

## INTRODUCCIÓN

En esta sección estudiaremos el funcionamiento de Git y diferentes criterios para realizar un correcto control de versiones.

Git es un software de control de versiones diseñado por Linus Torvalds. Fue pensado para que el mantenimiento de versiones de aplicaciones (cuando tienen un gran número de archivos de código fuente) sea eficiente y confiable. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

### ¿QUÉ ES UN SISTEMA DE CONTROL DE VERSIONES?

Imaginemos que comenzamos un nuevo proyecto que consiste en solo un archivo fuente. Lo más probable es que se puedan deshacer los cambios que se hicieron durante esa jornada.

Al día siguiente, se continúa con el desarrollo y se mejora el archivo, pero, en el intento, el programa comienza a presentar errores. Necesitamos volver a una versión anterior del proyecto, donde no existía el error. Sin embargo, al no contar con un Control de Versiones, no contamos con respaldo de las versiones previas.

Se denomina Control de Versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración de este. Una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación.

Un Sistema de Control de Versiones consiste en una especificación de un conjunto de mecanismos que garantiza el Control de Versiones, es decir las reglas y procedimientos para que se puede realizar el control de versiones.

Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas informáticas que faciliten esta gestión. Estas aplicaciones son implementaciones de las reglas y procedimientos establecidas por un Sistema de Control de Versiones; por este motivo también se las denomina Sistemas de Control de Versiones o VCS (del inglés Version Control System).

Ejemplos de este tipo de herramientas son CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar, Plastic SCM, Git, SCCS, Mercurial, Perforce, Fossil SCM, Team Foundation Server.

Además, algunos VCS están especializados para un tipo particular de aplicación, como ser los usados para el desarrollo de software, donde el control de versiones se realiza principalmente para controlar las distintas versiones del código fuente. Sin embargo, los mismos conceptos son aplicables a otros ámbitos como en trabajo de oficina tales como documentos, imágenes, sitios web, etc.

Git, entonces, ayuda con este aspecto del desarrollo. Ofrece herramientas para poder gestionar cada una de las etapas y versiones por las que va transitando un proyecto de desarrollo. Si bien el control de versiones es la principal característica de Git, actualmente, es un gran aliado en el Desarrollo Colaborativo.

### ¿QUÉ ES EL TRABAJO COLABORATIVO?

Es un modelo de desarrollo basado en la disponibilidad pública del código y la comunicación vía Internet. Este modelo se hizo popular a raíz de su uso para el desarrollo de Linux en 1991.

Tomando como contexto a Git, podríamos decir que el desarrollo colaborativo proporciona herramientas para que un gran número de individuos puedan hacer desarrollos en conjunto de una manera más fácil, menos propensa a errores y rápida de implementar.

De esta manera, siempre tenemos la opción de, por medio de algún software cliente VCS, publicar nuestro código junto con todas las etapas y versiones que nos llevó desarrollar el proyecto para que otras personas puedan sumar y aportar nuevas ideas a nuestro repositorio.

¿Qué es un repositorio? El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor. A veces se le denomina depósito o depot. Puede ser un sistema de archivos en un disco duro, un banco de datos, etc.

### EMPEZANDO A CONOCER A GIT

#### ¿CÓMO FUNCIONA GIT?: EL CONCEPTO DE GRAFO Y UNA BREVE DESCRIPCIÓN DE LA TEORÍA DE GRAFOS

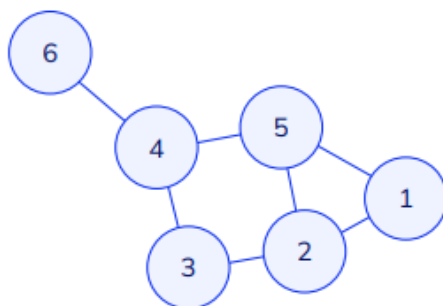
Anteriormente se ha comentado que un software de VCS es una implementación de una especificación para realizar control de versiones. Existen diversas especificaciones. Una de ellas, concretamente la que adopta el software Git está basada en el concepto del grafo.

En matemáticas y ciencias de la computación, un grafo (del griego grafos: dibujo, imagen) es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos que permiten representar relaciones binarias entre elementos de un conjunto. Son el objeto de estudio de la Teoría de Grafos.

Un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas (aristas). Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras.

Por ejemplo: una red de computadoras puede representarse y estudiarse mediante un grafo, en el que los vértices representan terminales y las aristas representan conexiones (las cuales, a su vez, pueden ser cables o conexiones inalámbricas).

Prácticamente, cualquier problema puede representarse mediante un grafo, y su estudio trasciende a las diversas áreas de las ciencias exactas y las ciencias sociales. El siguiente puede tranquilamente representar alguna situación problemática modelada mediante un grafo:

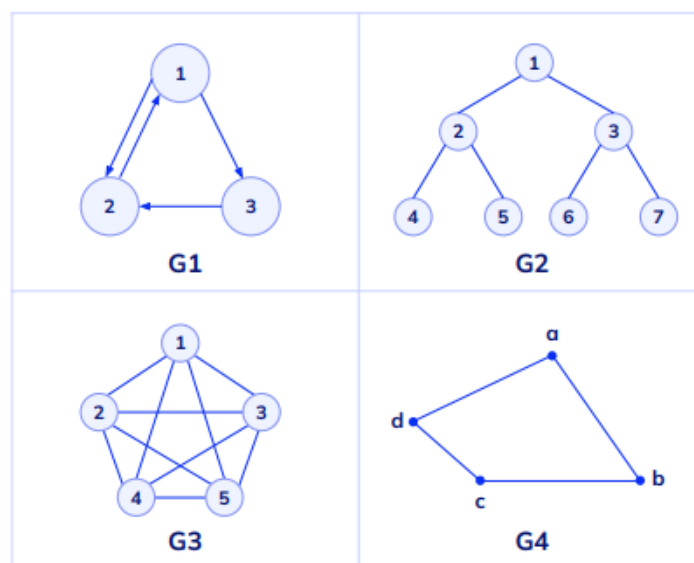


La Teoría de Grafos define formalmente un grafo  $G$  como un par ordenado  $G (V, N)$  donde:

- $V$  es un conjunto de vértices o nodos no vacío
- $N$  es un conjunto de aristas o arcos, que relacionan estos nodos

Como se ha mencionado,  $N$  consta de pares no ordenados de vértices, tales como  $\{x,y\}$  in  $N$ . Entonces se dice que  $x$  e  $y$  son adyacentes. Esto gráficamente, en el grafo se representa mediante una línea no orientada que une dichos vértices.

Si el grafo es dirigido se le llama dígrafo, se denota  $D$ , y entonces el par  $(x,y)$  es un par ordenado, esto se representa con una flecha que va de  $x$  a  $y$ , y se dice que  $(x,y)$  in  $N$ .



Un grafo está compuesto por:

- Aristas: son las líneas con las que se unen los vértices de un grafo.
- Aristas adyacentes: 2 aristas son adyacentes si convergen en el mismo vértice.
- Aristas paralelas: son dos aristas conjuntas si el vértice inicial y final son el mismo.
- Arista cíclica: es la arista que parte de un vértice para entrar en sí mismo.
- Cruce: son 2 aristas que cruzan en un mismo punto.
- Vértices: son los elementos que forman un grafo. Cada uno lleva asociada una valencia característica según la situación, que se corresponde con la cantidad de aristas que confluyen en dicho vértice.
- Camino: se denomina camino de un grafo a un conjunto de vértices interconectados por aristas. Dos vértices están conectados si hay un camino entre ellos.

Lo anterior permite que se pueda explicar la diferencia entre Git y cualquier otro VCS (Ej.: CVS, Subversion, Perforce, Bazaar, y otros). Y consiste en la forma en que procesa sus datos. Conceptualmente, **la mayoría de los otros sistemas almacenan información como una lista de cambios basados en archivos a través del tiempo**. Esto se puede observar en la figura 1.

Git no piensa ni almacena sus datos de esta manera. Git considera sus datos como un conjunto de snapshots (instantáneas) de un mini sistema de archivos. Cada vez que se confirma o se guarda el estado de un proyecto en Git, básicamente se toma una fotografía de cómo se ven

todos los archivos en ese momento y se almacena una referencia a esa instantánea. Para ser eficiente, si los archivos no han cambiado, Git no almacena el archivo nuevamente, solo un enlace al archivo idéntico anterior que ya ha almacenado. **Git maneja sus datos como una secuencia de copias instantáneas.** Esto se puede observar esquemáticamente en la figura 2.

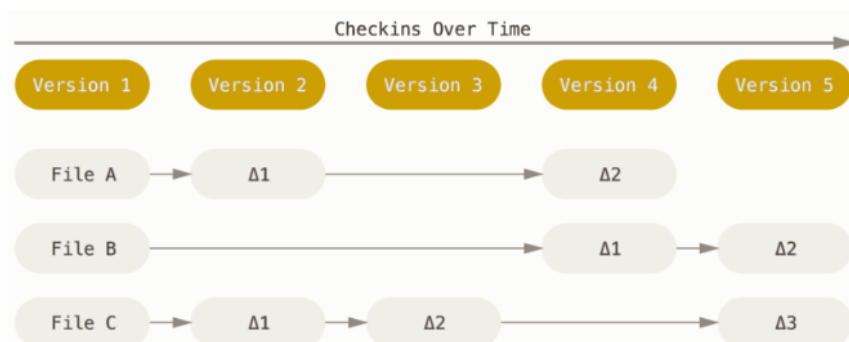


Figura 1. Almacenamiento de versiones basada en los cambios de cada archivo. Fuente: <https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git>

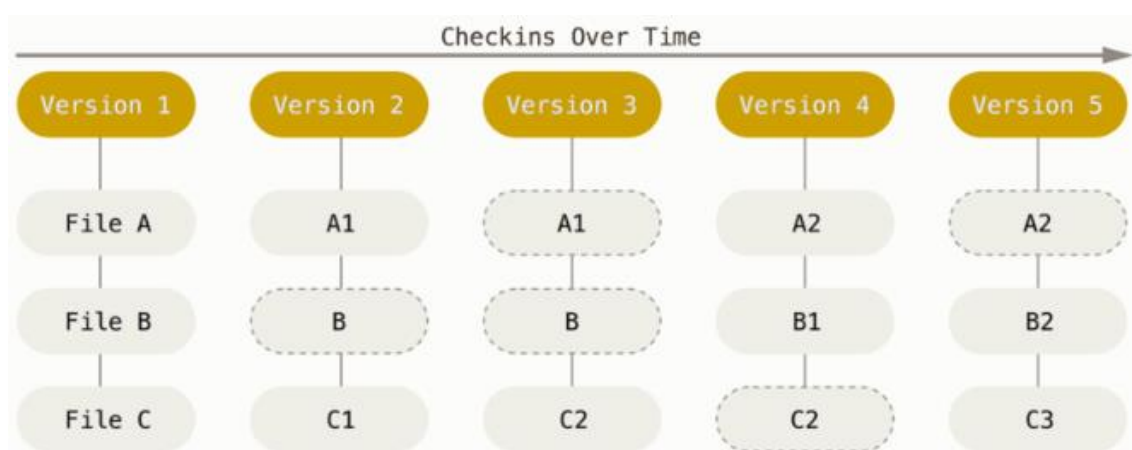


Figura 2. Almacenamiento de versiones como instantáneas del sistema de archivos en el tiempo. Fuente: <https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git>

Esta es una distinción importante entre Git y casi todos los demás VCS. Hace que Git reconsidere casi todos los aspectos del control de versiones que la mayoría de los otros sistemas copian de la generación anterior. Además, esta especificación plantea que los datos se organizan de manera similar a como lo hace un mini sistema de archivos con algunas herramientas increíblemente poderosas construidas sobre él, extendiendo las funcionalidades de un VCS.

¿Qué tiene que ver esta especificación con la teoría de grafos?

Precisamente, la gestión de estas instantáneas sigue las características de una estructura basada en grafos. Observe la figura 3. El primer nodo representa el recurso que sobre el cual se desea realizar un control de versiones. Además, se indica quien es el responsable de ese recurso (Jhon Doe). Por defecto Git denomina a las aristas que unen las versiones "oficiales" realizadas por Jhon con el nombre MASTER. Entonces el camino oficial que recorren las versiones del recurso está dadas por los nodos unidos por las aristas MASTER (color azul oscuro)

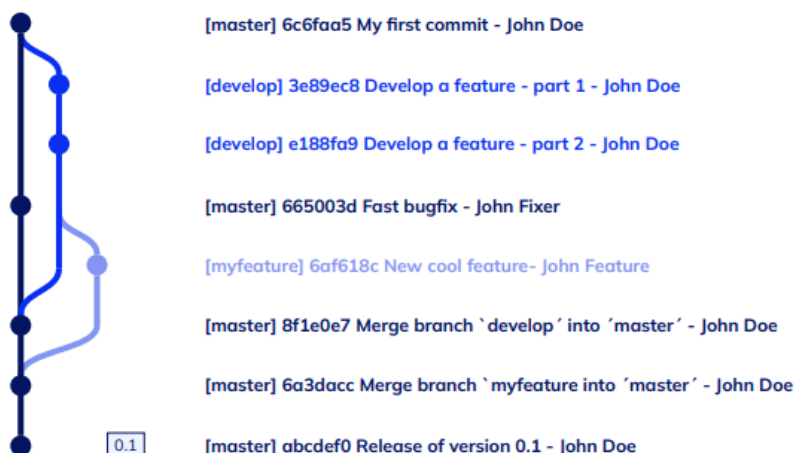


Figura 3. Ejemplo de representación mediante grafos de control de versiones que mantiene Git

Pero, además, Jhon puede generar “diferentes” versiones paralelas de sus aportes, que siguen “otros caminos”, por ejemplos la de color azul con intensidad intermedia, que representan instantáneas de cambios (versiones) realizadas sobre el nodo original.

Este concepto se denomina ramificación, y será estudiado en detalle más adelante.

Lo más importante a destacar es que gracias a la ramificación, Jhon puede gestionar de manera muy eficiente, por ejemplo, cambios que no ha decidido que sean “oficiales” hasta que esté seguro de que conformarán parte del recurso principal, lo cual en el esquema se puede observar se concreta en el tercer nodo de color azul oscuro. Allí es donde los cambios realizados de manera paralela se unen a los cambios principales. Además observe que este camino se puede seguir gracias a que Jhon le ha colocado el nombre de “Develop” a este camino o rama.

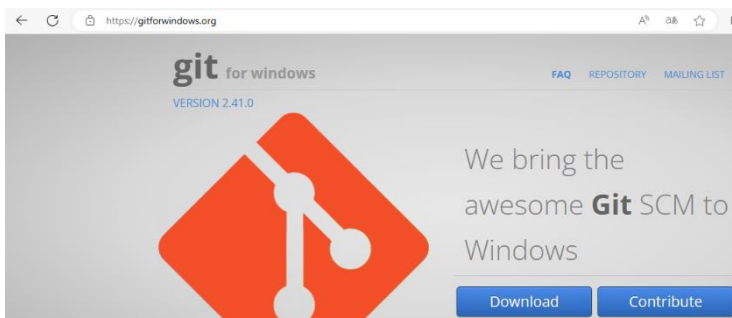
Incluso se puede crear ramas a partir de otras ramas, como la que se observa en la figura, donde la rama o camino MyFeature se ha creado a partir de Develop.

Estas ramificaciones, como puede intuir conformar el grafo de las versiones del recurso. Donde al final convergen a la última versión oficial del recurso.

## DESCARGA E INSTALACIÓN DE GIT

El siguiente contenido describe la instalación y configuración de las herramientas necesarias para trabajar con Git. Los pasos pueden variar ligeramente dado los cambios que puedan presentar el sitio de descarga, la herramienta de instalación, y algunos elementos de configuración.

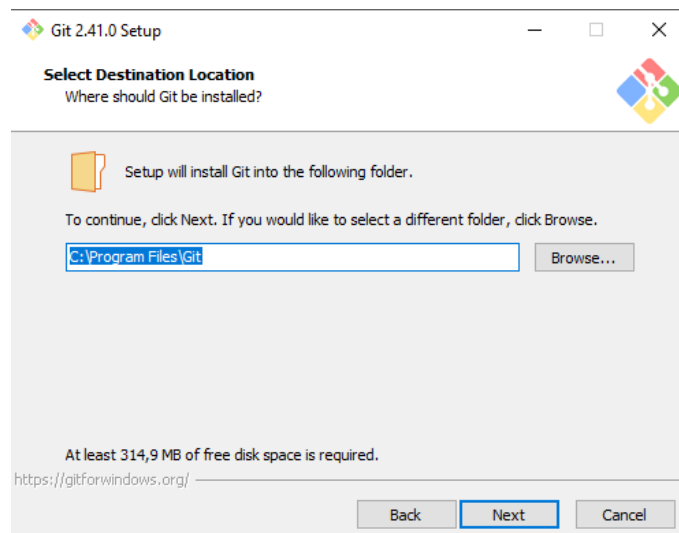
En primer lugar, acceda al sitio <http://gitforwindows.org/>. En este caso se mostrará la instalación en un entorno Windows de 64 bits. Al ingresar al sitio se elegirá el botón download, con lo cual iniciará el proceso de descarga.



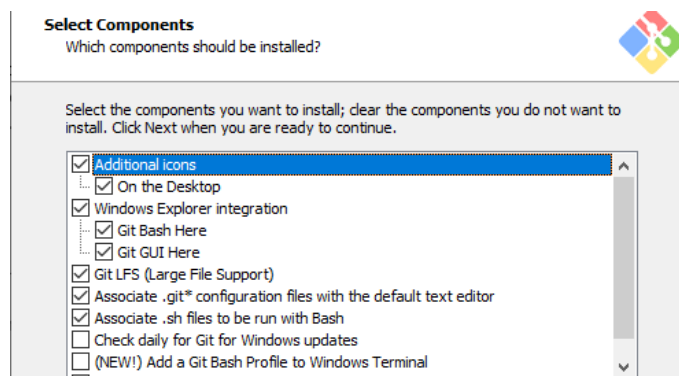
Una vez descargada la aplicación se debe ejecutar, esto generará una ventana similar a la siguiente, donde se debe aceptar los términos de licencia y uso



Al presionar el botón Next aparecerá la siguiente pantalla, donde se debe elegir el lugar donde se instalará la aplicación, que en general, salvo que el disco esté particionado y existan imitaciones con respecto a permisos, el lugar que por defecto indica el software de instalación es el adecuado

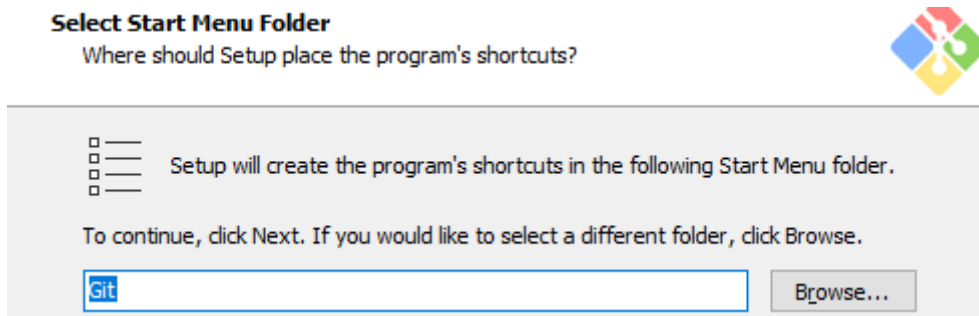


En la próxima ventana se debe seleccionar los componentes a instalar

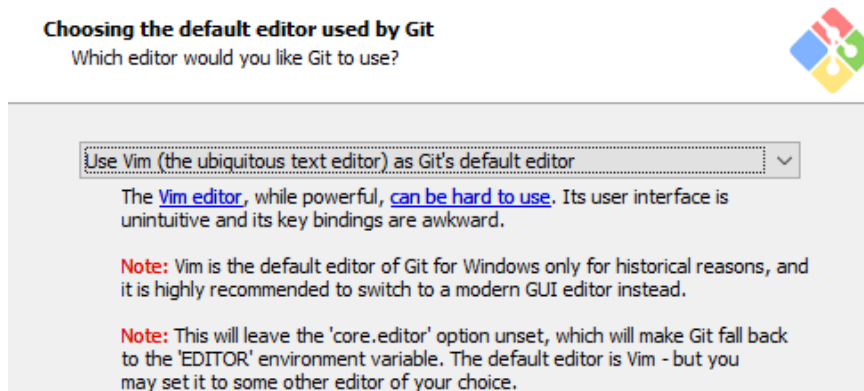




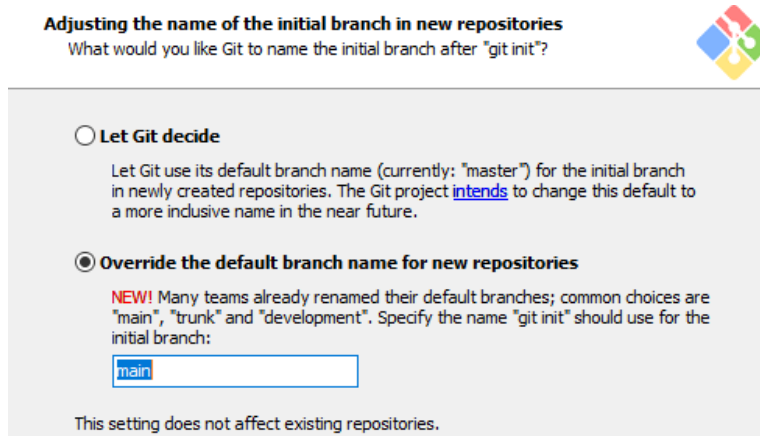
A continuación, nos consultará si deseamos que también se tenga acceso a la aplicación desde el menú de Windows



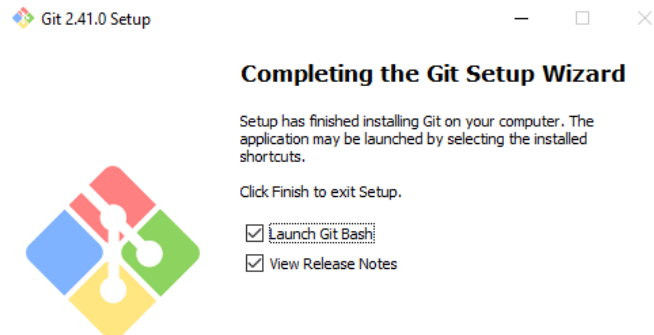
En la siguiente pantalla, nos permitirá elegir la ventana sobre la cual podremos ingresar los comandos de Git, en la cual en este caso dejaremos en Vim



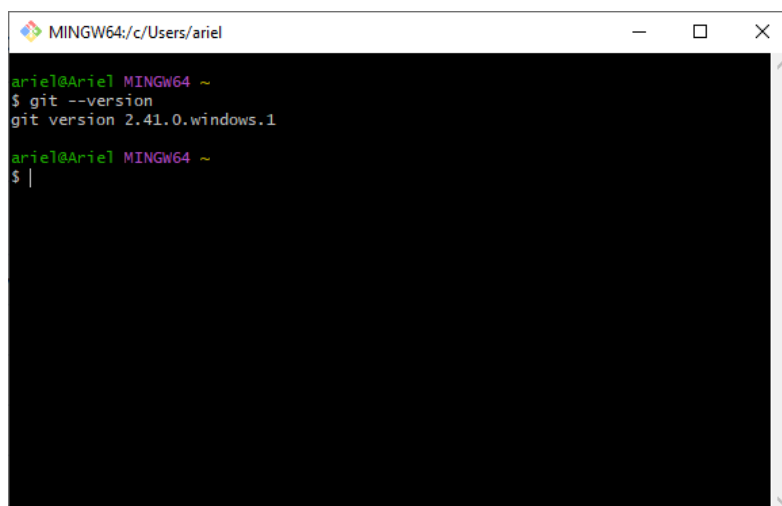
La siguiente pantalla es importante, ya que debemos elegir el nombre que por defecto Git usará para la rama principal. Como mencionamos anteriormente, al momento de que le indiquemos a Git que deba gestionar el control de versiones sobre un recurso, el camino formado por los nodos y arcos se identificará con un nombre. En el caso de Git en un principio la única opción era que este camino o rama se denomine por defecto Master. Por diferentes motivos, se ha popularizado utilizar un nuevo nombre: Main. En esta ventana, elegimos cual de estos dos nombres usaremos. En este caso, dado que usaremos más adelante Git-Hub como repositorio remoto de proyectos, el cual recomienda denominar como rama principal a Main, entonces seleccionaremos esta opción:



En las siguientes ventanas dejaremos seleccionadas las opciones por defecto. Al finalizar la ventana nos indicará lo siguiente:

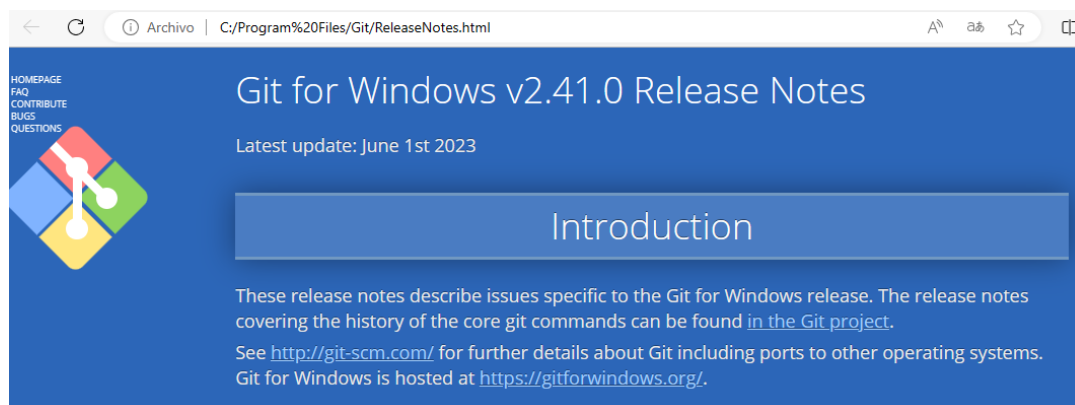


Con lo cual al presionar el botón finalizar, se nos abrirá dos ventanas:



Donde podemos ver que se ha ingresado el comando `git --version`, que luego de presionar la tecla ENTER nos devuelve la versión de la aplicación. Esto nos permite comprobar que la aplicación está trabajando de manera correcta.

La otra ventana, se abre en el browser, donde podemos observar un enlace a un minitutorial de uso de la aplicación:



Para la instalación en Linux y Mac puede consultar: [Git - Instalación de Git \(git-scm.com\)](https://git-scm.com)



## CONFIGURACION INICIAL

Ahora que tenemos Git instalado en nuestro Sistema Operativo se requiere realizar una **configuración para personalizar el entorno**.

Estas operaciones deberán realizarse una sola vez por computadora; van a quedar configuradas a través de actualizaciones. También podemos editarlas en cualquier momento con los mismos comandos.

Git viene con una herramienta **denominada git config** que permite obtener y configurar variables que controlan el aspecto visual de Git se va a ver, así como las operaciones que se podrán realizar

Estas variables pueden ser guardadas en tres lugares distintos que se detallan a continuación:

1. `/etc/gitconfig`: contiene los valores aplicados a todos los usuarios del Sistema Operativo y todos los repositorios. Si se utiliza la opción **--system** con **git config**, entonces se estará leyendo y escribiendo desde este archivo específicamente. Dado que es un archivo de configuración del sistema, se necesitan privilegios de superusuario o administrador para realizar cambios en él.
2. `~/.gitconfig` ó `~/.config/git/config`: contiene valores específicos del usuario. Puedes hacer que Git lea y escriba sobre este archivo específico usando la opción **--global** y estaremos afectando a todos los repositorios en los trabajemos en el sistema.
3. `./git/config`: este archivo puede ser encontrado dentro del mismo directorio del repositorio donde estamos actualmente trabajando y sirve para configurar específicamente un solo repositorio. Podemos forzar Git para que lea y escriba sobre este archivo usando la opción **--local** que, de hecho, es la opción por defecto.

Cada nivel anula los valores del nivel anterior, es decir que los valores en `.git/config` pesan más que aquellos en `/etc/gitconfig`.

En sistemas Windows, Git busca por el archivo `.gitconfig` en el directorio `$HOME` (`C:\Users\%USER` para la mayoría de las personas). También busca por `/etc/gitconfig`, que es relativo a la raíz de MSys, que hace referencia al directorio donde se ha decidido instalar Git.

Si estás usando la versión 2.x o superior de Git para Windows, hay también un archivo de configuración de nivel sistema en `C:\Documents and Settings\All Users\Application Data\Git\config` en Windows XP, y `C:\ProgramData\Git\config` en Windows Vista y superior. Este archivo de configuración puede ser cambiado únicamente por `git config -f <archivo>` como administrador. También podemos ver todas las configuraciones y dónde se encuentra cada una con:

```
git config --list --show-origin
```

La figura 4 muestra el resultado de la ejecución de este comando.

A continuación, se describe lo primero que tenemos que hacer cuando instalamos Git, que consiste en configurar nuestro nombre de usuario y dirección de correo electrónico. Esto es importante porque cada operación que realicemos con Git usará esta información para realizar un registro (log o bitácora) que permita en todo momento acceder a estos movimientos.

MINGW64:/c/Users/ariel

```
ariel@Ariel MINGW64 ~
$ git config --list --show-origin
file:C:/Program Files/Git/etc/gitconfig diff.astextplain.textconv=astextplain
file:C:/Program Files/Git/etc/gitconfig filter.lfs.clean=git-lfs clean -- %f
file:C:/Program Files/Git/etc/gitconfig filter.lfs.smudge=git-lfs smudge -- %f
file:C:/Program Files/Git/etc/gitconfig filter.lfs.process=git-lfs filter-process
file:C:/Program Files/Git/etc/gitconfig filter.lfs.required=true
file:C:/Program Files/Git/etc/gitconfig http.sslbackend=openssl
file:C:/Program Files/Git/etc/gitconfig http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
file:C:/Program Files/Git/etc/gitconfig core.autocrlf=true
file:C:/Program Files/Git/etc/gitconfig core.fscache=true
file:C:/Program Files/Git/etc/gitconfig core.symlinks=false
file:C:/Program Files/Git/etc/gitconfig pull.rebase=false
file:C:/Program Files/Git/etc/gitconfig credential.helper=manager
file:C:/Program Files/Git/etc/gitconfig credential.https://dev.azure.com.usehttppath=true
file:C:/Program Files/Git/etc/gitconfig init.defaultbranch=main
file:C:/Users/ariel/.gitconfig core.editor="C:\Users\ariel\AppData\Local\Programs\Microsoft VS Code\bin\code" --wait
file:C:/Users/ariel/.gitconfig user.name=Ariel A Vega
file:C:/Users/ariel/.gitconfig user.email=avega@fi.unju.edu.ar
file:C:/Users/ariel/.gitconfig filter.lfs.clean=git-lfs clean -- %f
file:C:/Users/ariel/.gitconfig filter.lfs.smudge=git-lfs smudge -- %f
file:C:/Users/ariel/.gitconfig filter.lfs.process=git-lfs filter-process
file:C:/Users/ariel/.gitconfig filter.lfs.required=true
ariel@Ariel MINGW64 ~
```

Para realizar el registro del usuario y el correo electrónico se recurre a los siguientes comandos

```
git config --global user.name "Mi nombre de usuario"
```

```
git config --global user.email "mi correo electrónico"
```

Por ejemplo

MINGW64:/c/Users/ariel

```
ariel@Ariel MINGW64 ~
$ git config --global user.name "avega@fi.unju.edu.ar"
ariel@Ariel MINGW64 ~
$ git config --global user.name "Ariel A Vega"
ariel@Ariel MINGW64 ~
$ git config --global user.email "avega@fi.unju.edu.ar"
ariel@Ariel MINGW64 ~
$
```

Podemos consultar el valor de cada una de estas variables, mediante los siguientes comandos

```
git config user.name
```

```
git config user.email
```

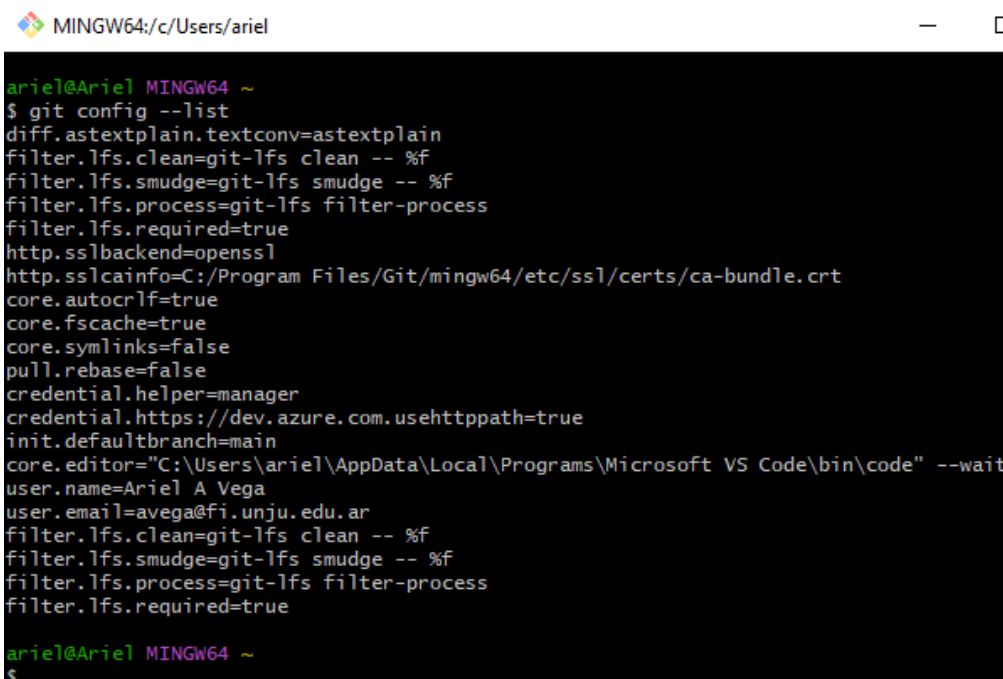
MINGW64:/c/Users/ariel

```
ariel@Ariel MINGW64 ~
$ git config user.name
Ariel A Vega
ariel@Ariel MINGW64 ~
$ git config user.email
avega@fi.unju.edu.ar
ariel@Ariel MINGW64 ~
$
```

También puedes consultar todas configuraciones realizadas en el sistema mediante el siguiente comando

```
git config --list
```

Esto va a mostrar una lista de todas las variables que estén configuradas y que Git puede detectar hasta ese momento:



```
MINGW64:/c/Users/ariel

ariel@Ariel MINGW64 ~
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=main
core.editor="C:\Users\ariel\AppData\Local\Programs\Microsoft VS Code\bin\code" --wait
user.name=Ariel A Vega
user.email=avega@fi.unju.edu.ar
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true

ariel@Ariel MINGW64 ~
$
```

## COMANDOS PARA MANEJAR EL SISTEMA DE ARCHIVOS

Los recursos sobre los que se realizará el control de versiones son archivos y carpetas. Por este motivo es necesario poder indicar como se pueden crear, modificar y eliminar archivos y carpetas desde la ventana de comandos Git; aunque esto se podría realizar también usando las órdenes visuales del sistema operativo. En esta sección veremos como realizarlo de ambas maneras.

La ventana de comandos de Git está configurada para reconocer comandos de gestión de sistemas de archivos de Linux. Por lo cual veremos los principales comandos

- **pwd:** El comando `pwd` se usa para localizar la ruta del directorio de trabajo en el que te encuentras. Seguramente al ingresar a la ventana de comandos en varios momentos querrás saber dónde te encuentras:



```
MINGW64:/c/Users/ariel

ariel@Ariel MINGW64 ~
$ pwd
/c/Users/ariel

ariel@Ariel MINGW64 ~
$
```

- **cd**: Del inglés **change directory** (cambio de directorio). Con la ayuda de este comando, podemos navegar por todos nuestros directorios en nuestro sistema. Es uno de los comandos más útiles, ya que por medio de él podremos cambiar de directorio ya sea a una rutas absolutas o rutas relativas, entre otras funcionalidades, por lo que nos centraremos en aquellas que sean de utilidad para realizar la gestión de los recursos a versionar.

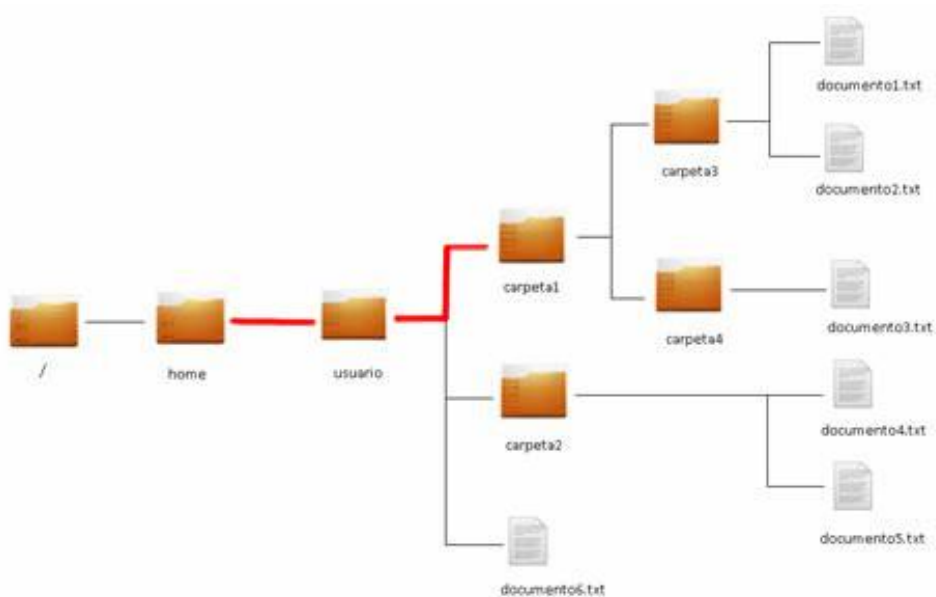
La sintaxis de este comando es

```
cd [opciones] [{ruta relativa}{ruta absoluta}{Directorios especiales}]
```

En primer lugar, veremos el concepto de ruta relativa y ruta absoluta:

**Las rutas relativas** indican el camino para encontrar un elemento, pero **basándonos en el directorio desde el que se ejecuta la orden**, es decir, desde el directorio en donde nos encontramos posicionados. Para saber si son correctas o no tenemos que saber siempre desde dónde se han utilizado (lo cual logramos a través del comando `pwd`). Para acuñar mejor el concepto, veamos algún ejemplo:

Suponga que se halla en el directorio `/home` y quiere ubicarse en la carpeta `/carpeta1`, tal como lo muestra la siguiente imagen



Para hacerlo puede usar el siguiente comando

```
cd usuario/carpeta1
```

Hemos usado una ruta relativa, ya que el punto de partida es la posición actual. Si realizamos un comando `pwd`, debería darnos un resultado similar a este

```
/home/usuario/carpeta1
```

En el ejemplo anterior nos hemos desplazado hacia adelante, hacia los directorios que dependen de nuestra ubicación actual. También es posible desplazarse hacia atrás usando rutas relativas. Pero esto requiere comprender el concepto de Directorio Padre. Los directorios padres son todos aquellos directorios que hay que recorrer para poder llegar a nuestra ubicación desde la raíz. Así en el ejemplo anterior, la carpeta `/usuario` es un directorio padre de `/carpeta1`. De la misma manera `/home` es directorio padre tanto de `/usuario` como de `/carpeta1`.



Ya que según el ejemplo estamos posicionados en /carpeta1, para volver al directorio padre inmediato usando ruta relativa deberíamos usar el siguiente comando:

```
cd ..
```

Si verificamos nuestra posición con el comando pwd, observaremos que estaremos en /home/usuario

Suponga que nuevamente está en /carpeta1 y desea volver hasta /home puede usar este comando:

```
cd ../../
```

Con lo cual le indicamos al sistema de archivos que suba en posición a dos directorios padres. Puede observar como los directorios padres se organizan en una jerarquía.

También por defecto con `cd /` podemos posicionarnos en la raíz del directorio, el cual representa el directorio desde donde nacen todos los demás directorios y que es único. Las rutas absolutas representan la ubicación de un archivo o directorio desde la raíz del sistema de directorios. El sistema de archivos es una estructura jerárquica que en el caso de Linux tiene una raíz que se indica cuando se pone solamente el carácter barra /. La raíz contiene los directorios principales del sistema que a su vez tendrán subdirectorios en su interior. Cuando yo quiero indicar dónde se encuentra un elemento usando una ruta absoluta, tendré que indicarle todos los directorios por los que hay que pasar empezando desde la raíz del sistema, SIEMPRE.

Suponga que está en /carpeta1 y desea ubicarse en /carpeta3 usando rutas absolutas. El comando necesariamente será el siguiente:

```
cd /home/usuario/carpeta1/carpeta3
```

- **mkdir:** Permite crear uno o varios directorios en la ubicación actual. A continuación vamos a crear un directorio, suponiendo que estamos en la unidad d:. En primer lugar abriremos una ventana de Git, suponiendo que usamos el enlace ubicado en el Escritorio de Windows. A continuación, se ejecutan los siguientes comandos

```
MINGW64:/d/proyectosgit
ariel@Ariel MINGW64 ~
$ cd /d

ariel@Ariel MINGW64 /d
$ pwd
/d

ariel@Ariel MINGW64 /d
$ mkdir proyectosgit

ariel@Ariel MINGW64 /d
$ cd proyectosgit

ariel@Ariel MINGW64 /d/proyectosgit
$ mkdir proyecto1

ariel@Ariel MINGW64 /d/proyectosgit
$ ls
proyecto1/

ariel@Ariel MINGW64 /d/proyectosgit
$ mkdir proyecto2 proyecto3

ariel@Ariel MINGW64 /d/proyectosgit
$ ls
proyecto1/ proyecto2/ proyecto3/

ariel@Ariel MINGW64 /d/proyectosgit
$ mkdir "proyectos java"

ariel@Ariel MINGW64 /d/proyectosgit
$ ls
proyecto1/ proyecto2/ proyecto3/ 'proyectos java'/

ariel@Ariel MINGW64 /d/proyectosgit
$ tree
bash: tree: command not found

ariel@Ariel MINGW64 /d/proyectosgit
$ |
```



Se puede observar que nos hemos ubicado en la unidad /d.

Posteriormente se ha creado un directorio denominado `proyectosgit` mediante el comando `mkdir proyectosgit`.

Luego nos ubicamos dentro de ese directorio y creamos el directorio `proyecto1`.

A continuación, se usa el comando `ls` en su forma más sencilla. Este comando lista todos los archivos y directorios pertenecientes a la ubicación actual.

Mediante `mkdir` es posible crear varios directorios a la vez, indicando el nombre de cada uno de ellos separados por un espacio. Esto se puede visualizar en cuando se crean las carpetas `proyecto2` y `proyecto3`.

Los directorios `proyecto1`, `proyecto2` y `proyecto3` son ejemplos de nombres simples, ya que no presentan caracteres especiales ni espacios.

Si deseamos crear un directorio con un nombre compuesto, debemos utilizar las dobles comillas. Esto se ve cuando se crea el directorio `"proyectos java"`.

Finalmente, podemos ver que este editor de comandos no reconoce el comando `tree` que permitiría ver la estructura de archivos en forma gráfica.

- `ls`: este es un comando muy útil, que hace referencia a la palabra inglesa `list`. Se utiliza para listar archivos o directorios. Este comando enumera el contenido del directorio que desee, archivos y otros directorios anidados. Su sintaxis es

```
ls [ option ...] [ archivo o directorio ]
```

Cuando se usa sin opciones ni argumentos, `ls` muestra una lista en orden alfabético de los nombres de todos los archivos en el directorio de trabajo actual.

Algunos resultados interesantes del comando al ser combinados con sus opciones son:

- `ls -a`: visualiza también los archivos ocultos. El comando los mostrará con un `.` delante del nombre del archivo.
- `ls -x`: permite mostrar los archivos y directorios organizados en varias columnas.
- `ls -l`: permite visualizar información detallada de cada archivo o directorio. Esto es:
  1. Los permisos de cada archivo.
  2. El número de archivos contenidos dentro de un directorio.
  3. El propietario del archivo.
  4. El tamaño de cada archivo.
  5. Fecha y hora de última modificación.
  6. Nombre y formato del archivo.
- `ls -group-directories-first`: De forma predeterminada, `ls` te ordena los archivos por orden alfabético de su nombre. Por lo tanto, pueden aparecer mezclados archivos y formatos de muy diversa índole, respondiendo exclusivamente a su letra inicial. Si queremos mostrar en el listado primero los directorios (o carpetas), y después el resto de los archivos ejecutamos este comando.
- `ls -t`: Permite ordenar el resultado por fecha de última modificación. Pero no nos muestra la fecha, por lo que deberíamos combinarlo con `-l` de la siguiente manera: `ls -t-l`.


También es posible combinar estas opciones utilizando un solo signo negativo y combinando las letras de opciones. Por ejemplo, es válido lo siguiente:

```
ls -atl
```

- touch: Este comando se usa para crear cualquier tipo nuevo de archivo. Es muy útil para los desarrolladores permitiendo crear archivos en el servidor. También, se usa para cambiar la hora de acceso y modificación de archivos. Su sintaxis es:

touch file\_name

Un ejemplo sería el siguiente

 MINGW64:/d/proyectosgit/proyecto1


```
ariel@Ariel MINGW64 /d/proyectosgit
$ cd proyecto1

ariel@Ariel MINGW64 /d/proyectosgit/proyecto1
$ touch archivo.txt

ariel@Ariel MINGW64 /d/proyectosgit/proyecto1
$ ls -l
total 0
-rw-r--r-- 1 ariel 197121 0 Jul 16 12:21 archivo.txt

ariel@Ariel MINGW64 /d/proyectosgit/proyecto1
$ |
```

- cat: Es uno de los más utilizados. El nombre del comando cat viene de “concatenate”, por su funcionalidad para concatenar archivos o unir, sumar. Sin embargo, nosotros la utilizaremos para leer el contenido de un archivo. También se puede usar para combinar y escribir contenidos de archivos en una salida estándar o copiar contenido de un archivo a otro, mostrar el número de línea o mostrar \$ al final de la línea. Suponga un archivo creado con touch cuyo nombre es archivo.txt y que posteriormente ha sido editado. Para ver su contenido procederemos de la siguiente manera:

 MINGW64:/d/proyectosgit/proyecto1

```
ariel@Ariel MINGW64 /d/proyectosgit/proyecto1
$ cat archivo.txt
Contenido del archivo
ariel@Ariel MINGW64 /d/proyectosgit/proyecto1
$ |
```

- rm: elimina cada archivo especificado en la línea de comando y directorios. Ten mucho cuidado al usarlo porque no se puede deshacer y es muy difícil recuperar archivos eliminados de esta manera. Para eliminar un archivo normal, debe escribir la siguiente sintaxis:

rm "archivo"

Por ejemplo

```
ariel@Ariel MINGW64 /d/proyectosgit/proyecto1
$ ls

ariel@Ariel MINGW64 /d/proyectosgit/proyecto1
$
```

El comando eliminó el archivo. Se puede verificar que la carpeta /proyecto1 está vacía como resultado de la ejecución de la aplicación.

Para eliminar directorios debe usar la opción -r

- vim: Es un editor de texto de terminal gratuito y de código abierto. Puedes usarlo como tu editor de código.



## FUNCIONAMIENTO Y USO DE VIM

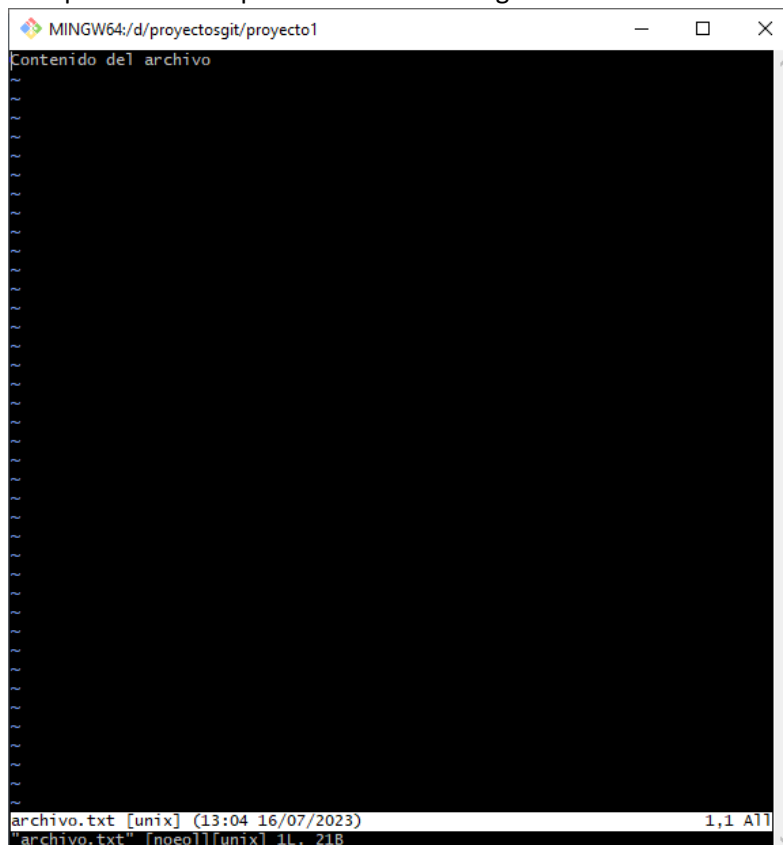
Vim es uno de los editores más usados en el terminal, junto con Nano. Aunque Nano es más sencillo de usar, la productividad que podemos alcanzar mediante el editor Vim es mucho mayor gracias a sus funcionalidades avanzadas, algo que podremos experimentar una vez aprendemos a utilizarlo de manera ágil. No obstante, los primeros pasos con Vim siempre son difíciles y esto provoca que muchas personas desistan de usarlo. Por esta razón en esta sección se brinda un espacio para que esta adaptación sea más sencilla y una referencia rápida para encontrar los comandos más utilizados en el día a día.

La clave para aprender a manejar Vim es acostumbrarse a sus distintos modos de funcionamiento. Esta es la parte que hace a Vim especial y difícil de entender para las personas que se inician.

- El Modo Comando: Es el modo al que se accede cuando un recurso es abierto para su edición con esta herramienta. Se consigue mediante el comando vim seguido del nombre del archivo. Por ejemplo

```
vim archivo.txt
```

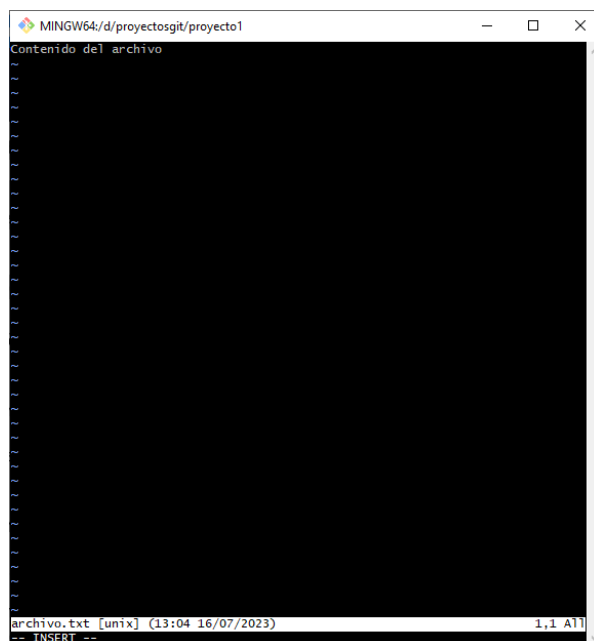
Esto generará que se vea una pantalla similar a la siguiente



Este modo permite realizar gran cantidad de acciones administrativas sobre el fichero, como buscar en el texto, salir, guardar, borrar líneas, etc.

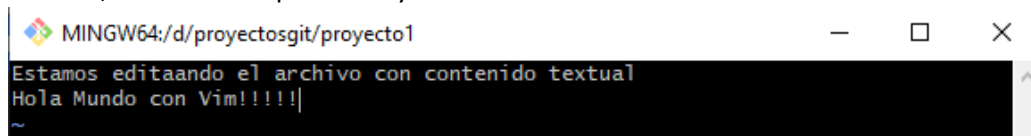
- Modo Inserción: Nos sirve cuando queremos editar el texto del archivo, añadiendo nuevo contenido con el teclado, o borrando carácter a carácter con las correspondientes teclas (Retroceso/Supr).

Estando en modo comando, pasamos a modo inserción pulsando la tecla i.

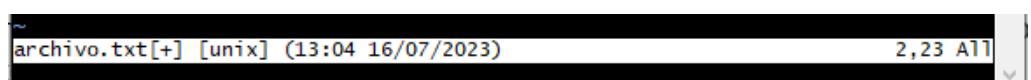
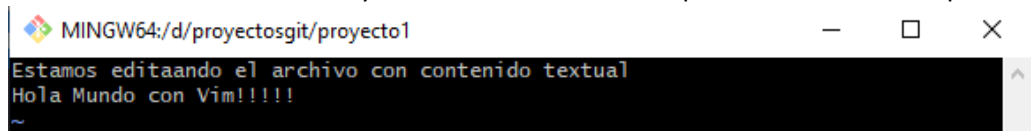


Podemos observar que en la parte inferior aparece la leyenda INSERT.

A partir de entonces el editor funcionará tal y como se esperaría en un editor de texto. Es decir, escribes cualquier cosa y se va introduciendo el texto en el archivo.



Para salir del modo inserción y volver al modo comando pulsamos la tecla escape Esc:



Comprendido ambos modos, ahora podemos describir los comandos más útiles y necesarios para que nuestros primeros pasos con Vim sean más llevaderos. **Recuerda que estos comandos los tienes que escribir en modo comando y no en modo inserción.**

Para algunos de estos comandos es necesario pulsar la tecla Enter para que realmente se ejecuten. En este caso, los comandos aparecerán en la parte de abajo del terminal, para que veamos qué es lo que se va a ejecutar:

Comandos asociados a Guardar y Cerrar:

- :w- Permite guardar el archivo



- :q – Salir de Vim. Si el archivo ha sido modificado pero no se ha guardado, nos advertirá y no podremos salir de Vim usando este comando

```
MINGW64:/d/proyectosgit/proyecto1
Estamos editando el archivo con contenido textual
Hola Mundo con Vim!!!!
Estoy agregando texto, que no será aún guardado
```

```
archivo.txt [unix] (09:07 17/07/2023) 2,23 A11
:q
```

Que generará el siguiente resultado

```
archivo.txt[+] [unix] (09:07 17/07/2023) 3,49-47 A11
E37: No write since last change (add ! to override)
```

- :q! - Salir de Vim, descartando posibles cambios no guardados que se hayan realizado en el archivo. Siguiendo desde el ejemplo anterior, veremos que si cierra el editor

```
archivo.txt[+] [unix] (09:07 17/07/2023) 3,49-47 A11
:q!
```

Volviendo a la línea de comandos

```
ariel@Ariel MINGW64 /d/proyectosgit/proyecto1
$
```

- :wq - Hace el guardado del archivo y después sale de Vim

Asociados a Deshacer y Rehacer

- u – Deshacer acción. Por ejemplo, en Modo Inserción se ha escrito el siguiente texto:

```
MINGW64:/d/proyectosgit/proyecto1
Estamos editando el archivo con contenido textual
Hola Mundo con Vim!!!!
Ingresando un nuevo texto
```

Luego de volver al modo comando, si ingresamos u veremos el siguiente resultado

```
MINGW64:/d/proyectosgit/proyecto1
Estamos editando el archivo con contenido textual
Hola Mundo con Vim!!!!
```

```
archivo.txt [unix] (09:07 17/07/2023) 2,23 A11
1 line less; before #1 09:17:18
```

- Crtl+r – Rehacer una acción. Siguiendo con el ejemplo anterior, al realizar esta acción en Modo Comando, obtendremos el siguiente resultado

```
MINGW64:/d/proyectosgit/proyecto1
Estamos editando el archivo con contenido textual
Hola Mundo con Vim!!!!
Ingresando un nuevo texto
```

```
archivo.txt[+] [unix] (09:07 17/07/2023) 2,23 A11
1 more line; after #1 09:17:18
```

Asociados con el movimiento por el contenido del archivo

- gg – Ponerse al inicio del fichero.
- Mayús+g – Ir a la última línea del fichero.
- Num+G – Ir a una línea determinada. Por ejemplo 14G llevaría el cursor a la línea 14.
- :set number – Hace que el editor muestre el número de las líneas.
- \$ – Ir al final de la línea.
- 0 – Ir al principio de la línea.

#### Borrar líneas

- dd – El comando permite borrar la línea actual, donde está el cursor.
- d+num – Este comando permite borrar un número de líneas. Por ejemplo, d3 borrará tres líneas.

Buscar: Vim tiene unas herramientas muy potentes para buscar texto en los archivos. Los comandos más útiles son los siguientes:

- /+texto – Al pulsar «/» se abre la función de búsqueda. Entonces podremos escribir el texto que queremos buscar. El editor resaltará todas las apariciones de este texto. Pulsamos enter y nos llevará a la siguiente aparición de la búsqueda, con respecto a la posición de nuestro cursor.
- n y N – Una vez hemos aceptada una búsqueda, el comando n nos lleva a la siguiente aparición de la cadena buscada. El comando N nos llevará a la anterior.

#### Otras ayudas

Con estos comandos te aseguramos que te podrás defender con Vim con cierta agilidad, de modo que puedas realizar las tareas de edición más comunes. Además, desde Vim puedes hacer lo siguiente:

:h – Abre la ayuda principal de Vim. Esto hace que nuestra ventana de terminal se divida en dos editores. En este momento nuestro cursor estará en el texto del archivo de ayuda de Vim. Podremos leer el fichero y utilizar las funciones de búsqueda comentadas con anterioridad. Cuando queramos salir tenemos que hacerlo como cualquier otro fichero, usando el comando correspondiente de Vim, por ejemplo :q.

Nota: Ud podría editar los archivos también usando otros editores disponibles en el Sistema Operativo. También, puede gestionar los archivos (crear, eliminar, buscar archivos y carpetas) usando las herramientas gráficas del sistema operativo.

### LOS REPOSITORIOS

Si queremos que Git realice el control de versiones de un conjunto de archivos y subdirectorios, en principio lo más recomendable es conformar un directorio que represente a un proyecto. Así, lo que le pediremos a Git es que realice el control de versiones de ese proyecto. En general la mayoría de los desarrollos tecnológicos trabajan con el concepto de proyecto.

Dentro de los mecanismos de trabajo que utiliza Git, lo que debemos hacer es convertir a esa carpeta de proyecto en un repositorio; para lo cual lo único que debemos hacer es agregar la carpeta .git/ dentro de ese proyecto. Es decir un repositorio es un proyecto que incluye la carpeta .git/.

Esta carpeta es muy importante, porque rastrea todos los cambios realizados en los archivos de un proyecto y crea un historial a lo largo del tiempo. Es decir, si se elimina la carpeta .git /, entonces desaparece el historial del proyecto.

El término correcto para indicar que una carpeta/proyecto se convierta en un repositorio es “Inicializar un repositorio”.

Usualmente, se puede obtener un repositorio de Git de dos maneras:

- Convertir un directorio local que, actualmente, no esté bajo ningún control de versión a un repositorio de Git.
- Clonar un repositorio de Git existente desde algún otro lugar.

Así, conseguiremos tener un repositorio de Git, en nuestra máquina local, listo para trabajar.

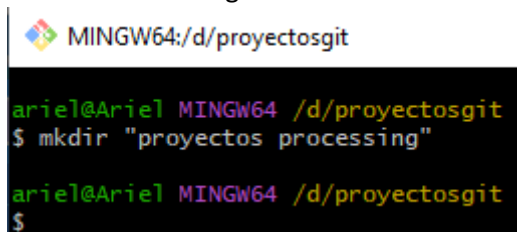
### USO DE GIT INIT

En el caso de tener un directorio de proyecto, y se desea comenzar a controlarlo con Git:

1. Primero, debemos dirigirnos a ese directorio.
2. Una vez allí, se debe correr el comando `git init`
3. Esto va a crear un nuevo subdirectorio con la carpeta .git/

Por ejemplo, observe la siguiente secuencia de comandos:

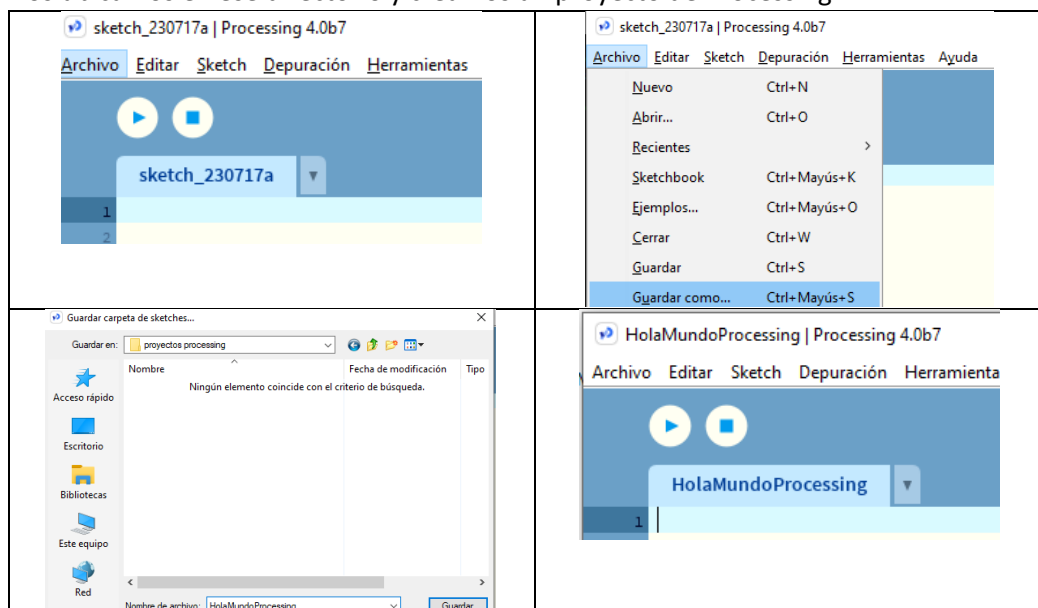
- 1) En el visor de comandos escribimos lo siguiente



```

MINGW64:/d/proyectosgit
ariel@Ariel MINGW64 /d/proyectosgit
$ mkdir "proyectos processing"
ariel@Ariel MINGW64 /d/proyectosgit
$
  
```

- 2) Nos ubicamos en ese directorio y creamos un proyecto de Processing



Con lo que tendremos en la carpeta proyectosgit se agregará la carpeta del proyecto que hemos creado

```
MINGW64:/d/proyectosgit

ariel@Ariel MINGW64 /d/proyectosgit
$ ls
'proyectos java'/ 'proyectos processing'/

ariel@Ariel MINGW64 /d/proyectosgit
$ |
```

- 3) Ahora ingresaremos dentro de esa carpeta y ejecutaremos el comando para inicializar el control de versiones

```
MINGW64:/d/proyectosgit/proyectos processing/HolaMundoProcessing

ariel@Ariel MINGW64 /d/proyectosgit
$ cd "proyectos processing"

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing
$ ls
HolaMundoProcessing/

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing
$ cd "HolaMundoProcessing"

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing
$ git init
Initialized empty Git repository in D:/proyectosgit/proyectos processing/HolaMundoProcessing/.git/

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$
```

Con lo anterior, podrás comprobar varias cosas:

- 1) Se ha agregado la etiqueta (main) en la ubicación del proyecto. Esto es, se ha creado una rama por defecto denominada main, que llevará el registro de todos los cambios que realicemos sobre este proyecto.
- 2) Se habrá agregado una carpeta .git/ dentro del proyecto que estará oculta

```
MINGW64:/d/proyectosgit/proyectos processing/HolaMundoProcessing

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ ls
HolaMundoProcessing.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ ls -a
./ ../ .git/ HolaMundoProcessing.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$
```

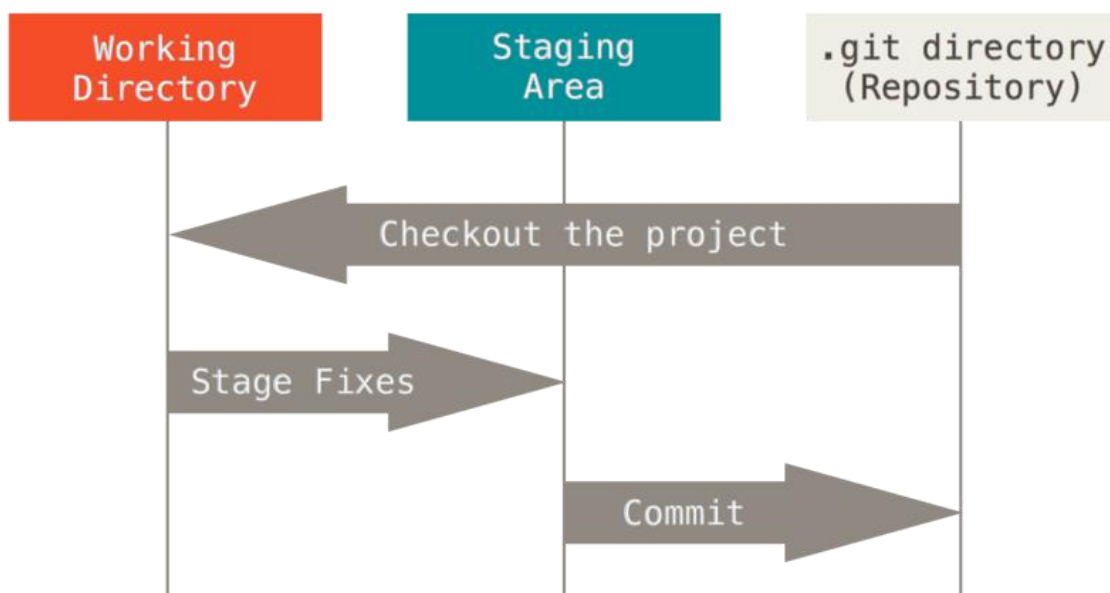
La operación de realizar el seguimiento de un proyecto se debe realizar solamente una vez.

## ESTADOS DE UN REPOSITORIO GIT

Esto es lo más importante que debes recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged).

- 1) Confirmado (Committed): significa que los datos están almacenados de manera segura en tu base de datos local.
- 2) Modificado (Modified): significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
- 3) Preparado (Staged): significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: El directorio de Git (.git directory), el directorio de trabajo (working directory), y el área de preparación (staging area):



El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se obtienen de la base de datos comprimida en el directorio de Git (haciendo un Checkout the Project), y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice ("index"), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

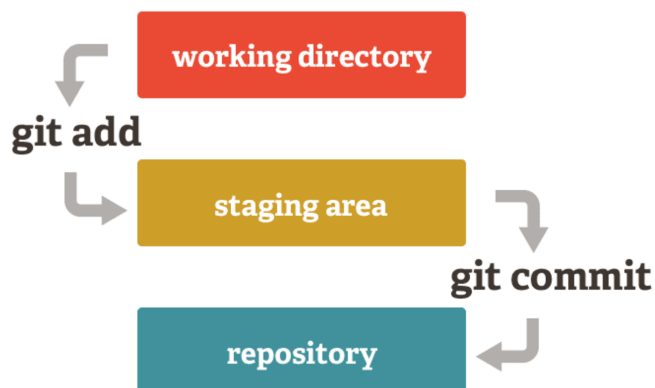
El flujo de trabajo básico en Git es algo así:

- 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 esa copia instantánea de manera permanente en tu directorio de Git.

Lo anterior esquemáticamente se representa de la siguiente manera:



Donde se observa que el pasaje de un estado a otro se realiza a través de comandos de Git. De manera resumida para pasar del working directory al staging area se utiliza el comando git add, mientras que para pasar de esta última área al .git directory hay que realizar un git commit.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

## GIT STATUS

Tomando como referencia el ejemplo de proyecto sobre el que se ha realizado el seguimiento mediante git init, podemos verificar que, aunque posee el archivo .pde que Processing crea dentro del directorio, este resultará desde el punto de vista de Git como un nuevo elemento dentro del proyecto. Esto lo comprobaremos al ingresar el siguiente comando

```

MINGW64:/d/proyectosgit/proyectos_processing/HolaMundoProcessing
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main

No commits yet

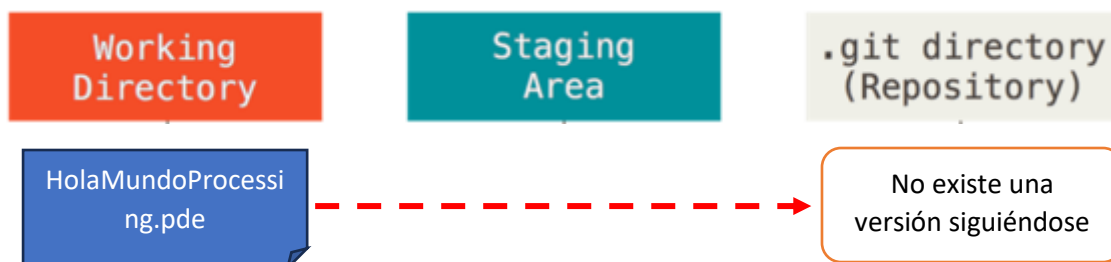
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        HolaMundoProcessing.pde

nothing added to commit but untracked files present (use "git add" to track)
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$
  
```

El comando nos brinda mucha información:

- 1) En primer lugar, nos indica que estamos parados en la Rama Principal, es decir el camino que por defecto Git realiza el seguimiento.
- 2) En segundo lugar, nos indica que no hemos realizado ninguna confirmación (commit). Los commits permiten confirmar los cambios realizados cuando los pasamos desde el staging area al .git directory o repositorio. En este ejemplo es natural, porque recién hemos creado el directorio.
- 3) En tercer lugar, nos informa que ha detectado archivos que no se están siguiendo (untracked files). Es decir, se han creado nuevos archivos, o aquellos existentes presentan modificaciones con respecto a la versión almacenada en el directorio de git. Para este ejemplo, significa que HolaMundoProcessing.pde está en el estado (modified), o lo que es lo mismo se halla en el working directory. Esto se debe a que Git observa que hay un archivo en el working directory del cual no encuentra una versión anterior (instantánea o snapshot) en el .git directory. **Por lo tanto, Git no lo va a incluir hasta que se lo indiquemos explícitamente.**

A su vez, brinda información sobre como poder colocar este cambio en el staging area mediante el comando git add. Esquemáticamente podemos visualizar esta situación de la siguiente manera:



Git status es la herramienta principal para determinar el estado de los archivos.

## GIT ADD

Se utiliza para comenzar a rastrear un archivo; por ejemplo puedes ejecutar lo siguiente:

```

MINGW64:/d/proyectosgit/proyectos processing/HolaMundoProcessing
$ git status
On branch main

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git add HolaMundoProcessing.pde
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   HolaMundoProcessing.pde
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$

```

Puede observar, además, que luego de ejecutar el comando, una nueva consulta de estados nos informa que existen cambios a ser confirmados. Específicamente el cambio radica en que existe un nuevo archivo a seguir (new file: HolaMundoProcessing.pde).

Esto significa que el comando add ha cambiado el estado del archivo, pasándolo de modified a staged:



El comando git add puede recibir tanto una ruta de archivo como de un directorio; si es de un directorio, el comando añade recursivamente los archivos que están dentro de él.

Si generamos un commit en este punto, la versión del archivo en el momento en que hayamos ejecutado el comando git add va a permanecer en el snapshot histórico anterior.

Pero si modificamos y guardamos los cambios del archivo HolaMundoProcessing.pde, por ejemplo, agregando un código similar al siguiente:

```

HolaMundoProcessing
1 void setup(){
2   size(400,400);
3   background(0);
4   text("Hola Mundo Processing",width/2,height/2);
5 }
  
```

Y luego ejecutamos nuevamente el comando git status obtendremos lo siguiente:

```

MINGW64:/d/proyectosgit/proyectos processing/HolaMundoProcessing
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main

No commits yet

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

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   HolaMundoProcessing.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ |
  
```

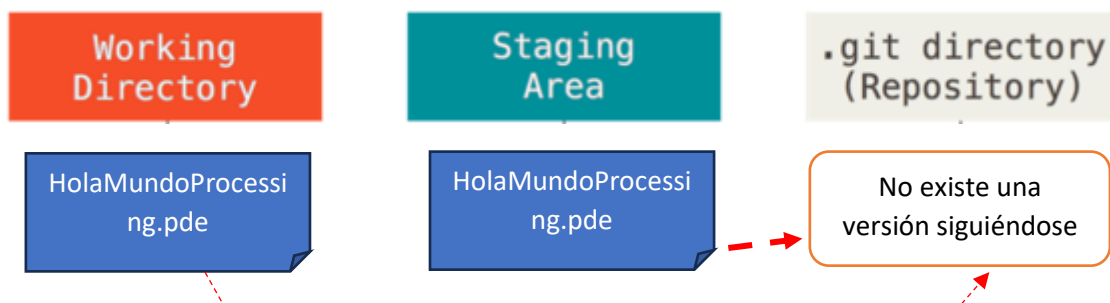
Ahora el archivo aparece tanto como un recurso que posee una versión a ser confirmada como también bajo la sección indicada como "Changes not staged for commit". Significa que un archivo con seguimiento ha sido modificado en el directorio de trabajo (working directory) pero

no ha sido preparado aún, con lo cual podríamos prepararlo utilizando nuevamente el comando `git add`.

Podemos entonces percibir que este comando es multipropósito ya que nos permite darles seguimiento a archivos nuevos y además preparar archivos que ya tenían seguimiento para el próximo commit.

Se debe tener en cuenta que al momento de realizar un commit, la versión del archivo que formará parte va a ser la que estaba cuando se ejecutó el comando `git add`. Es decir que, si se modifica un archivo luego de haber ejecutado el comando `git add`, se deberá ejecutar nuevamente el comando `git add` para agregar los cambios correspondientes.

Para nuestro ejemplo, esquemáticamente actualmente tenemos lo siguiente:



Es decir, el archivo sigue sin formar parte del repositorio (`.git directory`), pero además hay una versión lista a ser agregada (la del archivo sin contenido), mientras que hay otra versión que Git detecta en el working directory (aquella que tiene contenido luego de agregar el método `setup()`), y que como puede observar también se analiza como un archivo nuevo, que no existe en el repositorio.

Es más, suponga que realiza un nuevo `git add`, agregando de nuevo el archivo al Staging area:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git add HolaMundoProcessing.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main

No commits yet

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

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$
```

Como puede observar, Git detecta que tiene el archivo en el staging área. ¿Cuál será la versión que tomó? ¿La primera que estaba disponible para ser agregada o la más reciente? Vamos a dilucidar esta situación con el siguiente comando: `git commit`.

## GIT COMMIT

Una vez que el/los archivos/directorios se encuentran en el área de preparación (staging area), estamos listos para confirmar los cambios ejecutando un commit de la siguiente manera:

`git commit`

Sin embargo, vamos a recomendar que se realice un commit con definición de un mensaje, para lo cual se utiliza la opción -m. Para nuestro ejemplo, usaremos lo siguiente:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git commit -m "Agregando el archivo HolaMundoProcessing.pde"
[main (root-commit) 6180c2c] Agregando el archivo HolaMundoProcessing.pde
1 file changed, 5 insertions(+)
create mode 100644 HolaMundoProcessing.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ |
```

Como puede notar se genera una respuesta que indica que se ha agregado al archivo 5 nuevas líneas y que se han confirmado los cambios.

Específicamente nos brinda información acerca de:

- La rama (branch) donde confirmamos los cambios (main)
- El identificador SHA-1 del commit (6180c2c)
- Cuántos archivos fueron modificados
- Estadísticas acerca de las líneas que fueron insertadas o removidas en el commit.

Cada vez que hagamos un commit estaremos generando una nueva versión del proyecto, o snapshot, la cual podemos revertir o comparar luego.

Con lo anterior podemos deducir que Git ha tomado el último add realizado, por el hecho de que para él el archivo no existía en el repositorio, por lo que supone que el último add contiene la versión más reciente.

Podemos verificar el estado actual de nuestro repositorio con el comando `git status`

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
nothing to commit, working tree clean

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$
```

## GIT LOG

Antes de continuar viendo otras opciones de `git commit`, resulta conveniente estudiar el comando `git log`. Luego de haber creado varios commits, o mismo si hemos clonado un repositorio con un historial de commits existente, probablemente queramos mirar hacia atrás para ver qué ha pasado en el repositorio. Con el comando `git log` logramos esto:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git log
commit 6180c2c6ae9254f4cdeaafeb3ac6002b3b914c13 (HEAD -> main)
Author: Ariel A Vega <avega@fi.unju.edu.ar>
Date: Mon Jul 17 18:33:07 2023 -0300

    Agregando el archivo HolaMundoProcessing.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ |
```

Por defecto, sin parámetros, git log lista los commits hechos en un repositorio en orden cronológico inverso; esto es, el commit más reciente se verá primero. Como podemos ver, este comando enumera cada commit con su identificador SHA-1, el nombre de autor e email, la fecha de escritura y el mensaje del commit.

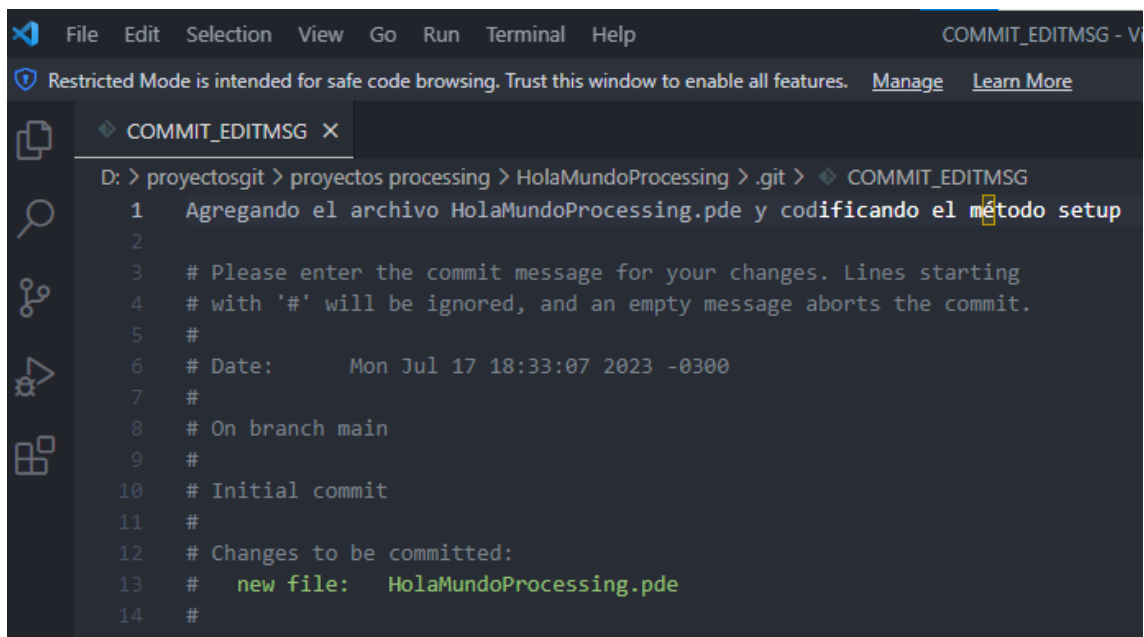
Este comando viene con muchos atajos y parámetros que podemos agregar para que la salida en la línea de comandos no sea tan abundante. Particularmente las opciones de --oneline y --graph son sumamente útiles respectivamente, para mostrar información abreviada sobre cada commit y para poder ver un gráfico ASCII mostrándonos nuestro historial de commits:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git log --oneline
6180c2c (HEAD -> main) Agregando el archivo HolaMundoProcessing.pde
```

Para --graph primero haremos más commits, por ejemplo

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git commit --amend
hint: Waiting for your editor to close the file... |
```

Que permite cambiar el mensaje del último commit, para lo cual abre un editor de texto, similar al siguiente donde podemos realizar los cambios, por ejemplo:



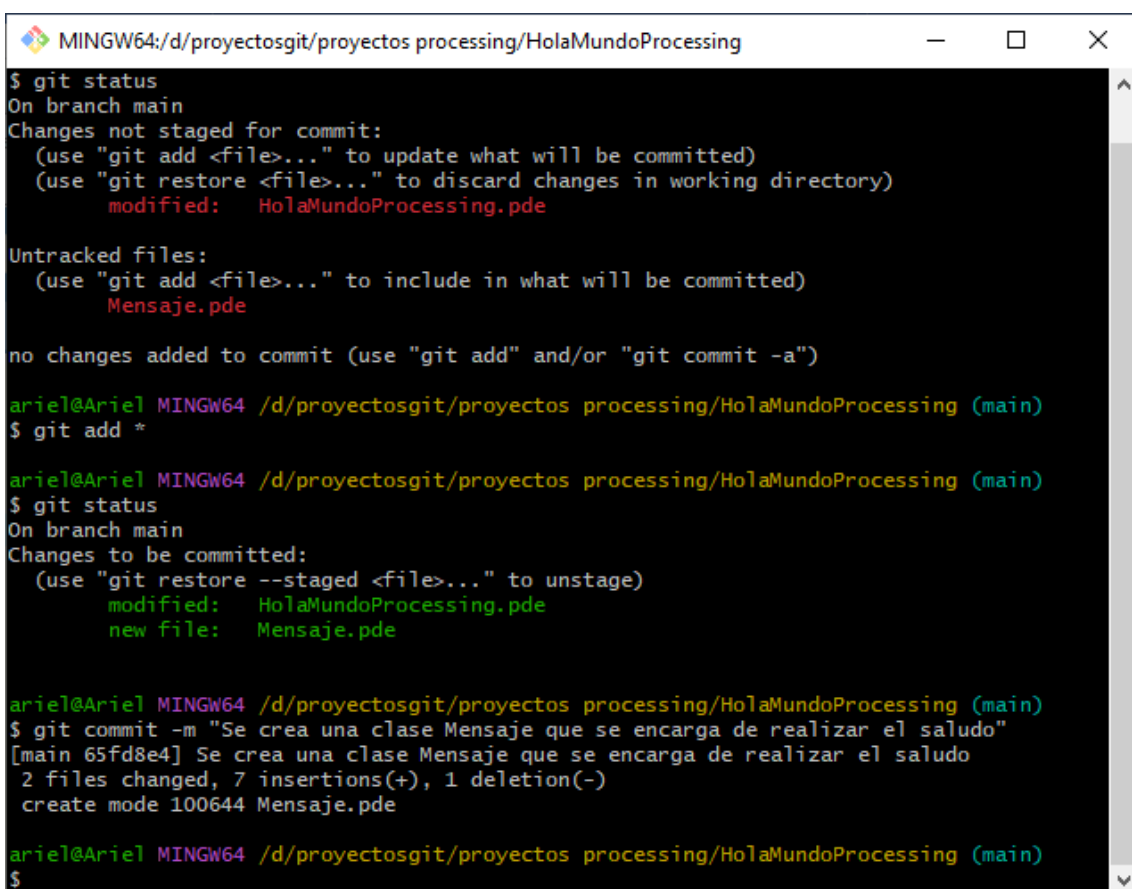
Que al guardar los cambios y salir del editor generará lo siguiente:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git commit --amend
[main 36a8bfd] Agregando el archivo HolaMundoProcessing.pde y codificando el método setup
Date: Mon Jul 17 18:33:07 2023 -0300
1 file changed, 5 insertions(+)
create mode 100644 HolaMundoProcessing.pde
```



Ahora agregaremos en el proyecto un archivo que represente la clase Mensaje, que será ahora la encargada de mostrar el mensaje. Luego guardaremos los cambios y procederemos a realizar el paso de colocar los cambios en el staging área y finalmente los confirmaremos:

<pre> 1  HolaMundoProcessing 2  Mensaje 3 4  1 Mensaje unMensaje; 5  2 void setup(){ 6  3     size(400,400); 7  4     background(0); 8  5     unMensaje = new Mensaje(); 9  6 } </pre>	<pre> 1  HolaMundoProcessing 2  Mensaje 3 4  1 class Mensaje{ 5  2     public Mensaje(){ 6  3         text("Hola Mundo Processing",width/2,height/2); 7  4     } 8  5 } </pre>
--	--



```

MINGW64:/d/proyectosgit/proyectos processing/HolaMundoProcessing
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   HolaMundoProcessing.pde

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

no changes added to commit (use "git add" and/or "git commit -a")

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git add *

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   HolaMundoProcessing.pde
    new file:   Mensaje.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git commit -m "Se crea una clase Mensaje que se encarga de realizar el saludo"
[main 65fd8e4] Se crea una clase Mensaje que se encarga de realizar el saludo
 2 files changed, 7 insertions(+), 1 deletion(-)
 create mode 100644 Mensaje.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$

```

Puede apreciar que Git detecta la presencia de un nuevo archivo (Mensaje.pde) a su vez que informa que HolaMundoProcessing.pde ha sido modificado.

Para agregar ambos cambios al staging área podemos utilizar el símbolo \* que representará a todos los recursos que se pueden agregar a esta área.

Finalmente hemos realizado un nuevo commit con su respectivo mensaje. Ahora visualizaremos los commits con la opción--graph



```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git log --graph
* commit 65fd8e48e0fec102d5e593d959521c8dbd40ba10 (HEAD -> main)
| Author: Ariel A Vega <avega@fi.unju.edu.ar>
| Date: Mon Jul 17 19:02:48 2023 -0300
|
| Se crea una clase Mensaje que se encarga de realizar el saludo
|
* commit 5b1e8821dcf004ede994aa908f3b4501e0eb019a
| Author: Ariel A Vega <avega@fi.unju.edu.ar>
| Date: Mon Jul 17 18:33:07 2023 -0300
|
| Agregando el archivo HolaMundoProcessing.pde y codificando el método setup
```

A medida que se realicen más commits el gráfico irá mostrando el sistema de grafos que componen el camino realizado.

Se pueden combinar ambas opciones en el comando git log. Por ejemplo:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git log --online --graph
* 65fd8e4 (HEAD -> main) Se crea una clase Mensaje que se encarga de realizar el saludo
* 5b1e882 Agregando el archivo HolaMundoProcessing.pde y codificando el método setup
```

## EJEMPLOS PRACTICOS

Para entender como funciona Git con estos comandos veamos un par de ejemplos más.

- 1) Agregar un nuevo archivo denominado OtroArchivo.pde. Agregarlo al repositorio. Luego eliminar físicamente el archivo y analizar el resultado de un git status

```
MINGW64:/d/proyectosgit/proyectos processing/HolaMundoProcessing
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  OtroArchivo.pde

nothing added to commit but untracked files present (use "git add" to track)

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git add OtroArchivo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   OtroArchivo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git commit -m "Se agrega el archivo OtroArchivo.pde"
[main 3d028cf] Se agrega el archivo OtroArchivo.pde
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 OtroArchivo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
nothing to commit, working tree clean

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ rm OtroArchivo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:   OtroArchivo.pde

no changes added to commit (use "git add" and/or "git commit -a")

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$
```

Como puede observar, Git detecta que en el working directory no se halla el archivo OtroArchivo.pde, el cual si está presente en el .git directory. Esto significa que la eliminación física en el directorio de trabajo no afecta el registro de versiones que esta el directorio de Git.

Para mantener la consistencia tenemos dos opciones: si hemos eliminado por error el archivo, podemos volver a traerlo desde el .git directory mediante el comando git restore. Por ejemplo

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git restore OtroArchivo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
nothing to commit, working tree clean

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ ls
HolaMundoProcessing.pde  Mensaje.pde  OtroArchivo.pde
```

O si deseamos eliminar el archivo en el repositorio, debemos usar el comando git rm. Por ejemplo

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ rm OtroArchivo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ ls
HolaMundoProcessing.pde  Mensaje.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:    OtroArchivo.pde

no changes added to commit (use "git add" and/or "git commit -a")

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git rm OtroArchivo.pde
rm 'OtroArchivo.pde'

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    OtroArchivo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git commit -m "Se elimina el archivo OtroArchivo.pde"
[main c237d25] Se elimina el archivo OtroArchivo.pde
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 OtroArchivo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git log --oneline
c237d25 (HEAD -> main) Se elimina el archivo OtroArchivo.pde
3d028cf Se agrega el archivo OtroArchivo.pde
65fd8e4 Se crea una clase Mensaje que se encarga de realizar el saludo
5b1e882 Agregando el archivo HolaMundoProcessing.pde y codificando el método setup
```

Varias cosas por observar: el comando git rm no confirma la eliminación del archivo del .git directory. Sino que lo pasa al estado Staged. Para confirmar la acción entonces debemos realizar un commit. De esta manera queda registrado su eliminación.

- 2) Suponga que desea crear un archivo denominado Auxiliar.pde. Confirma la acción y posteriormente elimina ese archivo del directorio de trabajo. Se da cuenta que es un error eliminarlo. Entonces, revierte esta acción del directorio de trabajo.

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Auxiliar.pde

nothing added to commit but untracked files present (use "git add" to track)

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git add Auxiliar.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git commit -m "Se agrega el archivo Auxiliar.pde"
[main 4d727fc] Se agrega el archivo Auxiliar.pde
1 file changed, 2 insertions(+)
create mode 100644 Auxiliar.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
nothing to commit, working tree clean

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ rm Auxiliar.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:    Auxiliar.pde

no changes added to commit (use "git add" and/or "git commit -a")

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git restore Auxiliar.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
nothing to commit, working tree clean
```

- 3) Suponga que agrega un nuevo archivo denominado Calculo.pde y confirma su pase al estado committed. Luego lo editas y los preparas (pasándolo al staging area) Luego modificas el archivo Auxiliar.pde. Por último, quieres poder observar en detalle los cambios que se han presentado. En primer lugar, agregamos el archivo indicado

HolaMundoProcessing	Auxiliar	Calculo	Mensaje	▼
<pre>1 class Calculo{ 2 }</pre>				

Luego lo agregamos al .git directory

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Calculo.pde

nothing added to commit but untracked files present (use "git add" to track)

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git add Calculo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git commit -m "Se agrega el archivo Calculo.pde"
[main 3260192] Se agrega el archivo Calculo.pde
1 file changed, 2 insertions(+)
create mode 100644 Calculo.pde
```

Ahora lo editamos de la siguiente manera:

```
HolaMundoProcessing Auxiliar Calculo Mensaje ▼
1 class Calculo{
2   private float variableDependiente;
3
4   public Calculo(float variableDependiente){
5     this.variableDependiente=variableDependiente;
6   }
7 }
```

Ahora lo enviamos al staging área

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Calculo.pde

no changes added to commit (use "git add" and/or "git commit -a")

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git add Calculo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Calculo.pde
```

A continuación, modificamos el archivo Auxiliar.pde

```
HolaMundoProcessing Auxiliar Calculo Mensaje ▼
1 class Auxiliar{
2   private Calculo calculo;
3   public Auxiliar(Calculo calculo){
4     println("Se usará para realizar calculos auxiliares");
5     this.calculo = calculo;
6   }
}
```

Si aplicamos el comando git status obtendremos lo siguiente:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Calculo.pde

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Auxiliar.pde
```

Para lo que solicita este ejemplo, el comando git status resulta impreciso para ti (se quiere ver exactamente qué ha cambiado, no solo cuáles archivos lo han hecho).

Para obtener el resultado deseado, se puede usar el comando git diff. Si bien, es probable que lo describamos en detalle más adelante, ahora nos permitirá responder estas dos preguntas: ¿Qué has cambiado, pero aún no has preparado? y ¿Qué has preparado y está listo para confirmar?

git diff: Permite visualizar qué has cambiado, pero aún no has preparado. Este comando compara lo que tienes en tu directorio de trabajo con lo que está en el área de preparación. El resultado te indica los cambios que has hecho pero que aún no has preparado. A continuación se observa la aplicación de este comando al ejemplo que hemos realizado:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git diff
diff --git a/Auxiliar.pde b/Auxiliar.pde
index 44733d9..e2f9919 100644
--- a/Auxiliar.pde
+++ b/Auxiliar.pde
@@ -1,2 +1,6 @@
class Auxiliar{
+ private Calculo calculo;
+ public Auxiliar(Calculo calculo){
+   println("Se usará para realizar calculos auxiliares");
+   this.calculo = calculo;
+ }
}
```

Si quieres ver lo que has preparado y será incluido en la próxima confirmación, puedes usar `git diff --staged`. Este comando compara tus cambios preparados con la última instantánea confirmada. Aplicado a nuestro ejemplo sería:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git diff --staged
diff --git a/Calculo.pde b/Calculo.pde
index eaa7192..7de949d 100644
--- a/Calculo.pde
+++ b/Calculo.pde
@@ -1,2 +1,7 @@
class Calculo{
+ private float variableDependiente;
+
+ public Calculo(float variableDependiente){
+   this.variableDependiente=variableDependiente;
+ }
+ }
}
```

En ambos casos vemos que aparecen a la izquierda de las líneas signos + o -. El + indica una línea agregada o modificada, mientras que la símbolo – indica una línea eliminada.

- 4) Intente eliminar el archivo `Mensaje.pde` usando `git rm` y analice el resultado

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ ls
Auxiliar.pde  Calculo.pde  HolaMundoProcessing.pde  Mensaje.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git rm Mensaje.pde
rm 'Mensaje.pde'

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   Calculo.pde
    deleted:    Mensaje.pde

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   Auxiliar.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ ls
Auxiliar.pde  Calculo.pde  HolaMundoProcessing.pde
```

Como puede observar, el archivo ha desaparecido del working directory y además la acción debe ser confirmada, ya que se encuentra el archivo eliminado en el staging área. Esto significa que implícitamente ejecuta el comando `rm`. También significa que podemos revertir esta acción o confirmarla.



```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Calculo.pde
        deleted:    Mensaje.pde

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Auxiliar.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git commit -m "Se modifica Calculo.pde y se elimina Mensaje.pde"
[main 4d872cc] Se modifica Calculo.pde y se elimina Mensaje.pde
2 files changed, 5 insertions(+), 5 deletions(-)
delete mode 100644 Mensaje.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Auxiliar.pde
```

Además, se termina de confirmar los otros cambios

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git diff
diff --git a/Auxiliar.pde b/Auxiliar.pde
index 44733d9..e2f9919 100644
--- a/Auxiliar.pde
+++ b/Auxiliar.pde
@@ -1,2 +1,6 @@
 class Auxiliar{
+ private Calculo calculo;
+ public Auxiliar(Calculo calculo){
+   println("Se usará para realizar calculos auxiliares");
+   this.calculo = calculo;
+ }

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git add Auxiliar.pde

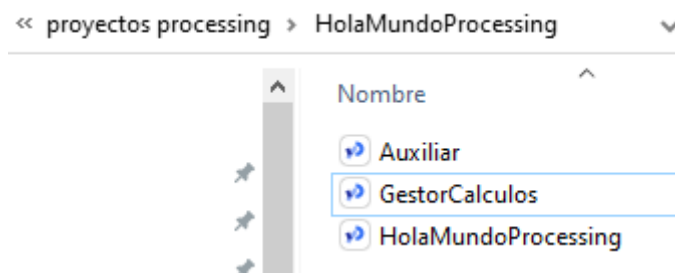
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git commit -m "Se modifica el archivo Auxiliar.pde"
[main e9d4dfd] Se modifica el archivo Auxiliar.pde
1 file changed, 4 insertions(+)

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git log --oneline
e9d4dfd (HEAD -> main) Se modifica el archivo Auxiliar.pde
4d872cc Se modifica Calculo.pde y se elimina Mensaje.pde
3260192 Se agrega el archivo Calculo.pde
4d727fc Se agrega el archivo Auxiliar.pde
c237d25 Se elimina el archivo OtroArchivo.pde
3d028cf Se agrega el archivo OtroArchivo.pde
65fd8e4 Se crea una clase Mensaje que se encarga de realizar el saludo
5b1e882 Agregando el archivo HolaMundoProcessing.pde y codificando el método setup
```

- 5) Cambiando el nombre de archivos: Al contrario que muchos sistemas VCS, Git no rastrea explícitamente los cambios de nombre en archivos. Si renombras un archivo en Git, no se guardará ningún metadato que indique que renombraste el archivo. Sin embargo, Git es bastante listo como para detectar estos cambios luego que los has hecho. Por ejemplo, si cambias el nombre del archivo en modo Windows el nombre del archivo Calculo por GestorCalculos obtendremos lo siguiente:  
Primero arreglamos el error que tenemos de haber eliminado el archivo Mensaje.pde

```
HolaMundoProcessing  Auxiliar  Calculo
1 //Mensaje unMensaje;
2 void setup(){
3     size(400,400);
4     background(0);
5     //unMensaje = new Mensaje();
6 }
```

Luego modificamos el nombre del archivo Calculo.pde



Finalmente eliminamos los errores de referencia en Auxiliar.pde y GestorCalculos.pde

<pre>HolaMundoProcessing  Auxiliar  GestorCalculos 1 class GestorCalculos{ 2     private float variableDependiente; 3 4     public GestorCalculos(float variableDependiente){ 5         this.variableDependiente=variableDependiente; 6     } 7 }</pre>	<pre>HolaMundoProcessing  Auxiliar  GestorCalculos 1 class Auxiliar{ 2     private GestorCalculos calculo; 3     public Auxiliar(Calculo calculo){ 4         println("Se usará para realizar calculos auxiliares"); 5         this.calculo = calculo; 6     } 7 }</pre>
---	---

Ahora veamos que nos dice Git

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Auxiliar.pde
        deleted:    Calculo.pde
        modified:   HolaMundoProcessing.pde

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

no changes added to commit (use "git add" and/or "git commit -a")
```

Para Git lo que se hizo fue eliminar Calculo.pde y luego agregar GestorCalculos.pde. Todo lo demás supone las modificaciones que hemos indicado.

Por lo que podemos realizar los comandos necesarios para que se confirmen las acciones:

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git add GestorCalculos.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git commit -m "Cambio nombre Calculo.pde -> GestorCalculos.pde y correccion de errores"
[main b8ca38b] Cambio nombre Calculo.pde -> GestorCalculos.pde y correccion de errores
1 file changed, 7 insertions(+)
create mode 100644 GestorCalculos.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos processing/HolaMundoProcessing (main)
$ git log --oneline
b8ca38b (HEAD -> main) Cambio nombre Calculo.pde -> GestorCalculos.pde y correccion de errores
e9d4dfd Se modifica el archivo Auxiliar.pde
4d872cc Se modifica Calculo.pde y se elimina Mensaje.pde
3260192 Se agrega el archivo Calculo.pde
4d727fc Se agrega el archivo Auxiliar.pde
c237d25 Se elimina el archivo OtroArchivo.pde
3d028cf Se agrega el archivo OtroArchivo.pde
65fd8e4 Se crea una clase Mensaje que se encarga de realizar el saludo
5b1e882 Agregando el archivo HolaMundoProcessing.pde y codificando el método setup
```





Sin embargo, Git provee el siguiente comando

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git mv GestorCalculos.pde Calculo.pde

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   Calculo.pde
    deleted:    GestorCalculos.pde

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   Auxiliar.pde
    modified:   HolaMundoProcessing.pde
```

El cual realiza ambas acciones, que sería el equivalente a lo siguiente:

```
$ mv GestorCaculos.pde Calculo.pde
```

```
$ git rm GestorCalculos.pde
```

```
$ git add Calculo.pde
```

Donde mv es un comando que permite modificar el nombre de un archivo.

Como puede observar, git mv realiza un conjunto de comandos. Vamos a confirmar los cambios

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git commit -m "Volvemos a cambiar nombre de archivo GestorCalculos.pde -> Calculo.pde"
[main 20dc765] Volvemos a cambiar nombre de archivo GestorCalculos.pde -> Calculo.pde
2 files changed, 2 insertions(+), 9 deletions(-)
delete mode 100644 GestorCalculos.pde
```

Note que estos cambios, no incluyen los nuevos cambios a realizar para corregir los errores que aparejan el cambio de nombre.

```
ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git log --graph --oneline
* afaa606 (HEAD -> main) Se modifican Calculo.pde y HolaMundoProcessing para corregir errores
* 20dc765 Volvemos a cambiar nombre de archivo GestorCalculos.pde -> Calculo.pde
* b8ca38b Cambio nombre Calculo.pde -> GestorCalculos.pde y correccion de errores
* e9d4dfd Se modifica el archivo Auxiliar.pde
* 4d872cc Se modifica Calculo.pde y se elimina Mensaje.pde
* 3260192 Se agrega el archivo Calculo.pde
* 4d727fc Se agrega el archivo Auxiliar.pde
* c237d25 Se elimina el archivo OtroArchivo.pde
* 3d028cf Se agrega el archivo OtroArchivo.pde
* 65fd8e4 Se crea una clase Mensaje que se encarga de realizar el saludo
* 5b1e882 Agregando el archivo HolaMundoProcessing.pde y codificando el método setup

ariel@Ariel MINGW64 /d/proyectosgit/proyectos_processing/HolaMundoProcessing (main)
$ git status
On branch main
nothing to commit, working tree clean
```

Con todos los elementos indicados se puede realizar el control de versionamiento de proyectos Processing para la rama main en una única máquina. En posteriores entregas se detallará el manejo de varias ramas, y además como gestionar el proyecto en un repositorio en la nube para que se pueda trabajar en modo colaborativo.