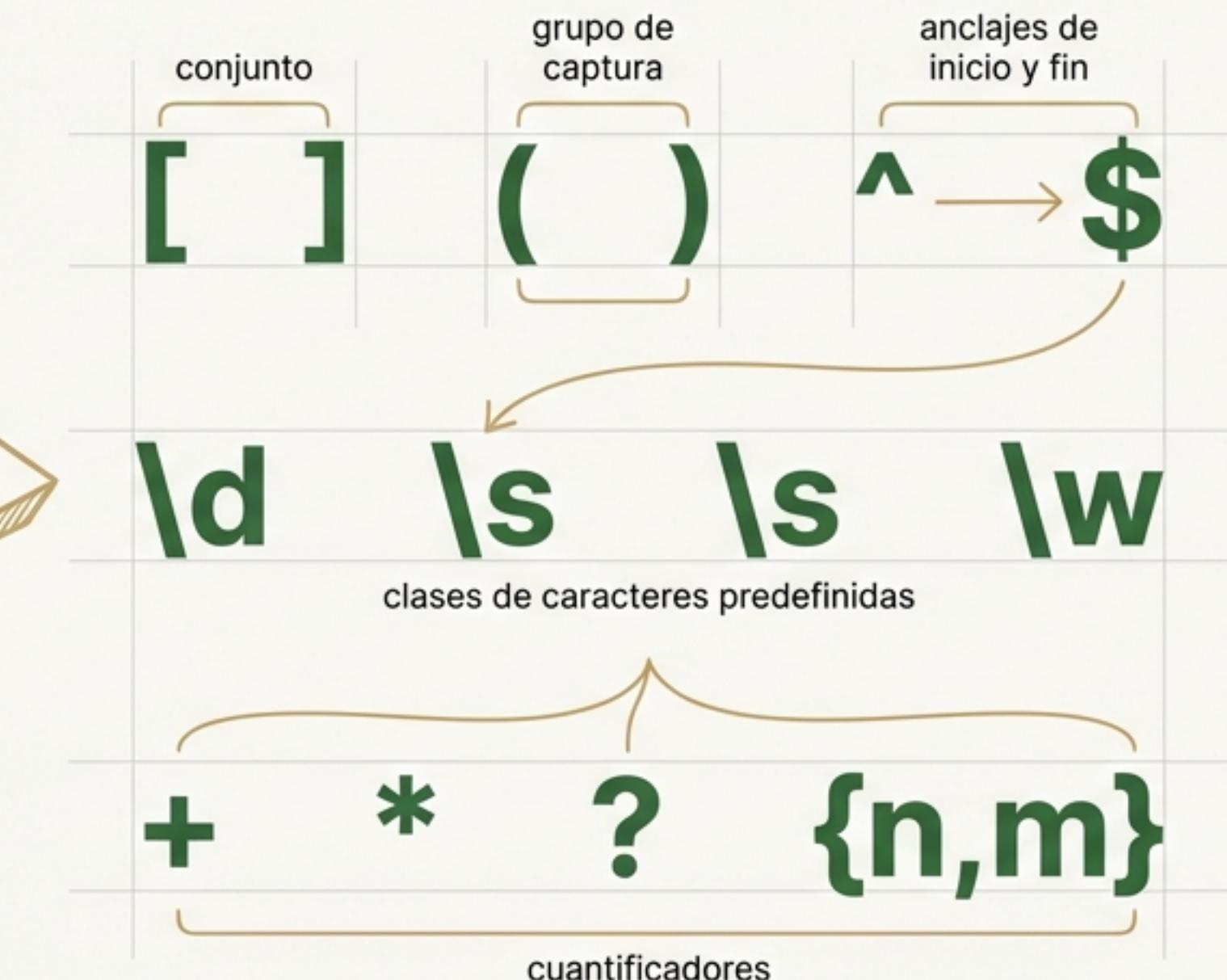


# De la Confusión a la Claridad

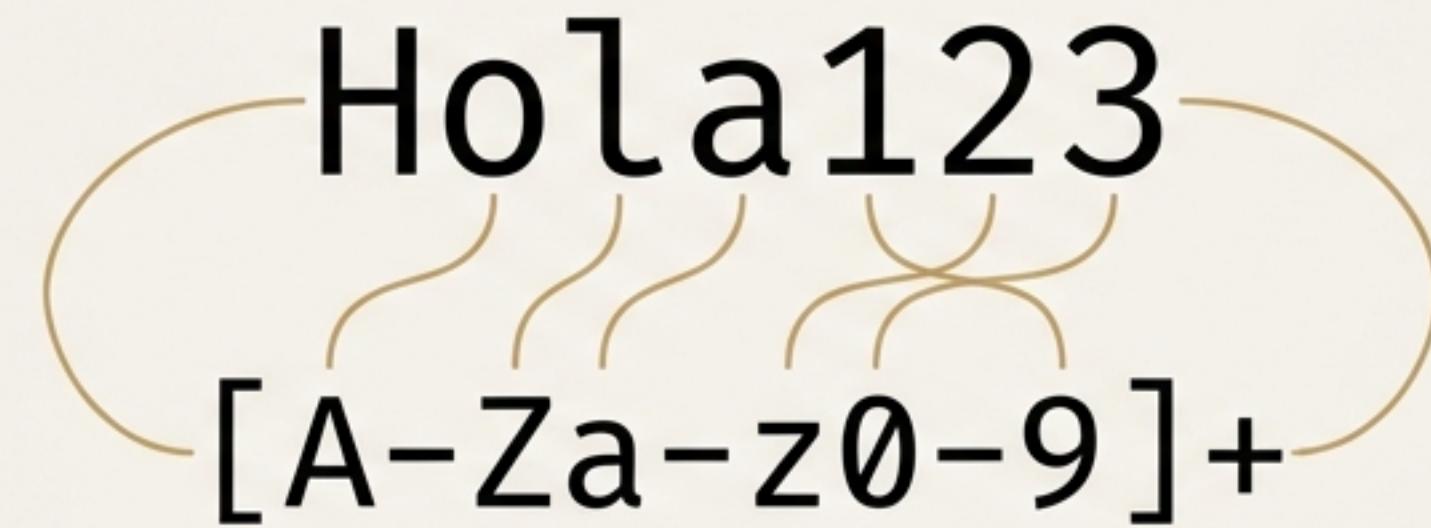
[ ] + \* \w  
\d \s \d ^\$  
] ^\$ \w ()  
[ ] { } .  
[ ] ? | \b



Apuntes Avanzados de Expresiones Regulares y Git

# ¿Qué es una Expresión Regular (Regex)?

Una expresión regular es una plantilla para buscar patrones en un texto. Imagina que es como un 'array' de caracteres donde cada índice es una letra o símbolo. El sistema itera sobre el texto (como un bucle `for`) para ver si coincide con el patrón que has definido.



- **Finalidad:** Validar formatos (IBAN, email, códigos hexadecimales).
- **Lógica:** En lugar de `if (letra == 'a' || letra == 'b' || ...) usas un rango [a-z].
- **Potencia:** Permite definir reglas complejas de validación en una sola línea.

# Caso de Estudio 1: El Validador de Dos Palabras con Caracteres Especiales

## El Problema

Se necesita validar una entrada de dos palabras que pueden contener caracteres especiales como 'ñ' o tildes.

*"Es que pone dos palabras... pero no funciona."*

## Puntos de Confusión

- El signo `+` está fuera de los corchetes, entre las dos definiciones de caracteres. ¿Qué se supone que hace ahí?
- No hay una definición explícita del espacio entre las palabras.

## El Código Inicial (Incorrecto)

```
// Entrada del usuario: "año mamut"
String patron = "[a-zA-ZñáéíóúÁÉÍÓÚ]+
[a-zA-ZñáéíóúÁÉÍÓÚ]+[a-zA-ZñáéíóúÁÉÍÓÚ]";  
  
if (entrada.matches(patron)) {
    System.out.println("Válido");
} else {
    System.out.println("No Válido"); //  
Resultado: "No Válido"
}
```

# Análisis y Depuración: La Extraña Sintaxis `//w` y la Negación Oculta

## Paso 1: Identificar el Verdadero Patrón del Profesor

Se revela que el código real visto en clase era diferente y más extraño:  
`//w[...]\*///w[...]`.

Comose conra la exista del tutor:  
*"Esto de aquí... yo no lo he visto nunca."*

## Paso 2: El Descubrimiento Clave

El código funcionaba "al revés". Entradas como "Antonio" (que no están en el set) daban `Válido`, y "año" (que sí está) daba `No Válido`.

La sintaxis `//w` está actuando como una negación. Significa "**que NO contenga**" los caracteres del conjunto.

## Paso 3: La Solución mediante la Lógica Inversa

Si el patrón significa "que no contenga" y queremos que valide cuando "sí contenga", necesitamos negar el resultado.

entrada.matches  
s("//w[...]"")  
-> true  
(si NO  
contiene 'ñ')



!entrada.matches  
es("//w[...]"")  
-> true  
(si SÍ  
contiene 'ñ')

```
// El patrón original funciona a la inversa.  
String patronInverso = "//w[a-zA-Z...ñ...ú...]+//w[a-zA-Z...ñ...ú...]+";  
  
// Para corregirlo, negamos el resultado de la condición.  
if (!entrada.matches(patronInverso)) {  
    System.out.println("Válido"); // Ahora "año mamut" es Válido  
} else {  
    System.out.println("No Válido"); // Y "Antonio Antonio" es No Válido  
}
```

# Solución y Principio: Entender la Negación y el Contexto

## La Solución Refinada

- Aunque negar con `!` funciona, la sintaxis `//w` es poco estándar en Java. Un enfoque más claro y universal es definir explícitamente lo que SÍ quieras que coincida.

### Código Canónico (La Forma Correcta)

```
// Principio: Define lo que buscas, no lo que no buscas.  
// El espacio se representa con \s  
String patronClaro = "^[a-zA-ZñáéíóúÁÉÍÓÚ]+\s[a-zA-ZñáéíóúÁÉÍÓÚ]+$";  
if (entrada.matches(patronClaro)) {  
    System.out.println("Válido");  
}
```

Indica el inicio de la cadena. Indica el final de la cadena. Define el set de caracteres permitidos. Cuantificador 'una o más veces'. Coincide con un carácter de espacio.

## El Principio Clave

Las expresiones regulares deben ser explícitas. Evita sintaxis ambiguas o no estándar (`//w`). Si un patrón funciona 'al revés', es probable que esté usando una lógica de negación que debe ser entendida o reemplazada por una afirmación directa.

# Caso de Estudio 2: El Validador de Nombre de Usuario

## El Problema

Contexto: Validar un nombre de usuario que debe contener letras y números, con una longitud mínima de 2 y máxima de 9 caracteres.

## El Código Inicial (Redundante)

```
String patron = "[a-zA-Z//d]{2,9}";  
// ...código de validación...
```



"Aquí hay ambigüedad... Si ya tienes `[a-zA-Z]`, y `//d` incluye números, ¿por qué repetir? Esto es redundancia."

## Pruebas y Resultados

Entrada	Resultado
`pepino`	Válido (6 caracteres)
`pi`	No Válido (error lógico inicial)
`pepino123`	Válido (9 caracteres)
`pepino1234`	No Válido (10 caracteres)

- El problema principal no es de funcionalidad, sino de claridad y eficiencia.

# Solución y Principio: La Claridad Vence a la Ambigüedad

## El Proceso de Simplificación

[a-zA-Z]

(Define letras)

+

\d

(Atajo para '[0-9]')

=

[a-zA-Z0-9]

(Combinación clara y sin redundancia)

## Código Corregido y Optimizado

```
// Se elimina la ambigüedad combinando los sets de caracteres.  
String patron = "[a-zA-Z0-9]{2,9}";
```

```
// Pruebas con el código corregido:
```

```
// "pi" -> Válido
```

```
// "pepino" -> Válido
```

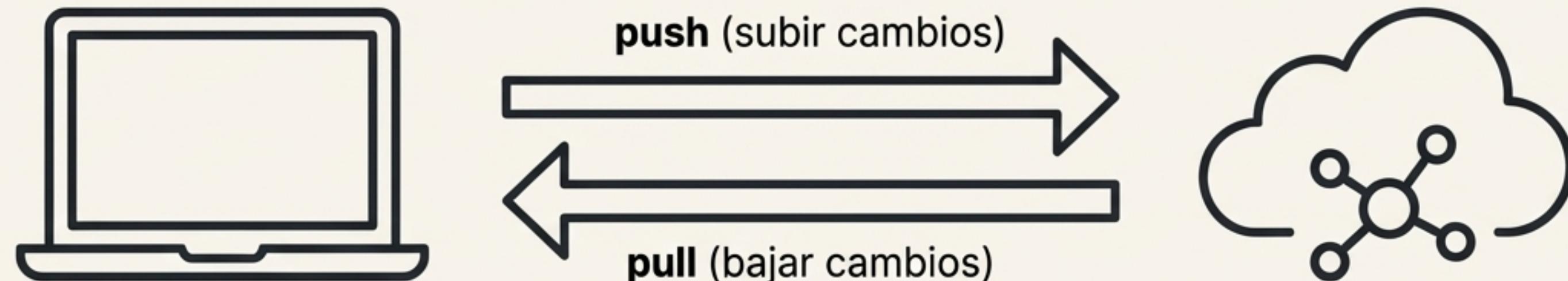
```
// "pepino1234" -> No Válido
```

## El Principio Clave

En programación, la ambigüedad y la redundancia son enemigas de la mantenibilidad. Busca siempre la expresión más simple y directa que cumpla el objetivo. Si dos partes de tu código hacen lo mismo, probablemente una de ellas sobre.

# Guía Práctica de Supervivencia para Git

Git no es solo algo que "debes saber que existe", es una herramienta que **debes usar**. Es un software de control de versiones. Su propósito es guardar un historial de todos los cambios en tu proyecto, permitiéndote colaborar y recuperarte de errores.



## Repositorio Local

Tu máquina. Donde trabajas y haces cambios.

## Repositorio Remoto (GitHub)

Un servidor centralizado. Donde se sincroniza el trabajo de todos y se guarda la versión "oficial".

# El Flujo de Trabajo Esencial (Parte 1): Sincronización

## Paso 0: Preparar el Terreno (Solo la primera vez)

```
git init
```

Convierte una carpeta normal en un repositorio de Git (local). Marca el punto de partida.

## Paso 1: Revisar Cambios Remotos

```
git fetch
```

Pregunta al repositorio remoto: '¿Hay algo nuevo?'. No descarga los cambios, solo actualiza tu "mapa" de lo que ha sucedido. Es un paso seguro para ver el estado del proyecto.

## Paso 2: Traer los Cambios a tu Local

```
git pull origin main
```

Descarga y fusiona todos los cambios desde el repositorio remoto (`origin`) y la rama principal (`main`) a tu repositorio local.

### Principio

Antes de empezar a trabajar, siempre haz `pull`. Asegúrate de tener la última versión para evitar conflictos.

# El Flujo de Trabajo Esencial (Parte 2): Guardar y Compartir tus Cambios

## Paso 3: Preparar tus Archivos para el "Guardado"

```
git add .
```

Añade todos los archivos modificados al "área de preparación" (Staging Area). Es como seleccionar qué fotos quieres incluir en un álbum antes de pegarlas. El `.` significa "todos los archivos en esta carpeta y subcarpetas".

## Paso 4: Confirmar tus Cambios en Local

```
git commit -m "Descripción clara y concisa del cambio"
```

Crea un punto de guardado permanente en tu historial local. El mensaje es crucial para que tú (y otros) sepáis qué se hizo.

*"No es lo mismo commit que push. Commit es un cambio en local."*

## Paso 5: Subir tus Cambios al Remoto

```
git push origin main
```

Envía todos tus commits locales al repositorio remoto para que otros puedan verlos y descargarlos.

# La Máquina del Tiempo: Cómo Revertir Errores

## Concepto: El Hash del Commit

- Cada `commit` tiene un identificador único llamado "hash".

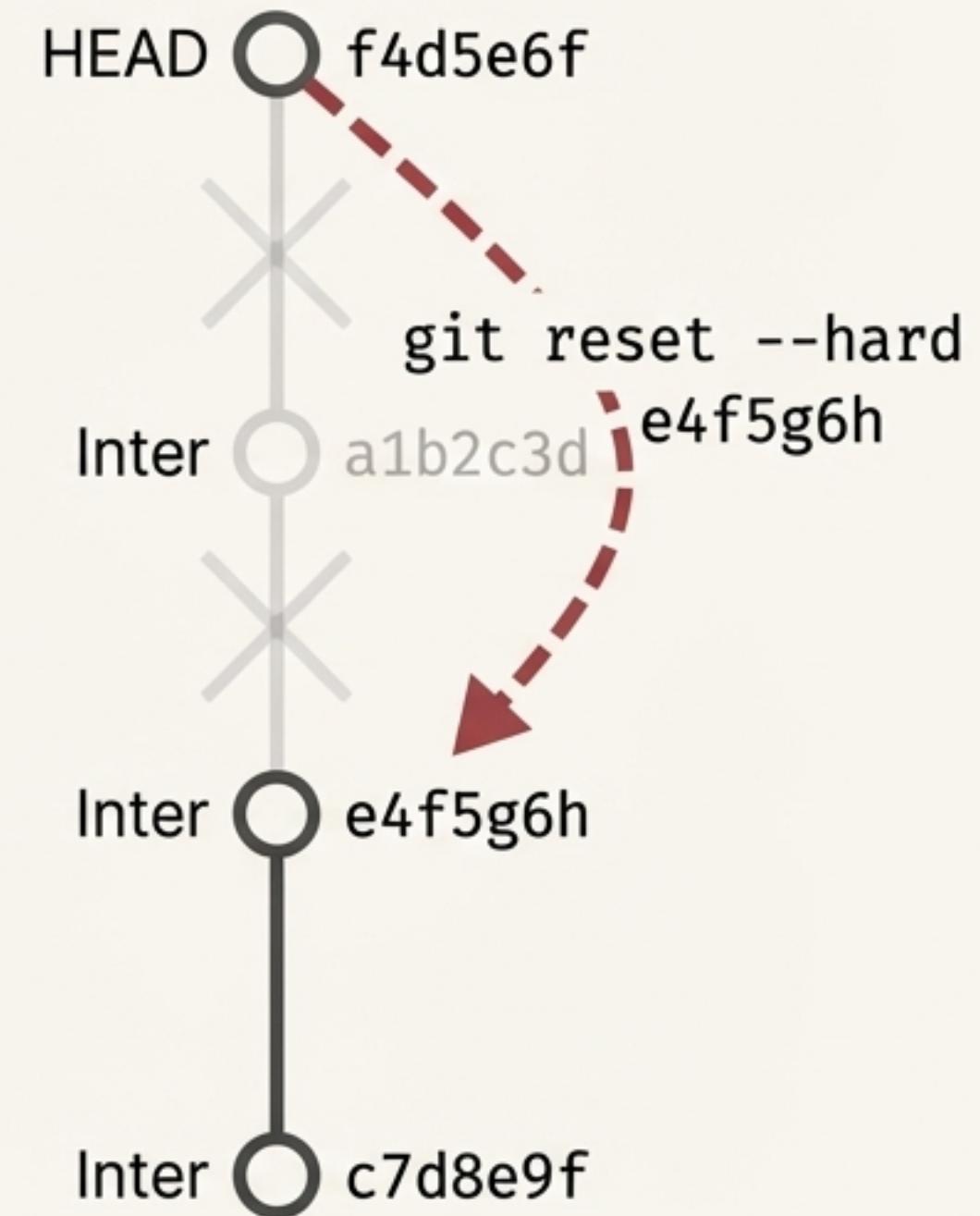
*“Es como un picadillo. Puedes resumir El Quijote entero en una línea de 256 bits. La probabilidad de que dos hashes diferentes coincidan es de 1 entre  $2^{256}$ , un número mayor que los átomos del universo observable.”*

## Comandos para Viajar en el Tiempo

- 1. Ver el Historial: `git log`
- 2. Volver a un Punto Anterior: `git reset --hard <hash\_del\_commit>`



Atención: esto elimina todos los cambios posteriores en tu local y no se pueden recuperar fácilmente. Úsalo con mucho cuidado.



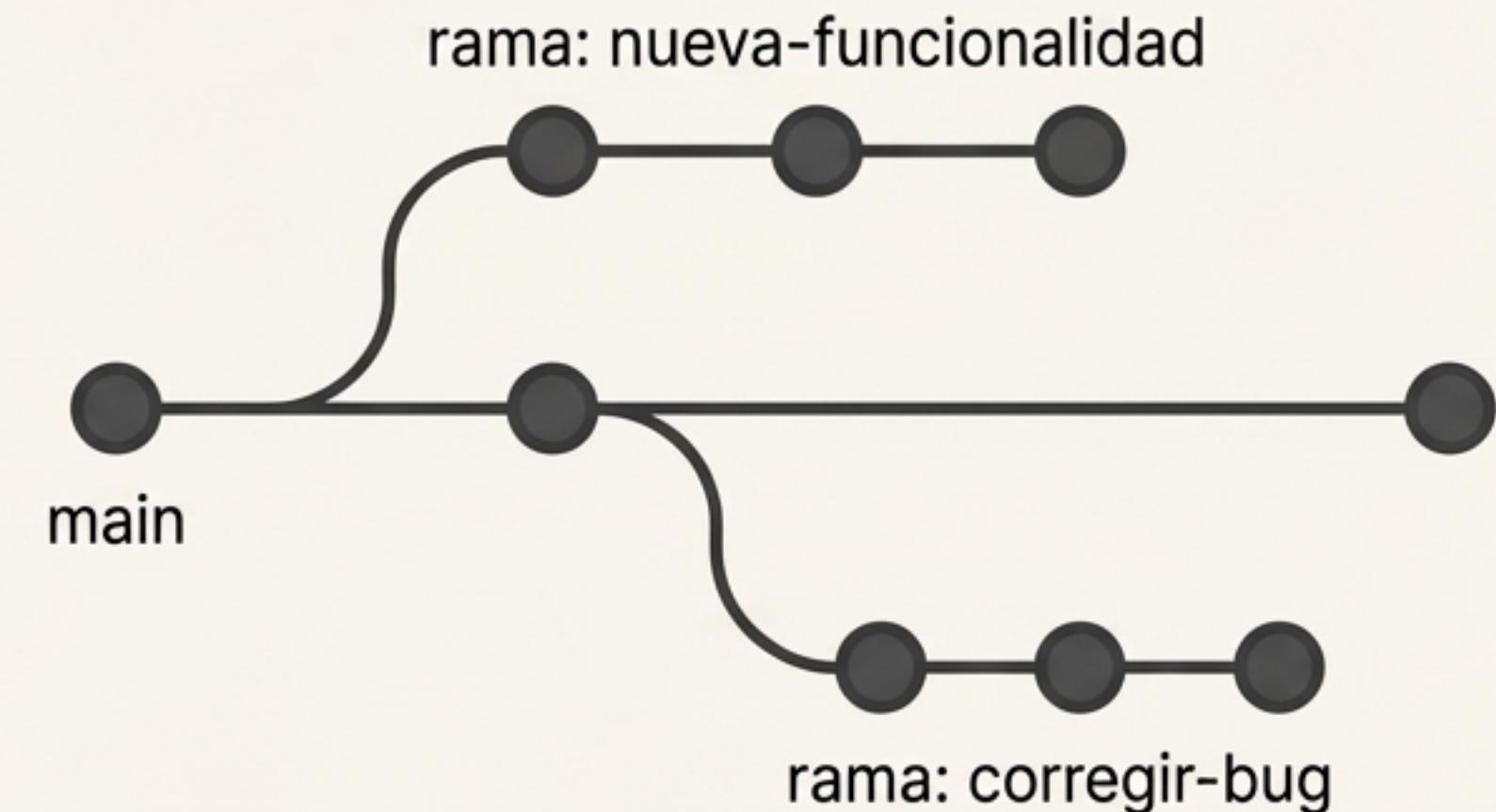
# Trabajo en Paralelo: Introducción a las Ramas (Branches)

## ¿Qué es una Rama?

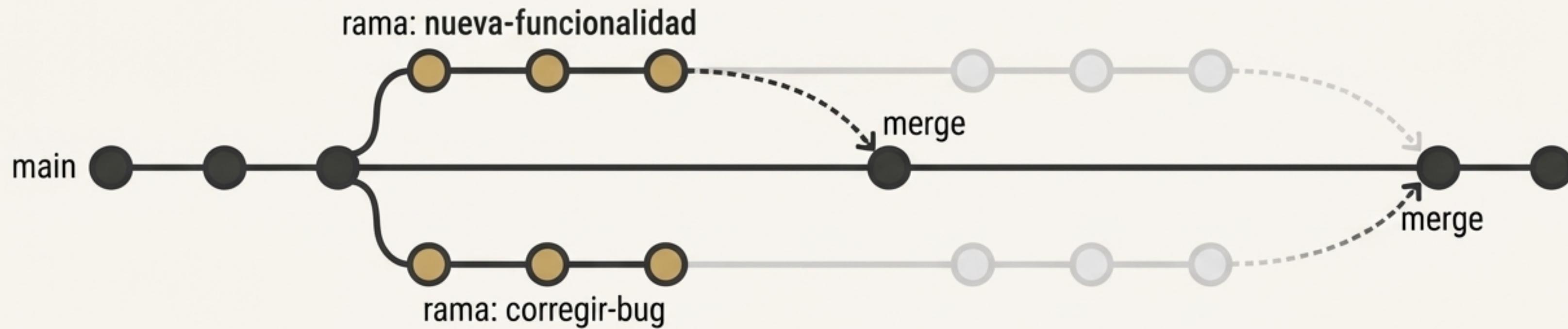
Una rama es una línea de desarrollo independiente. Te permite trabajar en una nueva funcionalidad o corregir un error sin afectar la rama principal (`main`), que siempre debe estar estable.

## ¿Por qué Usarlas?

- **Seguridad:** Experimenta sin miedo a romper el código principal.
- **Organización:** Cada tarea o funcionalidad puede tener su propia rama.
- **Colaboración:** Múltiples personas pueden trabajar en diferentes ramas simultáneamente sin interferir entre sí.



# El Ciclo de Vida de una Rama: Crear, Trabajar y Fusionar



## Comandos Fundamentales

**Crear una Rama:** `git branch <nombre-rama>`

**Moverse a la Rama:** `git checkout <nombre-rama>`

**Atajo (Crear y Moverse):** `git checkout -b <nombre-rama>`

## El Proceso de Fusión (Merge)

1. ...Haces tus cambios y commits en tu rama...
2. **Volver a la Rama Principal:** `git checkout main`
3. **Traer los Cambios:** `git merge <nombre-rama>`
  - Acción: "Integra todos los commits de tu rama en `main`."

## Limpieza

**Eliminar la Rama Local (una vez fusionada):**

`git branch -d <nombre-rama>`

# Mi Flujo de Trabajo a Prueba de Errores

Esta es mi manera, basada en la experiencia, para trabajar de forma segura y evitar conflictos, especialmente en equipo. Todo se prepara en local antes de subir nada al remoto.

## La Secuencia Maestra

### 1 Sincronizar

```
git fetch y luego git pull origin main
```

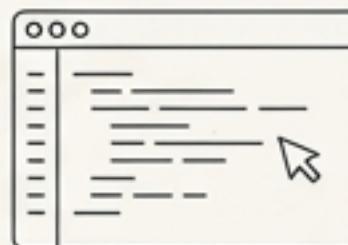
(Siempre empezar con la última versión)

### 2 Crear Rama Local

```
git checkout -b mi-nueva-tarea
```

(Aislar tu trabajo)

### 3 Trabajar



...modificar código, añadir archivos...

### 4 Confirmar en Local

```
git add . git commit -m "Descripción"
```

(Puedes hacer varios commits)

### 5 Preparar Fusión

```
git checkout main
```

(Volver a la base)

### 6 Actualizar la Principal de Nuevo

```
git pull origin main
```

(Asegurarse de tener lo último antes de fusionar)

### 7 Fusionar

```
git merge mi-nueva-tarea
```

(Integrar tu trabajo en la versión principal local)

### 8 Subir al Remoto

```
git push origin main
```

(Compartir el resultado final ya integrado)

### 9 Limpiar

```
git branch -d mi-nueva-tarea
```

(Mantener el repositorio ordenado)