¿P6. Qué es la Programación Orientada a Objetos?

La **Programación Orientada a Objetos (POO)** es un estilo de programación que organiza el software en **clases** (plantillas) y **objetos** (instancias). Su objetivo es estructurar los programas en piezas simples, reutilizables y fáciles de mantener.
A diferencia de los lenguajes más antiguos (como COBOL, Basic o C) que se centraban en la lógica, la POO —usada en lenguajes modernos como **Java, C# o Python**— integra datos y comportamientos en un mismo modelo.
En este enfoque, los programadores diseñan **clases** que agrupan información y funciones, y a partir de ellas crean **objetos** que interactúan entre sí, construyendo programas completos y más organizados.

La imagen que aparece en la presentación muestra **las principales características de Java**.
Características de Java

1. **Sencillo**
   - Fácil de entender.
   - Reduce la complejidad en comparación con otros lenguajes.
2. **Orientado a objetos**
   - Facilita la creación y mantenimiento de programas.
   - Basado en clases y objetos.
3. **Independiente de la arquitectura y portable**
   - El código en Java se compila a **Bytecode**, que puede ejecutarse en cualquier computadora con un intérprete de Java.
   - Permite portabilidad entre sistemas.
4. **Robusto**
   - Simplifica la gestión de la memoria.
   - Reduce errores relacionados con la administración de recursos.
5. **Multitarea**
   - Puede ejecutar varias líneas de código al mismo tiempo (multithreading).
6. **Dinámico**
   - No es necesario cargar todo el programa en memoria.
   - Las clases compiladas pueden cargarse bajo demanda durante la ejecución.

👉 En pocas palabras: **Java es un lenguaje sencillo, orientado a objetos, portable, robusto, multitarea y dinámico**, lo que lo hace muy usado en el desarrollo de aplicaciones modernas.

P7. Claves de la Programación Orientada a Objetos

La **Programación Orientada a Objetos (POO)** se basa en representar el mundo real mediante **objetos**.
En el caso de una **biblioteca**, los objetos serían:

- **Libro** → con datos como título, autor, año, y acciones como prestar o devolver.
- **Usuario** → con datos como nombre e historial, y acciones como pedir un libro.
- **Bibliotecario** → con funciones como registrar préstamos o devoluciones.

Cada objeto tiene **propiedades** (datos) y **métodos** (acciones).
La idea principal es que los objetos **interactúan entre sí**.
Ejemplo: un **Usuario** pide un libro, y el **Libro** cambia su estado a "prestado".

<En la imagen muestra un **ejemplo de clase en Java** con su explicación.

📌 **Explicación del código**

1. **Definición de la clase (línea 12):**
2. public class Producto {

Aquí se crea la **clase** llamada Producto.

- public → significa que puede ser usada desde cualquier parte del programa.
- class → palabra clave para definir una clase.
- Producto → es el nombre de la clase.

3. **Atributos (líneas 14–16):**
4. private int codigo;
5. private String nombre;
6. private double precio;

Estos son las **propiedades** del objeto Producto.

- codigo → número identificador (entero).
- nombre → texto que guarda el nombre del producto.

- precio → valor en dinero (decimal).

El modificador private significa que solo se pueden usar **dentro de la clase**, no directamente desde afuera.

7. **Métodos (desde la línea 18):**

    - **obtenerPrecio()**

    - public void obtenerPrecio(){
    - System.out.println("El precio es " + precio);
    - }

Muestra en pantalla el precio del producto.

    - **calcularTotal()**

    - public void calcularTotal(){
    - double total = precio * 5000;
    - }

Calcula un valor total multiplicando el precio por 5000 (ejemplo de operación con el atributo).

👉 En resumen:

- **Clase** = Molde o plantilla (Producto).
- **Atributos** = Datos que describen al objeto (codigo, nombre, precio).
- **Métodos** = Acciones que puede realizar (obtenerPrecio, calcularTotal).

P8. Clases, objetos e instancias

El primer y más importante concepto de la POO es la **distinción entre clase y objeto**. Una **clase** es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo. Por ejemplo, una clase para representar a animales puede llamarse 'animal' y tener una serie de **atributos**, como 'nombre' o 'edad' (que

normalmente son propiedades), y una serie con los comportamientos que estos pueden tener, como caminar o comer, y que a su vez se implementan como métodos de la clase (funciones).

Un **ejemplo sencillo de un objeto, Relación de la imagen con la POO**

1. **Clase (Animal):**
   Es el **molde o plantilla**.
   - Aquí se define qué características y comportamientos pueden tener todos los animales.
2. **Objetos (Perro y Gato):**
   Son **instancias de la clase**.
   - Ejemplo: un Perro y un Gato son objetos creados a partir de la clase Animal.
3. **Atributos (Pancho, 4 años / Paco, 3 años):**
   Son los **datos propios de cada objeto**.
   - Pancho (Perro) tiene nombre y edad.
   - Paco (Gato) también tiene su propio nombre y edad.
4. **Métodos (Comer, Caminar):**
   Son las **acciones o comportamientos** que los objetos pueden realizar.
   - El perro puede **comer**.
   - El gato puede **caminar**.

👉 En resumen:

- **Clase = Animal**
- **Objetos = Perro y Gato**
- **Atributos = nombre y edad de cada uno**
- **Métodos = acciones que realizan (comer, caminar, etc.)**

En la tabla se ven **Tipos de datos en Java**

🔹 **Números enteros**

- **byte** → 8 bits → valores pequeños (-128 a 127).
- **short** → 16 bits → enteros más grandes (-32,768 a 32,767).
- **int** → 32 bits → enteros comunes (el más usado).
- **long** → 64 bits → enteros muy grandes.

🔹 **Números reales (con decimales)**

- **float** → 32 bits → número decimal simple (menos preciso).
- **double** → 64 bits → número decimal doble (más preciso, el más usado).

🔹 **Cadenas de texto**

- **char** → 16 bits → almacena **un solo carácter** (‘a’, ‘b’, ‘1’).
- **String** → variable especial para **texto completo** (“Hola Mundo”).

🔹 **Otros**

- **boolean** → almacena solo **true o false** (verdadero o falso).

👉 En resumen:

- **Enteros** = byte, short, int, long.
- **Decimales** = float, double.
- **Texto** = char, String.
- **Lógico** = boolean.

P9. Principios de la Programación Orientada a Objetos

📌 **Encapsulación en POO**

La **encapsulación** significa que un objeto guarda toda su información **dentro de sí mismo** y solo permite mostrar al exterior lo que es necesario.

- **Atributos privados:** los datos internos se protegen y no se pueden modificar directamente desde fuera.
- **Métodos públicos:** se crean funciones para controlar cómo acceder o modificar esos datos de forma segura.

🚗 **Ejemplo con un coche**

- **Información pública (interfaz pública):** luces de freno o direccionales → muestran solo lo necesario a otros conductores.
- **Información privada (oculta):** motor, carburante o temperatura interna → están escondidos porque no todos necesitan saberlos.

👉 En resumen:

La encapsulación **protege los datos** y **controla el acceso** a ellos, mostrando solo lo esencial al exterior, tal como un coche muestra luces para comunicar, pero esconde su motor.

📌 **Abstracción en POO**

La **abstracción** consiste en mostrar al usuario **solo lo esencial** de un objeto y **ocultar la complejidad interna**.

- Permite trabajar con una **versión simplificada** del objeto (atributos y métodos importantes).
- El usuario no necesita saber cómo funciona internamente, solo cómo usarlo.
- Es como una evolución de la encapsulación, porque además de proteger los datos, **simplifica la interacción**.

🚗 **Ejemplo con un coche**

- Para conducir solo necesitas **usar el volante, pedales y palancas** (interfaz simple).
- No es necesario conocer cómo funciona el motor, la transmisión o el sistema eléctrico (detalles ocultos).

👉 En resumen:
La **abstracción** permite que el usuario trabaje con lo importante y entienda lo necesario, sin complicarse con los detalles internos.

📌 **Herencia en POO**
La **herencia** permite crear una **relación jerárquica entre clases**, donde una clase "padre" (o **superclase**) comparte sus **atributos y métodos** con otras clases "hijas" (o **subclases**).

- La **superclase** define lo común (ejemplo: todo Animal tiene nombre y edad).
- Las **subclases** heredan esas características, pero pueden **añadir o modificar** sus propios comportamientos (ejemplo: un Ave puede poner huevos, un Pez puede nadar).

👉 En resumen:
La herencia evita repetir código, porque permite **reutilizar lo común** en varias clases y **especializar lo particular** en cada subclase.

📌 **Polimorfismo en POO**
El **polimorfismo** significa que un mismo método o acción puede **comportarse de distintas formas** según el objeto que lo use.

- **Misma interfaz, diferentes comportamientos.**
- Se logra gracias a la **herencia**, donde las subclases pueden **modificar (anular)** o **adaptar** los métodos de la superclase.
- Existen dos formas principales:

1. **Sobrecarga de métodos:** mismo nombre de método, pero con distintos parámetros.
2. **Anulación de métodos:** una subclase redefine el comportamiento de un método heredado.

🐾 **Ejemplo con animales**

- Método hacerSonido():
- Un **Perro** → "ladrar()".
- Un **Gato** → "maullar()".
- Un **Pájaro** → "piar()".

Todos usan la misma interfaz (hacerSonido()), pero el resultado cambia según el objeto.

👉 En resumen:
El polimorfismo permite **usar el mismo método con diferentes resultados**, lo que hace el código más flexible y reutilizable.

Atributos y métodos de la clase pueden ser:

La imagen muestra los modificadores de acceso en Java, que definen quién puede ver y usar los atributos o métodos de una clase. Te lo resumo de forma clara:

📌 **Modificadores de acceso en Java**

1. **public (público):**
   - **Los atributos o métodos se pueden usar desde cualquier parte del programa.**
   - **Ejemplo: un método public void encender() lo puede llamar cualquier clase.**
2. **private (privado):**
   - **Solo se pueden usar dentro de la misma clase.**
   - **Es el más usado para proteger datos internos.**
   - **Ejemplo: un atributo private String contraseña; no puede ser accedido directamente desde fuera.**
3. **protected (protegido):**
   - **Se pueden usar dentro del mismo paquete y también en clases hijas (subclases) aunque estén en otro paquete.**
   - **Útil cuando se trabaja con herencia.**
   - **Ejemplo: un método protected void calcular() lo pueden usar las subclases.**

👉 **En resumen:**

- **public → acceso libre desde cualquier lugar.**
- **private → acceso solo dentro de la clase.**
- **protected → acceso dentro del paquete y por herencia.**

P10. Implementación de POO en distintos lenguajes

La **Programación Orientada a Objetos (POO)** se implementa de forma similar en varios lenguajes, aunque cada uno tiene su propia **sintaxis y particularidades**:

Java

- Usa class para definir clases.
- Implementa herencia con extends.
- Utiliza **interfaces**, polimorfismo y encapsulación para organizar y proteger el código.

C++

- Usa class para definir clases.
- Herencia con niveles de acceso (public, private, protected).
- Permite **herencia múltiple** y polimorfismo mediante **funciones virtuales**.
- Controla el acceso con modificadores de visibilidad.

Python

- Usa class para definir clases.
- Herencia indicando la clase base entre paréntesis.
- Admite **herencia múltiple** y polimorfismo de forma natural.
- Encapsulación mediante **propiedades** y **decoradores**.

C#

- Usa class para definir clases.
- Herencia con public y protected.
- Usa **interfaces** y polimorfismo (virtual y override).
- Encapsulación mediante modificadores de acceso (public, private, protected).

👉 En conclusión: aunque la POO se basa en los mismos principios (**clases, herencia, polimorfismo y encapsulación**), cada lenguaje ofrece **formas específicas** para implementarlos.

P11. En la tabla se muestra varios entornos de desarrollo Java (IDE).

Entornos de Desarrollo Java (IDE)

1. **NetBeans**

    - **IDE oficial muy utilizado para proyectos Java.**
    - **Ofrece soporte para múltiples lenguajes.**
    - **Descarga: NetBeans**
2. **Eclipse**
    - **Uno de los IDE más populares en programación Java.**
    - **Muy extensible gracias a sus plugins.**
    - **Descarga: Eclipse**
3. **BlueJ (JBlue en la tabla)**
    - **IDE diseñado principalmente para principiantes en Java.**
    - **Útil en entornos educativos por su simplicidad.**
    - **Descarga: BlueJ**

👉 **En conclusión: NetBeans y Eclipse son ideales para desarrolladores profesionales por sus múltiples herramientas, mientras que BlueJ es perfecto para el aprendizaje inicial de Java.**

P12. Sistemas de control de versiones

📌 **¿Qué es un sistema de control de versiones?**

Es una herramienta que permite **gestionar, registrar y organizar los cambios realizados en archivos o proyectos** (como software, documentos o bases de datos). Se usa mucho en desarrollo de software, pero también en cualquier trabajo colaborativo.

✅ **Beneficios principales que aparecen en la imagen:**

1. **Regresar a versiones anteriores**
   Permite recuperar documentos o código en un estado previo, evitando perder trabajo.

2. **Obtener información de los autores**
   Se puede saber quién realizó cada cambio, lo cual da trazabilidad y control.

3. **Identificar errores fácilmente**
   Al llevar un historial, es más simple detectar dónde y cuándo se introdujo un problema.

4. **Comparar cambios entre versiones**
   Es posible ver diferencias entre dos versiones de un archivo para entender qué se modificó.

5. **Recuperación de archivos defectuosos o eliminados**
   Si se borra o daña un archivo, se puede restaurar desde el historial del sistema.

👉 En conclusión:
Los sistemas de control de versiones **organizan el trabajo en equipo, protegen contra la pérdida de información y facilitan la detección y corrección de errores**, siendo esenciales en el desarrollo de software y proyectos colaborativos.

P13. ◈ ¿Qué es Git?

Git es un **sistema de control de versiones distribuido**.
Esto significa que es una herramienta que permite **guardar, controlar y administrar los cambios realizados en el código o en cualquier conjunto de archivos** a lo largo del tiempo.

Fue creado en el 2005 por **Linus Torvalds** (el mismo creador de Linux) para gestionar el enorme proyecto del kernel de Linux.

◈ ¿Para qué sirve Git?

Git se utiliza principalmente en el desarrollo de software, pero también puede usarse en cualquier proyecto con archivos que cambien con el tiempo.

✅ Sus principales usos son:

1. **Controlar versiones de archivos** 🗂️
   Puedes volver atrás en el tiempo y recuperar versiones anteriores de un proyecto.

2. **Trabajar en equipo** 👥
   Varios desarrolladores pueden trabajar en el mismo proyecto sin sobrescribir el trabajo de los demás.

3. **Ramas (branches)**
   Permite crear diferentes líneas de desarrollo para probar ideas nuevas o añadir funcionalidades, sin afectar el proyecto principal.

4. **Fusionar cambios (merge)**
   Une los trabajos realizados en distintas ramas o por diferentes personas.

5. **Historial completo**
   Cada cambio queda registrado con información de quién lo hizo, cuándo y por qué.

6. **Distribución**
   Cada desarrollador tiene una copia completa del repositorio, por lo que no depende de un servidor central para trabajar.

En resumen:
Git es una herramienta que sirve para gestionar versiones de proyectos, colaborar en equipo y mantener un historial de cambios seguro y organizado.
Es la base de plataformas muy usadas como **GitHub, GitLab o Bitbucket**.

P.16 Informe Técnico

Herramientas y Tecnologías de Versionamiento e Integración Continua

---

Portada

Servicio Nacional de Aprendizaje – SENA
Programa: Análisis y Desarrollo de Software
Ficha: [Número de ficha]
Aprendiz: [Tu nombre]
Instructor: [Nombre del instructor]
Fecha: [Día/Mes/Año]

Introducción

El desarrollo de software moderno requiere de herramientas que permitan garantizar la calidad, trazabilidad y eficiencia en la construcción de aplicaciones. Una de las prácticas fundamentales es el uso de **sistemas de control de versiones**, que permiten gestionar cambios en el código, colaborar en equipo y mantener un historial detallado de cada modificación.

En el presente informe se describen las herramientas y tecnologías de versionamiento seleccionadas para el proyecto, diferenciando entre sistemas de control de versiones **locales y remotos**, y relacionando su importancia en la **integración continua**. Asimismo, se presentan las herramientas de desarrollo que se emplearán en la codificación del software, asegurando un proceso organizado y eficiente.

Objetivo

Seleccionar e implementar herramientas de desarrollo y control de versiones que permitan gestionar los cambios del código fuente, favorecer la colaboración en equipo y aplicar prácticas de integración continua que garanticen la calidad y escalabilidad del software a desarrollar.

Diferencia entre Versionamiento Local y Remoto

- ### Versionamiento Local:

1. El control de versiones se realiza en el equipo del desarrollador.
2. Se mantiene un historial de cambios solo en el repositorio local.
3. Ejemplo: usar **Git** sin conexión a una plataforma como GitHub.
4. **Ventaja:** rapidez y control personal sobre los cambios.
5. **Limitación:** dificulta la colaboración y sincronización entre equipos.

- ### Versionamiento Remoto:

1. El repositorio se almacena en un servidor o en la nube, accesible a todos los integrantes del equipo.
2. Permite sincronizar cambios entre diferentes desarrolladores.
3. Ejemplos: **GitHub, GitLab, Bitbucket**.
4. **Ventaja:** facilita la colaboración, la trazabilidad y la integración continua.
5. **Limitación:** depende de conexión a internet y plataformas externas.

Selección de Herramientas de Desarrollo para la Codificación del Software

1. **Visual Studio Code (VS Code):**

    - Editor de código multiplataforma.
    - Ligero, personalizable mediante extensiones.
    - Integración con Git para control de versiones.
2. **Java (Lenguaje de programación):**

    - Lenguaje orientado a objetos, robusto y multiplataforma.
    - Se adapta a sistemas empresariales y escalables.
3. **MySQL (Base de datos):**

    - Sistema de gestión de bases de datos relacional.
    - Permite estructurar y consultar datos del software de forma eficiente.
4. **Node.js (para backend o servicios):**

    - Plataforma que permite ejecutar JavaScript en el servidor.
    - Ideal para construir APIs y servicios complementarios.

Selección y Descripción de Herramientas de Versionamiento

1. **Git:**

    - Sistema de control de versiones distribuido.
    - Permite gestionar el historial de cambios de forma local y remota.
    - Soporta trabajo con ramas (branches) y fusiones (merge).
    - Es rápido, seguro y eficiente para proyectos de cualquier tamaño.
2. **GitHub:**

    - Plataforma en la nube que complementa a Git.
    - Permite almacenar repositorios remotos, colaborar en equipo y gestionar versiones.
    - Ofrece tableros de proyectos, control de incidencias y revisiones de código.
3. **GitHub Actions (Integración Continua):**

    - Herramienta para automatizar flujos de trabajo.
    - Permite ejecutar pruebas automáticas, validar código y realizar despliegues continuos.
    - Asegura calidad del software desde las primeras fases del desarrollo.

Conclusión

El uso combinado de **Git (versionamiento local)** y **GitHub (versionamiento remoto)** permite llevar un control eficiente del código y facilita la colaboración entre los integrantes del proyecto. Además, la integración de **GitHub Actions** en el flujo de trabajo asegura que el software sea probado, validado y desplegado de manera automática, aplicando principios de **integración continua**.

Las herramientas de desarrollo seleccionadas, como **Visual Studio Code, Java y MySQL**, garantizan un entorno de programación moderno y confiable, alineado con los objetivos del proyecto. En conjunto, estas tecnologías optimizan el trabajo en equipo, aumentan la productividad y mejoran la calidad del producto final.

P19. 📄 Informe Técnico

Estándar de Codificación para el Desarrollo del Software

Portada

Servicio Nacional de Aprendizaje – SENA
Programa: Análisis y Desarrollo de Software
Ficha: [Número de ficha]
Aprendiz: [Tu nombre]
Instructor: [Nombre del instructor]
Fecha: [Día/Mes/Año]

Introducción

En el desarrollo de software, los estándares de codificación son fundamentales para garantizar la legibilidad, mantenibilidad y calidad del código fuente. El paradigma **orientado a objetos (POO)** establece principios como la encapsulación, herencia y polimorfismo, que requieren de una escritura clara y organizada para facilitar la colaboración en equipos de trabajo y la comprensión del sistema en el futuro.

El presente informe define los estándares de codificación que se aplicarán al proyecto, tomando como base el lenguaje **Java**, ampliamente utilizado en entornos empresariales. Se abordan normas de nombramiento de variables, clases, métodos, constantes y comentarios, siguiendo buenas prácticas de programación.

Objetivo

Establecer un conjunto de estándares de codificación en Java que garanticen la claridad, uniformidad y mantenibilidad del software desarrollado bajo el paradigma orientado a objetos.

Estándares de Codificación

🔹 1. Nombramiento de Variables

- Usar **camelCase**: la primera palabra en minúscula y las siguientes con inicial en mayúscula.
- Deben ser descriptivas y claras.
- Evitar abreviaciones innecesarias.

Ejemplo:

- int numeroClientes;
- String nombreUsuario;

🔹 2. Declaración de Clases

- Usar **PascalCase**: cada palabra comienza con mayúscula.
- El nombre debe ser un **sustantivo** que represente el objeto.

Ejemplo:

- public class Cliente { }
- public class ProductoInventario { }

🔹 3. Declaración de Métodos

- Usar camelCase.
- El nombre debe ser un verbo que indique la acción del método.
- Incluir parámetros descriptivos.

Ejemplo:

- public void calcularTotalPedido() { }
- public String obtenerNombreCliente(int idCliente) { }

◈ 4. Declaración de Constantes

- Usar mayúsculas con guiones bajos.
- El nombre debe ser representativo del valor que almacena.

Ejemplo:

- public static final int MAX_CLIENTES = 100;
- public static final String MENSAJE_ERROR = "Ocurrió un error en la transacción";

◈ 5. Comentarios

- Usar comentarios para explicar bloques de código y métodos.
- Se deben usar tres tipos:
  - Comentarios de línea: //
  - Comentarios de bloque: /* ... */
  - Comentarios de documentación (JavaDoc): /** ... */

Ejemplo:

```
// Este método calcula el total de la compra
public double calcularTotal(double precio, int cantidad) {
   return precio * cantidad;
}



/**
 * Clase Cliente que almacena información básica.
 * @author Nombre
 * @version 1.0
 */
public class Cliente {
   private String nombre;
   private int edad;

}
```

🔷 6. Estándares Adicionales

- Indentación de **4 espacios** por nivel.
- Máximo de **80-100 caracteres por línea**.
- Cada clase debe guardarse en un archivo independiente con el mismo nombre de la clase.
- Uso de nombres en inglés cuando el proyecto lo requiera para estandarización.

Conclusión

El uso de estándares de codificación asegura que el software desarrollado bajo el paradigma orientado a objetos sea claro, mantenible y escalable. La aplicación de normas como **camelCase para variables y métodos, PascalCase para clases y mayúsculas para constantes**, junto con comentarios adecuados, garantiza un código limpio y entendible para todo el equipo de desarrollo.

Estos lineamientos permitirán que el proyecto mantenga coherencia en su desarrollo, faciliten la integración de nuevos programadores y aseguren buenas prácticas en la industria del software.

P.22 📄 Informe Técnico

Diferencias entre Git Local y Git Remoto

Portada

Servicio Nacional de Aprendizaje – SENA
Programa: Análisis y Desarrollo de Software
Ficha: [Número de ficha]
Aprendiz: [Tu nombre]
Instructor: [Nombre del instructor]
Fecha: [Día/Mes/Año]

Introducción

En el desarrollo de software, el uso de sistemas de control de versiones es fundamental para garantizar la correcta administración de los cambios en el código fuente. Git es

uno de los sistemas de control de versiones distribuidos más utilizados en la industria, gracias a su flexibilidad, eficiencia y capacidad de trabajo colaborativo.

Git puede trabajar en dos modalidades: **local** y **remoto**. Git local permite al desarrollador trabajar directamente en su máquina, gestionando versiones y ramas sin necesidad de conexión a internet. Git remoto, por otro lado, permite la colaboración en equipo a través de servidores centralizados como GitHub, GitLab o Bitbucket.

Este informe presenta las diferencias, características y comandos básicos de Git local y remoto.

Objetivo

Identificar y describir las diferencias entre Git local y Git remoto, así como sus características principales y los comandos más utilizados en cada uno, con el fin de fortalecer las competencias en integración continua y control de versiones.

Diferencias entre Git Local y Git Remoto

Aspecto	Git Local	Git Remoto
Definición	Es el repositorio almacenado en la máquina del desarrollador.	Es el repositorio alojado en un servidor externo (GitHub, GitLab, Bitbucket).
Acceso	No requiere conexión a internet.	Requiere internet para sincronizar cambios.
Uso principal	Guardar y administrar cambios de forma individual.	Compartir, colaborar y sincronizar con otros desarrolladores.
Control	El control está únicamente en el equipo local.	Permite trabajo colaborativo con múltiples usuarios.
Seguridad	Riesgo de pérdida si no se hacen copias externas.	Copia de seguridad en la nube o servidor.
Velocidad	Más rápido al no depender de red.	Puede ser más lento al subir/bajar cambios.

Características de Git Local y Git Remoto

Git Local	Git Remoto
Manejo independiente de commits y ramas.	Permite integrar cambios de varios colaboradores.
No requiere autenticación.	Necesita autenticación (SSH, HTTPS, tokens).
Ideal para pruebas y desarrollo individual.	Esencial en proyectos colaborativos.
Historial accesible sin internet.	Historial accesible desde cualquier lugar con internet.

Comandos Básicos de Git Local

Comando	Descripción
git init	Inicializa un nuevo repositorio en la carpeta local.
git add .	Agrega los archivos modificados al área de preparación (staging area).
git commit -m "mensaje"	Registra los cambios en el historial local.
git status	Muestra el estado actual de los archivos del repositorio.
git log	Muestra el historial de commits realizados en el repositorio local.
git branch	Lista, crea o elimina ramas en el repositorio local.

Comandos Básicos de Git Remoto

Comando	Descripción
git clone <url>	Clona un repositorio remoto a la máquina local.
git remote add origin <url>	Conecta el repositorio local con uno remoto.
git push origin main	Envía los commits locales al repositorio remoto.
git pull origin main	Descarga y fusiona los cambios desde el repositorio remoto.
git fetch	Descarga cambios del remoto sin fusionarlos.
git push -u origin <rama>	Sube una rama local al repositorio remoto y la vincula.

Conclusión

Git local y Git remoto son dos componentes complementarios del control de versiones. Mientras el primero permite trabajar de manera individual, el segundo facilita la

colaboración y la integración continua en equipos de desarrollo. El conocimiento y dominio de los comandos básicos de ambos es fundamental para garantizar un flujo de trabajo ágil, organizado y seguro en los proyectos de software.

Curso GitHub.

https://www.youtube.com/watch?v=eQMcIGVc8N0