**Parte Teórica**

Investigar qué es markdown

GIT:

1. ¿Qué es un repositorio en Git y cómo se diferencia de un proyecto “normal”?

2. ¿Cuáles son las tres áreas principales de Git (working directory, staging area/index y repository) y qué papel cumple cada una?

3. ¿Cómo representa Git los cambios internamente? (objetos blob, tree, commit y tag).

4. ¿Cómo se crea un commit y qué información almacena un objeto commit?

5. ¿Cuál es la diferencia entre git pull y git fetch?

6. ¿Qué es un branch (rama) en Git y cómo Git gestiona los punteros a commits?

7. ¿Cómo se realiza un merge y qué conflictos pueden surgir? ¿Cómo se resuelven?

8. ¿Cómo funciona el área de staging (git add) y qué pasa si omito este paso?

9. ¿Qué es el archivo .gitignore y cómo influye en el seguimiento de archivos?

10. ¿Cuál es la diferencia entre un “commit amend” (--amend) y un nuevo commit?

11. ¿Cómo se utiliza git stash y en qué escenarios es útil?

12. ¿Qué mecanismos ofrece Git para deshacer cambios (por ejemplo, git reset, git revert, git checkout)?

13. ¿Cómo funciona la configuración de remotos (origin, upstream) y qué comandos uso para gestión de forks?

14. ¿Cómo puedo inspeccionar el historial de commits (por ejemplo, git log, git diff, git show)?

**Programación:**

15. ¿Cuáles son los tipos de datos primitivos en Java?

16. ¿Cómo funcionan las estructuras de control de flujo como if, else, switch y bucles en Java?

17. ¿Por qué es importante usar nombres significativos para variables y métodos?

18. ¿Qué es la Programación Orientada a Objetos (POO)?

19. ¿Cuáles son los cuatro pilares de la Programación Orientada a Objetos?

20. ¿Qué es la herencia en POO y cómo se utiliza en Java?

21. ¿Qué son los modificadores de acceso y cuáles son los más comunes en Java?

22. ¿Qué es una variable de entorno y por qué son importantes para Java o la programación en general?

**1. ¿Qué es un repositorio en Git y cómo se diferencia de un proyecto “normal”?**

Un repositorio en Git es un lugar donde puedes almacenar el código, los archivos y el historial de revisiones de cada archivo. Los repositorios pueden contar con múltiples colaboradores, pueden ser públicos como privados y permite volver a versiones anteriores del código.

Además es una carpeta de proyecto que contiene un subdirectorio oculto llamado .git donde se guarda todo el historial, configuración y metadatos del control de versiones.  
La diferencia entre con un **Proyecto “normal”**, es que este solo es una carpeta con archivos; no tiene control de versiones ni historial.

**2. Áreas principales de Git**

* **Working directory (Directorio de trabajo)**: es la copia local de los archivos del proyecto en la máquina. Aquí es donde se realizan modificaciones, crear nuevos archivos o eliminar los que ya no son necesarios. Estos cambios son detectados por Git, pero aún no forman parte del historial oficial hasta que son confirmados.
* **Staging área / Index**: es una zona intermedia que almacena temporalmente los cambios que se desean incluir en el próximo commit. Al usar git add, se añade archivos o modificaciones específicas a esta área. Esto permite seleccionar de manera precisa qué cambios serán confirmados, facilitando la organización y claridad en el historial de versiones.
* **Repository (Repositorio local)**: es la base de datos interna donde Git guarda todos los commits y el historial completo del proyecto. Cuando se ejecuta git commit, los cambios almacenados en la staging area se registran de forma permanente en el repositorio local. Cada commit incluye un snapshot de los archivos en ese momento, junto con información sobre el autor y un mensaje descriptivo.

**3. ¿Cómo representa Git los cambios internamente? (objetos blob, tree, commit y tag).**

**Blob (Binary Large Object):**  Guarda el contenido exacto de un archivo, pero sin información sobre su nombre ni ubicación. Cada versión distinta de un archivo genera un blob distinto, incluso si el cambio es mínimo.

**Tree** : Representa el contenido de un directorio. Contiene referencias a otros trees (subdirectorios) y a blobs (archivos), junto con los nombres y permisos. Es como un “índice” que dice qué archivos existen y dónde.

**Commit** : Es un objeto que guarda un snapshot (foto) del proyecto en un momento específico. Apunta a un objeto tree (el estado de la carpeta raíz), al commit anterior (o commits padres), y almacena metadatos como autor, fecha y mensaje.

**Tag**: Es una etiqueta que apunta a un commit específico. Puede ser ligera (solo referencia) o anotada (incluye información adicional como nombre del creador, fecha y mensaje).

**4. Creación de un commit y contenido de un objeto commit**

* Flujo:
  + git add para pasar cambios al **staging**.
  + git commit -m "mensaje" para confirmar.
* Un commit almacena:
  + Puntero al **tree** con el estado de los archivos.
  + Referencia(s) a commit(s) padre(s).
  + Autor y fecha.
  + Mensaje descriptivo.

**5. Diferencia entre git pull y git fetch**

* **git fetch**: Descarga cambios del remoto, pero **no** los fusiona con tu rama actual. Solo actualiza referencias.
* **git pull**: Hace fetch + merge (o rebase si se configura). Trae y aplica los cambios al instante.

**6. Branch (rama) y gestión de punteros**

* Un **branch** es simplemente un puntero que apunta a un commit específico.
* Git gestiona las ramas moviendo ese puntero al nuevo commit cada vez que haces uno.
* HEAD es el puntero que indica en qué rama/commit estás trabajando.

**7. Merge y conflictos**

## **Cómo se realiza un merge**

1. Ubicarse en la rama de destino (la que recibirá los cambios) usando: git checkout main
2. Iniciar el proceso de merge indicando la rama origen con: git merge nombre\_rama
3. Git intentará combinar automáticamente los cambios de ambas ramas.
   * Si no hay conflictos, el merge finaliza y se crea un commit de merge.
   * Si hay conflictos, será necesario resolverlos manualmente.

### **2. Tipos de conflictos que pueden surgir**

* **Conflictos de contenido** : Cuando las mismas líneas de un archivo fueron modificadas en ambas ramas de forma diferente.
* **Conflictos de eliminación** : Una rama elimina un archivo mientras la otra lo modifica.
* **Conflictos de renombrado** : El mismo archivo es renombrado de forma diferente en cada rama.

### **3. Resolución de conflictos**

1. Git marcará los archivos con indicadores para mostrar las diferencias:  
   <<<<<<< HEAD

contenido en la rama actual

=======

contenido en la rama que se une

>>>>>>> nombre\_rama

1. Revisar cada archivo en conflicto y decidir qué cambios conservar, modificar o combinar.
2. Marcar cada archivo como resuelto usando: git add nombre\_archivo
3. Finalizar el merge para registrar la combinación de cambios con: git commit

**8. Área de staging (git add) y omitir el paso**

**Función del área de staging**

* El área de staging (o staging area) es una zona intermedia donde se guardan los cambios que se incluirán en el próximo commit.
* Permite seleccionar de forma precisa qué modificaciones de archivos formarán parte de ese commit.
* Se actualiza con el comando git add nombre\_archivo o git add . para añadir todos los cambios.

**Qué sucede cuando se usa git add**

* Git toma una “fotografía” del contenido actual del archivo y la guarda en el área de staging.
* El archivo queda marcado para ser incluido en el siguiente commit.
* Si el archivo se modifica nuevamente después de ser añadido, esos cambios no estarán en el staging hasta volver a usar git add.

**Qué pasa si se omite este paso**

* Los cambios permanecerán únicamente en el directorio de trabajo y no se incluirán en el próximo commit.
* El commit se generará únicamente con los cambios que estén en el staging, por lo que podrían quedar modificaciones fuera del historial.
* Es comparable a redactar una carta pero no colocarla en el sobre antes de enviarla.

**9. .gitignore**

Es un archivo de texto que contiene una lista de patrones que indican a Git qué archivos o carpetas deben ser ignorados.

Funciones:

* Permite excluir archivos que no deben formar parte del repositorio, como: Archivos temporales o de caché, Archivos generados automáticamente por el sistema o el entorno de desarrollo, Configuraciones locales y credenciales, Archivo que lista patrones de archivos/carpetas que Git debe ignorar y Evita que se rastreen archivos temporales, credenciales, builds, etc.

**10. commit --amend vs nuevo commit**

* **Nuevo commit**: Añade un registro más al historial.
* **--amend**: Reemplaza el último commit (mensaje y/o contenido), modificando el historial. Útil para corregir errores recientes.

**11. git stash**

* Guarda temporalmente cambios no confirmados (y limpia el working directory).
* Escenarios útiles: Cuando necesitas cambiar de rama rápidamente pero no quieres hacer commit de cambios incompletos.
* Se recupera con git stash pop o git stash apply.

**12. Mecanismos para deshacer cambios**

* **git reset**: Mueve punteros y opcionalmente borra cambios del staging o del working directory.
* **git revert**: Crea un commit nuevo que revierte uno anterior, preservando historial.
* **git checkout** (o git restore en versiones nuevas): Restaura un archivo a una versión específica.

**13. Configuración de remotos y forks**

* **origin**: Nombre por defecto del remoto principal de tu repositorio.  
  **upstream**: Nombre usado comúnmente para el remoto del repositorio original del cual hiciste un fork.
* Comandos:
  + git remote add origin <url>
  + git remote add upstream <url>
  + git fetch upstream para traer cambios del original.

**14. Inspección de historial**

* git log: Lista commits (puede filtrarse por autor, fecha, etc.).
* git diff: Muestra cambios entre commits o entre el working directory y el staging/repo.
* git show <commit>: Muestra detalles y cambios de un commit específico.

**15. Tipos de datos primitivos en Java** Java tiene 8 tipos primitivos:

* **byte** (8 bits, enteros pequeños)
* **short** (16 bits, enteros medianos)
* **int** (32 bits, enteros)
* **long** (64 bits, enteros grandes)
* **float** (32 bits, decimales de precisión simple)
* **double** (64 bits, decimales de doble precisión)
* **char** (16 bits, un carácter Unicode)
* **boolean** (true / false)

**16. Estructuras de control de flujo**

* **if / else**: Ejecutan bloques de código según condiciones booleanas.
* **switch**: Evalúa una expresión y ejecuta el caso (case) que coincida, con opción de default.
* **Bucles**:
  + for: Repite un bloque un número definido de veces.
  + while: Repite mientras la condición sea verdadera.
  + do-while: Similar a while pero ejecuta al menos una vez antes de evaluar la condición.
  + for-each: Recorre colecciones o arreglos.

**17. Importancia de nombres significativos**

* Mejoran la **legibilidad** y **mantenibilidad** del código.
* Ayudan a otros desarrolladores (y a ti mismo en el futuro) a entender la lógica sin tener que leer cada detalle de la implementación.
* Evitan confusiones y errores al dar contexto sobre lo que hace la variable o método.

**18. Programación Orientada a Objetos (POO)** Paradigma de programación que organiza el software en **objetos**, los cuales combinan datos (atributos) y comportamientos (métodos). Se basa en modelar elementos del mundo real o abstractos.

**19. Cuatro pilares de la POO**

1. **Encapsulamiento** → Ocultar detalles internos y exponer solo lo necesario.
2. **Abstracción** → Representar ideas esenciales sin incluir detalles innecesarios.  
   **Herencia** → Reutilizar código mediante clases que heredan propiedades y métodos de otras.
3. **Polimorfismo** → Capacidad de que diferentes clases respondan de forma distinta al mismo método.

**20. Herencia en POO y en Java**

* Permite crear una clase (subclase) a partir de otra (superclase), heredando atributos y métodos.

En Java se usa la palabra clave **extends**:  
class Animal {

void hacerSonido() { System.out.println("Sonido genérico"); }

}

class Perro extends Animal {

void hacerSonido() { System.out.println("Ladrar"); }

}

* Se utiliza para **reutilizar código** y **especializar comportamientos**.

**21. Modificadores de acceso más comunes en Java**

* **public**: Accesible desde cualquier clase.
* **protected** : Accesible desde la misma clase, subclases y mismo paquete.
* **default** (sin palabra clave) : Accesible solo dentro del mismo paquete.
* **private**: Accesible solo dentro de la misma clase.

**22. Variable de entorno**

* Es un valor almacenado en el sistema operativo que los programas pueden leer.
* Se usan para configurar comportamientos (por ejemplo, JAVA\_HOME para indicar dónde está instalado Java).
* Son importantes porque permiten **configuración sin modificar el código**, facilitan la **portabilidad** y ayudan a manejar credenciales, rutas y configuraciones sensibles.