directa

```
$ mkdir directorio_trabajo
$ cd directorio_trabajo
$ git init
// En .git tenemos realmente el repositorio
```

- Por clonación de repositorio remoto (el directorio de trabajo debe estar vacío)

```
$ mkdir DIRECTORIO_TRABAJO
$ git clone URL_repositorio_remoto DIRECTORIO_TRABAJO

// O bien:
$ mkdir DIRECTORIO_TRABAJO
$ cd DIRECTORIO_TRABAJO
$ git clone URL_repositorio_remoto .  // ¡¡¡Obsérvese el punto (.)!!!

// O  bien:
$ git clone URL_repositorio_remoto
// crea automáticamente un directorio con el mismo nombre que el repositorio remoto
// (sin extensión .git, si la tuviera)
```

  - Se lleva a cabo una copia integra del repositorio remoto en el local
  - Sólo se configura la rama por defecto (suele ser `master`)
  - Si se desea trabajar en otra rama conocida a priori

```
$ git clone -b rama URL_repositorio_remoto
```

  - A posteriori:

```
$ git clone URL_repositorio_remoto [...]
$ cd DIRECTORIO_TRABAJO
$ git checkout -b rama_local servidor_remoto/[rama_remota|PUNTERO|COMMIT]
// Ejemplos concreto:
//    git checkout -b rama_x origin/rama_1
//    git checkout -b rama_y origin/HEAD
//    git checkout -b rama_z 345baf3...
```

# Creación de un repositorio remoto

- Creación (en máquina *remota*):

```
$ mkdir repositorio
$ cd repositorio
$ git init --bare
// En este directorio tenemos ya el contenido equivalente de .git de un repositorio local
```

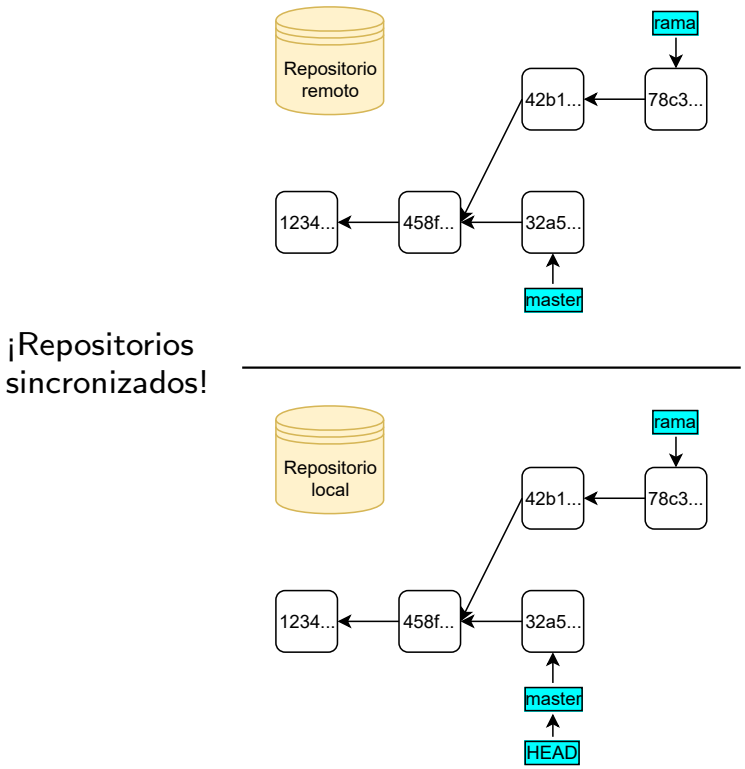
- Un repositorio *local* puede hacer las veces de *remoto* pero sólo con permisos lectura

```
$ mkdir repo1
$ cd repo1
$ git init
// Rellenado de repositorio
$ cd .. // (DIRECTORIO NO HIJO DE repo1, hermano, en el ejemplo)
$ mkdir repo2
$ cd repo2
$ git clone ../repo1 .
// El repositorio local ../repo1 ha hecho las veces de
// remoto (solo lectura);  no se podrá hacer push
```

# Índice

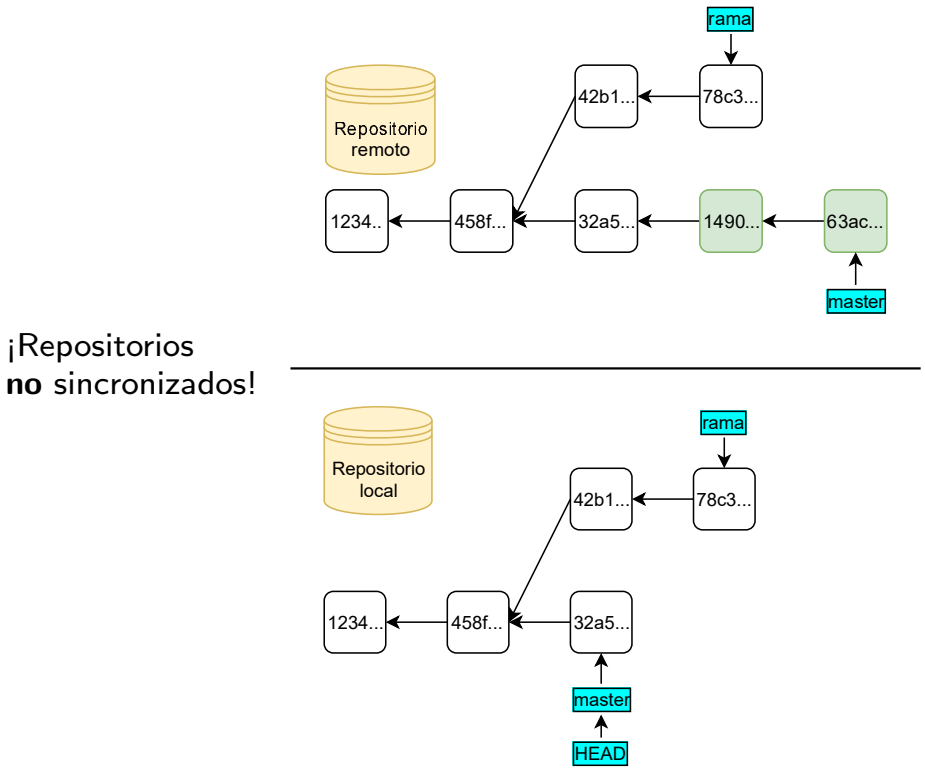
# Descarga de repositorio remoto (I)

- Situación de partida: repositorios sincronizados



# Descarga de repositorio remoto (II)

- Repositorio remoto actualizado (pero no el local)



# Descarga de repositorio remoto (III)

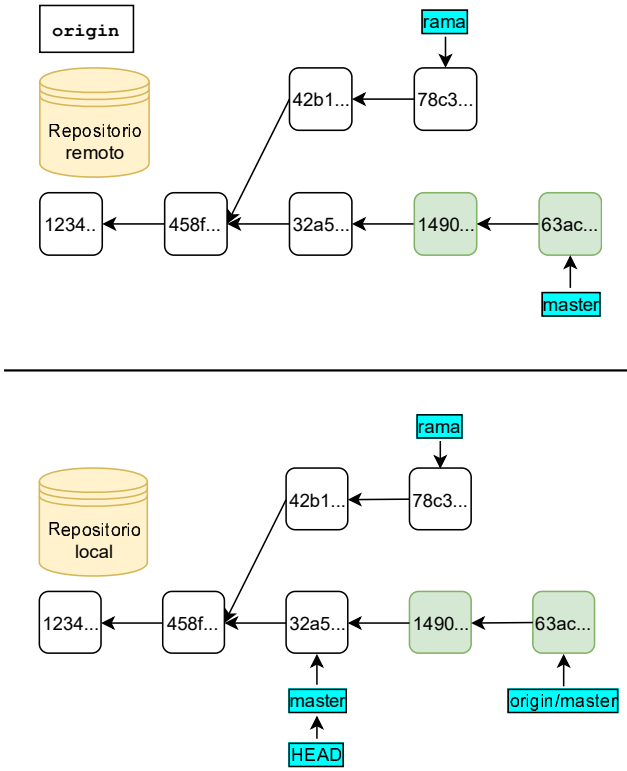
- Crear alias para repositorio remoto (sólo es necesario una única vez)

```
$ git remote add ALIAS_REMOTO URL_REPOSITORIO_REMOTO
```

- A partir de ahora, `ALIAS_REMOTO` y `URL_REPOSITORIO_REMOTO` serán intercambiables
- Un nombre típico para `ALIAS_REMOTO` es **origin**

# Descarga de repositorio remoto (IV)

- Descarga de rama `master` desde servidor `origin`: `fetch`



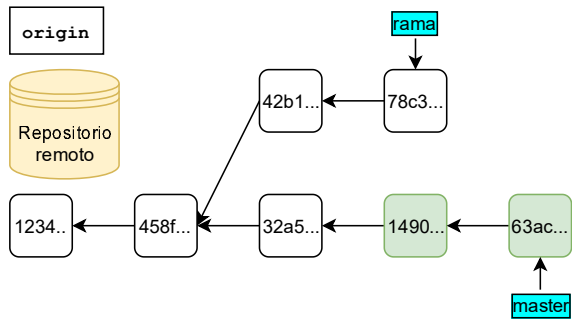

```
$ git fetch origin master
```

`git fetch origin`: descarga todas las ramas de `origin`

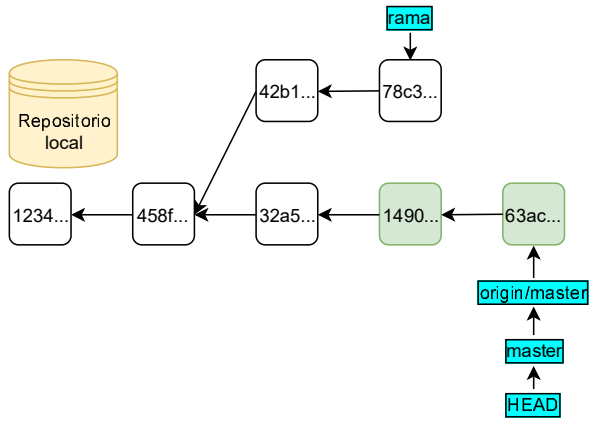
Estas ramas descargadas no son *editables*, sí *fusionables*

# Descarga de repositorio remoto (V)

- Fusión entre la rama `master` y `origin/master`: `merge` con *fast-forward* ¡Sincronizados!




¡Repositorios sincronizados!




```
$ git merge origin/master
```

# Comando equivalente: pull

- La secuencia de comandos

```
$ git fetch origin master
$ git merge origin/master
```

es equivalente a

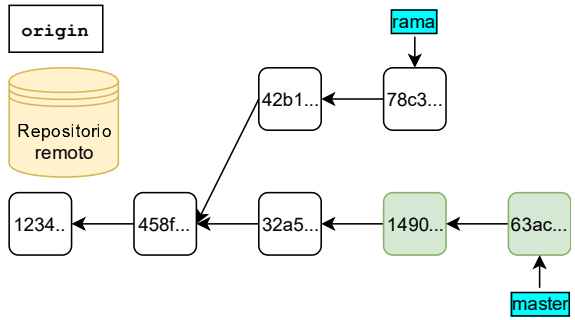
```
$ git pull origin master
```

Si no se especifica rama, se *traen* todas las ramas:

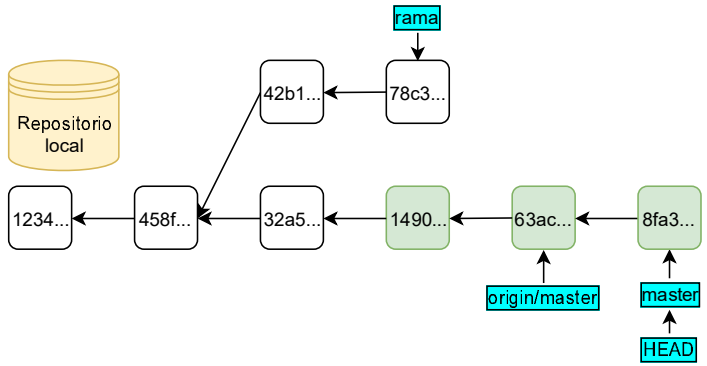
```
$ git pull origin
```

# Carga hacia repositorio remoto

- Actualización del respositorio local



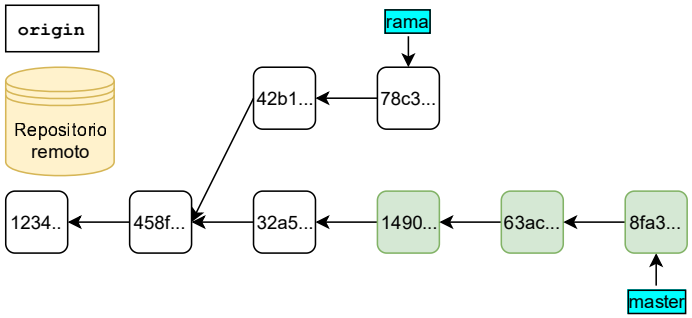

¡Repositorios **no** sincronizados!



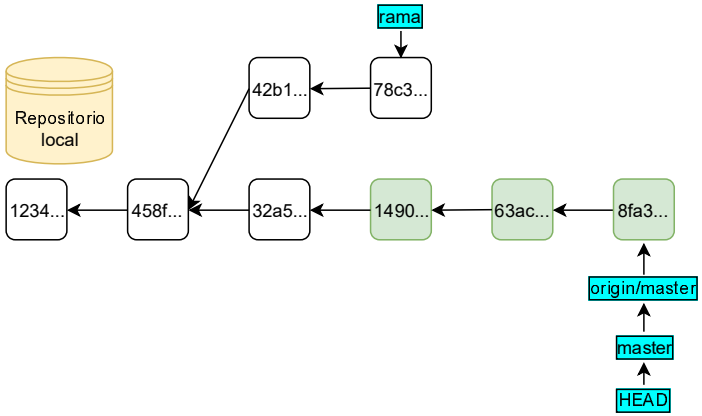

```
$ git commit -am "..."
```

# Carga hacia repositorio remoto

- Actualización del respositorio remoto: `push`



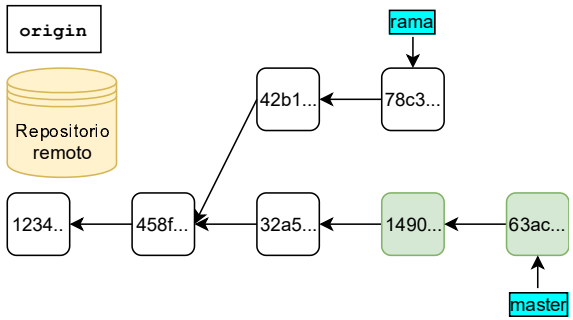

¡Repositorios sincronizados!



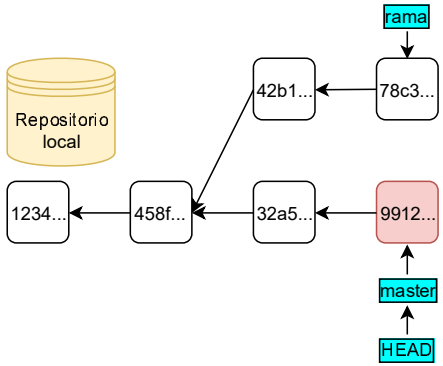

```
$ git push origin master
```

# Error frecuente

- Empezamos a trabajar en WD sin haber actualizado el repositorio:



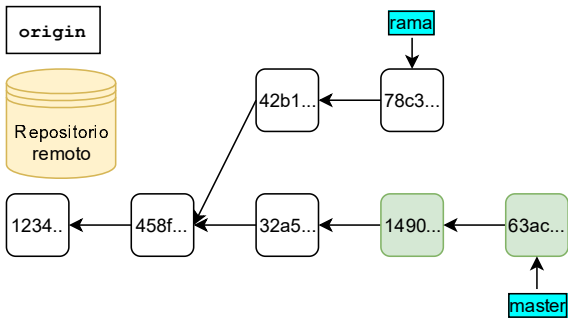

¡Repositorios no sincronizados!



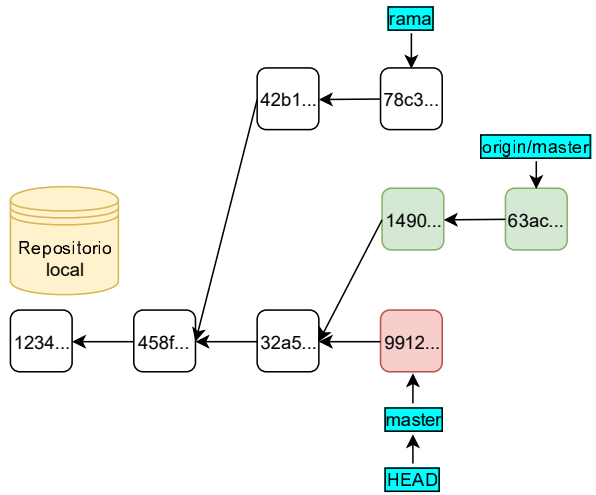

```
$ git commit -am -M "..."
```

# Error frecuente

- Si dado el escenario, desea sincronizar con el repositorio remoto, probable CONFLICTO:



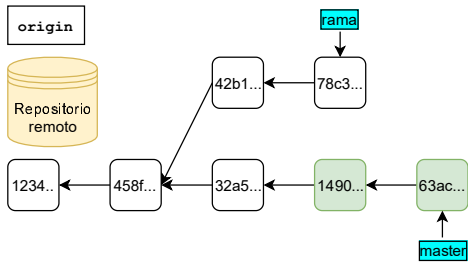

¡Repositorios no sincronizados!



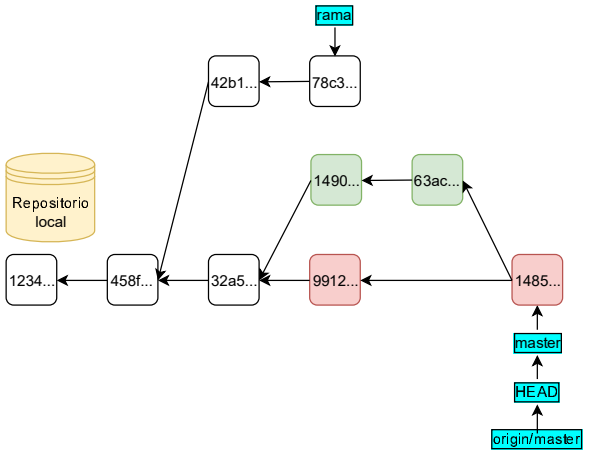

*fast-forward* imposible entre `master` y `origin/master`

# Error frecuente

- Conflicto. Resolución:
  - Hay que bajar repositorio remoto `pull/fetch`
  - Fusionarlo con el local `pull/merge`
  - Subir fusión al repositorio remoto `commit` ⇒ `push`




¡Repositorios no sincronizados!




¡Resolver conflicto!

```
$ git pull origin master
```

O bien

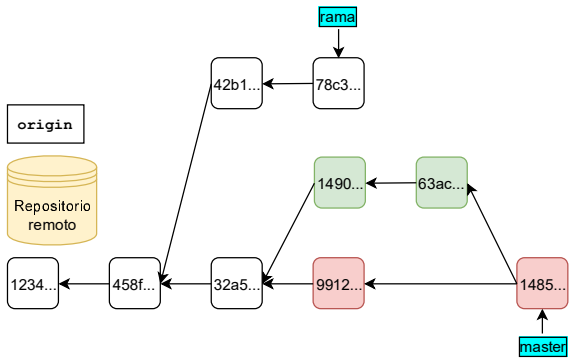
```
$ git fetch origin master
$ git merge origin/master
```

y después...

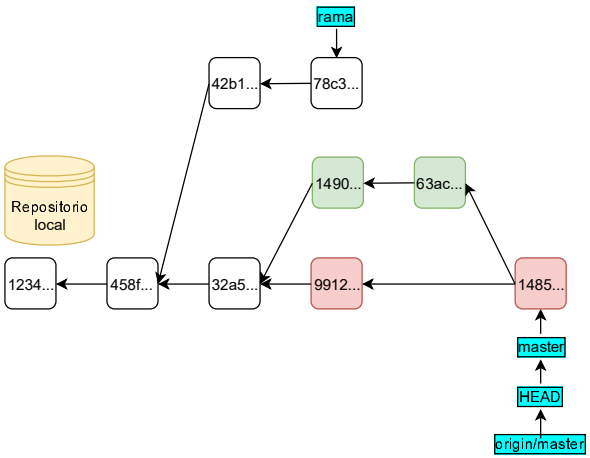
```
$ git commit -am "Mensaje"
```

# Error frecuente

- Actualización en repositorio remoto: `push`




¡Repositorios sincronizados!




```
$ git push origin master
```

# Más sobre repositorios remotos

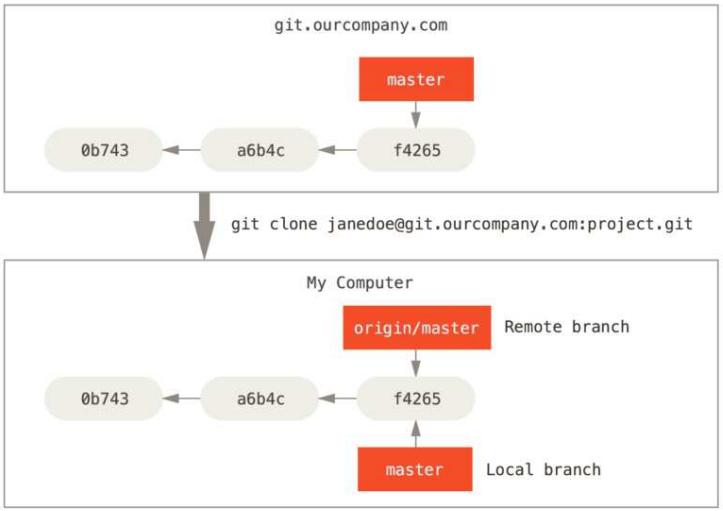
```
// Se pueden añadir mas alias de repositorios
$ git remote add   otro_alias_repositorio   otra_url_repositorio_origen

// Se pueden utilizar posteriormente:
$ git pull/fetch otro_alias_repositorio rama

// O borrarlos (desasociarlos)
$ git remote remove alias_repositorio
// O renombrarlos
$ git rename alias_actual alias_nuevo

// Muestra todos los repositorios remotos asociados con nuestro repositorio local
// que fueron añadidos mediante add o mediante clone (origin)
$ git remote
// O bien, con mas detalle
$ git remote -v

// O más detalles todavía
$ git remote show alias_repositorio
```

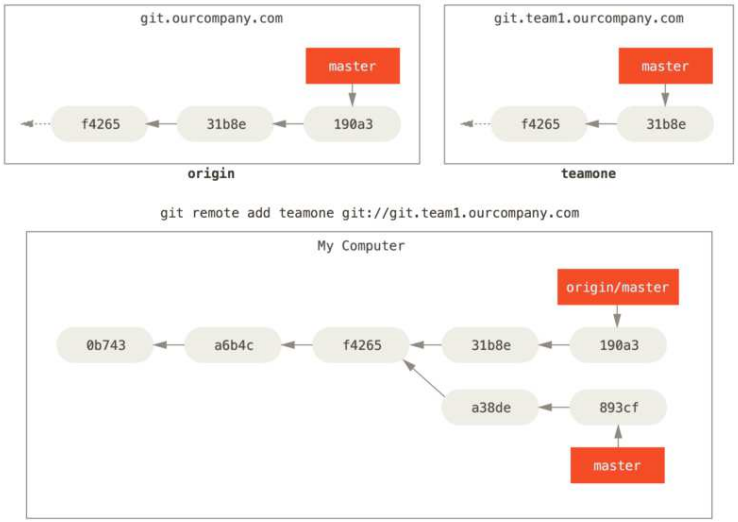
# Índice

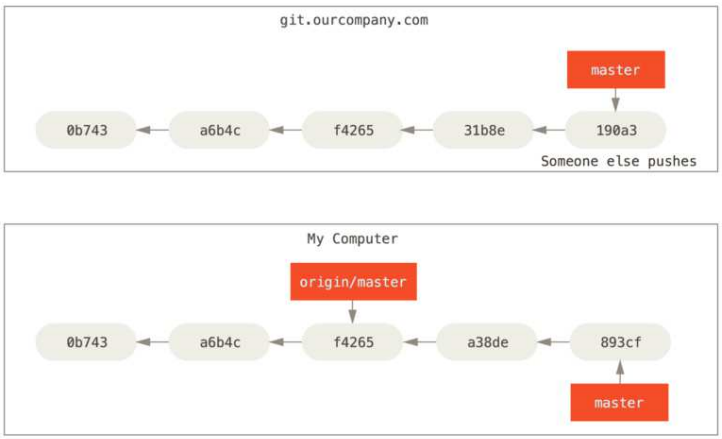
# *Remote-tracking branches*

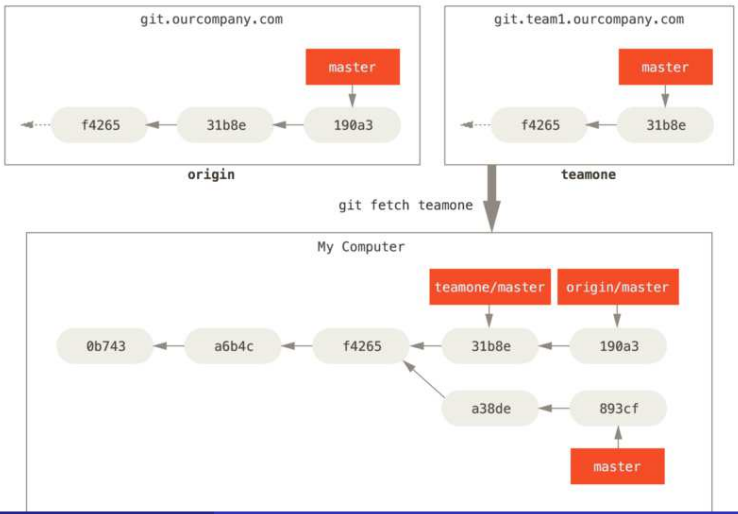
- Son referencias (no cambiables localmente) al estado de ramas en un repositorio remoto
- Sintaxis: `<repoRemoto>/<rama>`. Ejemplo: `origin/master`
- Con clonación/pulling/fetching: adquisición local de estas referencias

# *Remote-tracking branches*: ejemplos



# Ramas (locales) de seguimiento *(tracking branches)*

- Rama de seguimiento: rama resultante de hacer un *checkout* de una rama de seguimiento remota (denominada *upstream branch*).
- Cuando se clona un repositorio: creación automática de rama local `master` que sigue a rama remota `origin/master`.
- Igualmente con `checkout` (previo `fetch`):

```
$ git checkout -b rama remoto/rama
// Alternativamente
$ git checkout --track remoto/rama
// O si no hay ambigüedad (si no existe localmente y sí remotamente en un solo repositorio...)
$ git checkout rama
```

- Y con `push`

```
$ git push -u origin master
```

- Para conocer todas las ramas con seguimiento

```
$ git branch -vv
```

- Cuando se establece el *seguimiento* no es necesario especificarlo en parámetros:

```
$ git push
$ git pull
$ git fetch
```

# Índice

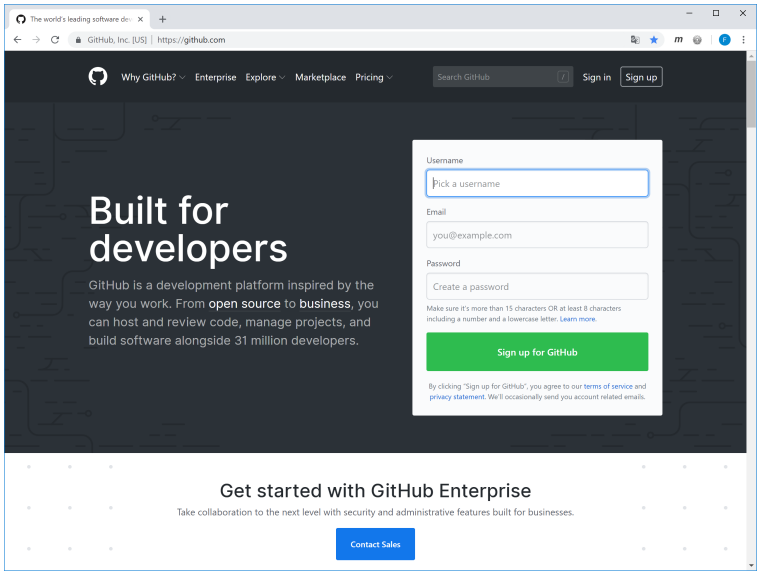
# Conceptos básicos

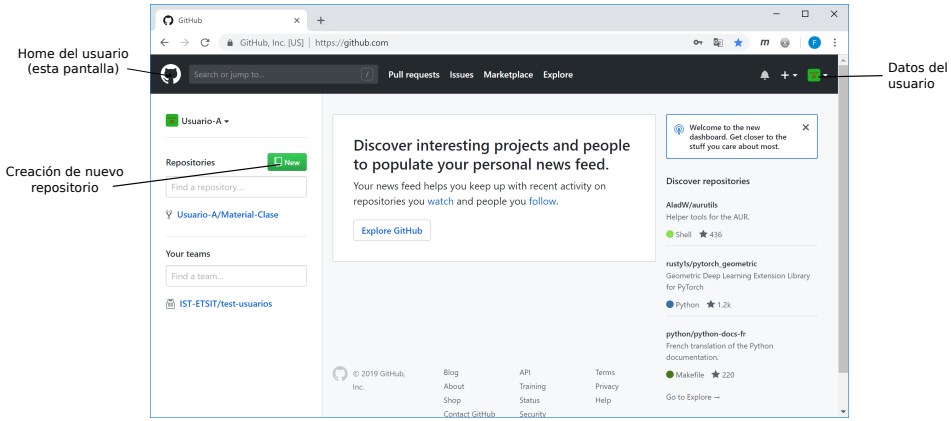
- `http://github.com`
- GitHub: repositorio remoto de Git con, adicionalmente, interfaz web
- Algunas diferencias operativas y notacionales

# Índice

# Apertura de cuenta

- Sugerencia nombre de usuario: email de UPV en *Sign up* de http://github.com

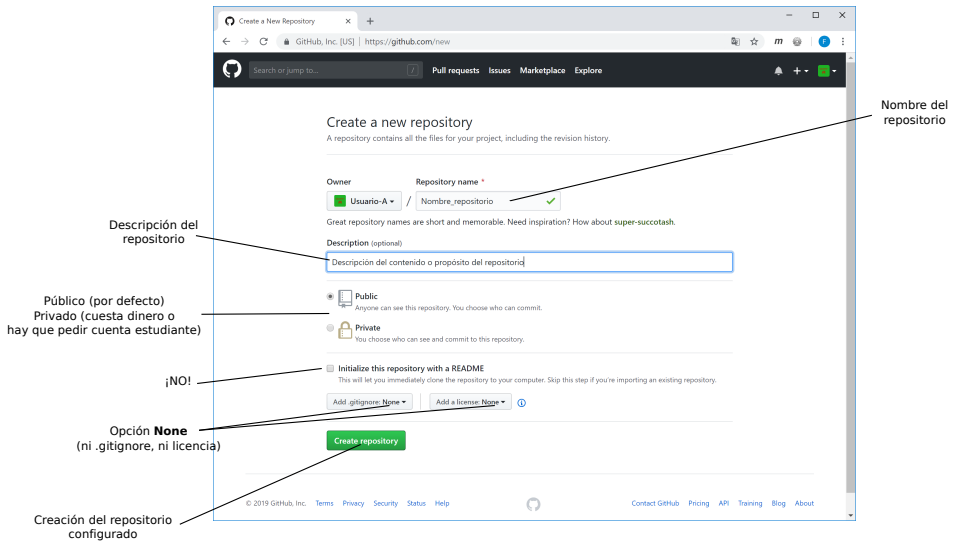


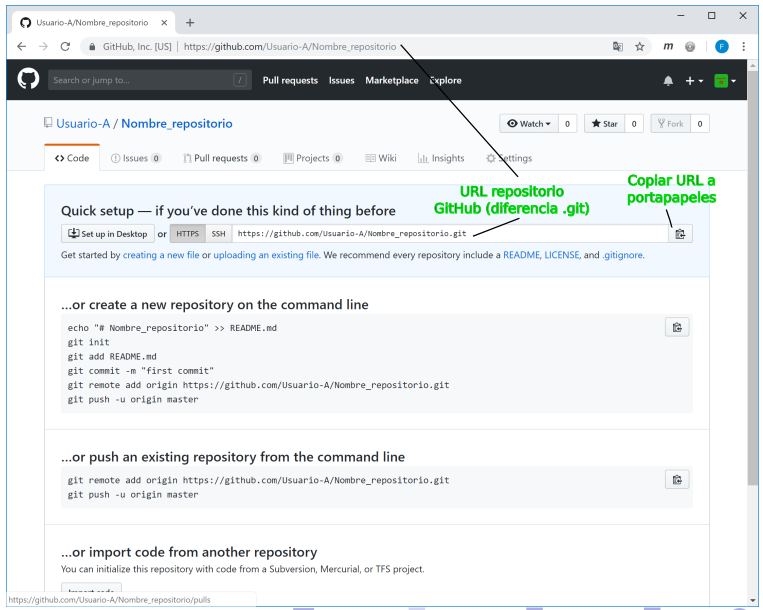




- Posibilidad de tener repositorios privados
- Los respositorios compartidos con la asignatura (organización GitHub) serán **siempre** privados.
- Próximamente se explicará interacción con organización: `AATT-ETSIT`

# Gestión de repositorios: inicialización local (I)

- Debe quedar claro quién crea **inicialmente** el repositorio.
- Si inicialización en local, entonces
  - En local: `git init` y acceso a repositorio convencional
  - En GitHub:
    - Creación de repositorio nuevo: New
    - Rellenar detalles de creación
    - Importante: **SIN** `README`, `.gitignore` ni `licencia`
    - Seguir indicaciones en página tras `Create repository`



# Gestión de repositorios: inicialización local (II)

- Debe quedar claro quién...
- Si inicialización en local, entonces
  - En local...:
  - En GitHub ...:
  - En local:

```
$ git remote add NOMBRE_SERVIDOR URL_GITHUB
```

    - `NOMBRE_SERVIDOR`: el nombre (alias) que queramos, `origin`, `github`, `remoto`, ...
    - `URL_GITHUB`: URL (desde navegador o copiar a portapapeles)

  - En local, ya interaccion convencional:

```
$ git push/pull [-u] NOMBRE_SERVIDOR RAMA
```

# Gestión de repositorios: inicialización en GitHub

- Si inicialización en GitHub (opción 1)
  - Si se inicializa repositorio en GitHub: con `README`, `.gitignore` o `licencia`, y pulsar Create repository
  - En local: `git clone URL_GITHUB [WD]`
  - En local, ya interacción convencional
- Si inicialización en GitHub (opcion 2)
  - Si se inicializa repositorio en GitHub: con `README`, `.gitignore` o `licencia`
  - En local: `git init`
  - En local: `git remote add ORIGEN URL_GITHUB`
  - En local: `git pull [-u] ORIGEN NOMBRE_RAMA`
  - En local, ya interacción convencional

# Interaccción entre repositorios de GitHub

- *Clonación en GitHub*: Fork
- Fusiones: *Pull request* a uno mismo o a un repositorio clonado (mediante *fork*)
  - Ejemplos
- Entorno intrínsecamente colaborativo