

Módulo profesional entornos de  
desarrollo

UD 6.1 –  
Control de  
versiones

# Modus operandi clásico



# Modus operandi clásico

## Trabajando en grupo

- Enviar cambios por mail, o
- Sincronizar cambios por Dropbox, o
- Sincronizar cambios por Google Docs.



# Sistema de control de versiones

Los **Sistemas de Control de Versiones** son programas que permiten **manejar los cambios** en el código fuente de un proyecto a lo largo del tiempo.

# Sistema de control de versiones

Llevan un **seguimiento** de las modificaciones que hacemos, y en caso de que nos equivoquemos, es posible volver atrás y comparar el código actual con versiones anteriores para ayudar a arreglar el error.

# Sistema de control de versiones

También permiten que distintas personas modifiquen el código a la vez y **compartan los cambios**, tratando de prevenir conflictos, y en caso de que los hubiera, ayudando a identificarlos y resolverlos.

# Sistema de control de versiones

Es decir, permiten...

- Arreglar *accidentes* y volver a versiones anteriores del código.
- Compartir código con otras personas.

# Sistema de control de versiones

## ¿Porqué usar un control de versiones nos hará felices?

- Proporciona **copias de seguridad** automáticas de los ficheros.
- Permite **volver a un estado anterior** de nuestros ficheros.
- Ayuda a trabajar de una forma **más organizada**.
- Permite **trabajar de forma local**, sin conexión con servidor. (en distribuídos).
- Permite que **varias personas** trabajen en los **mismos ficheros**.
- Permiten trabajar en **varias funcionalidades en paralelo separado** (ramas).



# Cómo empezar con GIT y trabajar con un repositorio remoto git

**GIT** es un sistema de control de versiones (Version Control System)

Un **VCS** sirve como repositorio de códigos de programa, incluidas todas las revisiones históricas. Registra los cambios en los archivos cuando hacemos confirmaciones (commit) y los registra en un log para que se pueda recuperar cualquier archivo en cualquier punto de confirmación (commit)



# Cómo empezar con GIT y trabajar con un repositorio remoto git

**Git** fue diseñado y desarrollado inicialmente por **Linus Torvalds**, en 2005, para apoyar el desarrollo del kernel de Linux. GIT es un sistema de control de versiones **distribuido (DVCS)**.

Otros VCS populares son:

- Sistemas centralizados de control de versiones cliente-servidor (**CVCS**):
  - **CVS** (sistema de versiones concurrentes),
  - **SVN** (Subversion)
  - Perforce.
- VCS distribuidos (**DVCS**):
  - GIT,
  - Mercurial,
  - Bazar,
  - Darcs.

El sitio de Git es <http://git-scm.com>

# Sistemas de control de versiones distribuidos vs centralizados

Distribuido vs. Centralizado

El control de versiones **distribuido** toma un enfoque entre iguales (peer-to-peer), opuesto al enfoque de **cliente-servidor de los sistemas centralizados**.

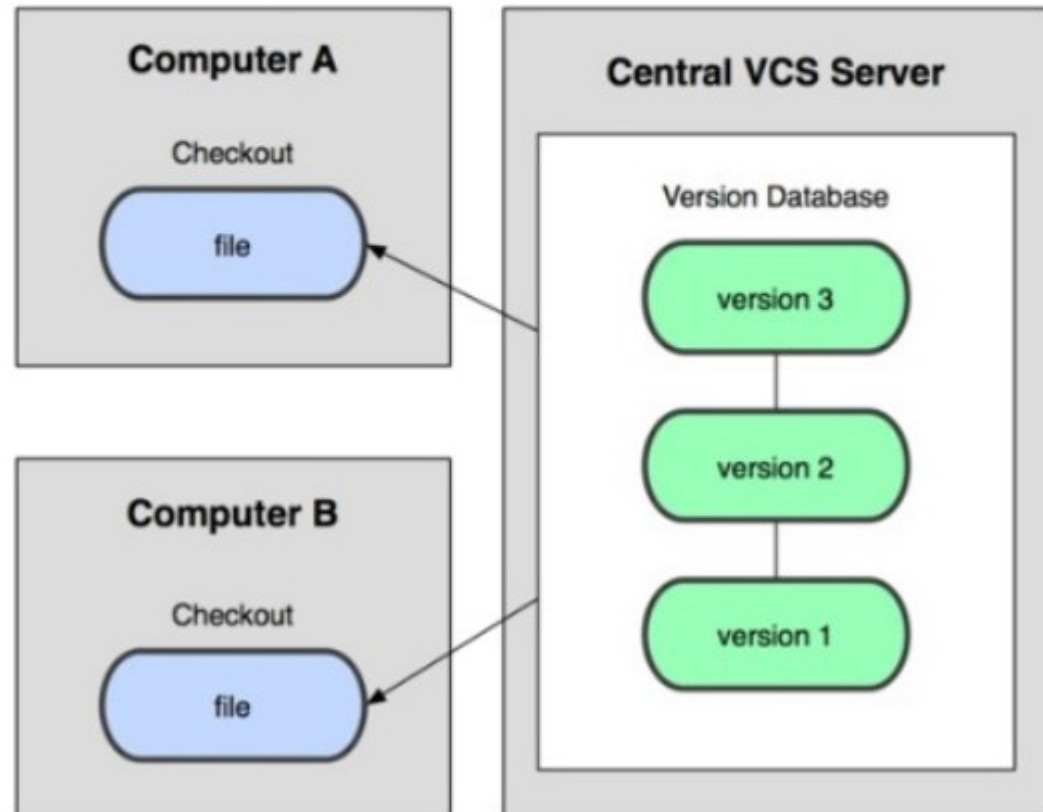
En lugar de un único repositorio central en el cual los clientes se sincronizan, la copia local del código base de cada igual es un repositorio completo. El control de versiones distribuido sincroniza los repositorios intercambiando ajustes (conjuntos de cambios) entre iguales.

**Distribuidos:** cada usuario tiene una copia completa del repositorio (pueden ser utilizados como copias de respaldo de emergencia). Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos.

**Centralizados:** existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos)

## Sistemas de Control de Versiones Centralizados (CVCS)

Ejemplos: CVS, Subversion, Perforce, SourceSafe, ...

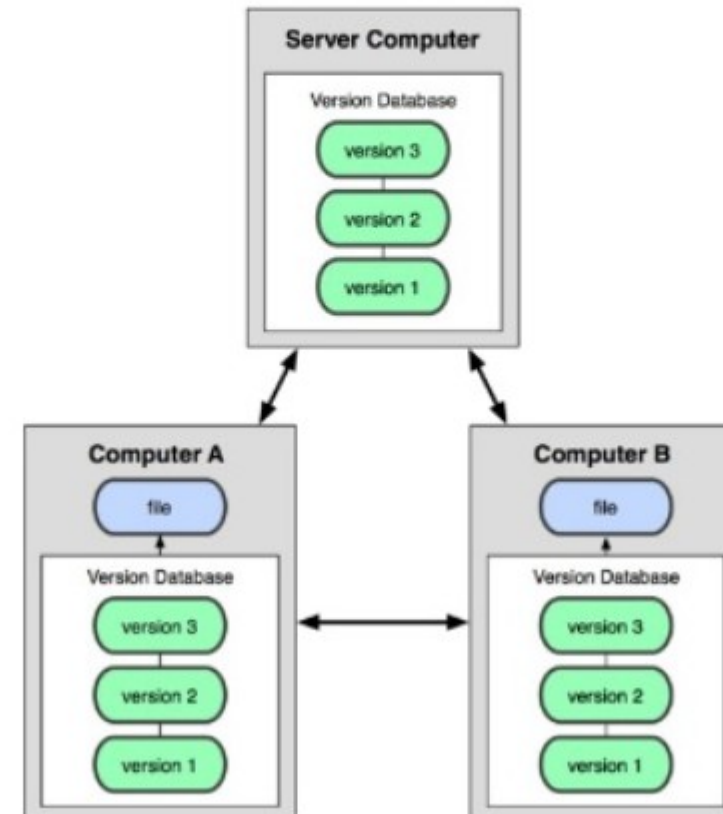


# Sistemas de Control de Versiones Distribuidos (DVCS)

Ejemplos: git, Mercurial, Bazaar, BitKeeper,...

**Distribuidos:** cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos.

**Centralizados:** existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos)



# Conceptos generales

Antes de nada, vamos explicar conceptos generales de los sistemas de control de versiones:

## **Repositorio**

El **repositorio** es el lugar en el que se almacenan los datos actualizados e históricos de cambios.

# Conceptos generales

## Ramas

Branches o ramas, son **bifurcaciones** en el desarrollo del proyecto.

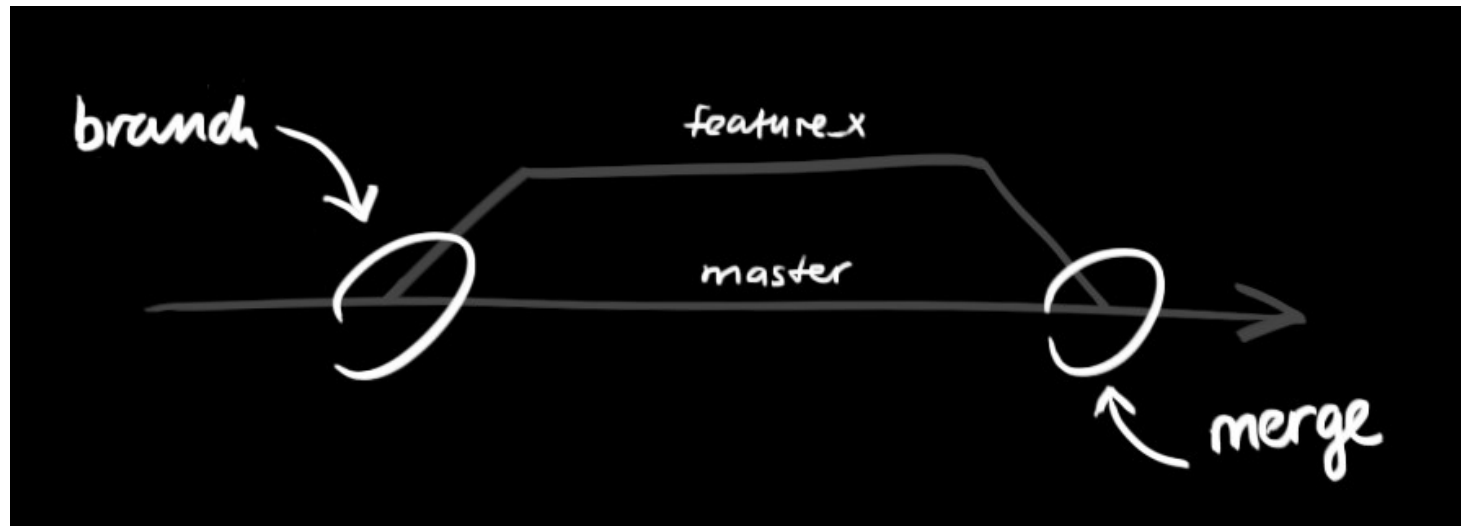
Supongamos que estamos trabajando en un proyecto y queremos añadir nueva funcionalidad al mismo. La forma **adecuada** de hacerlo es crear una **nueva rama** con el nombre de la nueva funcionalidad donde añadiremos nuestros cambios

Hemos testado profundamente nuestra nueva funcionalidad y estamos listos para moverla de la rama nueva funcionalidad a la rama principal master. Dicha acción requiere que **mezcleemos** el contenido de una de las ramas dentro de la otra.

# Conceptos generales

## Ramas

Cuando inicializamos un proyecto con Git automáticamente nos encontramos en una rama a la que se denomina "master".





# Conceptos generales

## **Commits**

Los commits son los puntos de guardado que se van realizando sobre las ramas en los que fijamos los cambios realizados en el código. Podremos volver a ellos para recuperar el estado del código en un determinado momento.

## **Merge**

Un merge es la acción de juntar una rama sobre otra y unificar los cambios. Por ejemplo para añadir una funcionalidad sobre el proyecto principal.

## **Sincronización (Pull/Push)**

Son los procesos para sincronizar el estado de nuestros sistemas locales con el servidor de código principal, tanto para subir los cambios locales como para descargar las últimas actualizaciones.

# Configurando Git

Debes configurar Git en su máquina local, de la siguiente manera:

## 1. Descargar e instalar:

- Para Windows y Mac, descargue el instalador desde <http://git-scm.com/downloads> y ejecute el instalador descargado.
- Para Ubuntu, ejecute el comando "sudo apt-get install git".
- Para Windows, use el shell de comandos "Git Bash" incluido con Git Installer para emitir comandos. Para Mac/Ubuntu, use el "Terminal".

# Configurando Git

## 2. Personalizar Git:

Ejecute el comando **"git config"** (para Windows, ejecute "Git Bash" desde el directorio instalado de Git. Para Ubuntu / Mac, inicie un "Terminal"):

```
// Set up your username and email (to be used in labeling your commits)
$ git config --global user.name "your-name"
$ git config --global user.email "your-email@youremail.com"
```

- La configuración se mantiene en "<GIT\_HOME>/etc/ gitconfig" (del directorio instalado de GIT) y "<USER\_HOME>/ .gitconfig" (del directorio de inicio del usuario)
- Puede ejecutar **"git config --list"** para enumerar los ajustes:

```
$ git config --list
user.email=xxxxxx@xxxxxx.com
user.name=xxxxxx
```

# Configurando Git

## Tu identidad

Es importante establecer nuestro **nombre y email**, ya que estos van a ir asociados con los cambios que hagamos:

```
git config --global user.name "Guybrush Threepwood"  
git config --global user.email guybrush@example.com
```

# Fundamentos de Git

## Comandos de Git

Git proporciona un conjunto de comandos simples, distintos e independientes desarrollados de acuerdo con la filosofía "Unix toolkit": construya herramientas pequeñas e interoperables.

Para ejecutar un comando, inicie un "Terminal" (para Ubuntu / Mac) o "Git Bash" (para Windows):

```
$ git <command> <arguments>
```

# Fundamentos de Git

## Ayuda y Manual

La mejor manera de obtener ayuda en estos días es sin duda **googlear**.

Para obtener **ayuda sobre los comandos de Git**:

```
$ git help <command>  
// or  
$ git <command> --help
```

El manual de GIT se incluye con el software (en el directorio "doc") y también está disponible en línea en <http://git-scm.com/docs>.

# Git. Empezando con un repositorio local

Hay 2 formas de iniciar un proyecto administrado por Git:

- Comenzando tu **propio proyecto**;
- **Clonando un proyecto existente** desde un host GIT.

Comenzaremos con "**Comenzando su propio proyecto**" y cubriremos "**Clonación**" más adelante

# Git. Empezando con un repositorio local

## Configurar el directorio de trabajo para un nuevo proyecto

Comencemos un proyecto de programación en el directorio de trabajo llamado "**hello-git**", con un archivo fuente "Hello.java" de la siguiente manera:

```
// Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world from GIT!");
    }
}
```

Compile "Hello.java" en "Hello.class"



# Git. Empezando con un repositorio local

También se recomienda encarecidamente proporcionar un archivo "README.md" (un archivo de texto en una sintaxis llamada "Markdown" como "GitHub Flavored Markdown") para describir tu proyecto:

```
// README.md  
This is the README file for the Hello-world project.
```

Ahora, tenemos 3 archivos en el árbol de trabajo: "Hello.java", "Hello.class" y "README.md". No deseamos hacer seguimiento de archivos ".class", ya que pueden reproducirse desde ".java"

# Git. Empezando con un repositorio local

## Inicializar un nuevo Git Repo (git init)

Para administrar un proyecto bajo Git, ejecute **"git init"** en el directorio raíz del proyecto (es decir, "hello-git") (a través de "Git Bash" para Windows, o "Terminal" para Ubuntu / Mac):

```
// Change directory to the project directory
$ cd /path-to/hello-git

// Initialize Git repo for this project
$ git init
Initialized empty Git repository in /path-to/hello-git/.git/

$ ls -al
drwxr-xr-x  1 xxxxx  xxxxx  4096 Sep 14 14:58 .git
-rw-r--r--  1 xxxxx  xxxxx  426 Sep 14 14:40 Hello.class
-rw-r--r--  1 xxxxx  xxxxx  142 Sep 14 14:32 Hello.java
-rw-r--r--  1 xxxxx  xxxxx   66 Sep 14 14:33 README.md
```

# Git. Empezando con un repositorio local

Se creará un **subdirectorio oculto llamado ".git"** en el directorio raíz de tu proyecto (como se muestra en el listado "ls -a" anterior), que contiene TODOS los datos relacionados con Git.

Ten en cuenta que CADA repositorio de Git está asociado con un directorio de proyecto (y sus subdirectorios). El repositorio de Git está completamente contenido dentro del directorio del proyecto.

Por lo tanto, es seguro copiar, mover o renombrar el directorio del proyecto. Si su proyecto usa más de un directorio, puedes crear un repositorio de Git para CADA directorio, o usar enlaces simbólicos para vincular los directorios, o ... (?!).

# Git. Empezando con un repositorio local

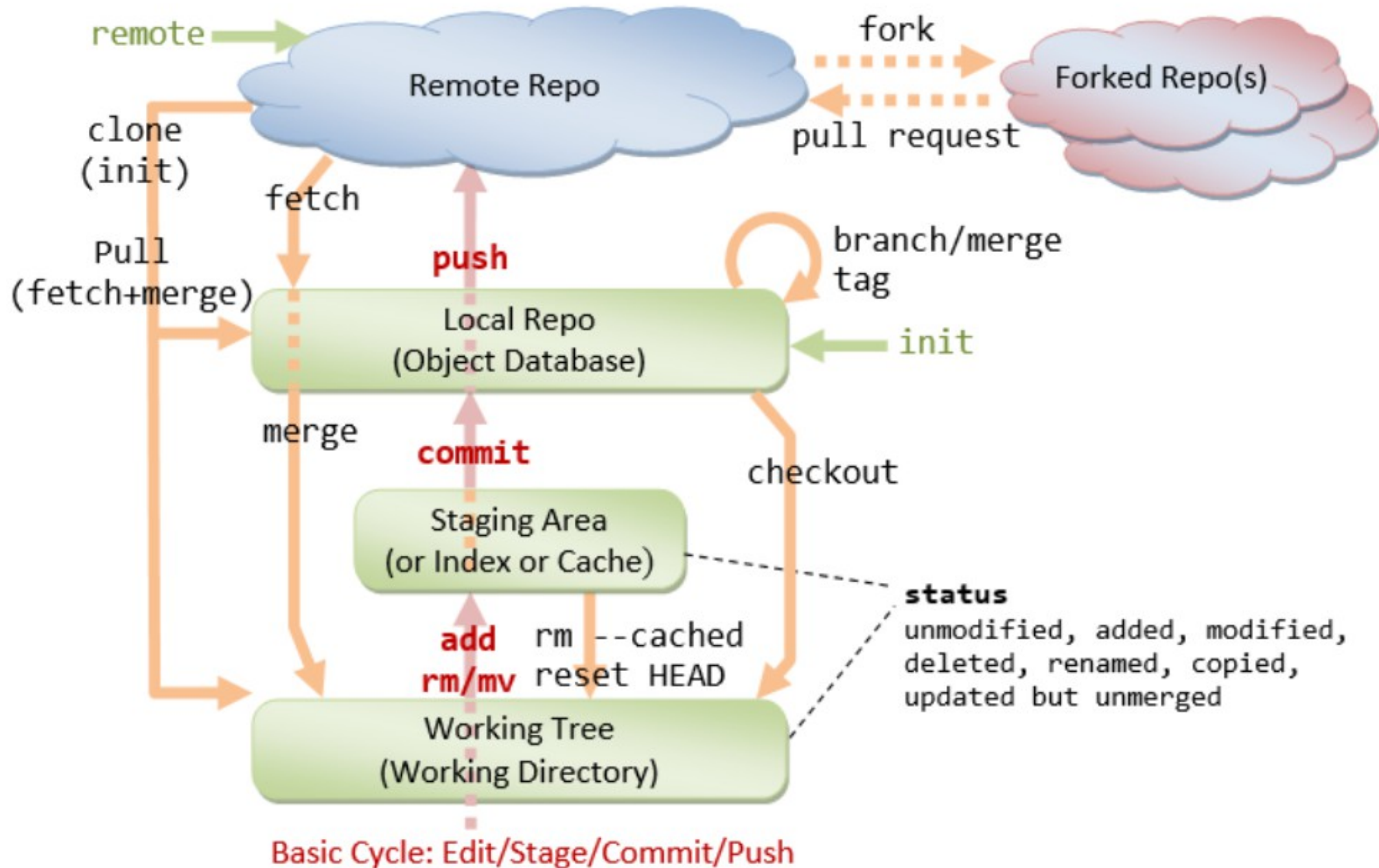
```
git init
```

Crea un repositorio local vacío. Un lienzo en blanco, por así decirlo.

- 1 Nos paramos en el directorio que queremos convertir en un repositorio.
- 2 Ejecutamos `git init`.

Esto crea un subdirectorio `.git` que tiene todos los archivos necesarios de Git.

N



# Modelo de almacenamiento Git

El **repositorio local después de "git init" está vacío.**  
Necesitas depositar explícitamente los archivos en el repositorio.

Antes de continuar, es importante subrayar que Git gestiona los cambios en los archivos entre los **commit** (confirmaciones).

En otras palabras, es un sistema de control de versiones que te permite realizar un seguimiento de los cambios en el archivo cuando hacemos commit(confirmar)

# Preparando cambios en archivos para el seguimiento (git add <file>...)

Ejecute el comando "**git status**" para mostrar el estado de los archivos:

```
$ git status
On branch master
Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
    Hello.java
    README.md
nothing added to commit but untracked files present (use "git add" to track)
```

Por defecto, comenzamos en una rama llamada "**master**". Vamos a discutir "rama" más tarde.

# Git. Empezando con un

En Git, los archivos en el directorio trabajo o tienen **seguimiento** o **no tienen seguimiento**.

Actualmente, los 3 archivos están sin seguimiento.

Para preparar un nuevo archivo para el seguimiento, usa el comando "**git add <file> ...**".

```
// Add README.md file
$ git add README.md

$ git status
On branch master
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
    Hello.java

// You can use wildcard * in the filename
// Add all Java source files into Git repo
$ git add *.java

// You can also include multiple files in the "git add"
// E.g.,
// git add Hello.java README.md

$ git status
On branch master
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   Hello.java
    new file:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
```

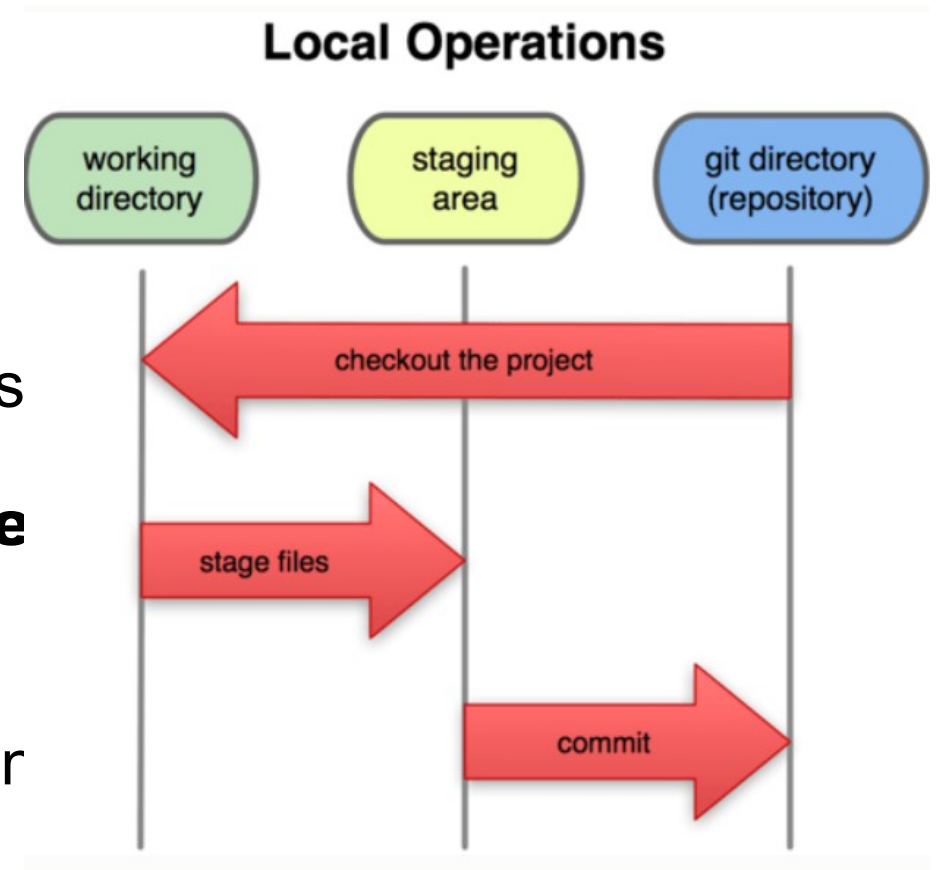


# Git. Empezando con un repositorio local

El comando "**git add <file> ...**" toma uno o más nombres de archivo o rutas con posiblemente un patrón de comodines.

También puedes usar "**git add**". para agregar todos los archivos en el directorio actual (y todos los subdirectorios). Pero esto incluirá **"Hello.class", que no deseamos hacer e seguimiento.**

Cuando se agrega un nuevo archivo, está preparado (o se indexa o almacena en caché) en el área de preparación (como se muestra en el modelo de almacenamiento GIT), pero aún NO se ha confirmado (committed)



# Git. Empezando con un repositorio local

`git add`

Una vez que tenemos cambios hechos, tenemos que marcarlos como preparados antes de confirmarlos. En la jerga de Git, decimos que pasamos los cambios a *staged*.

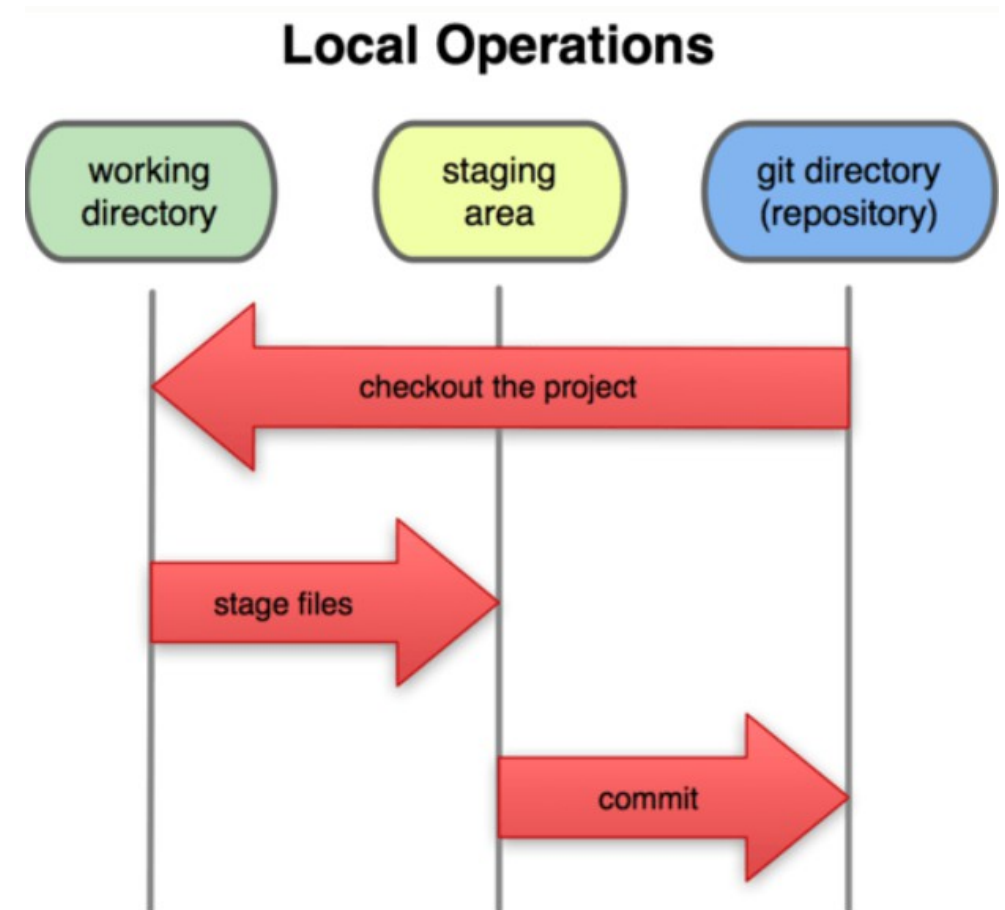
- 1 Creamos/modificamos el archivo en cuestión.
- 2 Ejecutamos `git add [nombre del archivo]`.

# Git. Empezando con un repositorio local

Git usa dos etapas para confirmar(commit) cambios en el archivo:

- **"git add <file>"** para preparar los cambios de archivos en el área de preparación, y
- **"git commit"** para confirmar TODOS los cambios de archivos del área de preparación al repositorio local.

El área de preparación le permite agrupar cambios de archivos relacionados y confirmarlos juntos.



# Confirmando cambios en archivos (git commit)

El comando **"git commit"** confirma TODOS los cambios de archivos en el área de preparación. Utiliza una opción -m para proporcionar un mensaje para la confirmación (commit)

```
$ git commit -m "First commit" // -m to specify the commit message
[master (root-commit) 858f3e7] first commit
2 files changed, 8 insertions(+)
create mode 100644 Hello.java
create mode 100644 README.md

// Check the status
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
nothing added to commit but untracked files present (use "git add" to track)
```

# Confirmando cambios en archivos (git commit)

git commit

Una vez que tenemos ciertos cambios en *staged*, podemos confirmarlos ejecutando `git commit -m [mensaje]`.

Donde [mensaje] es una breve descripción de los cambios que acabamos de confirmar.

# Viendo los datos de confirmación (git log)

Git registra varios metadatos en cada commit, que incluye un mensaje de registro, una marca de **hora**, el nombre de usuario del **autor** y el **correo electrónico** (establecido durante la personalización).

Puede usar **"git log"** para listar los datos de confirmación; o "git log --stat" para ver las estadísticas del archivo

```
$ git log
commit 858f3e71b95271ea320d45b69f44dc55cf1ff794
Author: username <email>
Date:   Thu Nov 29 13:31:32 2012 +0800
    First commit

$ git log --stat
commit 858f3e71b95271ea320d45b69f44dc55cf1ff794
Author: username <email>
Date:   Thu Nov 29 13:31:32 2012 +0800
    First commit
Hello.java | 6 ++++++
README.md  | 2 ++
2 files changed, 8 insertions(+)
```

# Viendo los datos de confirmación (git log)

Cada **commit** se **identifica** mediante un **código hash** SHA-1 de 40 dígitos hexadecimales.

Pero usualmente utilizamos los primeros **7 dígitos hexadecimales** para hacer referencia a un commit, como se resalta.

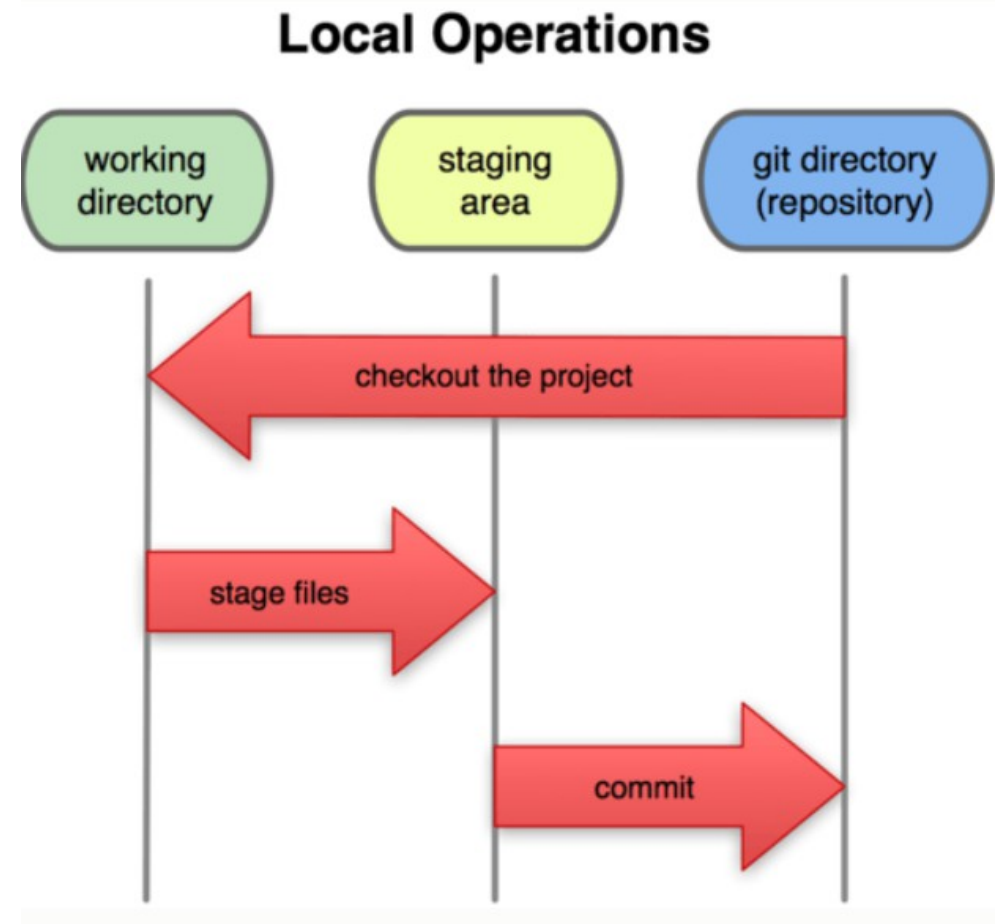
Para ver los detalles de la confirmación, use **"git log -p"**, que enumera todos los parches (o cambios).

```
$ git log -p
commit 858f3e71b95271ea320d45b69f44dc55cf1ff794
Author: username <email>
Date:   Thu Nov 29 13:31:32 2012 +0800
    First commit
diff --git a/Hello.java b/Hello.java
new file mode 100644
index 0000000..dc8d4cf
--- /dev/null
+++ b/Hello.java
@@ -0,0 +1,6 @@
+// Hello.java
+public class Hello {
+    public static void main(String[] args) {
+        System.out.println("Hello, world from GIT!");
+    }
+}
diff --git a/README.md b/README.md
new file mode 100644
index 0000000..9565113
--- /dev/null
+++ b/README.md
@@ -0,0 +1,2 @@
+// README.md
+This is the README file for the Hello-world project.
```

# Flujo de trabajo

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu **directorio de trabajo**.
2. Preparas los archivos, añadiéndolos a tu **área de preparación**.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git (**repositorio local**)





# Información de estado de archivos (git status)

¿Está preparado, confirmado o ninguna de las dos?

```
git status
```

¡No es lo mismo!

Las modificaciones que hacemos pueden estar en **4 estados** distintos:

- **Sin seguimiento (untracked)**: archivos que nunca fueron agregados al repositorio, por ejemplo archivos nuevos.

Output de ejemplo

```
Untracked files:
  README
```

# Información de estado de archivos (git status)

git status

¡No es lo mismo!

Las modificaciones que hacemos pueden estar en **4 estados** distintos:

- **Modificado (modified)**: las modificaciones todavía no están marcadas como *staged*.

Output de ejemplo

```
Changes not staged for commit:  
  modified:   README
```

# Información de estado de archivos (git status)

git status

¡No es lo mismo!

Las modificaciones que hacemos pueden estar en **4 estados** distintos:

- **Preparado (staged)**: las modificaciones están en *staged* e irán en la próxima *confirmación de cambios (commit)*.

Output de ejemplo

```
Changes to be committed:  
  new file:   README
```

# Información de estado de archivos (git status)

git status

¡No es lo mismo!

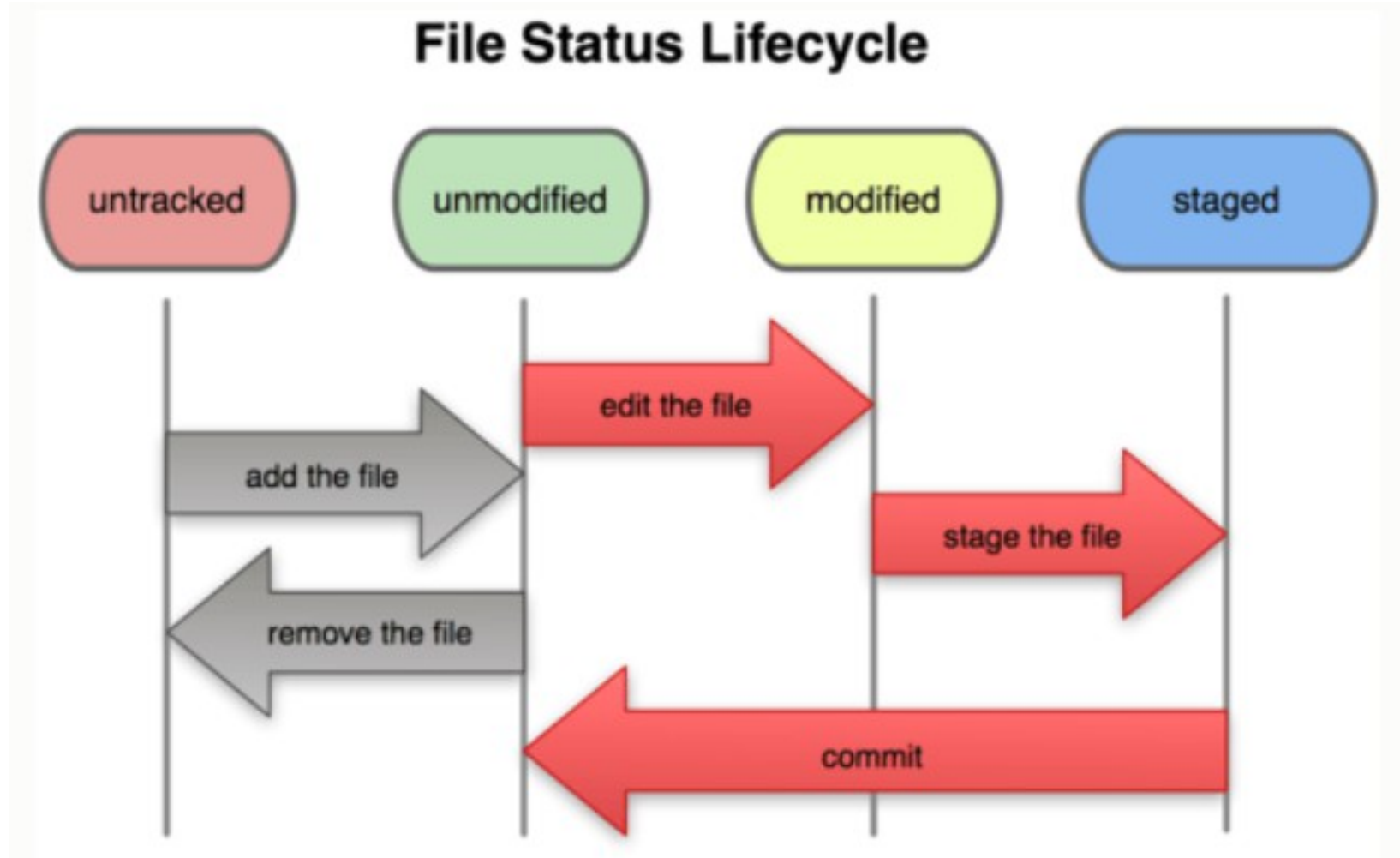
Las modificaciones que hacemos pueden estar en **4 estados** distintos:

- **Confirmado (committed)**: las modificaciones están guardadas con un *mensaje* que explica los cambios realizados.

Output de ejemplo

```
nothing to commit, working directory clean
```

# Ciclo de vida del estado de un archivo



# Información de estado de archivos (git status)

Por ejemplo, realiza algunos cambios en el archivo "Hello.java" y verifica el estado nuevamente:

```
// Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world from GIT!");
        System.out.println("Changes after First commit!");
    }
}
```

# Información de estado de archivos (git status)

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   Hello.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
no changes added to commit (use "git add" and/or "git commit -a")
```

El archivo "Hello.java" está marcado como modificado en el directorio de trabajo (en "Cambios no preparados para confirmación"), pero no modificado en el área de preparación (no se muestra "Cambios por confirmar").

# Información de estado de archivos (git status)

Puede inspeccionar todos los cambios sin preparar usando el comando "**git diff**" (o "**git diff <file>**" para el archivo especificado). Muestra los cambios del archivo en el directorio de trabajo desde la última confirmación (commit):

```
$ git diff
diff --git a/Hello.java b/Hello.java
index dc8d4cf..f4a4393 100644
--- a/Hello.java
+++ b/Hello.java
@@ -2,5 +2,6 @@
 public class Hello {
     public static void main(String[] args) {
         System.out.println("Hello, world from GIT!");
+        System.out.println("Changes after First commit!");
     }
 }
```



# Información de estado de status)

```
$ git diff
diff --git a/Hello.java b/Hello.java
index dc8d4cf..f4a4393 100644
--- a/Hello.java
+++ b/Hello.java
@@ -2,5 +2,6 @@
 public class Hello {
     public static void main(String[] args) {
         System.out.println("Hello, world from GIT!");
+        System.out.println("Changes after First commit!");
     }
 }
```

La versión anterior (a partir de la última confirmación) está marcada como --- y la nueva como +++. Cada parte de los cambios está delimitada por

"@@ - <old-line-number>, <number-of-lines> + <new-line-number>, <number-of-lines> @@".

Las líneas agregadas se marcan como + y se eliminan como -.

En la salida anterior, se comparan la versión anterior (a partir de la última confirmación) de la línea 2 para 5 líneas y la versión modificada de la línea 2 para 6 líneas. Se agrega una línea (marcada como +).

# Información de estado de archivos (git status)

Prepare los cambios de "Hello.java" ejecutando el comando "**git add <file> ...**":

```
$ git add Hello.java

$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   Hello.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
```

# Información de estado de archivos (git status)

Ahora, está marcado como modificado en el área de preparación ("Cambios por confirmar"), pero no modificado en el directorio de trabajo (no se muestra "Cambios no preparados para confirmación").

Ahora, los cambios se han puesto en el área de preparación. Al ejecutar "**git diff**" para mostrar los resultados de los cambios no preparados muestra una salida vacía.

Puede inspeccionar el cambio preparado (en el área de preparación) mediante el comando "**git diff - staged**":

```
// List all "unstaged" changes for all files (in the working tree)
$ git diff
    // empty output - no unstaged change

// List all "staged" changes for all files (in the staging area)
$ git diff --staged
diff --git a/Hello.java b/Hello.java
index dc8d4cf..f4a4393 100644
--- a/Hello.java
+++ b/Hello.java
@@ -2,5 +2,6 @@
 public class Hello {
     public static void main(String[] args) {
         System.out.println("Hello, world from GIT!");
+        System.out.println("Changes after First commit!");
     }
 }

// The "unstaged" changes are now "staged".
```

# Git commit

Confirmar TODOS los cambios de archivos en el área de preparación través de "**git commit**":

```
$ git commit -m "Second commit"
[master 96efc96] Second commit
 1 file changed, 1 insertion(+)

$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
nothing added to commit but untracked files present (use "git add" to track)
```

# git status

Una vez que se **confirman** los cambios en el archivo, se marca como **no modificado** en el **área de preparación** (no se muestra "Cambios por confirmar").

Tanto "**git diff**" como "**git diff --staged**" devuelven una salida vacía, lo que indica que no hay cambios "sin preparar" y ni "preparados".

Los **cambios preparados** se **borran** cuando se **confirman** los cambios; mientras que los **cambios no preparados** se **borran** cuando los cambios se **envían al área de preparación**.

# Git log

Ejecuta **"git log"** para listar todos las confirmaciones (commit)

```
$ git log
commit 96efc96f0856846bc495aca2e4ea9f06b38317d1
Author: username <email>
Date:   Thu Nov 29 14:09:46 2012 +0800
    Second commit

commit 858f3e71b95271ea320d45b69f44dc55cf1ff794
Author: username <email>
Date:   Thu Nov 29 13:31:32 2012 +0800
    First commit
```

# El archivo .gitignore

Todos los archivos en el directorio de Git se les hace un seguimiento o no se les hace un seguimiento.

Para hacer que los archivos (como **.class**, **.o**, **.exe** que podrían reproducirse desde la fuente) se **ignoren y eliminarlos de la lista de archivos sin seguimiento**, crea un archivo **".gitignore"** en el directorio de su proyecto, que enumere los archivos a ser ignorado, de la siguiente manera:

```
# .gitignore

# Java class files
*.class

# Executable files
*.exe

# Object and archive files
# Can use regular expression, e.g., [oa] matches either o or a
*.[oa]

# temp sub-directory (ended with a directory separator)
temp/
```

# El archivo .gitingnore

Ahora, ejecuta el comando "**git status**" para verificar los archivos sin seguimiento.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitingnore
nothing added to commit but untracked files present (use "git add" to track)
```

Ahora, "Hello.class" no se muestra en "Archivos sin seguimiento".



# El archivo .gitignore

Normalmente, también le hacemos seguimiento y confirmamos el archivo .gitignore.

```
$ git add .gitignore

$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   .gitignore

$ git commit -m "Added .gitignore"
[master 711ef4f] Added .gitignore
 1 file changed, 14 insertions(+)
 create mode 100644 .gitignore

$ git status
On branch master
nothing to commit, working directory clean
```

# Actividad guiada. Deshacer cambios

## **Preparando el repositorio para hacer ejercicios de deshacer cambios**

1. Comprobar el estado del repositorio (**git status**)
2. Agregar el fichero idiomas/idiomas1.txt.
3. Comprobar el estado del repositorio (**git status**)
4. Añadir los cambios a la zona de preparación (**git add**) y hacer un commit con el mensaje "idiomas " (**git commit -m "idiomas"**)
5. Comprobar el estado del repositorio (**git status**)

# Actividad guiada. Git. Deshacer cambios

## Antes de llevarlos al área de preparación

1. Comprobar el estado del repositorio (**git status**)
2. Eliminar la última línea del fichero Hello.java y guardarlo
3. Comprobar el estado del repositorio (**git status**)
4. Deshacer los cambios realizados en el fichero Hello.java para volver a la versión anterior del fichero (**git checkout - - Hello.java**)
5. Volver a comprobar el estado del repositorio (**git status**)

<https://aprendeconalf.es/docencia/git/manual/deshacer-cambios/>

# Reemplaza cambios locales (git checkout)

En caso de que hagas algo mal (lo que seguramente nunca suceda ;) puedes reemplazar cambios locales usando el comando

```
git checkout -- <filename>
```

Este comando reemplaza los cambios en tu directorio de trabajo con el último contenido commit. Los cambios que ya han sido agregados al área de preparación, así como también los nuevos archivos, se mantendrán sin cambio.

# Actividad guiada. Git. Deshacer cabios

## Antes de la fase de confirmación

1. Comprobar el estado del repositorio (**git status**)
2. Eliminar la última línea del fichero Hello.java y guardarlo
3. Añadir los cambios a la zona de preparación (**git add**)
4. Comprobar el estado del repositorio (**git status**)
5. Quitar los cambios de la zona de preparación, pero mantenerlos en el directorio de trabajo (**git reset Hello.java**)
6. Comprobar el estado del repositorio (**git status**)
7. Deshacer los cambios realizados en el fichero para volver a la versión anterior del fichero. (**git checkout - - Hello.java**)
8. Comprobar el estado del repositorio (**git status**)

# Actividad guiada. Git. Deshacer cambios

## Antes de la fase de confirmación

1. Comprobar el estado del repositorio (**git status**)
2. Eliminar la última línea del fichero Hello.java y guardarlo
3. Añadir los cambios a la zona de preparación (**git add**)
4. Comprobar el estado del repositorio (**git status**)
5. Quitar los cambios de la zona de preparación, pero mantenerlos en el directorio de trabajo (**git reset Hello.java**)
6. Comprobar el estado del repositorio (**git status**)
7. Deshacer los cambios realizados en el fichero para volver a la versión anterior del fichero. (**git checkout - - Hello.java**)
8. Comprobar el estado del repositorio (**git status**)

# Actividad guiada. Git. Deshacer cambios

## Antes de la fase de confirmación

1. Eliminar la última línea del fichero Hello.java y guardarlo.
2. Eliminar el fichero idiomas/idiomas1.txt.
3. Añadir un fichero nuevo idiomas/idiomas2.txt. vacío.
4. Añadir los cambios a la zona de preparación (**git add .**)
5. Comprobar de nuevo el estado del repositorio (**git status**)
6. Quitar los cambios de la zona de preparación, pero mantenerlos en el directorio de trabajo. (**git reset**)
7. Comprobar de nuevo el estado del repositorio. (**git status**)
8. Deshacer los cambios realizados para volver a la versión del repositorio. (**git checkout -- .**)
9. Volver a comprobar el estado del repositorio (**git status**)

# Actividad guiada. Git. Deshacer cambios

## Después de la fase de confirmación

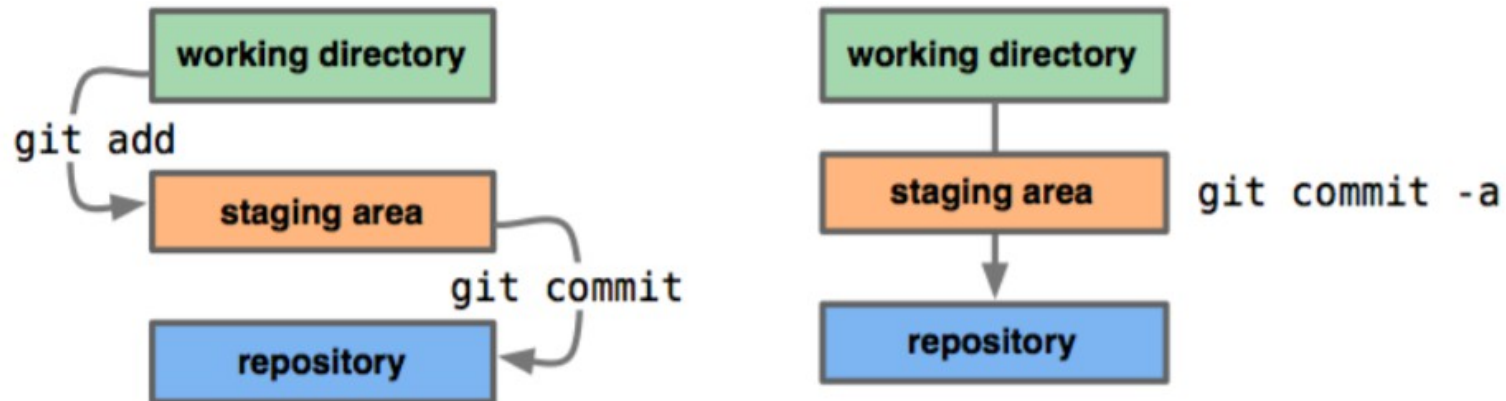
1. Comprobar el estado del repositorio (**git status**)
2. Eliminar la última línea del fichero Hello.java y guardarlo.
3. Eliminar el fichero idiomas/idiomas1.txt
4. Añadir los cambios a la zona de preparación y hacer un commit con el mensaje “borrado accidental” (git **commit -a -m** “borrado accidental”)
5. Comprobar el historial y el estado del repositorio. (git status y git log)
6. Deshacer el último commit pero mantener los cambios anteriores en el directorio de trabajo y la zona de preparación (git **reset - -soft** HEAD~1)
7. Comprobar el historial y el estado del repositorio. (git status y git log)
8. Volver a hacer el commit con el mismo mensaje de antes (git commit -m “borrado accidental”)
9. Comprobar el historial y el estado del repositorio. (git status y git log)
10. Deshacer el último commit y los cambios anteriores del directorio de trabajo volviendo a la versión anterior del repositorio. (git **reset - -hard** HEAD~1)
11. Comprobar de nuevo el historial y el estado del repositorio. (git status y git log)



# Más sobre Confirmar cambios (git COMMIT)

## **commit -a -m “mensaje”**

El parámetro **-a** hace un **git add** antes de hacer commit de todos los archivos modificados o borrados (de los nuevos no), con lo que nos ahorramos un paso



# Más sobre Confirmar cambios (git COMMIT)

La he liado en el mensaje del último commit!!!

`git commit - - amend -m "new commit message"`

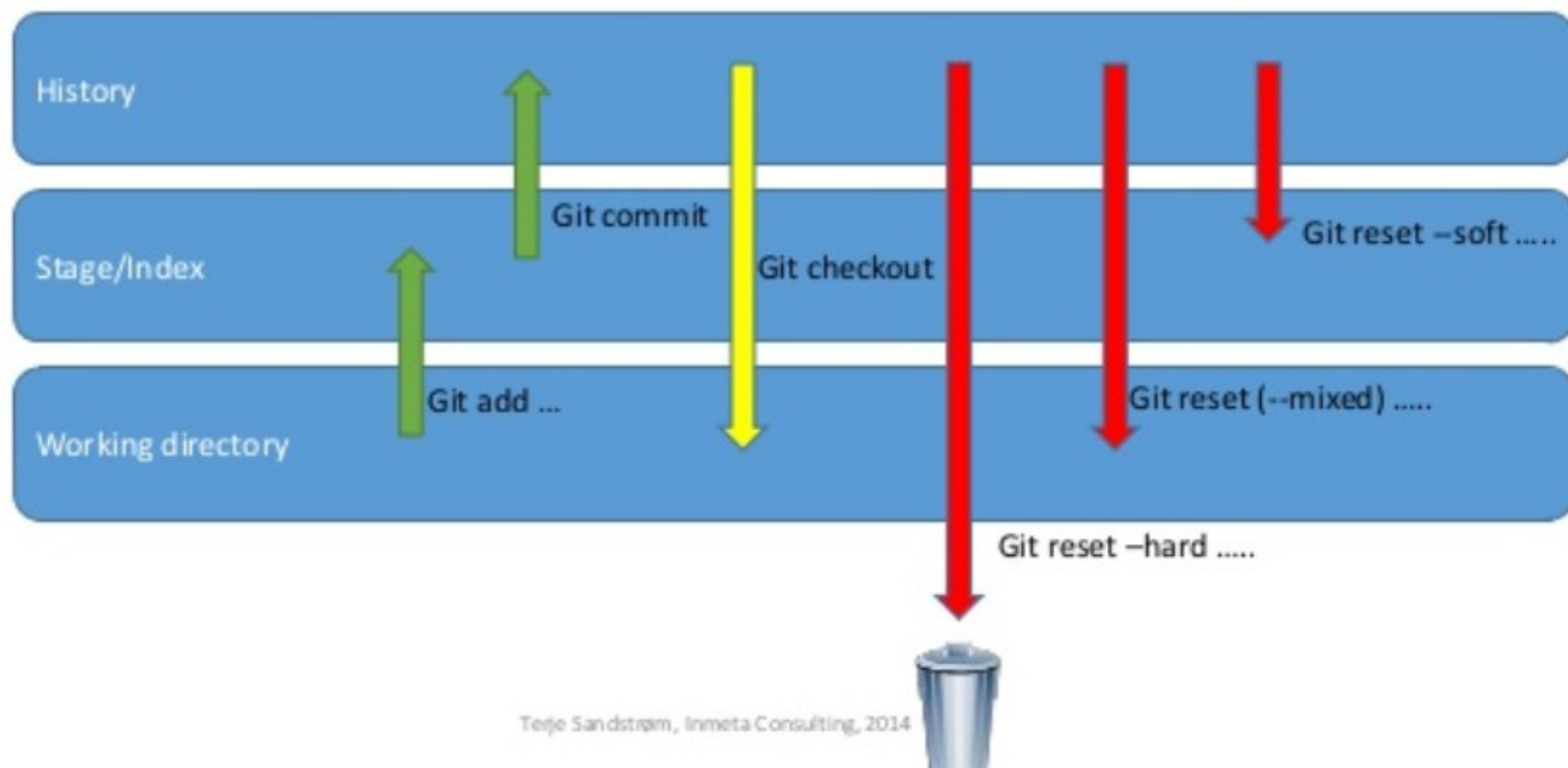
# git RESET

Eliminar un archivo del área de preparación sin perder las modificaciones en el área de trabajo

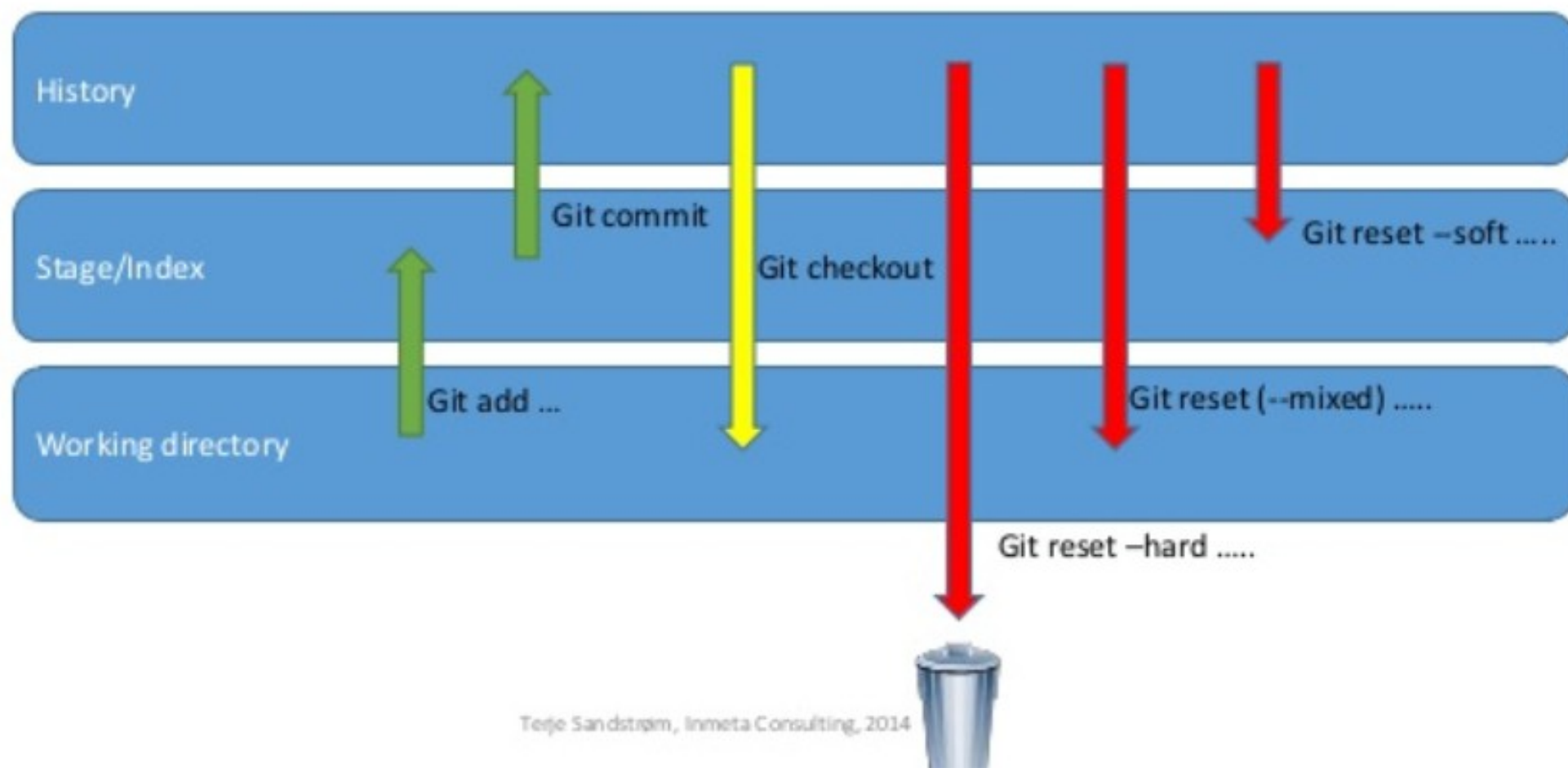
**git reset archivo**

Útil por si no queremos hacer commit de este archivo

# Git tree movements visualized

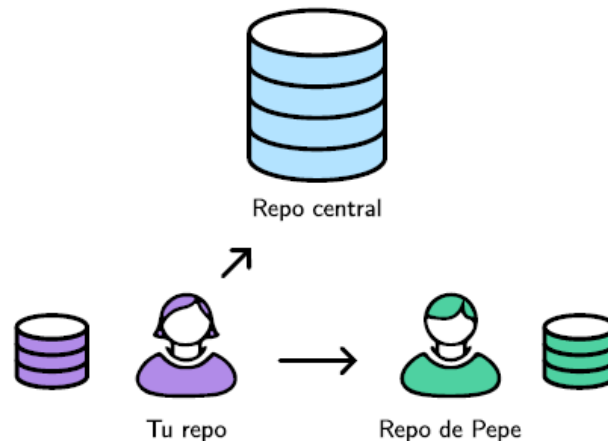


# Git tree movements visualized




# Colaborando con otras personas. Repositorios remotos

Los repositorios remotos son *copias* de nuestro proyecto a las cuales accedemos a través de Internet. Puede haber varios, cada uno de los cuales puede ser de solo lectura o de lectura/escritura, según los permisos que tengamos.



Colaborar con otros implica gestionar estos repositorios remotos, y mandar (**push**) y recibir (**pull**) datos de ellos cuando necesites compartir cambios.

# Git. Configurando un repositorio remoto

1. Regístrate en un **host GIT**, como
  -  **Github** <https://github.com/signup/free> (ilimitado para proyectos públicos; tarifa para proyectos privados); o
  -  **BitBucket** <https://bitbucket.org/> (Usuarios ilimitados para proyectos públicos; 5 usuarios gratuitos para proyectos privados; limitado para el Plan Académico); entre otros.
2. Inicia una sesión en el host GIT. Crea un **nuevo repositorio** remoto llamado "**prueba**".

# Git. Configurando un repositorio remoto

3. En tu repositorio local (continuemos trabajando en nuestro proyecto "hello-git"), configura el nombre y la URL del repositorio remoto a través del comando "**git remote add <remote-name> <remote-url>**".

Por convención, nombraremos nuestro repositorio remoto como "**origen**". Puede encontrar la URL de un repositorio remoto desde el host Git. La URL puede tomar la forma de **HTTPS** o **SSH**. Utilice HTTPS para la simplicidad.



# Git. Configurando un repositorio remoto

```
// Change directory to your local repo's working directory
$ cd /path-to/hello-git

// Add a remote repo called "origin" via "git remote add <remote-name> <remote-url>"
// For examples,
$ git remote add origin https://github.com/your-username/test.git           // for GitHub
$ git remote add origin https://username@bitbucket.org/your-username/test.git // for Bitbucket
```

# Git. Configurando un repositorio remoto

Puedes enumerar todos los nombres remotos y sus URL correspondientes a través de "git remote -v", por ejemplo,

```
// List all remote names and their corresponding URLs  
$ git remote -v  
origin  https://github.com/your-username/test.git (fetch)  
origin  https://github.com/your-username/test.git (push)
```

Ahora, puede administrar la conexión remota, utilizando un nombre simple en lugar de la URL compleja.

# Git. Configurando un repositorio remoto

4. Empuja las confirmaciones (commit) **del repositorio local al repositorio remoto** a través de "**git push -u <nombre-remoto> <local-branch-name>**".

Por convención, la rama principal de nuestro repositorio local se llama "**master**" (como se ve en la salida anterior de "git status"). Discutiremos "rama" más tarde.

```
// Push all commits of the branch "master" to remote repo "origin"
$ git push origin master
Username for 'https://github.com': *****
Password for 'https://your-username@github.com': *****
Counting objects: 10, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (10/10), 1.13 KiB | 0 bytes/s, done.
Total 10 (delta 1), reused 0 (delta 0)
To https://github.com/your-username/test.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

# Git. Configurando un repositorio remoto

```
git push
```

Para enviar los cambios **desde nuestro repositorio local a algún repositorio remoto**, ejecutamos: `git push [remoto] [branch]`.

# Git. Configurando un repositorio remoto

5. Inicia sesión en el host de GIT y seleccione "prueba" del repositorio remoto, encontrarás todos los archivos confirmados.
6. En su sistema local, realiza algunos cambios (por ejemplo, en "Hello.java"); prepara y confirma los cambios en el repositorio local; y empújalo repositorio remoto. Esto se conoce como el ciclo  
."Editar / Preparar(add) / Confirmar(commit) / Empujar(push)".

# Git. Configurando un repositorio remoto

```
// Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world from GIT!");
        System.out.println("Changes after First commit!");
        System.out.println("Changes after Pushing to remote!");
    }
}
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working dire
    modified:   Hello.java
no changes added to commit (use "git add" and/or "git commit -a")

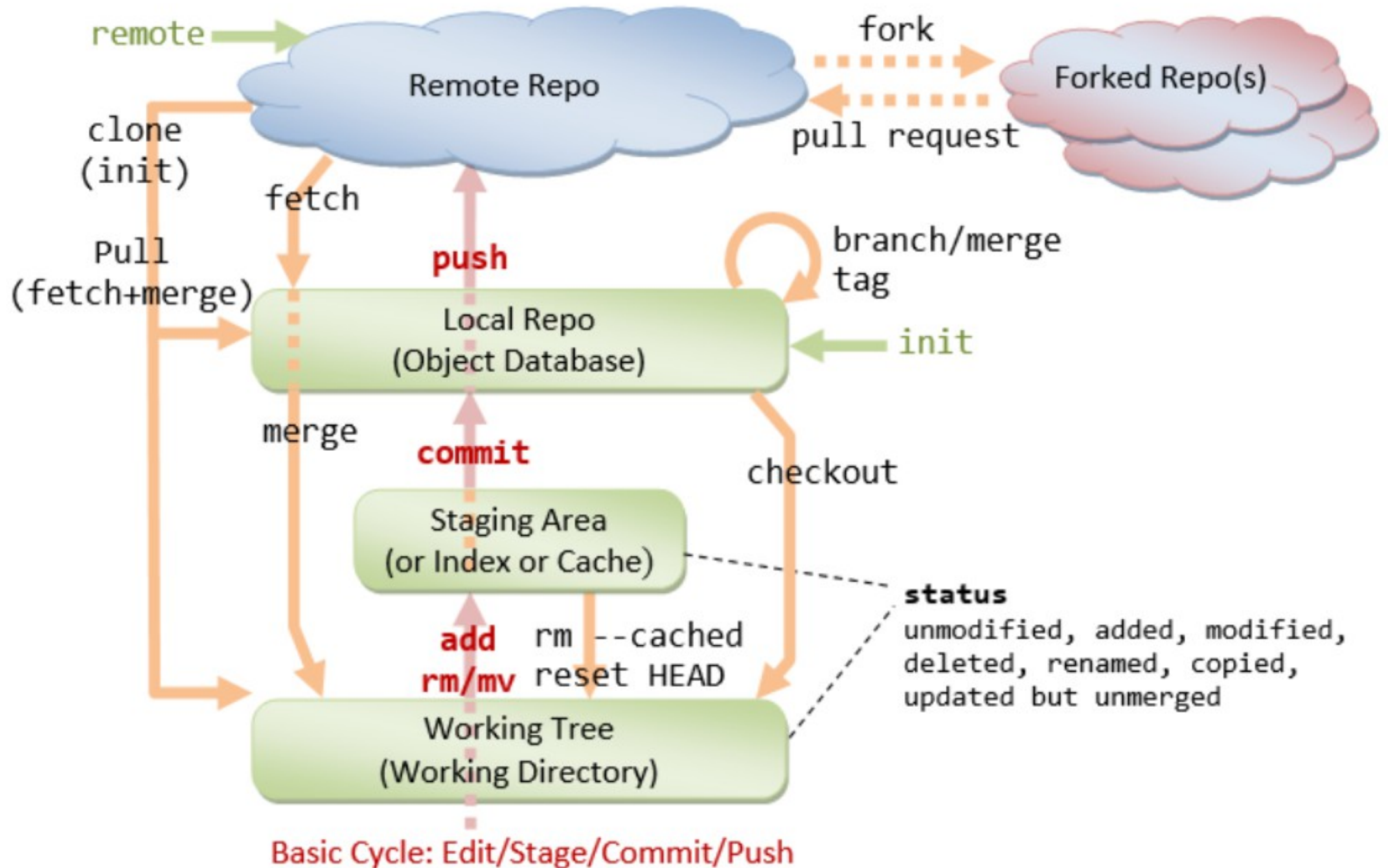
// Stage file changes
$ git add *.java

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   Hello.java

// Commit all staged file changes
$ git commit -m "Third commit"
[master 744307e] Third commit
1 file changed, 1 insertion(+)

// Push the commits on local master branch to remote
$ git push origin master
Username for 'https://github.com': *****
Password for 'https://username@github.com': *****
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 377 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/your-username/test.git
711ef4f..744307e  master -> master
```





# Clonar un proyecto de un repositorio remoto (**git clone <remote-url>**)

Como se mencionó anteriormente, puede iniciar un repositorio GIT local ejecutando "git init" en su propio proyecto o "git clone <remote-url>" para hacer una copia desde un proyecto existente.

Cualquier persona que tenga acceso de lectura a su repositorio remoto puede clonar tu proyecto. También puede clonar cualquier proyecto en cualquier repositorio remoto público.

El comando "**git clone <remote-url>**" inicializa un repositorio local y copia todos los archivos en el directorio de trabajo. Puedes encontrar la URL de un repositorio remoto desde el host Git.

# clonar

## sintaxis

```
// SYNTAX
// =====
$ git clone <remote-url>
  // <url>: can be https (recommended), ssh or file.
  // Clone the project UNDER the current directory
  // The name of the "working directory" is the same as the remote project name
$ git clone <remote-url> <working-directory-name>
  // Clone UNDER current directory, use the given "working directory" name

// EXAMPLES
// =====
// Change directory (cd) to the "parent" directory of the project directory
$ cd path-to-parent-of-the-working-directory

// Clone our remote repo "test" into a new working directory called "hello-git-cloned"
$ git clone https://github.com/your-username/test.git hello-git-cloned
Cloning into 'hello-git-cloned'...
remote: Counting objects: 13, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 13 (delta 2), reused 13 (delta 2)
Unpacking objects: 100% (13/13), done.
Checking connectivity... done.

// Verify
$ cd hello-git-cloned

$ ls -a
.  ..  .git  .gitignore  Hello.java  README.md

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

# Clonar un proyecto de un repositorio remoto

El "clon de git" crea automáticamente un nombre remoto llamado "origen" asignado a la URL remota clonada. Puedes comprobarlo a través de "**git remote -v**":

```
// List all the remote names
$ git remote -v
origin  https://github.com/your-username/test.git (fetch)
origin  https://github.com/your-username/test.git (push)
```

## Resumen del ciclo básico "Editar / preparar/ Confirmar / Empujar"

```
// Edit (Create, Modified, Rename, Delete) files,  
// which produces "unstaged" file changes.  
  
// Stage file changes, which produces "Staged" file changes  
$ git add <file> // for new and modified files  
$ git rm <file> // for deleted files  
$ git mv <old-file-name> <new-file-name> // for renamed file  
  
// Commit (ALL staged file changes)  
$ git commit -m "message"  
  
// Push  
$ git push <remote-name> <local-branch-name>
```

OR,

```
// Stage ALL files with changes  
$ git add -A // OR, 'git add --all'  
  
$ git commit -m "message"  
$ git push
```

OR,

```
// Add All and Commit in one command  
$ git commit -a -m "message"  
  
$ git push
```

# Git pull

```
git pull
```

Para traer cambios **desde un repositorio remoto a nuestro repositorio local**, ejecutamos: `git pull [remoto] [branch]`.

Veremos su uso en la siguiente actividad

# Actividad

En parejas

1. **Alumno A:** crea proyecto y empuja cambios a un **repositorio remoto**
2. **Alumno B:** **clona** proyecto
3. **Alumno A:** realiza **cambios** en el proyecto y los **empuja** al repositorio remoto

(El alumno A debe compartir el repositorio y agregar los usuarios con los que desea compartir ese repositorio remoto)

4. **Alumno B:** **trae** cambios del repositorio remoto
5. **Alumno B:** modifica archivo y **empuja** cambios al **repositorio remoto compartido**
6. **Alumno A:** **trae** cambios del repositorio remoto

# Git herramientas de interfaz gráfica

## Git-GUI (Windows)

Para mayor comodidad, Git proporciona una herramienta GUI, llamada **git-gui**, que se puede usar para realizar todas las tareas y ver el registro de confirmación gráficamente.

### Instalar "Git-Gui"

Para ejecutar git-gui, puede hacer clic derecho en la carpeta del proyecto y elegir "Git Gui"; o inicia el shell Git-bash y ejecuta el comando "git gui".

# Git herramientas de interfaz gráfica

Para ver el registro, seleccione "Repositorio" ⇒ "Visualizar el historial maestro", que inicia el "gitk". Puedes ver los detalles de cada commit.

También puede ver cada uno de los archivos a través del "Repositorio" ⇒ "Buscar archivos maestros" ⇒ Seleccione un archivo.

**Git-gui se incluye con Git.** Para iniciar git-gui, haga clic derecho en el directorio de trabajo y elija "git gui", o ejecute el comando "git gui" en el shell de Git-Bash.



# actividad

Utilizar el soporte de GIT en Netbeans siguiendo el siguiente tutorial





<https://netbeans.org/kb/docs/ide/git.html>

Utilizar el soporte de GIT en IntelliJ siguiendo el siguiente tutorial

<https://www.jetbrains.com/help/idea/set-up-a-git-repository.html#put-existing-project-under-Git>

# Y que pasa si... <BOOM!

## A veces hay conflictos

- Supongamos que dos personas ( y ) están trabajando en un mismo proyecto. Es decir, ambos tienen una *copia* en su máquina.
- Ahora imaginemos, por ejemplo, que  modifica la línea 23 del archivo README y confirma los cambios.
- Sin saberlo,  también modifica la línea 23 del archivo README, pero pone algo distinto y confirma dichos cambios.
- ¿Qué va a pasar cuando quieran compartir lo que hicieron?  
**Va a haber un conflicto**, ya que dos personas modificaron de forma distinta la misma línea.
- ¿Qué va a hacer Git?  
**Se va a quejar**. A alguno de los dos le va a tocar **incorporar a mano los cambios del otro**.

# Y que pasa si... <BOOM!

## ¿Qué hago?

- Decido cómo tiene que quedar el archivo final
- Hago add
- Despues commit normalmente, con un mensaje como 'Merge'

## Ejercicio de a 2 máquinas (preferiblemente 2 personas): 🐙 y 🤖

- 1 🐙 y 🤖: crear un repositorio local vacío.
- 2 🐙: crear un repositorio nuevo en [GitLab](#), y darle permiso a 🤖 para hacer *push*.
- 3 🐙 y 🤖: asociar el repositorio remoto recién creado.
- 4 🐙 y 🤖: crear un archivo *README* con contenidos distintos en la primera línea.
- 5 🐙: hacer *push* de los cambios al repositorio remoto.
- 6 🤖: intentar hacer *push* de los cambios al repositorio remoto. ¿Qué pasó?
- 7 🤖: bajarse los cambios del repositorio remoto. ¿Anduvo?
- 8 🤖: resolver los conflictos que haya.
- 9 🤖: añadir y confirmar el archivo que tenía conflicto.
- 10 🤖: *pushear* estos nuevos cambios.
- 11 🐙: bajarse los nuevos cambios.