**CURSO DE GIT Y GITHUB**

**Configuración.**

Lo primero que hay que hacer una vez instalado git, es abrir **git bash** y configurar cual va a ser nuestro **alias** y nuestro **correo electrónico**. Porque cuando nosotros hacemos un cambio (como subir un archivo a un repositorio, realizar una edición o crear una rama nueva, etc) ese cambio va a tener nuestra firma. esta firma va a tener nuestro correo electrónico y nuestro alias, entonces si no definimos por defecto cual va a ser nuestro correo y nuestro alias, git no va a saber a quien atribuirle los cambios.

Otra cosa importante es que nuestro repositorio de Github y el de Git deben tener el mismo correo.

Existen 3 tipos de jerarquías, que son: local, system, y global.

**Local:** es el repositorio en el que estamos trabajando en nuestra computadora (osea digamos es como tener una carpeta). Cada repositorio local tiene una firma diferente.

**Global:** configura una misma firma para todos los repositorios de un usuario del sistema. Osea que todos los repositorios que tengamos en un usuario x de nuestra pc tienen nuestra firma.

**System:** configura una misma firma para todos los repositorios de todos los usuarios en el sistema.

algo muy importante a destacar es que estas jerarquías están organizadas de manera que se les de **prioridad a la configuración más chica**, es decir, es posible tener una configuración inferior que sobreescriba la superior. O sea que podemos tener una firma configurado en system y otra firma configurada en global, lo mismo con global y local.

Para elegir que configuración usar escribimos:

$git config --jerarquía

Todas las configuraciones que se hagan a partir de este comando, se harán en la jerarquía indicada.

Por defecto, lo más común es usar **global** para que todos los repositorios de nuestro usuario funcionen así.

para modificar los datos de nuestro usuario usamos estos comandos (siguiendo el comando anterior):

$git config --global user.name "nombre de usuario"

Elegimos nuestro email:

$git config --global user.email "correo"

Y listo, ya tenemos configurado nuestro nombre de usuario o alias, y nuestro email. Cabe mencionar que para modificar un dato del usuario hay que usar **user.algo** (si lo pensás, es parecido a llamar a la propiedad de un objeto).

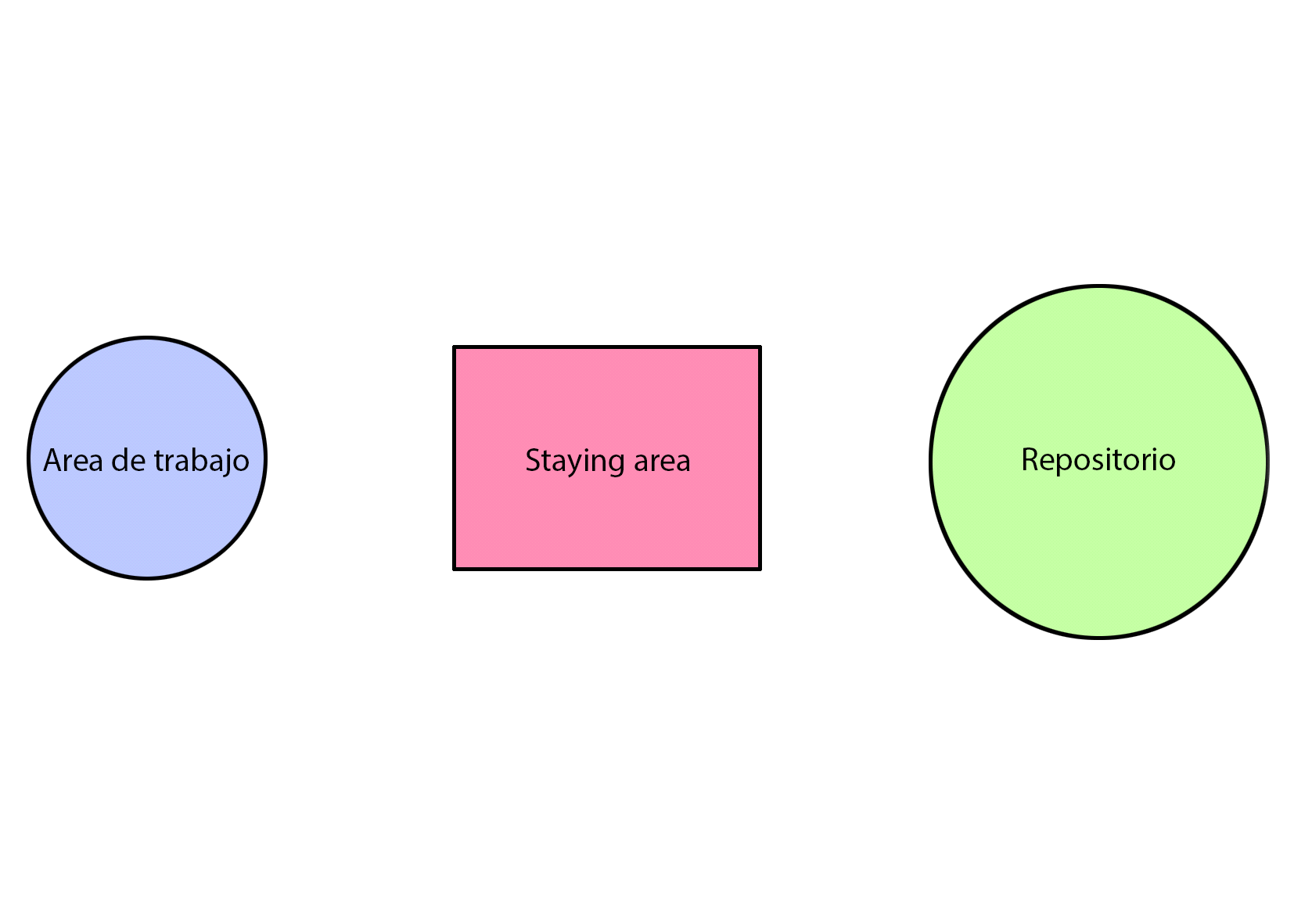
Configuración de compatibilidad entre sistemas: Esto es un detalle que normalmente pasa desapercibido, pero que es importante para que la persona que baje nuestros archivos en su sistema pueda utilizarlos sin problemas, ya sea windows, macOS, o Linux.

$git config --global core.autocrlf true

**Creación de repositorios.**

Empecemos por lo más básico, ¿Qué es un repositorio? básicamente es un espacio que usamos para guardar nuestros archivos, datos, codigos fuente, imagenes, diseños, etc. Lo que necesitemos guardar. Y este nos permite llevar un control de versiones, podemos pensar esto como los checkpoints de los juegos, al llegar a cierto punto guardamos la partida y podemos volver a ese momento en el tiempo para hacer las cosas de otra manera y tal.

Existen tres areas de en las que se guardan nuestros archivos, y son: el area de trabajo, el staying area, y el repositorio.



**Pase al staying area:**

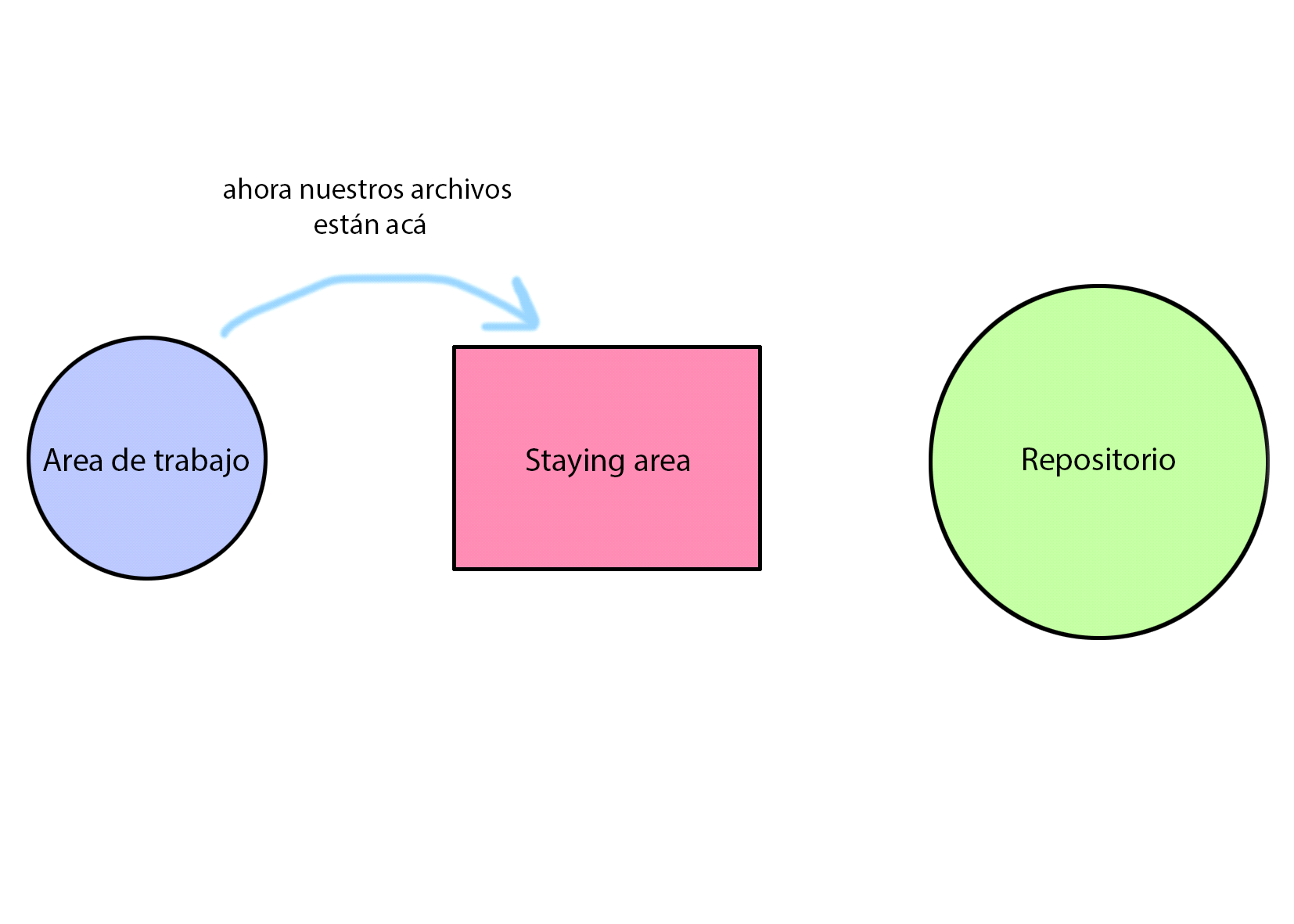
Area de trabajo: es una carpeta situada en algún lugar de nuestra computadora, como el escritorio por ejemplo, en donde nosotros codeamos normalmente.

Staying area: es el punto medio entre el area de trabajo y repositorio, a donde van nuestros archivos antes de subirse al repositorio. acá es donde los archivos que modificamos durante nuestro día de trabajo se analizan y prueban para ver si están aptos para ser subidos al repositorio.

Repositorio: es el lugar en el que se encuentran los archivos con el proyecto terminado.

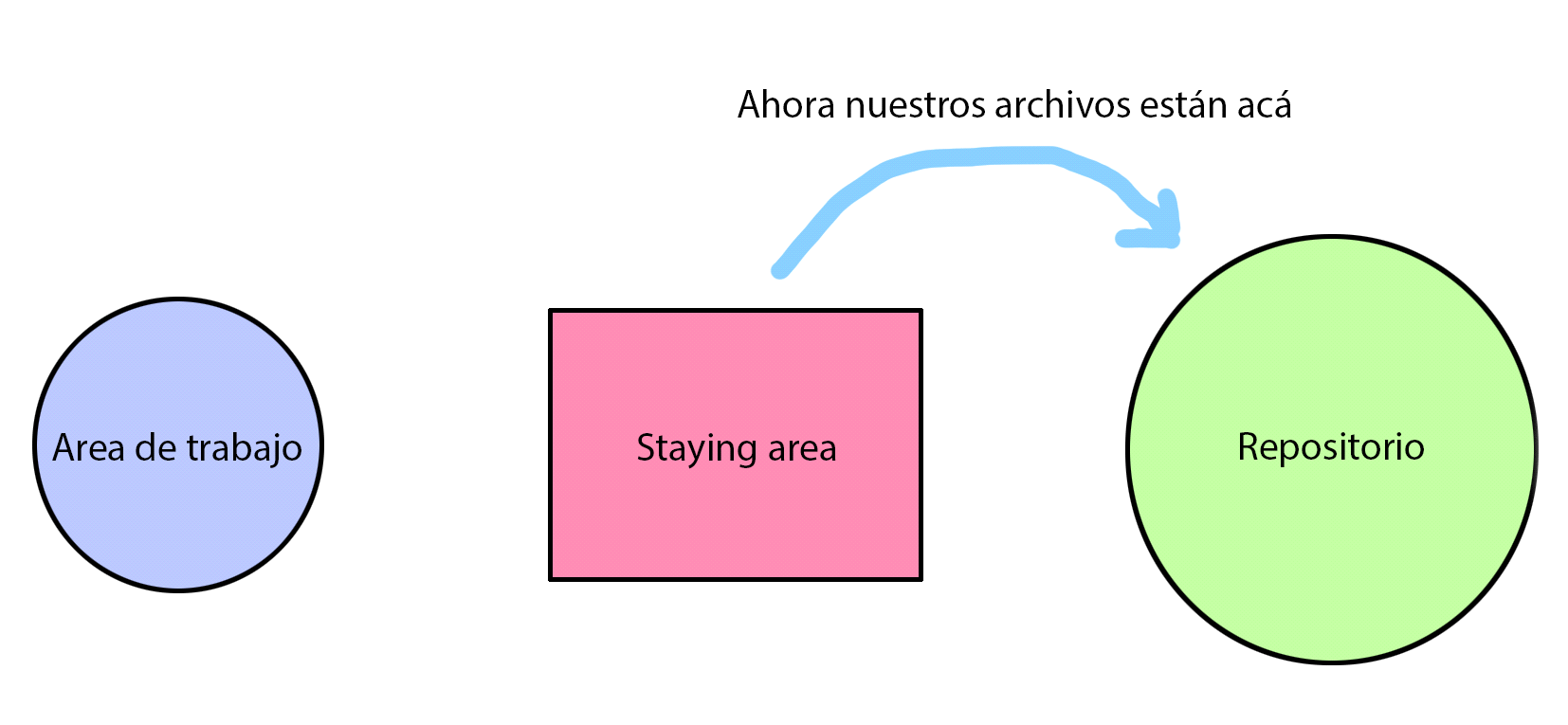
Para iniciar nuestra area de trabajo tenemos que movernos en git hasta la carpeta que queremos utilizar para estas tareas, e introducir el comando **git init**. De esta manera inicializamos la rama master, que vendría siendo como la linea de tiempo principal. Si en la carpeta que elegimos se creó una carpeta oculta .git, significa que está todo ok.

Ahora, para añadir un archivo al staying area usamos **git add nombreArchivo** (o add . si queremos añadir todos los archivos). También podemos concatenar archivos mediante espacios, de esta manera **git add nombreArchivo nombreArchivo2 nombreArchivo3** y así sucesivamente (lo mismo se puede hacer para removerlos con **rm --cached**).

Una vez hecho esto podemos ver con **git status** los archivos que están, o no, en el staying area. Si los nombres de los archivos se muestran en rojo significa que NO ESTÁN en el staying area, y si se muestran en verde significa que SI LO ESTÁN. Los archivos que estén en rojo no se subirán al repositorio al hacer el commit.

**Pase al repositorio:**

Una vez que los archivos estén en el staying area podemos mandarlos al repositorio con **git commit**, este comando también puede escribirse como **git commit -m "mensaje"** para incluir un mensaje corto que acompañe nuestro commit, y de esa manera, tener una pequeña descripción de que cambios se agregaron en ese commit. En caso de querer commitear algo sin tener que pasar por el staying area (no es recomendable a menos que sea el primer commit del proyecto) podemos ingresar **git commit -a**.



En caso de querer eliminar un archivo de un repositorio, podemos eliminarlo en la carpeta del area de trabajo y luego añadirlo al staying area, de esa manera notificaremos a git nuestra desición de eliminar ese archivo y al no encontrarlo en nuestros cambios lo eliminará del repositorio final.

Para recuperar el archivo podemos usar **git restore nombreArchivo** esto restaura el archivo que subimos previamente al staying area.

Para volver a nuestro último commit podemos usar **git checkout**, podemos ver esto como un viaje en el tiempo hacia el pasado. Puede usarse para restablecer una versión anterior de todos los archivos, o concaternarse con el nombre de uno o más archivos específicos para restablecer solo esos.

Para volver a una versión anterior en el area de trabajo, podemos usar **git reset --hard,** este comando saca todo del staying area y lo restaura al último commit que hicimos.

Para cambiarle el nombre a un archivo podemos usar **git mv archivoAEditar nombreNuevo.**

**Git diff:**

Para usar este comando primero debemos saber que archivos están commiteados. Para esto podemos usar **git show** que nos muestra la información de un archivo que YA ESTÁ commiteado. O sea que si commiteo un archivo que diga "Hola", y lo leo con git show, me va a mostar el "hola". Pero si lo edito a "hola2", no lo commiteo, y lo leo con git show me va a mostrar la versión original.

El comando **git diff --staged** lo que hace es mostrarnos la diferencia entre un archivo que está en el staying area, y uno que está en el commit. Los archivos del staying area se mostrarán en verde, mientras que los que están en el commit se mostrarán en rojo.

Para elegir que commits queremos comparar necesitamos saber la identificación del commit, para esto usamos **git log**, este comando nos va a mostrar la identifiacación de nuestros commits, el commit en el que estamos situados en el presente, los datos del autor, la fecha y hora en la que se hizo el commit, y el mensaje que le dejamos.

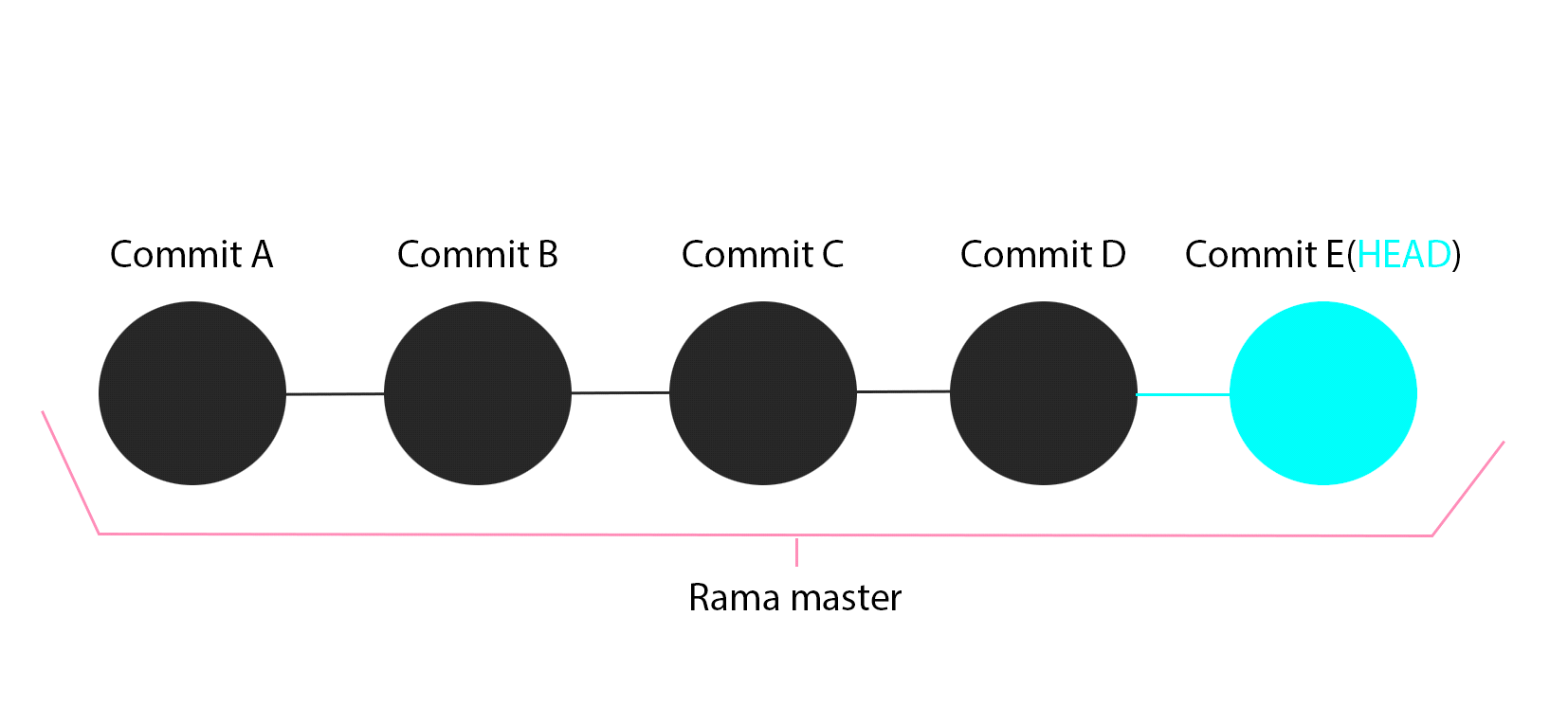
Algo como esto:

commit 4427bc7fbf483a81ebad90075327ed5921976edd (HEAD -> master)

Author: Alias <Email>

Date: Tue January 29 16:29:22 2024 -0300

miMensaje

Head indica que estamos situados en ese commit, y master indica que estamos en la rama master.

Para simplificar esta información podemos ingresar **git log --oneline** que nos va a devolver la lista de commits pero más resumida, y el codigo de identificación se reducirá a los primeros 7 caracteres. O sea que del codigo anterior solo quedaría 4427bc7 (este codigo se le llama hash). Algo importante a mencionar es que hay cierta probabilidad de que el identificador de dos o más identificadores coincidan, mientras más caracteres pueda abarcar el código, menor es esta probabilidad. Si trabajamos en un servidor con muchos commits lo mejor es extender el código, pero como vamos a trabajar en proyectos pequeños no hace falta, incluso podemos darnos el lujo de acortarlo.

Para extender o acortar el código podemos ingresar **git config --global core.abbrev numeroDeCaracteresDeseado** y elegir la cantidad de caracteres que queremos que se muestren.

Entonces para ver la comparación entre dos o más commits y su información ingresamos **git diff codigo1 codigo2**. Si solo quisieramos ver el nombre de los commits podemos escribir **git diff --name-only codigo1 codigo2**.

Si escribimos **git diff --word-diff codigo1 codigo2** se nos van a mostrar las diferencias entre un archivo y otro línea por línea, o sea, nos muestra los cambios que hicimos. El texto en color rojo corresponde al commit del **codigo1**, y el texto verde corresponde al del **codigo2**.

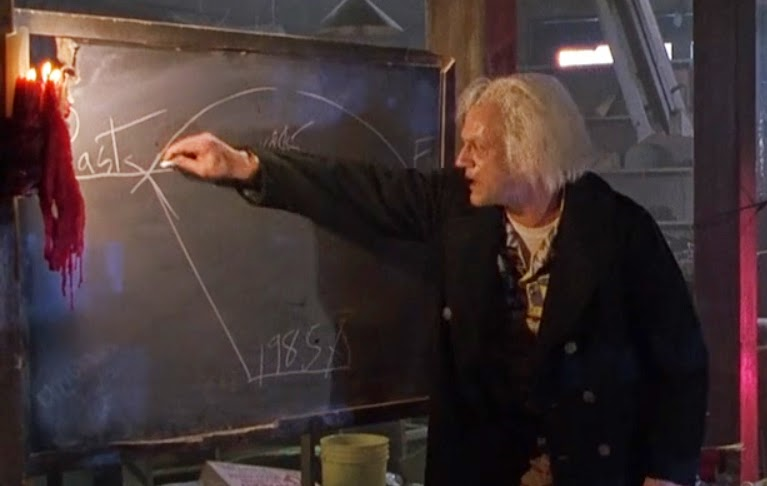
**Modificar y deshacer commits:**

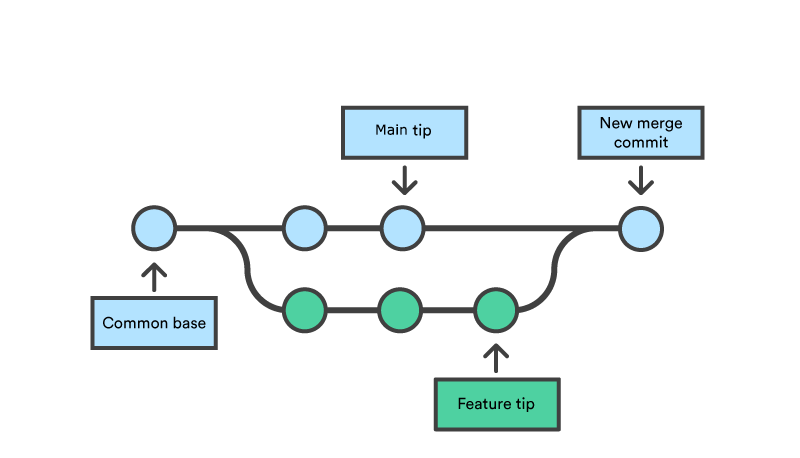
Para deshacer uno o más commits previos al Head podemos usar **git reset --soft codigo (o soft head-numeroDeCommitsADeshacer)**, hay que tener en cuenta que una vez hecho esto, los commits previos al Head que hayamos elegido desaparecerán. Otra cosa importante es que este comando, aparte de deshacer commits, traslada el commit Head al staying area, por si queremos hacerle algún cambio. Luego podemos crear el commit y listo.

Algo a destacar es que los archivos que no estén repetidos permanecerán en el nuevo Head.

**Branchs (ramas):**

Ya que conocemos lo que es una rama, más específicamente, lo que es la rama principal, lo que sigue son las ramas paralelas o bifurcaciones. Podemos pensar esto como una tangente, es decir una linea del tiempo alterna en la que podemos agregar funcionalidades nuevas y hacer pruebas, y una vez que estémos seguros de que queremos añadir esas funcionalidades al programa final, podemos unir esa rama con la rama principal.





Para ver las ramas que tenemos creadas podemos usar el comando **git branch**, el nombre de rama que esté en verde indica que estamos trabajando en esa rama.

Y para crear una rama nueva escribimos **git branch nombre-rama,** es una buena practica escribir los nombres de las ramas separando las palabras con guiones. A este formato se le llama “Kebbab case”. Entonces esta nueva rama que creamos va a ser una copia de la rama principal que utilizaremos para hacer cambios sin tocar la rama principal.

Para movernos a la rama que hayamos creado, ingresamos **git switch nombre-rama** y al usar **git branch** nuevamente vamos a ver que el nombre de nuestra nueva rama ahora está en verde. Así mismo podemos crear ramas de las ramas.

Una manera más corta de crear una rama y moverse a esta es usando **git switch -c rama-nueva**.

Para eliminar una rama primeramente debemos estar afuera de la misma, ya que si eliminamos la rama en la que nos encontramos posicionados vamos a tener problemas. Una vez fuera usamos el comando **git branch -d nombre-rama** y ya quedaría eliminada.

Para modificar el nombre de una rama de la lista podemos usar **git branch -m nombre-rama nuevo-nombre**. En caso de querer modificar el nombre de la rama en la que nos encontramos posicionados, podemos usar **git branch -m nuevo-nombre**.

Ahora, para fusionar una rama (hacer merge) a la rama master, primero hay que tener un commit hecho en la rama paralela. Una vez esté listo el commit podemos ingresar **git merge nombre-rama-paralela**, entonces el commit de esa rama se va a fusionar con el commit head de la rama master (o sea que no se fucionan las ramas como tal, sino que se fusionan los commits). Entonces al llmar a **git log –-oneline** nos va a mostrar algo como esto:

e37f0 (HEAD -> master, nombre-rama-paralela)

Una vez hecho esto vamos a ver que los cambios que se hicieron en la rama paralela ahora se fusionaron con lo que había en la rama master, sobrescribiendo o reemplazando lo que hayamos establecido.

Si quisieramos volver al estado de la rama master previo a la fusión (o sea desfusionar las ramas), podemos usar **git reset --hard** con el código del anteúltimo commit al final. O sea así: **git reset --hard codigo**. Y si volvemos a entrar a la rama paralela vamos a ver que seguimos teniendo los archivos como los dejamos antes de la fusión, y estos van a seguir commiteados ya que no se eliminan los commits, solo se defusionan.

Hay que aclarar que al poner **git log** en una rama paralela, vamos a ver no solo los commits de esa rama, sino también de su rama padre.

**Merge conflicts:**

Si al tener dos ramas llamadas A y B, donde A es la master y B la paralela, nosotros desarrollamos nuestras funcionalidades en la rama B, pero paralelamente seguimos desarrollando la rama A, vamos a tener un conflicto al intentar fusionarlas y git no va a saber con que cambios quedarse ya que en ambas ramas hay cambios.

Al hacer merge entre las dos ramas, git nos avisará que hay un conflicto y si tenemos nuestro editor de código abierto, este nos va a mostrar donde inicia y termina el conflicto, así como también marcar cuales son los cambios que tiene cada rama, y nos va a permitir elegir con cual de los dos cambios quedarnos o si queremos combinarlos (esto ya depende de lo que considere mejor el dev). Los cambios de la rama master se remarcaran con un color y los de la rama hija con otro. Luego de elegir los cambios que queremos conservar, commiteamos y listo, esto se puede hacer todo desde visual studio o visual studio code.

**.gitignore:**

Podemos crear un archivo llamado .gitignore que va a tener como función decirle a git que archivos queremos que se ignoren al momento de hacer un commit.

Dentro del archivo (que podemos editar como archivo de texto) podemos escribir los nombres de los archivos específicos que queramos ignorar, como por ejemplo “nombres.txt”. Si quisieramos ignorar cierto tipo de archivos podemos escribir **\*finalDelArchivo**, esto no se limita al formato del archivo, sino a la terminación de su nombre. O sea, si tengo algo así:

Archivo1.txt

Archivo1(copia1).txt

Archivo1(copia2).txt

Podriamos escribir **\*).txt,** de esta manera solo se ignorarían los archivos que terminan con parentesís y .txt, y el que termina con .txt no se ignoraría. Es otra forma de especificar, aunque no es una buena práctica.

Para que git NO ignore un archivo, o designorar un archivo, podemos escribir **!archivo**.

Para ignorar un directorio completo podemos escribir simplemente **carpeta/**.

También podemos ignorar un archivo específico dentro de un directorio así **carpeta/archivo**.

Es importante saber que si ya commiteamos un archivo y luego lo ignoramos en le .gitignore, este no se ignorará al hacer otro commit.

Tambien podemos crear un **.gitignore** y configurarlo para que sea **global**, de esta manera los archivos que estén en este, se ignorarán en todos los repositorios del usuario. Para configurar el archivo hay que ingresar **git config --global core.excludesfile directorioDelGitignore**. En caso de querer que este archivo no afecte a un repositorio en particular, porque va a tener su propio .gitignore, podemos crear un nuevo archivo en el directorio de dicho repositorio y automáticamente va a pisar al archivo global (ya que recordemos que las configuraciones **locales** tienen prioridad).

**Alias:**

Podemos pensar en un alias como una variable, solo que en vez de guardar datos, guardaremos comandos. Esto se hace para darle un nombre a ese comando y poder llamarlo de una forma más corta.

Ejemplo:

**git log --oneline --all --graph –-pretty=format:”%C(auto)%h%d %s %C(black)%C(bold)%c”**

Este comando es larguísimo, y sería muy tedioso tener que escribirlo una y otra vez ¿no? Bueno, para simplificar la escritura podemos hacer lo siguiente:

**Git config --global alias.log-mejorado “--oneline --all --graph –-pretty=format:’%C(auto)%h%d %s %C(black)%C(bold)%c’”**

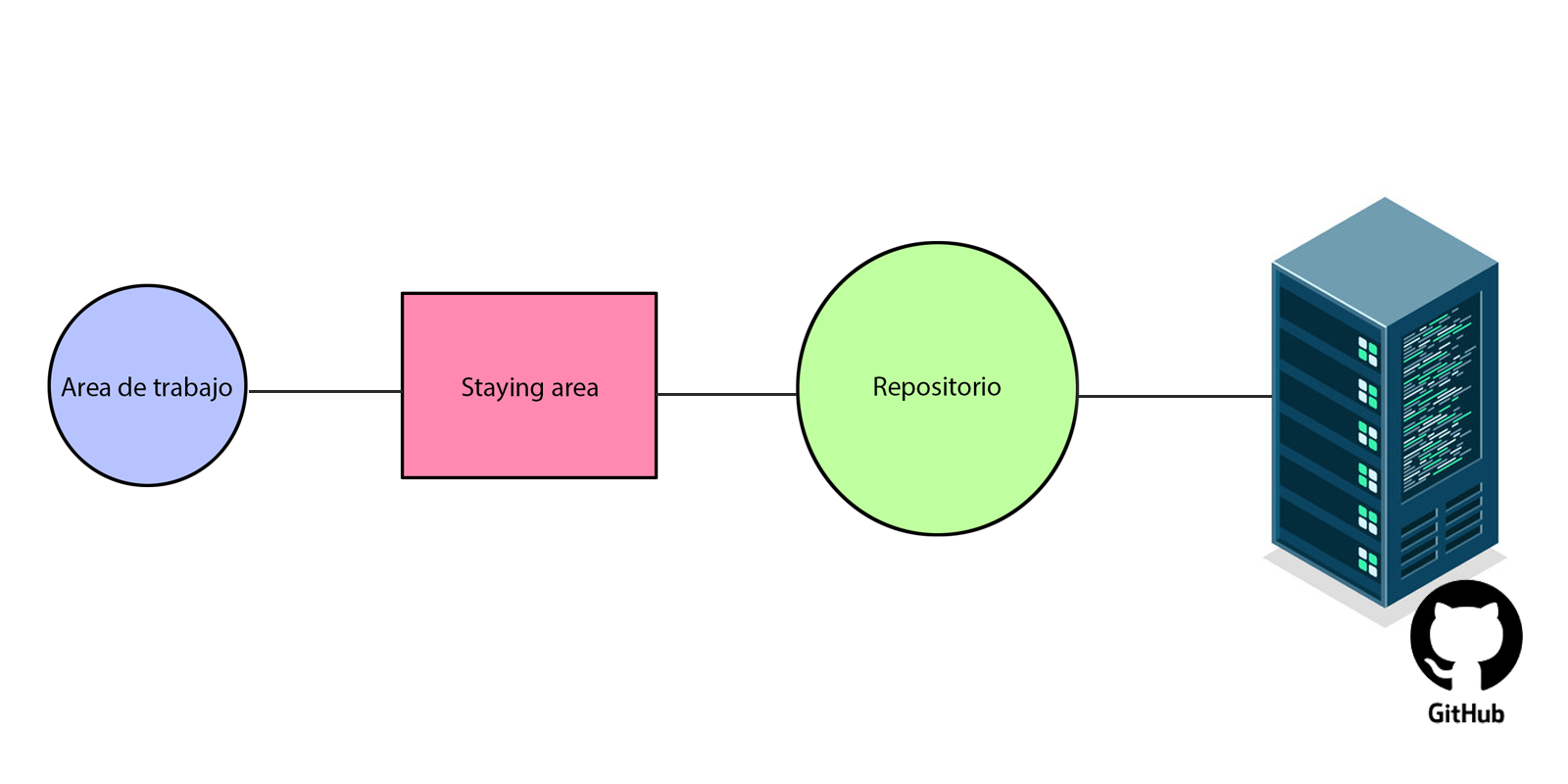
Entonces podremos llamarlo haciendo **git log-mejorado** y nos va a mostrar lo mismo que el comando largo. Esto puede hacerse con cualquier comando que queramos.

Algo importante a tener en cuenta es que es una buena practica escribir el alias con el tipo de comando (en este caso **log**), un guion, y la especificación que queramos darle.

**GitHub:**

Habiendo “terminardo” con la etapa de git, ahora entramos en la etapa de GitHub, que sería como el lugar final a la que van a ir a parar nuestros archivos y los de nuestros compañeros de trabajo.

Basicamente un repositorio remoto GitHub es un servidor central en la nube, en el que se van a concentrar todos cambios que realice cada integrante de un equipo (o persona individual), y al que todos van a estar sincronizados al mismo tiempo.



Para empezar a trabajar con GitHub primero debemos crearnos una cuenta, luego crear un repositorio, y copiar el enlace que se encuentra en el botón verde que dice “<> Code”. Este enlace es la URL de nuestro repositorio.

**Comandos de git:**

Comandos para moverse por las carpetas del sistema:

**cd carpetaElegida** = moverse a una carpeta

**cd ../** = retroceder una carpeta

**mkdir nombreRepositorio** = crear una carpeta

**rmkdir nombreRepositorio** = eliminar una carpeta

**git init** = inicializa la rama principal en el area de trabajo

**ls** = nos muestra los archivos que tenemos en el area de trabajo

**pwd** = nos muestra el directorio en la que nos encontramos

**ls -a** = nos muestra los archivos ocultos

**git add** = añade UN archivo del area de trabajo al staying area

**git add .** = añade TODOS los archivos del area de trabajo al staying area

**Git status** = nos muestra los archivos que están en el staying area

**git rm --cached nombreArchivo** = remueve el archvo del staying area para seguir editandolos

**git status -s o --short** = muestra el estado de los archivos de una manera más resumida y limpia

**git log --oneline** = muestra los commits de la rama resumidos

**git log --oneline --all** = muestra los commits de todas las ramas

**git log --oneline --all --graph** = muestra todos los commits con un gráfico de las ramas que tenemos

**git log --oneline --all --graph –-pretty=format:”%C(auto)%h%d %s %C(black)%C(bold)%c”** = muestra lo mismo que el anterior pero añadiendo hace cuanto tiempo se hizo cada commit