

# Conceptos principales

Git es un sistema de control de versiones que permite a los desarrolladores trabajar en colaboración en un mismo proyecto y mantener un historial detallado de todos los cambios realizados. Algunos de los conceptos básicos de Git incluyen:

- Repositorio: un espacio donde se almacena todo el historial de cambios de un proyecto.
- Commit: una confirmación de los cambios realizados en el repositorio.
- Branch: una rama independiente de desarrollo que se deriva del repositorio principal.
- Merge: la integración de dos ramas de desarrollo.
- Pull: la acción de obtener los últimos cambios de un repositorio remoto y fusionarlos con el repositorio local.
- Push: la acción de enviar los cambios realizados en el repositorio local a un repositorio remoto.
- Clone: la acción de copiar un repositorio remoto en el repositorio local.
- Header: Se refiere a un apuntador, nos indica en donde nos encontramos (versión de commit de la rama actual).
- Master: Se refiere a la rama principal de desarrollo.

Git también utiliza un sistema de identificación de versiones basado en SHA-1, que garantiza la integridad de los archivos y la seguridad de los datos. Además, Git cuenta con herramientas para gestionar conflictos y resolver problemas de integración en los cambios realizados por diferentes colaboradores en un proyecto.

Estos conceptos se estarán explicando y sus códigos en las secciones de esta documentación.

## Comandos de inicialización y configuración básica

Estos comandos son configuraciones iniciales que debemos usar en Git, es importante memorizarlos. **Y MUY IMPORTANTE** siempre el primer paso es configurar el nombre de usuario y el correo globales, sino, git no nos permitirá hacer commits porque no podrá identificar a la persona que realice cambios.

El git init es opcional dependiendo de si eres el dueño del repositorio o eres un colaborador, los colaboradores no necesitan hacer git init, pero eso lo veremos más adelante.

```
git init = Iniciar el repositorio en la carpeta seleccionada  
git config --global user.name "nombre_ej": permite configurar el nombre del usuario.  
git config --global user.email "correo_ej": permite configurar el correo del usuario.
```

Estos comandos son los básicos para empezar a trabajar nuestro git.

```
git add . : Nos permite añadir todos los cambios a la sección de staging.  
git commit -m "mensaje_ejemplo": nos permite crear un nuevo commit con los últimos cambios.  
git commit -am "mensaje_ejemplo": nos permite hacer un add y un commit al mismo tiempo.
```

La diferencia principal entre git add . y git commit -am es que el segundo comando realiza ambos pasos (agregar los cambios y hacer un commit) en un solo comando, mientras que el primer comando solo agrega los cambios al área de preparación, y aún necesitas usar el comando git commit para hacer el commit con un mensaje.

## Comandos para crear ramas de desarrollo

```
git branch (nombre deseado para branch): nos permite crear una nueva rama de desarrollo.  
git branch -d (nombre del branch): nos permite eliminar una rama creada anteriormente.  
git merge (nombre de la rama para el merge): Nos permite unir los cambios de una rama con otra.
```

## Comandos para gestionar Git

gitk: abre un software gráfico que nos da un log e igualmente una representación gráfica en arbol de nuestro git.

git status: Nos permite saber el status actual de nuestro repositorio.

git log: nos tener un listado de todos nuestros commits.

git log --oneline: nos permite tener un listado de nuestros commits pero resumidos, ideal si ya tenemos muchos commits.

Git checkout (hash del commit/branch): nos permite mover el Header de la branch actual hacia el último commit realizado de dicha rama, nos permite movernos entre versiones de nuestros o commits o bien a otra rama de desarrollo.

Alias: Ésto nos permite crear variables reutilizables con códigos de git, así no tendremos que escribir códigos super largos cada vez, veamos un ejemplo...

```
alias arbol ="nuestro código"
```

Ahora cuando escribamos la palabra "arbol" se ejecutará el código que hayamos puesto entre las comillas, sin tantas complicaciones.

## Eliminación de archivos

Git reset y git rm son comandos con utilidades muy diferentes, pero se pueden confundir muy fácilmente.

git rm --cached: Elimina los archivos de nuestro repositorio local y del área de staging, pero los mantiene en nuestro disco duro. Básicamente le dice a Git que deje de trackear el historial de cambios de estos archivos, por lo que pasaran a un estado `untracked`.

git rm --force: Elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

## Creación de repositorios remotos





# Llaves publicas y privadas

## Crear llaves localmente (localmente)

Lo primero que necesitamos para crear un repositorio remoto es crear unas llaves privadas esto es relativamente sencillo aunque hay que ser específicos con el código. El código para crear una llave ssh es con git es:

```
ssh-keygen -t rsa -b 4096 -C "correo_GitHub"
```

Una vez hecho eso, Git nos recomendará la carpeta donde guardar, así que solamente damos enter. Le podemos poner una contraseña a esa llave ssh, ésto es seguridad adicional, por practicidad no la pondremos, pero es recomendable sí ponerla, por buenas prácticas más que nada. La carpeta designada contendrá archivos como los que se muestran a continuación.

<input type="checkbox"/> Nombre ^	Fecha de modificación	Tipo	Tamaño
 id_rsa	18/02/2023 07:31 p. m.	Archivo	4 KB
 id_rsa.pub	18/02/2023 07:31 p. m.	Microsoft Publisher Document	1 KB
 known_hosts	18/02/2023 07:45 p. m.	Archivo	1 KB
 known_hosts.old	18/02/2023 07:45 p. m.	Archivo OLD	1 KB

id\_rsa: Llave privada > La llave que no debemos compartir y debemos cuidar demasiado.

id\_rsa.pub: llave pública > la que usaremos publicamente y sí podemos compartir, se usará para nuestras cuentas de GitHub u otras.

Necesitamos comprobar el proceso y agregarlo a Windows:

```
eval $(ssh-agent -s)
ssh-add ~/.ssh/id_rsa
```

El simbolo "~" es un shortcut absoluto, que buscará dentro de la carpeta home de nuestro sistema. El comando ssh-add añadirá nuestra llava privada, si tiene otro nombre hay que sustituirlo.

Para añadir el proyecto remoto usamos el siguiente código:

```
git remote add "nombre_ej" git@github.com:"nuestro proyecto"
```

Es importante notar que en "nombre\_ej" podemos poner cualquier nombre que queramos, en este caso se llamará teamProject pero puede ser cualquier nombre, como podemos ver a continuación.

Ejemplo (con el proyecto actual):

```
git remote add teamProject git@github.com:Baalgarthem/proyectoCompilador.git
```

Si en alguna ocasión necesitamos cambiar la dirección de nuestro proyecto remoto podemos usar el siguiente comando:

```
git remote set-url origin git://new.url.here
```

Donde "origin" sería el nombre que hayamos elegido, en este caso fue *teamProject*.

Si deseamos no cambiar la dirección sino eliminar nuestro vinculo remoto, podemos usar el siguiente comando:

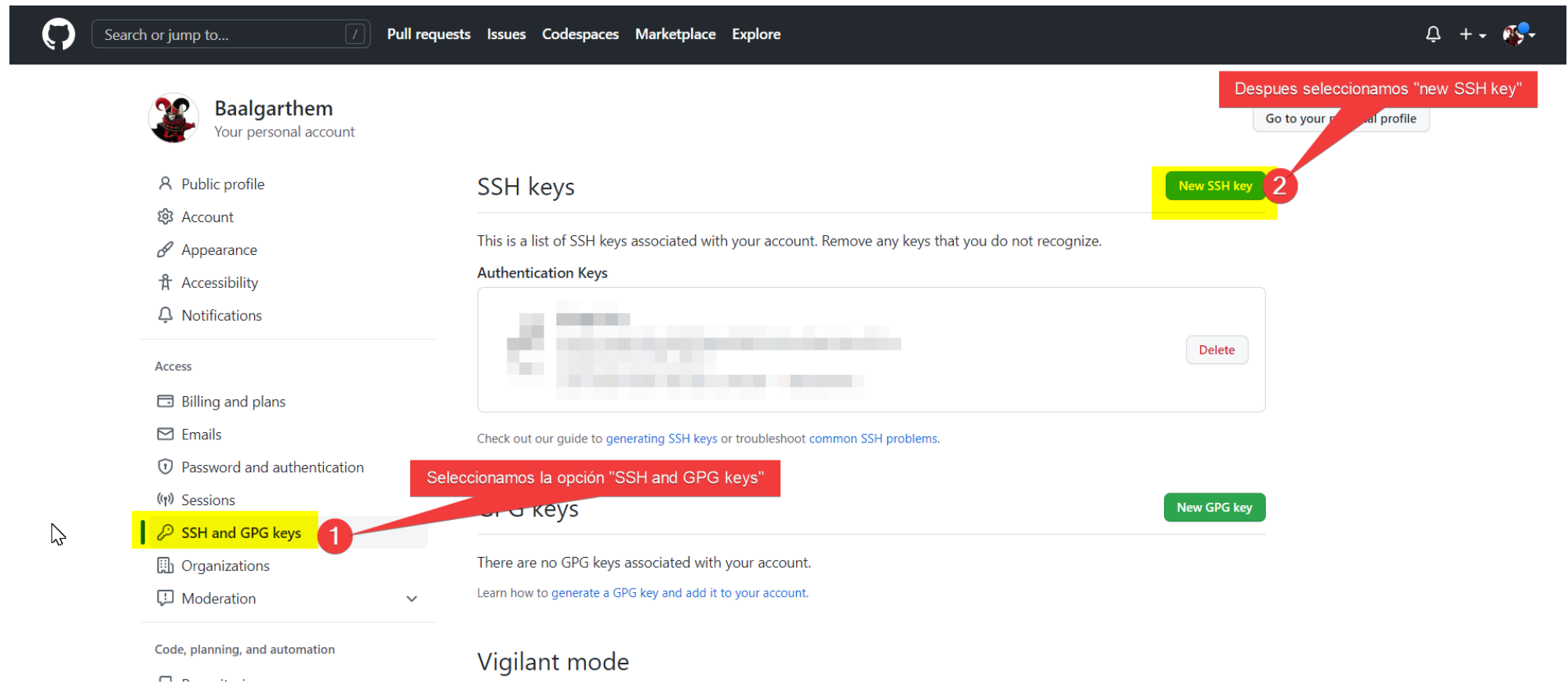
```
git remote remove origin
```

# Conectar con repositorio remoto

Ahora el siguiente paso importante para configurar nuestro repositorio con GitHub es ir a la página oficial:

<https://github.com/settings/keys>

Y hacer lo que se muestra en la siguiente imagen:



Cuando seleccionemos "New SSH key" metemos el contenido de nuestra llave publica. En la carpeta que creamos abrimos nuestra llava la que tiene el formato "pub" de publico. Es recomendable abrir el archivo con el bloc de notas, posterior a eso se copiará el contenido de ese archivo y se pegará en el recuadro de SSH keys, tal cual se muestra en la imagen de arriba.

Una vez hecho lo anterior y para comprobar que todo está correcto usamos el siguiente comando

```
git remote -v
```

Si hicimos todo correctamente, ya deberíamos poder hacer comandos remotos, tenemos dos direcciones remotas, una para hacer fetch y otra para hacer pull. Si nos aparecen como en la siguiente imagen, significa que ya terminamos de configurar el ámbito remoto.

```
6de27be se añade un documento de proyecto compilador, para estructurar y documentar el proyecto
Alush@Reign MINGW64 /d/Academico/Lenguajes y automatas 11/Proyecto Compilador (ejercicios)
$ git remote add teamProject git@github.com:Baalgarthem/proyectoCompilador.git

Alush@Reign MINGW64 /d/Academico/Lenguajes y automatas 11/Proyecto Compilador (ejercicios)
$ git remote -v
teamProject      git@github.com:Baalgarthem/proyectoCompilador.git (fetch)
teamProject      git@github.com:Baalgarthem/proyectoCompilador.git (push)

Alush@Reign MINGW64 /d/Academico/Lenguajes y automatas 11/Proyecto Compilador (ejercicios)
$
```

Si ejecutamos

```
git remote
```

veremos solamente la conexión remota (hacia nuestro GitHub) que hayamos configurado, tal cual se muestra a continuación:

```
Alush@Reign MINGW64 /d/Academico/Lenguajes y automatas 11/Proyecto Compilador (ejercicios)
$ git remote
teamProject
Alush@Reign MINGW64 /d/Academico/Lenguajes y automatas 11/Proyecto Compilador (ejercicios)
```

## Comandos necesarios para trabajar con repositorios remotos.

Para recapitular, entendamos que los comandos add, commit, trabajan sobre nuestros archivos locales. O sea que no afectarán al repositorio remoto de ninguna forma.

El comando para enviar archivos al repositorio remoto es "push". Mientras que el comando para jalar o bien descargar todos los cambios más recientes del repositorio es "pull". Siempre es recomendable que hagamos un pull (antes de comenzar a trabajar) para descargar los ultimos cambios antes de hacer un push, esto más que nada para no provocar cambios o incosistencias en el codigo (cuando se trabaja en equipos grandes, si son menos de cinco personas esto podría omitirse o bien si una sola persona se encargará de una rama, entonces no hay necesidad de cumplir esta regla sin embargo es una buena práctica y es recomendable acostumbrarse a ello)

```
git push teamProject master  
git pull teamProject master
```

Recordemos que teamProject es solamente el nombre que elegimos, casi siempre se suele llamar origin, pero podemos poner cualquier nombre que deseemos.

## Añadir colaboradores en en Github

Una vez que un miembro del equipo ya ha creado el repositorio, los demás miembros del equipo seguramente estarán emocionados por comenzar a usar la herramienta git. Cada miembro del equipo necesitaría una cuenta ya sea en GitHub, o en GitLab, o cualquier otro sistema de su preferencia, pero en esencia el funcionamiento es igual en todos lados. Así que en éste caso estoy usando GitHub.



Search or jump to... / Pull requests Issues Codespaces Marketplace Explore

Baalgarthem / proyectoCompilador Public

Pin Unwatch 1

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master 3 branches 0 tags

Go to file Add file <> Code

**Baalgarthem** Se marca el inicio del ejercicio 2 54be8b8 last week 11 commits

#Diagramas	cambios de ejemplo de prácticas	last week
#Documentación	Punto de respaldo del proyecto	last week
#Ejercicios en JavaCC	Se marca el inicio del ejercicio 2	last week
.gitignore	Se añade el primer ejercicio de los ejercicios en Java CC, en donde e...	last week

Help people interested in this repository understand your project by adding a README. Add a README

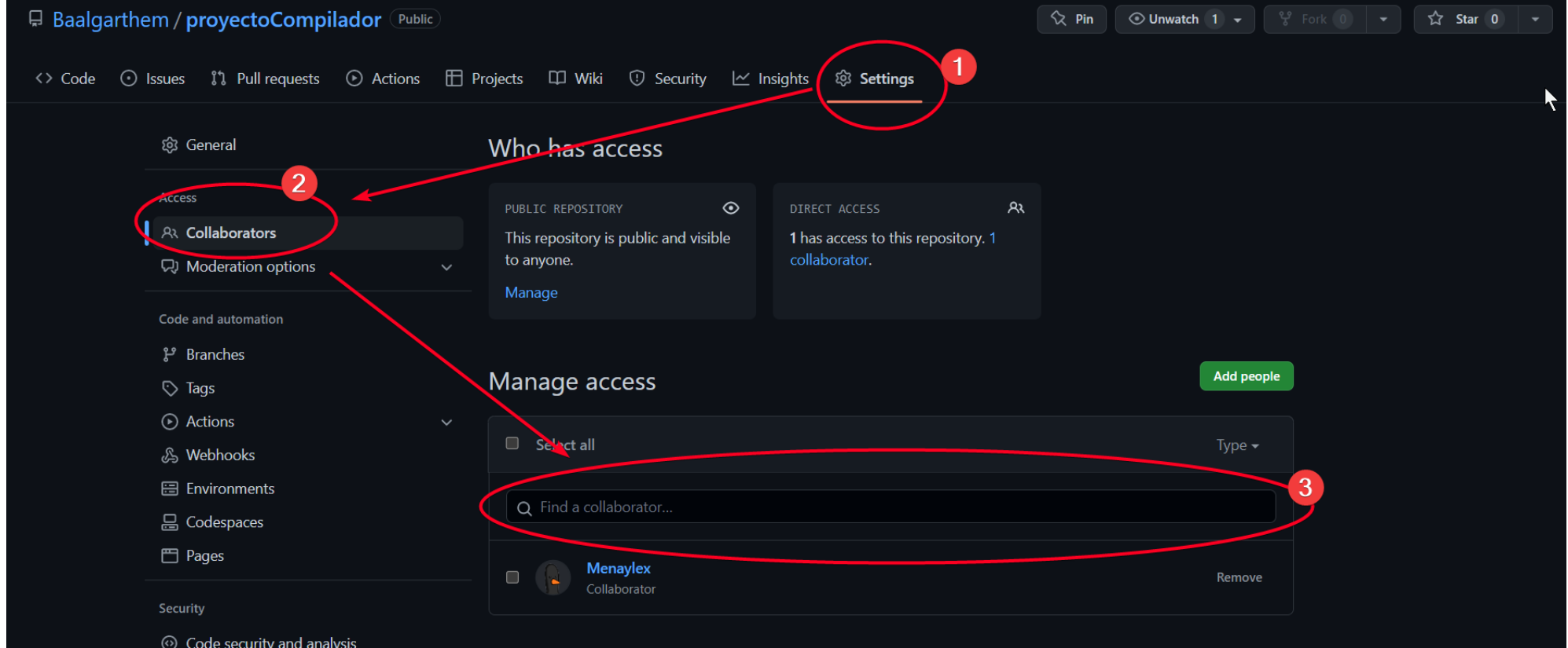
**About**  
Proyecto compilado  
automatas II | TECN

0 stars  
1 watching  
0 forks

**Releases**  
No releases published  
Create a new release

**Packages**  
No packages published  
Publish your first package

Para colaborar de forma remota en GitHub lo que se necesita es (desde las configuraciones de nuestro proyecto) añadir un colaborador, para esto el colaborador debe darnos su nombre de usuario en GitHub y debemos añadirlo como colaborador al proyecto. Una vez hecho eso esta persona ya tendría permisos de push.



Para comenzar a trabajar se necesita crear una nueva carpeta e iniciar git bash en ella. **PERO (IMPORTANTE LEER)** no debe hacer un git init, porque ésta persona no va a iniciar un repositorio, sino que va a jalar (pull) un repositorio ya existente. Si se inicia un repositorio y luego se hace un clone provocará una inconsistencia en git que nos dirá "estas creando un repositorio dentro de otro repositorio". Para evitar ésto, simplemente no hagamos un git init. Eso se utiliza solamente cuando nosotros vamos a ser los masters.

En resumen:

- Para añadir un colaborador, la forma más fácil es con su nombre de usuario publico en github, desde las configuraciones (de nuestro repositorio al que lo queremos añadir) y buscando su nombre de usuario.
- Un colaborador no debe usar el comando "git init"
- Una vez que el colaborador ha sido invitado (y aceptó) a participar en el proyecto debe hacer un pull por primera vez.

- Finalmente, debe hacer su primer push y checar el log para comprobar que sus cambios se han realizado con éxito.

# Trabajar con ramas en Git y GitHub

Para trabajar con ramas tanto en git como en github necesitamos un conjunto de codigos que se diferencia entre, codigos locales y codigos remotos. Los locales trabajan unicamente sobre nuestro equipo, donde tengamos nuestros archivos de trabajo, mientras que los comandos remotos afectarán al repositorio online.

## Ramas Locales

Para saber cuales ramas existen en nuestro repositorio local podemos escribir este comando en nuestro git bash.

```
git branch
```

Eso nos mostrará un listado de todas las ramas que tenemos actualmente, si en cambio queremos crear una nueva rama es bastante sencillo

```
git branch nombre_rama
```

Solamente tendríamos que reutilizar el comando branch y añadirle el nombre de la rama que queremos crear. ¿Pero y qué pasa si queremos borrar la rama porque ya no la vemos necesaria? eso es simple también. Hacemos el mismo comando pero añadiendo una flag -d (de delete) en caso de que haya complicaciones git nos podría recomendar usar una forma forzada de eliminación -D. Veamos los códigos.

```
git branch -d nombre_rama : este codigo borrará una rama
```

```
git branch -D nombre_rama : este codigo también borrará una rama (pero de forma forzada)
```

Cabe recalcar, que al ser una bandera, no importa si la ponemos despues o antes del nombre, lo importante es ponerla. También, podemos añadir muchas banderas a diferentes comandos en git. Si esto se combina con el comando "alias" puede ser una forma increíblemente rapida y efectiva de trabajo. En todo caso, continuemos.

Si queremos renombrar una rama podemos usar el siguiente comando:

```
git branch -m nombrePrevio nombreNuevo
```

Hay un comando que nos permite unir cambios de una rama con otra, es un comando común y que se usará a menudo una vez que se ha iniciado en Git, este comando es es:

```
git merge nombre_rama
```

`git merge` es un comando de Git que permite combinar cambios de diferentes ramas. La idea detrás del merge es que puedes trabajar en diferentes ramas en paralelo y luego combinar los cambios en una sola rama, para así mantener todo el historial de cambios.

El proceso de `git merge` implica tres pasos:

1. Primero, se debe asegurar que la rama que se quiere mezclar se encuentra actualizada con los cambios más recientes de la rama destino. Esto se puede hacer con el comando `git fetch`.
2. A continuación, se debe ir a la rama destino y usar el comando `git merge` para mezclar los cambios de la otra rama. Esto crea un commit de "mezcla" que combina los cambios de ambas ramas y crea una nueva instantánea en la rama destino.
3. Finalmente, si hay conflictos entre los cambios de ambas ramas, Git muestra un mensaje para que el usuario resuelva los conflictos manualmente. Una vez que se resuelven los conflictos, se puede realizar un commit de mezcla para finalizar el proceso.

Es importante tener en cuenta que `git merge` puede generar conflictos que deben ser resueltos manualmente. Estos conflictos ocurren cuando dos ramas han modificado la misma sección del código de manera diferente. En estos casos, el usuario debe decidir cómo mezclar los cambios para poder continuar con el proceso de merge.

En resumen, `git merge` es una herramienta útil para combinar diferentes cambios en diferentes ramas en una sola rama. Permite mantener el historial de cambios y trabajar en paralelo, aunque es importante estar atento a los posibles conflictos que puedan surgir.

## Ramas Remotas

Antes de enviar cambios al repositorio remoto es recomendable hacer un pull (descargar los últimos cambios)

el comando pull se puede usar así

```
git pull nombre_proyecto
```

(donde proyecto llevará el nombre del vínculo remoto que hayamos creado con anterioridad)

Después podemos usar el git push para enviar los cambios, todos los cambios, si ocupamos enviar solo una rama podemos añadir las ramas

```
git push nombre_proyecto nombre_rama
```

El nombre de rama es solamente si queremos enviar una rama, si queremos enviar todos los cambios que haya, entonces se envía solamente el push del proyecto, sin embargo por buena práctica es mejor solamente enviar cambios de las ramas que se nos han asignado, de esa forma no estorbaremos ni crearemos conflictos con otros archivos del proyecto.

Cuando haces un pull a un repositorio de GitHub, se traerá la rama en la que estás trabajando actualmente y sus cambios asociados, pero no necesariamente todas las ramas del repositorio.

Sin embargo, si ejecutas el comando:

```
git pull --all
```

Entonces se descargarán todas las ramas del repositorio. De esta manera, tendrás acceso a todas las ramas locales del repositorio en tu sistema y podrás trabajar con ellas.

**¿Que comando de git puedo usar para que se envíe mi repositorio local tal cual está y forzar al repositorio remoto a eliminar cualquier cosa que tenga diferente de mi repositorio local?**

```
git push --force
```

Ten en cuenta que este comando es peligroso, ya que puede sobrescribir cualquier cambio que se haya realizado en el repositorio remoto sin advertencia. Por lo tanto, es recomendable tener cuidado al utilizar este comando y asegurarse de que no se estén perdiendo cambios importantes. Además, es importante comunicar a los colaboradores del repositorio que se está utilizando este comando para evitar conflictos.

Es importante tener en cuenta que, en algunos casos, es posible que necesites ejecutar el comando `git fetch` para actualizar tu repositorio local con las ramas nuevas del repositorio remoto. Una vez que hayas hecho esto, puedes utilizar el comando `git branch -a` para listar todas las ramas locales y remotas en tu repositorio y elegir la rama que deseas usar con el comando `git checkout`.

## Como limpiar archivos innecesarios de nuestro Git

En el flujo de trabajo, quizás hayamos creado archivos de más, o simplemente querramos eliminar cualquier cosa que no estemos usando en nuestro proyecto, para ello podemos usar los siguientes comandos

```
**git clean --dry-run** Para visualizar qué archivos se van a borrar. Archivos, no carpetas  
**git clean -f** Para borrar los archivos que visualizaste: Ignorará lo que esté dentro del archivo  
.gitignore  
**git clean -df** Borra archivos y carpetas
```

## Como restaurar todo a un estado anterior

Esta es la parte que a todos les debe interesar y seguramente lo que más les emociona a todos, como bien sabemos los commits nos ayudan a hacer una especie de punto de control al que le añadimos al comentario ¿Qué pasa cuando (desde nuestra rama alternativa o nuestra rama master) echamos algo a perder y ya nada funciona? y ahora necesitamos volver a ese tiempo en donde las cosas sí funcionaban.

Para aplicar ese trágico y caotico concepto de tener que reiniciar todo (reiniciarlo desde el punto 'commit' que nosotros deseemos) podemos usar unos simples comandos, los cuales se listarán a continuación:

```
git reset --HARD HS1
```

En donde hs1 es el hash de nuestro commit, simplemente podemos copiarlo y pegarlo. La bandera --HARD forzará el reinicio. Es muy importante saber que cuando usemos un reset perderemos todos los commits anteriores que hayamos hecho. Volveremos a ese punto, pero perderemos todos los commits que hayamos hecho antes de ese punto. Veamos una imagen representativa.

```
bash: -git: command not found
Alush@Ocean MINGW64 /d/Academico/Lenguajes y automatas 11/Proyecto Compilador (master)
$ git clean --dry-run
Alush@Ocean MINGW64 /d/Aca
$ git clean -f
Alush@Ocean MINGW64 /d/Academico/Lenguajes y automatas 11/Proyecto Compilador (master)
$ git log --oneline
73821e (HEAD -> master) Para master actualizada y unificada con las otras ramas, se ha actualizado la estructura de la carpeta de trabajo, se creó un archivo log para anotar registros (para los usuarios en gen
7a4a3ba (proyectoAutomatas/master) A punto de hacer merge
e99f6e6 el ejercicio 11 faltaba, ya se terminó
7b99ed9 Ejercicio 10 terminado, todos los ejercicios han sido terminados, ahora se procede a hacer revisión y unir con master en prontitud, despues de revisión el día de 09/03/2023
962a784 Ejercicio 9 terminado, se revisaron los demas ejercicios para ver posibles mejoras, se continua con el ejercicio 10
c150c02 Ejercicio 7 terminado
8d4ab21 Ejercicio 5 terminado, revisando otros pendientes
9b63628 Ejercicio 18 terminado, quedan pendientes solo unos cuantos, revisando problemario
0c4d5a7 Ejercicio 17 terminado, se sigue al ejercicio 18
caa9749 Ejercicio 16 terminado, sigue el ejercicio 17
b7866b8 Ejercicio 15 terminado, sigue el ejercicio 16
e90b362 terminé el ejercicio 14, sin embargo queria añadir una funcionalidad extra, la dejaré pendiente
dc197cf Ejercicio 13 ya terminado, prosigue: ejercicio 14
e893a9a Ejercicio 12 ya terminado, se sigue al ejercicio 13
1e9fd10 Ejercicio 3 terminado
14a2e3e Ejercicio 4 resuelto, se re-escribio el codigo 4 y se mejoró su sintaxis, ahora se declara como terminado
c340f88 Se terminó el ejercicio número 4 y ahora hay que re-escribirlo
3526d8c Ejercicio 6 final
3dd342d se ordenaron nuevamente los ejercicios, el ejercicio 1 ya funciona y ahora empezaremos con el ejercicio 2
edc2e02 1.1 Se hizo un token y un ejercicio para contar numeros pares, ya que se ha logrado compilar se puede avanzar a realizar diferentes ejercicios
dde4c79 se logra compilar (aunque con errores) para saber numeros pares)
3920223 desarrollando el problemario
8794a31 creando un main que pueda leer los tokens del proyecto
299e6b5 tutoriales terminados con todos sus archivos y clases
0a0d098 ejercicios tutoriales terminados
600f800
20383f5
c0a56ed
54be8b8
bdd6eee
ab73fe5 cambios de ejemplo de prácticas
e6e6ad0 Ejercicio 1 terminado
c4e0ca1 Ejercicio 2 terminado, muchas gracias
7b8ec18 Hice un ejercicio de prueba para explicarlo
b139021 Punto de respaldo del proyecto
8db1cbf Se termina el ejercicio 1 de JavaCC, su documentación y código
fbd2dcc Se añade el primer ejercicio de los ejercicios en Java CC, en donde el archivo test.jj se encargará de lanzar mensajes de errores
ba94ecb Carpeta de ejercicios en JavaCC añadida
6de27be se añade un documento de proyecto compilador, para estructurar y documentar el proyecto
```

Esto es un hash, es el que debemos copiar y pegar para poder volver a ese estado

Suponiendo que hacemos un reset a éste punto  
entonces perderiamos todos los commits que vemos encerrados en verde, para siempre, eliminados sin poder hacer nada.

A grandes rasgos, con todos estos comandos, ya se puede en esencia trabajar de forma colaborativa usando git y apoyandonos con github, podemos enriquecer el aprendizaje, hacer flujos de trabajo profesionales, evidenciar el trabajo de varios integrantes del equipo así como resolver problematicas y asignar tareas de forma eficiente y eficaz.

# Solución de errores comunes en Git

## Eliminar referencias invalidas

Primeramente este codigo nos ayudará a conocer las referencias invalidas, lo que obtengamos de aqui lo podemos pegar en un archivo de texto:



```
git for-each-ref --format='%(refname) %(objectname)' refs | awk '$2 ==  
"0000000000000000000000000000000000000000" {print $1}'
```

Ahora veamos una explicación de como funciona este codigo.

- git for-each-ref es un comando que lista referencias de Git. En este caso, estamos diciéndole que liste todas las referencias (tanto branches, tags, etc.) en el repositorio.
- --format='%(refname) %(objectname)' es una opción que le dice a Git cómo queremos que nos muestre la información de cada referencia. En este caso, le estamos pidiendo que nos muestre el nombre completo de la referencia (es decir, el refname) seguido del hash SHA-1 del objeto al que apunta la referencia (es decir, el objectname).
- refs es el ámbito de las referencias que queremos listar. En este caso, estamos diciéndole que liste todas las referencias que comienzan con refs/ (lo que incluirá todas las referencias de branches, tags, remotes, etc.).
- | es el símbolo de la tubería (pipe), que toma la salida del comando anterior y la pasa como entrada al siguiente comando.
- awk es un programa de procesamiento de texto que se utiliza comúnmente en Unix/Linux. En este caso, estamos utilizando awk para filtrar la salida de git for-each-ref.
- '\$2 == "00" {print \$1}' es una expresión awk que le dice a awk qué hacer con cada línea de entrada. En este caso, estamos diciéndole que si la segunda columna (es decir, el objectname) es igual a una cadena de ceros, entonces imprima la primera columna (es decir, el refname). Esta expresión filtra todas las referencias que apuntan a un objeto inexistente (es decir, el objeto "00"), que es lo que ocurre cuando eliminamos una referencia de Git.

En resumen, el comando de Git git for-each-ref lista todas las referencias del repositorio, --format especifica el formato en el que se mostrarán las referencias, refs establece el ámbito de las referencias, y la tubería con awk filtra las referencias que apuntan a objetos inexistente.

Ahora con esos datos obtenidos de nuestro archivo de texto, podemos usar algo como lo que veremos a continuación para eliminar una o más referencias invalidas de nuestro repositorio (Las referencias invalidas suceden cuando eliminamos archivos de forma manual sin usar el comando rm que trae git):

```
$ rm -f .git/refs/desktop.ini .git/refs/heads/desktop.ini .git/refs/remotes/desktop.ini  
.git/refs/remotes/proyectoAutomatas/desktop.ini .git/refs/tags/desktop.ini
```

En este caso yo use mis propias referencias invalidas para arreglar los problemas, este comando es peligroso porque no pide confirmaciones y borra tajantemente las referencias invalidas, es muy importante estar seguros de que esos archivos no son importantes para nosotros y no afectarán en nada al repositorio, de lo contrario es mejor hablarlo con el equipo de trabajo y pedir asesorias sobre alternativas.

Despues de limpiar las referencias invalidas se puede hacer un nuevo commit, hacer un push y luego un fetch para comprobar que todo esta funcionando bien en el envio y recepción de datos de nuestro repositorio.

La detección de referencias inválidas en un proyecto de Git es importante por varias razones:

La importancia de detectar las referencias invalidas en git incluyen lo siguiente.

- **Integridad de los datos:** las referencias son una parte fundamental de la estructura interna de un repositorio Git, ya que almacenan información acerca del historial y estado del código. Si las referencias se corrompen o se vuelven inválidas, esto puede afectar la integridad de los datos y llevar a situaciones en las que se pierdan cambios o se produzcan errores en la aplicación.
- **Rendimiento del repositorio:** las referencias inválidas pueden tener un impacto negativo en el rendimiento del repositorio, ya que pueden hacer que las operaciones de Git sean más lentas o incluso provocar que se produzcan errores durante la ejecución de las mismas.
- **Calidad del código:** la detección y corrección de referencias inválidas puede mejorar la calidad del código, ya que ayuda a prevenir errores y asegura que el historial del repositorio esté completo y sin corrupciones.

En resumen, la detección de referencias inválidas es importante para garantizar la integridad, el rendimiento y la calidad del código en un proyecto de Git.

Git es una herramienta que en un ambito de proyectos, es de gran ayuda no solo para alumnos, sino tambien para docentes. Al mismo tiempo que es una buena práctica en los entornos estudiantiles y laborales.

—Ing. Arturo Ramirez, TECNM - ALVARADO , 2023