

Guía de Estudio: Java, Git y SQL

Conceptos Clave y Código Práctico de Nuestra Sesión



Tu Ruta de Aprendizaje



Módulo 1: Dominando Java Avanzado

Exploraremos enumeraciones, herencia y el uso correcto de la palabra clave 'this'.



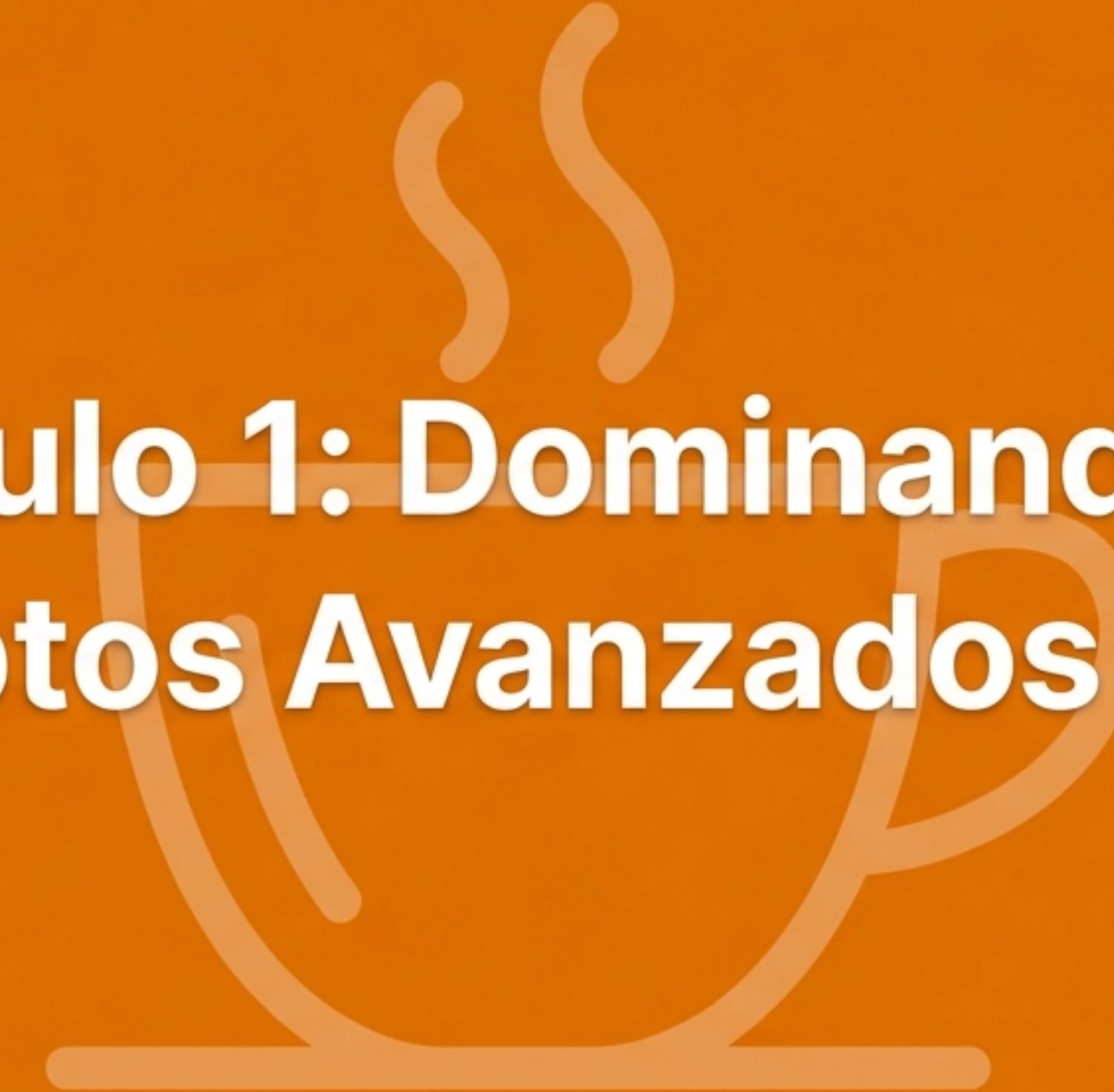
Módulo 2: Tu Flujo de Trabajo con Git

Una guía paso a paso para versionar y colaborar en tus proyectos con Git y GitHub.



Módulo 3: Recetario de Consultas SQL

Aprenderás a filtrar, agrupar y ordenar datos como un profesional.



Módulo 1: Dominando los Conceptos Avanzados de Java

Enumeraciones: Definiendo Constantes con Seguridad

Concepto

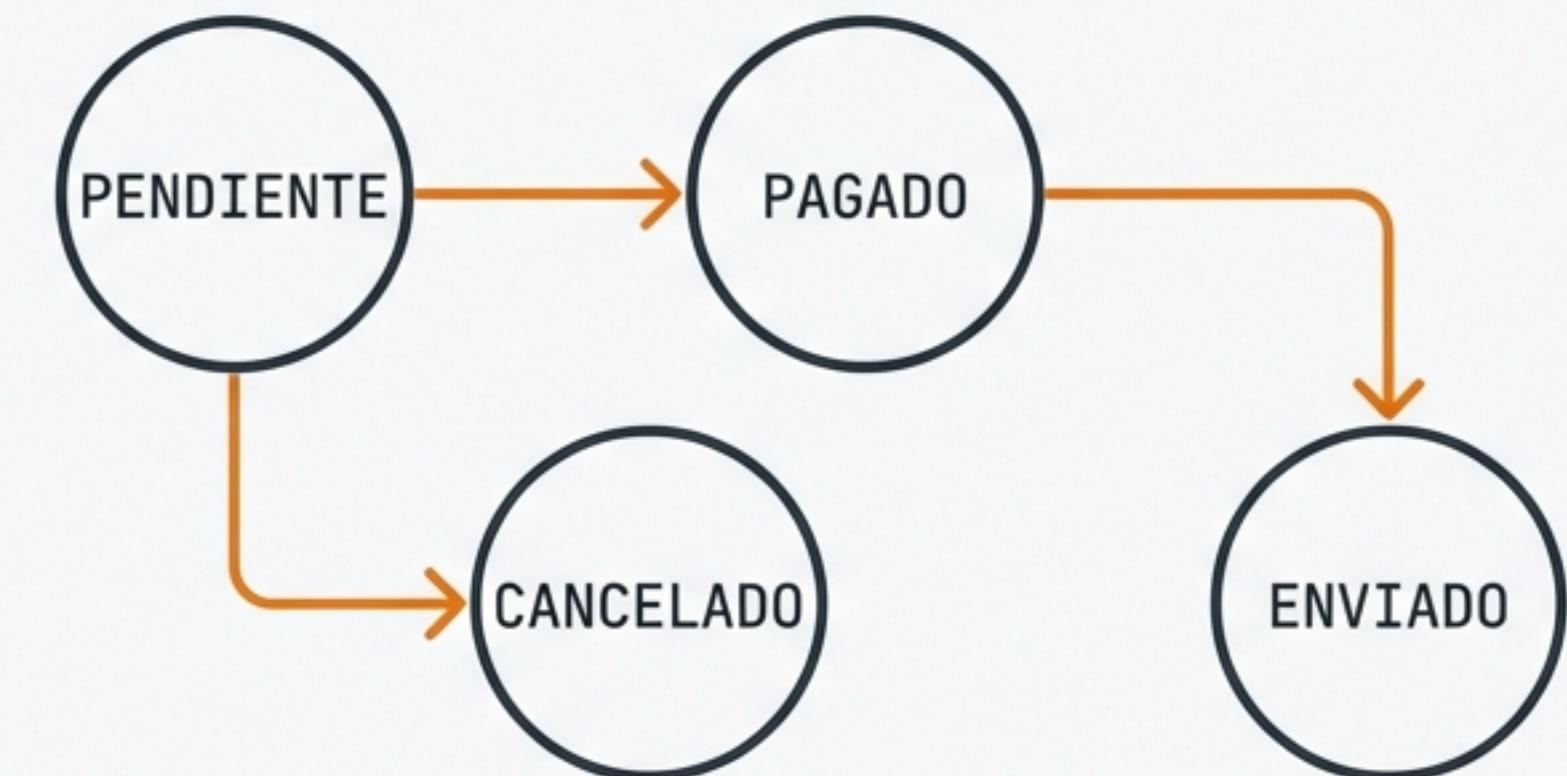
¿Qué son?

Una enumeración es un tipo de dato personalizado e inamovible que representa un conjunto fijo de constantes.

¿Por qué usarlas?

Para mejorar la legibilidad del código y evitar errores con valores inválidos. En lugar de usar números como `1` para 'Pendiente', usas un término claro y explícito.

Contexto de Uso



Perfectas para gestionar estados finitos, como los de un pedido: `PENDIENTE`, `PAGADO`, `ENVIADO`, `CANCELADO`.

Enumeraciones en Acción: El Estado de un Pedido

1. La Declaración del Enum

```
// Declaras tu propio tipo de dato
public enum EstadoPedido {
    PENDIENTE,
    PAGADO,
    ENVIADO,
    CANCELADO
}
```

Un nuevo tipo, tan válido como `String` o `int`.

2. Su Uso en la Clase 'Pedido'

```
public class Pedido {
    private EstadoPedido estado; // Usas el enum como tipo de atributo

    public Pedido() {
        // El estado inicial es inamovible
        this.estado = EstadoPedido.PENDIENTE;
    }

    // Cambias el estado con un setter, pasando otro valor del enum
    public void setEstado(EstadoPedido nuevoEstado) {
        // Aquí iría la lógica para permitir solo cambios de estado válidos
        // (p. ej. de PENDIENTE a PAGADO, pero no de CANCELADO a ENVIADO)
        this.estado = nuevoEstado;
    }
}
```

El atributo `estado` solo puede ser uno de los cuatro valores definidos.

Se asigna un valor del enum de forma segura y legible.

La Palabra Clave `this`: Resolviendo la Ambigüedad

El único propósito de `this` es diferenciar entre un atributo de la clase y un parámetro de un método o constructor cuando ambos tienen el mismo nombre. Evita la **ambigüedad**.

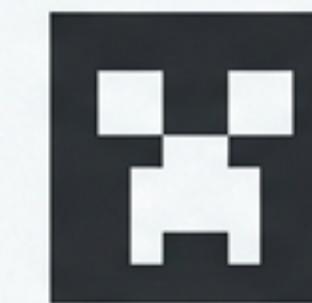
Uso Necesario (Hay Ambigüedad)

```
public class Asamblea {  
    private String nombre;  
  
    public Asamblea(String nombre) {  
        // 'this.nombre' es el atributo de la clase.  
        // 'nombre' es el parámetro del constructor.  
        this.nombre = nombre;  
    }  
}
```

Uso Innecesario (No Hay Ambigüedad)

```
public class Asamblea {  
    private int numeroDeAula;  
  
    public Asamblea() {  
        // No hay parámetro, no hay ambigüedad.  
        // 'this' aquí es opcional y redundante.  
        this.numeroDeAula = 0;  
    }  
}
```

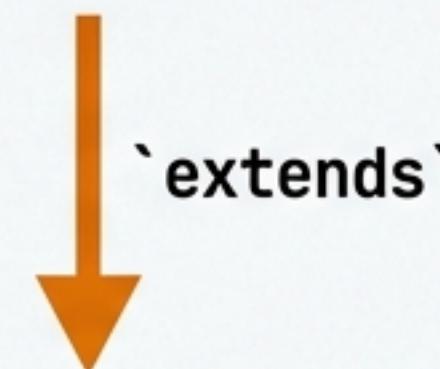
Herencia: La Analogía de Minecraft



Clase Padre: `Monstruo`

Atributos: `id`, `fuerza`

Métodos: `hacerSonido()`



Clase Hija: `Zombie`

Hereda: `id`, `fuerza`, `hacerSonido()`

+ **Añade Métodos Propios:** `quemarseConElSol()`

🔗 **Modifica Métodos Heredados:** `hacerSonido()` (con un sonido diferente)

Una clase hija (`Zombie`) hereda todas las características de su clase padre (`Monstruo`), pero también puede tener sus propias características únicas y modificar las heredadas.

Herencia en Código: `extends` y `@Override`

Clase Padre (`Monstruo.java`)

```
public class Monstruo {  
    protected int id;  
    protected int fuerza;  
  
    public void hacerSonido() {  
        System.out.println("Sonido de monstruo genérico");  
    }  
}
```

Clase Hija (`Zombie.java`)

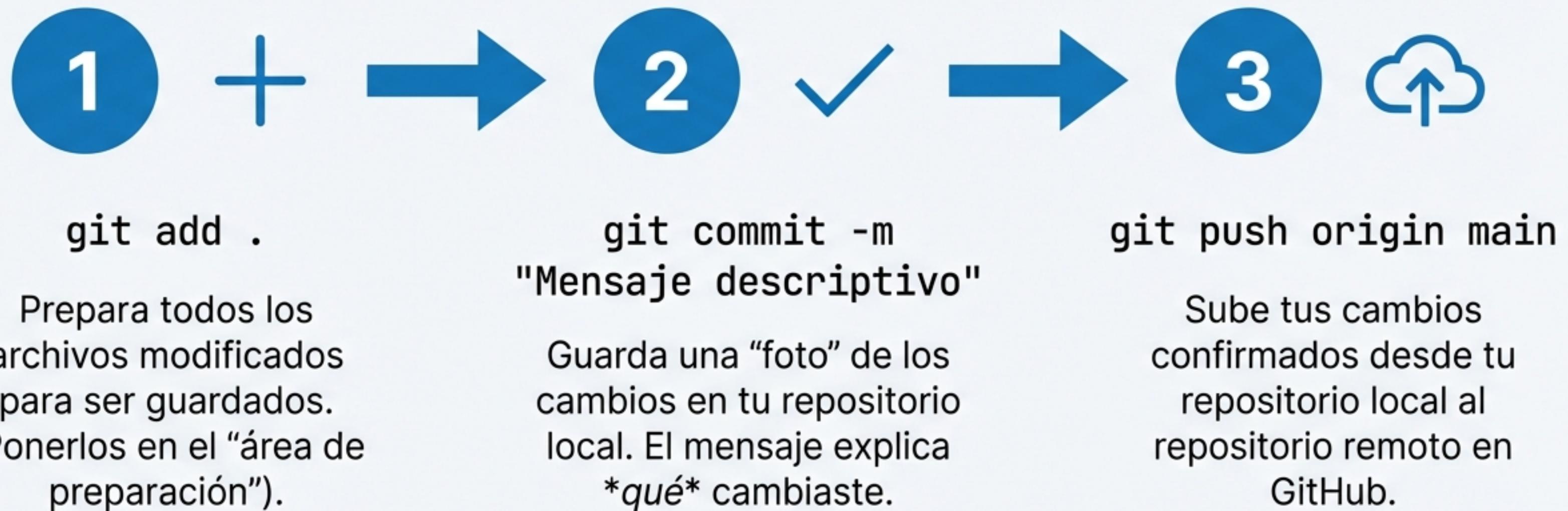
```
// Zombie 'extiende' de Monstruo, heredando sus miembros  
public class Zombie extends Monstruo {  
  
    // Método propio de Zombie  
    public void quemarseConElSol() {  
        System.out.println("El zombie se quema con el sol.");  
    }  
  
    // '@Override' indica que estamos modificando un método heredado  
    @Override  
    public void hacerSonido() {  
        System.out.println("Arrrggghhh..."); // Sonido específico  
    }  
}
```



Módulo 2: Tu Flujo de Trabajo Esencial con Git y GitHub

Flujo de Trabajo con Git y GitHub

El Ciclo Básico de Git: Añadir, Confirmar, Subir



Guía Práctica: Comandos Útiles para el Día a Día

Para Empezar (Solo una vez por proyecto)

- `git init`: Inicia un repositorio local.
- `git remote add origin [URL]`: Conecta tu repositorio local con el remoto en GitHub.

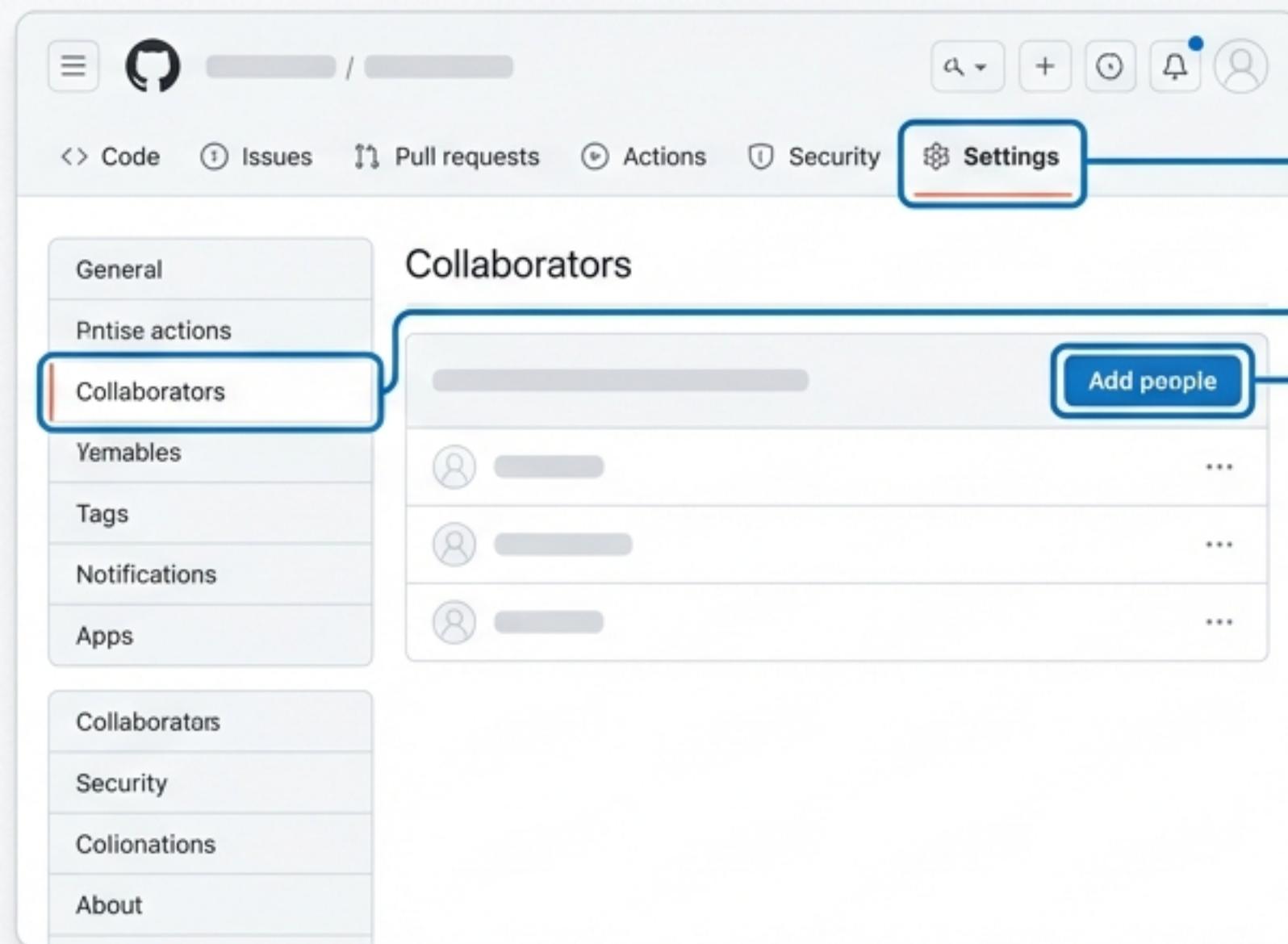
Para Verificar el Estado

- `git status`: Muestra qué archivos han sido modificados, cuáles están preparados y cuáles no.
- `git log --oneline`: Muestra el historial de commits de forma compacta, en una sola línea por commit.

! Nota Práctica (¡Ojo!)

Git puede dar problemas al inicializar un repositorio en un disco duro externo. Es recomendable trabajar siempre en una carpeta local de tu ordenador (como el Escritorio) y hacer copias de seguridad al disco externo.

Colaborando en GitHub: Invitando a tu Equipo



1. En tu repositorio en GitHub, ve a la pestaña **Settings**.
2. En el menú lateral, selecciona **Collaborators**.
3. Haz clic en el botón **Add people**.
4. **Introduce el nombre de usuario** de GitHub de la persona que quieras invitar.

La persona recibirá una invitación por correo electrónico que deberá aceptar. Mientras tanto, su estado aparecerá como 'Pending Invite'.



Módulo 3: Recetario de Consultas SQL

El Arte de Filtrar: `WHERE`, `IN`, `BETWEEN` y `LIKE`

Filtrar por Condición Numérica (`WHERE`)

-- Selecciona empleados con un salario mayor a 50000
`SELECT * FROM empleados WHERE salario > 50000;`

Filtrar por una Lista de Valores (`IN`)

-- Selecciona empleados de los departamentos de Ventas o Marketing
`SELECT * FROM empleados WHERE departamento IN ('Ventas', 'Marketing');`

Filtrar por Rango (`BETWEEN`)

-- Selecciona empleados con salarios entre 30000 y 60000
`SELECT * FROM empleados WHERE salario BETWEEN 30000 AND 60000;`

Filtrar por Patrón de Texto (`LIKE`)

-- Selecciona clientes cuyo nombre empieza con 'A'
`SELECT nombre FROM clientes WHERE nombre LIKE 'A%';`

Agrupando Datos: `GROUP BY` y `HAVING`

Usa `GROUP BY` para agrupar filas que tienen los mismos valores en columnas específicas en un registro resumen. Usa `HAVING` para filtrar estos grupos después de que las funciones de agregación (como `AVG` o `SUM`) se hayan calculado.

Regla de Oro



WHERE filtra filas individuales **antes** de la agrupación.



HAVING filtra grupos **después** de la agrupación.

Código de Ejemplo

```
-- Muestra los departamentos cuyo salario medio es superior a 40000
```

```
SELECT
```

```
    departamento,
```

```
    AVG(salario) AS salario_promedio
```

```
FROM
```

```
    empleados
```

```
GROUP BY
```

```
    departamento
```

```
HAVING
```

```
    AVG(salario) > 40000; -- Filtra los resultados del grupo
```