

## TEMA 4. OPTIMIZACIÓN Y DOCUMENTACIÓN.

---

### Objetivos

- Concepto de refactorización.
- Documentación con Javadoc en Eclipse.
- Concepto de control de versiones.
- Introducción a GIT.

### Contenidos

#### 1.- Refactorización.

La **refactorización** (del inglés *refactoring*) es una técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo. Informalmente se denomina *limpiar el código*.

Se realiza a menudo como parte del proceso de desarrollo del software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad.

Los tests aseguran que la refactorización no cambia el comportamiento del código.

Es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad.

**Objetivo:** mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro.

El código debe ser limpio y bien estructurado.

En Java, existe una serie de recomendaciones en el estilo de la programación, también llamadas **convenciones**.

**Eclipse y Netbeans** nos proporciona algunas herramientas para mejorar la calidad del código:

- Vamos a centrarnos en algunas herramientas automáticas de formateo y refactorización del código.
- Con una simple combinación de teclas se puede llevar a cabo algunas de las buenas prácticas de Java y mejorar el código manteniendo la funcionalidad.

#### **Atajos de teclado**

##### **Optimización de imports:**

Se reorganizan los imports de la clase actual. Se eliminan los que no se utilizan y añade los que falten.

**Eclipse:** CTRL + MAYUS + O

**Netbeans:** CTRL + MAYUS + I

**Es importante repasar los imports una vez hecho esto, ya que si encuentra dos clases que se llaman igual pero están en distintos paquetes, se puede liar e importar el que no queremos. Por ejemplo, la clase Date, que existe en los paquetes util y sql de Java.**

### Formateo del código:

Un buen formato y tabulado de código facilita su lectura.

**Eclipse:** CONTROL + MAYUS + F

**Netbeans:** ALT + MAYUS + F

### Cambio de nombres de variables o clases:

Si hemos creado una variable o clase y nos hemos saltado la nomenclatura que debería tener, se puede arreglar fácilmente con Eclipse.

Se puede renombrar seleccionando el nombre de la clase o la variable y pulsando:

**Eclipse:** ALT + MAYÚSCULAS + R.

**Netbeans:** CTRL + R


El mismo IDE se encargará de buscar todas las ocurrencias de la misma dentro de su ámbito y renombrarlas.

En el caso de las clases también cambia el nombre del fichero, ya que debe coincidir con el de la clase que contiene.

Si encuentra conflictos avisará para que intervenga el usuario.


### Mejoras en el código

a) Gastar el mínimo número de variables, es decir, solo las necesarias:


<pre>int res1; int res2; int res3; int res4;  if(opcion == 1){     res1 = num1 + num2;     System.out.println(res1); }else if(opcion == 2){     res2 = num3 + num4;     System.out.println(res2); }else if(opcion == 3){     res3 = num5 + num6;     System.out.println(res3); }else if(opcion == 4){     res4 = num7 + num8;     System.out.println(res4); }</pre>	 <b>Refactorizado</b>	<pre>int resultado;  if(opcion == 1){     resultado = num1 + num2;     System.out.println(resultado); }else if(opcion == 2){     resultado = num3 + num4;     System.out.println(resultado); }else if(opcion == 3){     resultado = num5 + num6;     System.out.println(resultado); }else if(opcion == 4){     resultado = num7 + num8;     System.out.println(resultado); }</pre>
---	---	--

b) Evitar repetir código (A partir del ejemplo anterior (refactorizado)):

El **System.out.println(resultado)** se está repitiendo en cada opción, por lo que podríamos extraerlo al final.

<pre>int resultado;  if(opcion == 1){     resultado = num1 + num2;     System.out.println(resultado); }else if(opcion == 2){     resultado = num3 + num4;     System.out.println(resultado); }else if(opcion == 3){     resultado = num5 + num6;     System.out.println(resultado); }else if(opcion == 4){     resultado = num7 + num8;     System.out.println(resultado); }</pre>	<p>Refactorizado</p> 	<pre>int resultado = 0;  if(opcion == 1){     resultado = num1 + num2; }else if(opcion == 2){     resultado = num3 + num4; }else if(opcion == 3){     resultado = num5 + num6; }else if(opcion == 4){     resultado = num7 + num8; }  System.out.println(resultado);</pre>
--	--	--

c) Intentar gastar la mejor estructura posible: Siempre que queramos evaluar la misma variable muchas veces, poner un **if-else if- else if- else if...** no es la mejor opción. Para eso existe la estructura **switch**.

<pre>int resultado = 0;  if(opcion == 1){     resultado = num1 + num2; }else if(opcion == 2){     resultado = num3 + num4; }else if(opcion == 3){     resultado = num5 + num6; }else if(opcion == 4){     resultado = num7 + num8; }  System.out.println(resultado);</pre>	<p>Refactorizado</p> 	<pre>int resultado = 0;  switch(opcion){     case 1:         resultado = num1 + num2;         break;     case 2:         resultado = num3 + num4;         break;     case 3:         resultado = num5 + num6;         break;     case 4:         resultado = num7 + num8;         break; }  System.out.println(resultado);</pre>
--	---	--

d) Nombres de variables descriptivos: Cuanto más descriptivo sea el nombre de la variable, menos comentarios tendremos que realizar. En el ejemplo anterior, las variables resultado y opcion, muestran realmente lo que almacenan.

<p style="color: green; text-align: center;"><b>Bien</b></p> <pre>int resultado = 0;  switch(opcion){     case 1:         resultado = num1 + num2;         break;     case 2:         resultado = num3 + num4;         break;     case 3:         resultado = num5 + num6;         break;     case 4:         resultado = num7 + num8;         break; }  System.out.println(resultado);</pre>	<p style="color: red; text-align: center;"><b>Mal</b></p> <pre>int a = 1; int b = 0;  switch(a){     case 1:         b = num1 + num2;         break;     case 2:         b = num3 + num4;         break;     case 3:         b = num5 + num6;         break;     case 4:         b = num7 + num8;         break; }  System.out.println(b);</pre>
---	--

e) Hacer solo las operaciones necesarias: No siempre tenemos que realizar todas las operaciones, tan solo las que se necesiten en el flujo de ejecución del programa. En el siguiente ejemplo, solo deberíamos hacer la operación que seleccione el usuario.

```
System.out.print("Introduce número 1: ");
num1 = tec.nextDouble();
tec.nextLine();
System.out.print("Introduce número 2: ");
num2 = tec.nextDouble();
tec.nextLine();

resMedia = (num1 + num2) / 2;
resRaiz = Math.sqrt(num1 + num2);

System.out.println("¿Qué desea realizar? (1 = Media, 2 = Raiz cuadrada)");
opcion = tec.nextLine();

if(opcion.equals("1")){
    System.out.println(resMedia);
}else if(opcion.equals("2")){
    System.out.println(resRaiz);
}

↓ Refactorizado

System.out.print("Introduce número 1: ");
num1 = tec.nextDouble();
tec.nextLine();
System.out.print("Introduce número 2: ");
num2 = tec.nextDouble();
tec.nextLine();

System.out.println("¿Qué desea realizar? (1 = Media, 2 = Raiz cuadrada)");
opcion = tec.nextLine();

if(opcion.equals("1")){
    resMedia = (num1 + num2) / 2;
    System.out.println(resMedia);
}else if(opcion.equals("2")){
    resRaiz = Math.sqrt(num1 + num2);
    System.out.println(resRaiz);
}
```

f) Usar tipos primitivos: Los tipos primitivos son más eficientes y ocupan menos memoria, por lo que deberíamos usarlos, **siempre que podamos**.

En el anterior ejemplo, podríamos capturar la opción en formato **int**, para usar un tipo primitivo, que sea más fácil de comparar y más eficiente.

```

System.out.print("Introduce número 1: ");
num1 = tec.nextDouble();
tec.nextLine();
System.out.print("Introduce número 2: ");
num2 = tec.nextDouble();
tec.nextLine();

System.out.println("¿Qué desea realizar? (1 = Media, 2 = Raiz cuadrada)");
opcion = tec.nextLine();

if(opcion.equals("1")){
    resMedia = (num1 + num2) / 2;
    System.out.println(resMedia);
}else if(opcion.equals("2")){
    resRaiz = Math.sqrt(num1 + num2);
    System.out.println(resRaiz);
}

```



**Refactorizado**

```

System.out.print("Introduce número 1: ");
num1 = tec.nextDouble();
tec.nextLine();
System.out.print("Introduce número 2: ");
num2 = tec.nextDouble();
tec.nextLine();

System.out.println("¿Qué desea realizar? (1 = Media, 2 = Raiz cuadrada)");
opcion = tec.nextInt();

if(opcion == 1){
    resMedia = (num1 + num2) / 2;
    System.out.println(resMedia);
}else if(opcion == 2){
    resRaiz = Math.sqrt(num1 + num2);
    System.out.println(resRaiz);
}

```

El programador se pone a programar alguna parte del programa y le hace una primera prueba, compilando y viendo que funciona. Después se dispone a arreglar el código, cambiando código de sitio, haciendo métodos más pequeños, etc, etc. Al final deja el código funcionando exactamente igual que antes, pero hecho de otra manera que le facilita seguir con su trabajo.

### ***Ventajas de la refactorización***

Si hacemos refactoring cada vez que veamos código feo, el código ***se mantiene más elegante y sencillo***.

Aunque inicialmente parece una pérdida de tiempo arreglar el código, al final ***se gana tiempo***.

Las modificaciones y añadido tardan menos y se pierde mucho **menos tiempo en depurar y entender el código**.

**Minimiza comentarios:** Si el código es legible y con nombres descriptivos, no es necesario añadir comentarios ni tanta documentación.

### **Un libro sobre refactoring**

El libro por excelencia sobre refactoring es "refactoring" de Martin Fowler.

La web de refactoring es <http://www.refactoring.com>.

Algunas de las cosas que comenta son:

- **Código duplicado.** Si hay líneas de código exactamente iguales en varios sitios, se deberían extraer en métodos.
  - **Métodos muy largos.** Se deben partir en métodos más pequeños.
  - **Clases muy grandes.** En estos casos debería tratar de identificarse qué cosas hace esa clase, ver si realmente todas esas cosas tienen algo que ver la una con la otra y si no es así, hacer clases más pequeñas, de forma que cada una trate una de esas cosas.
- Métodos que necesitan muchos parámetros.** Suele ser buena idea hacer una clase que contenga esos parámetros y pasar la clase en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.

## **2.- Documentación con Javadoc en Netbeans.**

Javadoc es un paquete de desarrollo de java que nos genera documentación sobre nuestro proyecto de cada una de nuestras clases en una jerarquía de árbol. Su uso es recomendable para todo proyecto, incluido los proyectos pequeños porque nos ayuda a tener siempre un código bien documentado.

Javadoc funciona por medio de marcas que le ayudan a reconocer las partes que queremos documentar y crear los archivos HTML:

<p><b>@author</b> - El nombre del autor del proyecto</p> <p><b>@version</b> - La versión del proyecto</p> <p><b>@see</b> - Añade una referencia a una clase, método o enlace web</p> <p><b>@param</b> - Nombre parámetro utilizado en un método incluido significado</p> <p><b>@return</b> - El resultado de un método incluido su descripción</p> <p><b>@exception</b> - Nombre de la excepción mas una descripción</p> <p><b>@throws</b> - Nombre de la excepción mas una descripción</p> <p><b>@deprecated</b> - Añade una alerta al usuario de que el método que sigue a continuación ya no debe usarse y que será eliminado en versiones posteriores</p>
---

Para indicarle a Javadoc que queremos incluir documentación, debemos comenzar los comentarios de la siguiente manera:

<pre>/**  * Esto para Javadoc  */</pre>
---

Al comentar una clase, en el encabezado colocamos el autor de la clase, la instrucción @see con un enlace a una página web, también podemos agregar la versión de la clase:

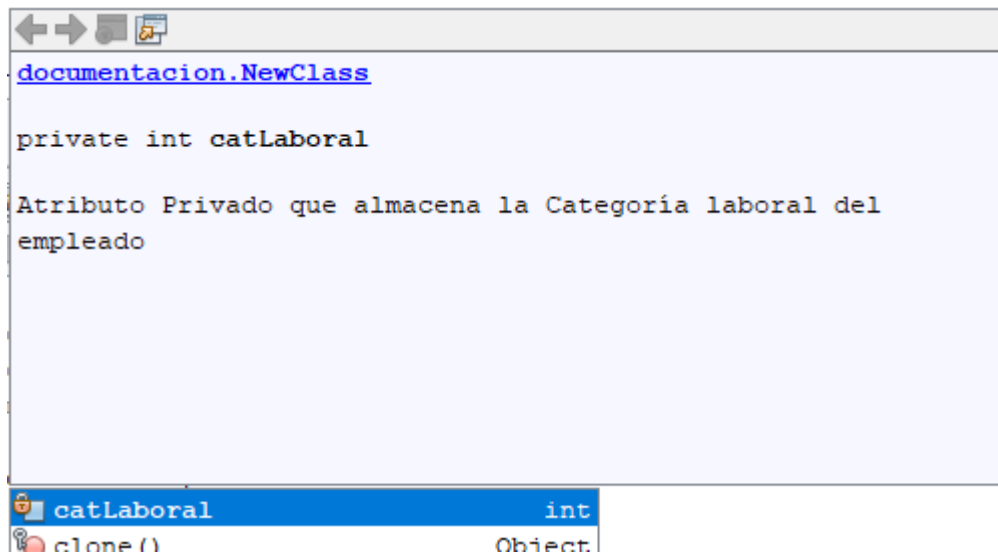
```
/**
 * @author Mario
 * @see <a href="http://www.campusaula.com">http://www.campusaula.com</a>
 * @version 1.2 07 de Mayo de 2013
 */
public class Clase_Java {
    ...
}
```

Para mostrar el autor y la versión en Javadoc debemos configurar el proyecto con el botón derecho sobre el proyecto y Propiedades. En la categoría Documentando marcamos mostrar Autor y Versión.

También podemos documentar las atributo

```
/**
 * Atributo Privado que almacena la Categoría laboral del empleado
 */
private int catLaboral;
```

Cuando se haga uso de atributos comentados, en el editor de Netbeans mostrará:



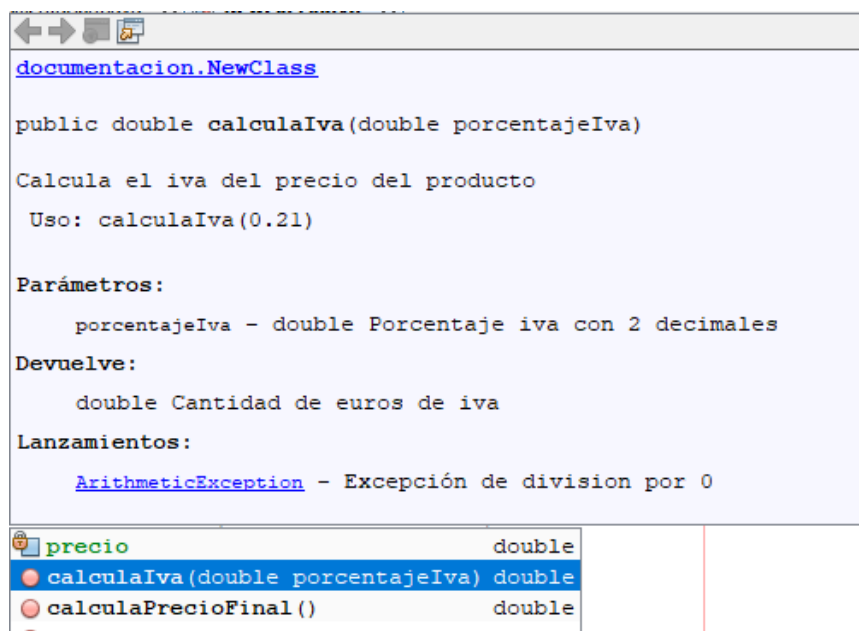
Debemos también comentar el constructor de clase:

```
/**
 * Constructor de clase
 */
public Clase_Java(){
    ...
}
```

Cuando creamos un método es recomendable documentar los parámetros de entrada, si es que los tuviera, y si este método retorna algún resultado, utilizar la marca `@return`, si por el contrario el método es de la forma VOID, no se usa nada. Así también para las excepciones que puedan ocurrir se usa `@exception`. En la descripción del método, se puede incluir un ejemplo de uso encerrado en las etiquetas PRE, por ejemplo:

```
/**
 * Calcula el iva del precio del producto
 * <pre> Uso: calculaIva(0.21)</pre>
 * @param porcentajeIva double Porcentaje iva con 2 decimales
 * @return double Cantidad de euros de iva
 * @exception ArithmeticException Excepción de división por 0
 */
public double calculaIva(double porcentajeIva) {
    return this.precio*porcentajeIva;
}
```

Cuando utilicemos este método, podremos ver que Netbeans nos despliega toda la información en cuando se hace referencia al nombre `calculaIva()`.



Cuando tenemos varias versiones de un proyecto y vamos actualizando los métodos, antes de eliminar los métodos que ya no se harán uso, es preferible, primero alertar al usuario que ciertas funciones dejarán de existir en versiones posteriores, para esto, en la descripción se añade la marca `@deprecated`, y la marca `@see`, que hace referencia a la función de reemplazo, es decir, supongamos que en una primera versión tenemos un método llamado Suma pero que en una nueva versión se decide crear una nueva versión `suma_positivos`, entonces lo que debemos hacer es añadir en la documentación de Suma, la alerta de que esta función ya no se utilizará más y que es mejor usar la nueva función `suma_positivos`, es decir:



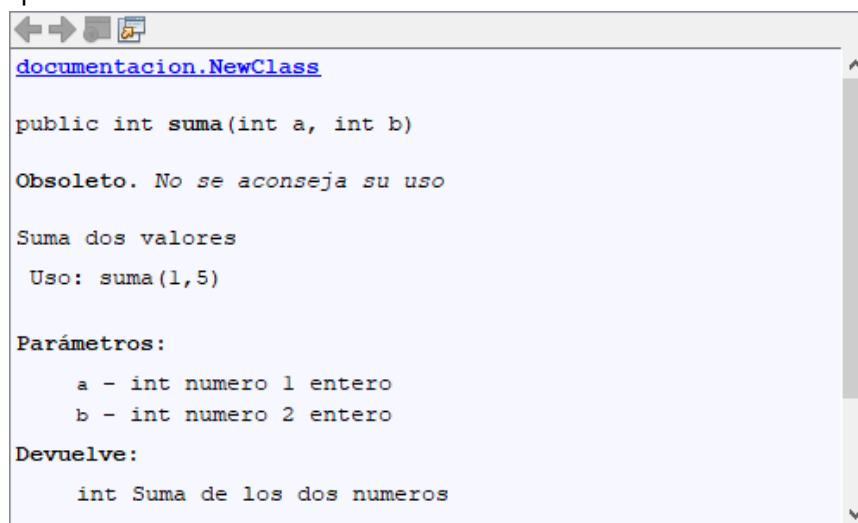
```

/**
 * Suma dos valores
 * <pre> Uso: suma(1,5)</pre>
 * @param a int numero 1 entero
 * @param b int numero 2 entero
 * @return int Suma de los dos numeros
 * @exception ArithmeticException Excepción de división por 0
 * @deprecated No se aconseja su uso
 * @see suma_positivos(int, int)
 */
public int suma(int a, int b){
    int suma = a+b;
    return suma;
}

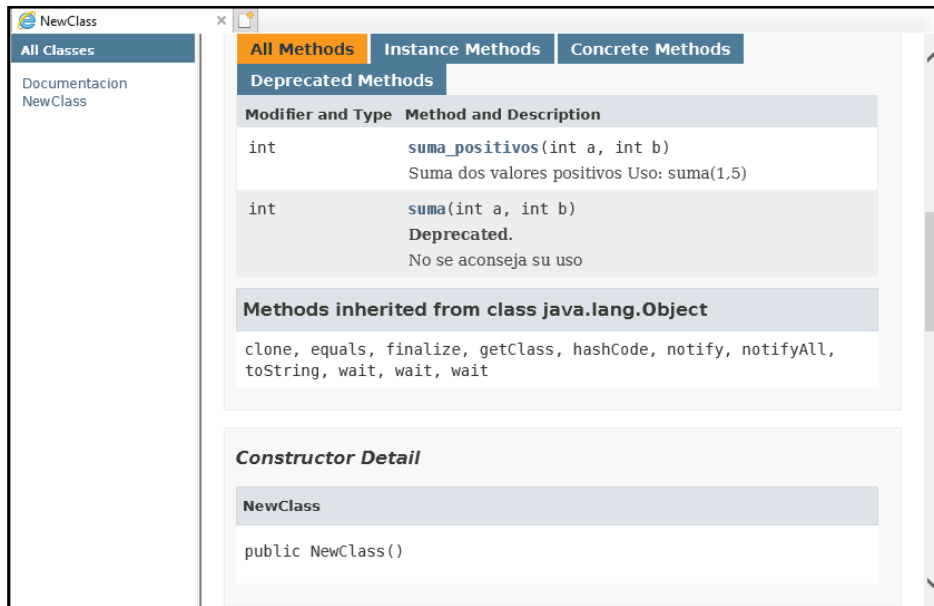
/**
 * Suma dos valores positivos
 * <pre> Uso: suma(1,5)</pre>
 * @param a int numero 1 entero
 * @param b int numero 2 entero
 * @return int Suma de los dos numeros o 0 si alguno de los números es
negativo
 */
public int suma_positivos(int a, int b){
    int suma=0;
    if(a>=0 && b>=0){
        suma=a+b;
    }
    return suma;
}

```

Ahora, cuando se haga uso del método Suma, este nos aparecerá con un subrayado y con la alerta correspondiente:



Para generar los HTML, vamos a Ejecutar -> Generar JavaDoc. y esperamos unos segundos a que nos genere todos los archivos.



### 3.- Documentación con Javadoc en Eclipse

Cuando desarrollamos aplicaciones, de un modo paralelo debemos construir su documentación. Esta documentación:

- No debe ser demasiado extensa.
- Debe explicar de una manera resumida los métodos y/o clases más importantes.

Desde las primeras versiones de Java e incluso para documentar las propias librerías del lenguaje, se utiliza un mecanismo llamado **Javadoc** para garantizar que el código y su documentación están sincronizados.

Este sistema *consiste en incluir comentarios en el código, utilizando unas etiquetas especiales (entre **/\*\*** y **\*/**), que después pueden procesarse y genera un HTML navegable.*

#### Ejemplo:

Crea un nuevo proyecto llamado **PruebaJavadoc**.

Crea una clase llamada **Principal** y marca el check de generar comentarios:

**NOTA:** Esto hará que se generen automáticamente el comentario de autor.

```
/**  
  
/**  
 * @author Mario G  
 *  
 */  
public class Principal {  
  
}
```

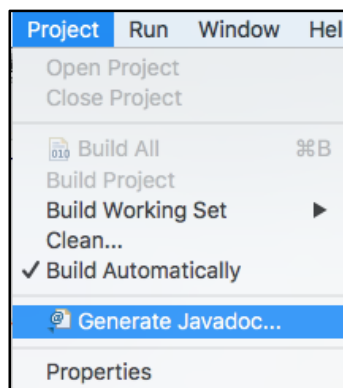
**NOTA:** Este autor lo obtiene del usuario del equipo.

Añade el método main que pida 2 números enteros por teclado y los sume. Después añade justo encima de la cabecera del método un comentario de javadoc indicando lo que hace el método.

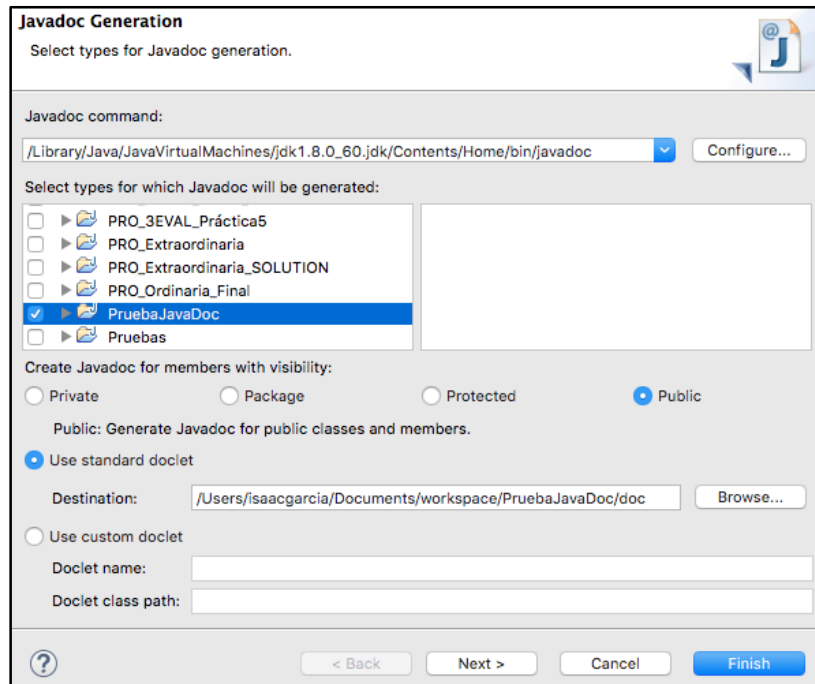
El resultado sería el siguiente:

```
/**  
 * Este método es el principal del programa.  
 * Es un sencillo programa que pide dos números enteros y muestra el resultado de su suma.  
 */  
public static void main(String args[]){  
    int numEntero1, numEntero2;  
  
    Scanner teclado = new Scanner(System.in);  
    System.out.println("Número 1: ");  
    numEntero1 = teclado.nextInt();  
    System.out.println("Número 2: ");  
    numEntero2 = teclado.nextInt();  
  
    System.out.println("El resultado es " + (numEntero1 + numEntero2));  
}
```

Por último vamos a generar la documentación desde el menú de **proyecto – Generar Javadoc**

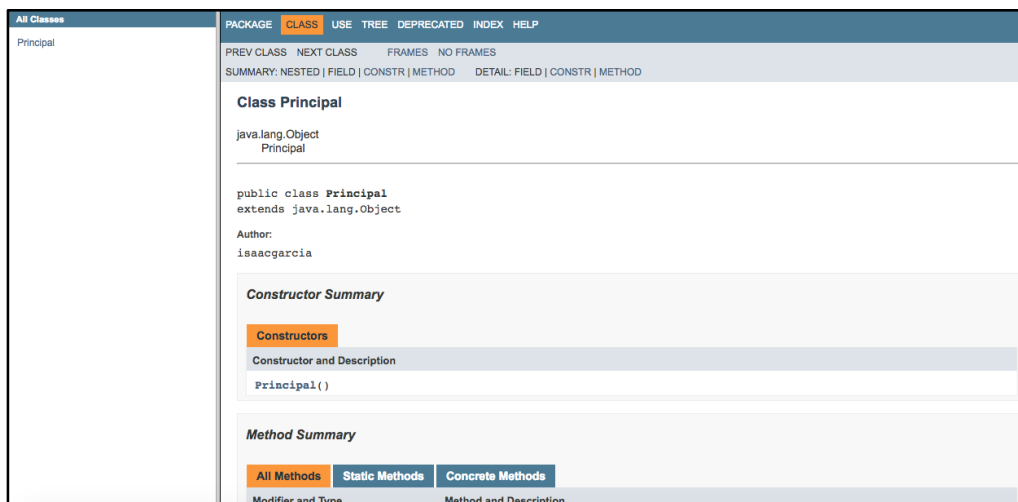


Esto abrirá un asistente donde seleccionaremos el proyecto y la carpeta donde se genera (por defecto la carpeta **doc** dentro del proyecto). El comando Javadoc se encuentra en la carpeta C:\Program Files\Java\jdk1.8.0\_191\bin



Finalizamos y confirmamos que queremos sobrescribir la carpeta.

Si abrimos ahora la carpeta del proyecto desde el explorador, habrá una nueva carpeta llamada doc con la documentación en formato html. Para verla es tan sencillo como abrir el fichero **index.html** y podremos navegar por todas las clases.



#### 4.- Control de versiones.

Se puede definir el control de versiones como la capacidad de recordar todos los cambios que se han realizado tanto en la estructura de directorios como en el contenido de los archivos. Es muy útil cuando se necesita mantener un cierto control de los cambios que se realizan sobre documentos, archivos o proyectos que comparten varias personas o un equipo de trabajo. Con el uso de un Sistema de Control de Versiones podemos saber qué cambios se hacen, quien los ha efectuado y cuando los ha realizado.

El Sistema Control de versiones registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Tipos:

- **Centralizados:**
  - Un único repositorio centralizado de todo el código.
  - Poca flexibilidad: todas las decisiones fuertes necesitan la aprobación del responsable.
  - Algunos ejemplos son CVS, Subversion o Team Foundation Server.
- **Distribuidos:**
  - Cada usuario tiene su propio repositorio.
  - Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos.
  - También existe un repositorio centralizado, que sirve de punto de sincronización de los distintos repositorios locales.
  - Ejemplos: Git y Mercurial.

## 5.- Git.

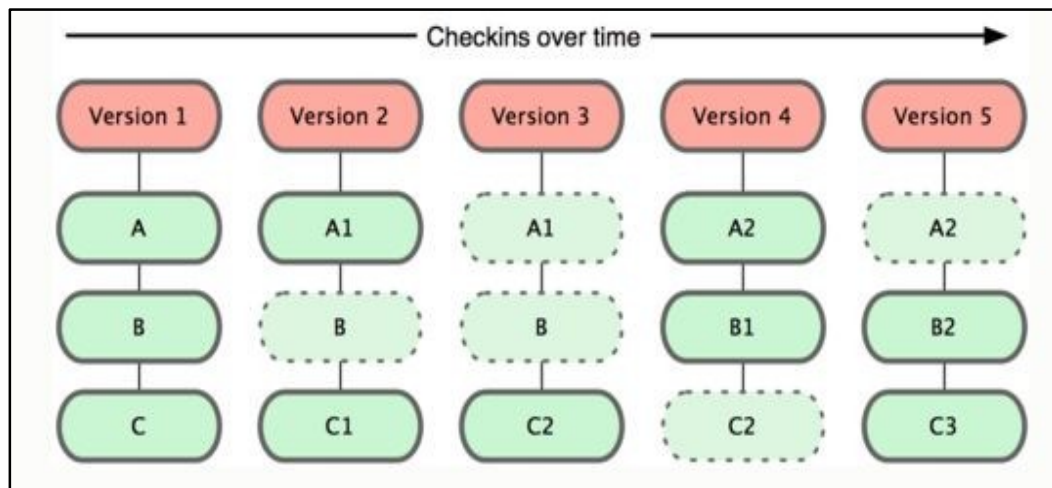
Git es el sistema de control de versiones por excelencia.

Potente y muy versátil: Todo gran proyecto que se precie, tendrá un repositorio git.

Empresas como Apple o Google lo utilizan a diario.

Diseñado por Linus Torvalds.

Git almacena la información como instantáneas del proyecto a lo largo del tiempo.



### Estados de un fichero

Git tiene tres estados principales en los que se pueden encontrar tus archivos:

- Modificado (**modified**): El archivo ha sido modificado pero todavía no se ha confirmado en tu repositorio local.
- Preparado (**staged**): El archivo modificado ha sido marcado para que se guarde en la próxima confirmación.

- Confirmado (**committed**): El fichero ha sido confirmado en el repositorio local, por lo que se ha creado una nueva versión.

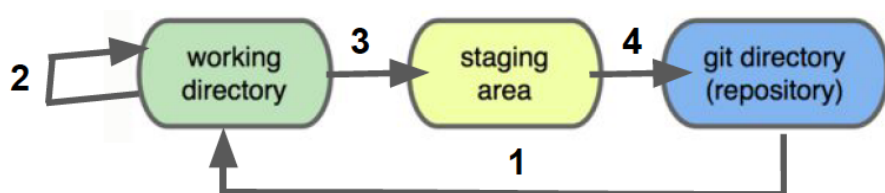
### Secciones principales de un proyecto Git

Estos estados nos llevan a tres secciones principales de un proyecto de Git:

- Directorio de trabajo (**working directory**): copia local de una versión del repositorio.
- Área de preparación (**staging area**): Archivo, contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación.
- Directorio .git (**repository**): Almacena los metadatos y el repositorio local de tu proyecto.

### Flujo de trabajo básico

- 1.- **Clonar o Crear** un repositorio en forma local.
- 2.- **Modificar** una serie de archivos en tu directorio de trabajo.
- 3.- Preparar los archivos, **añadiéndolos** a tu área de preparación.
- 4.- **Confirmar** los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git.

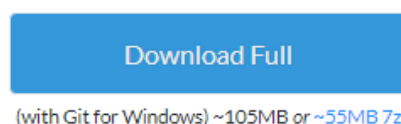


### Instalación

Existen muchas formas de instalar git, incluso tiene una consola propia desde donde ejecutar todos sus comandos.

Nosotros usaremos **Cmder** que es **una consola de linux en windows**, es decir, podremos ejecutar todos los comandos que existen en linux y además, existe la posibilidad de descargarla con **git integrado**.

La descarga se realiza desde la web <http://cmder.net/> pulsando el botón de descarga:



### Configuración inicial

#### Identidad.

Lo primero que debemos hacer cuando instalamos Git es establecer el nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que envías.

# Configuración de Git

## git config

- Establecer el nombre de usuario  
`git config --global user.name "Your-Full-Name"`
- Establecer el correo del usuario  
`git config --global user.email "your-email-address"`
- Activar el coloreado de la salida  
`git config --global color.ui auto`
- Mostrar la configuración  
`git config --list`

```
git config --global user.name "Mario Garcia"
git config --global user.email _mario.garcia@campusaula.com
git config user.name //Muestra el nombre de usuario guardado
```

## Creación de repositorios

### Creación de un repositorio nuevo

## git init

- **git init <nombre-repositorio>** crea un repositorio nuevo con el nombre <nombre-repositorio>.

Este comando crea una nueva carpeta con el nombre del repositorio, que a su vez contiene otra carpeta oculta llamada **.git** que contiene la base de datos donde se registran los cambios en el repositorio.

Si estás empezando el seguimiento en Git de un proyecto existente, necesitas ir al directorio del proyecto y escribir

```
git init
```

Esto crea un nuevo subdirectorio llamado **.git** que contiene todos los archivos necesarios del repositorio.

## Añadir cambios a la zona de intercambio temporal

### **git add**

- **git add <fichero>** añade los cambios en el fichero **<fichero>** del directorio de trabajo a la zona de intercambio temporal.
- **git add <carpeta>** añade los cambios en todos los ficheros de la carpeta **<carpeta>** del directorio de trabajo a la zona de intercambio temporal.
- **git add .** añade todos los cambios de todos los ficheros no guardados aún en la zona de intercambio temporal.

## Estado e historia de un repositorio

### Mostrar el estado de un repositorio

### **git status**

- **git status** muestra el estado de los cambios en el repositorio desde la última versión guardada. En particular, muestra los ficheros con cambios en el directorio de trabajo que no se han añadido a la zona de intercambio temporal y los ficheros en la zona de intercambio temporal que no se han añadido al repositorio.

## Añadir cambios al repositorio

### **git commit**

- **git commit -m "mensaje"** confirma todos los cambios de la zona de intercambio temporal añadiéndolos al repositorio y creando una nueva versión del proyecto. **"mensaje"** es un breve mensaje describiendo los cambios realizados que se asociará a la nueva versión del proyecto.
- **git commit --amend -m "mensaje"** cambia el mensaje del último commit por el nuevo mensaje **"mensaje"**.



## Mostrar el historial de versiones de un repositorio

### `git log`

- **git log** muestra el historial de commits de un repositorio ordenado cronológicamente. Para cada commit muestra su código hash, el autor, la fecha, la hora y el mensaje asociado.  
Este comando es muy versátil y muestra la historia del repositorio en distintos formatos dependiendo de los parámetros que se le den. Los más comunes son:
  - **--oneline** muestra cada commit en una línea produciendo una salida más compacta.
  - **--graph** muestra la historia en forma de grafo.

## Mostrar los datos de un commit

### `git show`

- **git show** muestra el usuario, el día, la hora y el mensaje del último commit, así como las diferencias con el anterior.
- **git show <commit>** muestra el usuario, el día, la hora y el mensaje del commit indicado, así como las diferencias con el anterior.

## Mostrar las diferencias entre versiones

### `git diff`

- **git diff** muestra las diferencias entre el directorio de trabajo y la zona de intercambio temporal.
- **git diff --cached** muestra las diferencias entre la zona de intercambio temporal y el último commit.
- **git diff HEAD** muestra la diferencia entre el directorio de trabajo y el último commit.

## Deshacer cambios

### Eliminar cambios del directorio de trabajo o volver a una versión anterior

#### **git checkout**

- **git checkout <commit> -- <file>** actualiza el fichero <file> a la versión correspondiente al commit <commit>. Suele utilizarse para eliminar los cambios en un fichero que no han sido guardados aún en la zona de intercambio temporal, mediante el comando **git checkout HEAD -- <file>**.

### Eliminar cambios de la zona de intercambio temporal

#### **git reset**

- **git reset <fichero>** elimina los cambios del fichero <fichero> de la zona de intercambio temporal, pero preserva los cambios en el directorio de trabajo.

Para eliminar por completo los cambios de un fichero que han sido guardados en la zona de intercambio temporal hay que aplicar este comando y después **git checkout HEAD -- <fichero>**.

- **git reset <commit>** actualiza el HEAD al commit <commit>, es decir, elimina todos los commits posteriores a este commit, pero no elimina los cambios del directorio de trabajo.

## Ramas

Inicialmente cualquier repositorio tiene una única rama llamada **master** donde se van sucediendo todos los commits de manera lineal.

Una de las característica más útiles de Git es que permite la creación de ramas para trabajar en distintas versiones de un proyecto a la vez.

Esto es muy útil si, por ejemplo, se quieren añadir nuevas funcionalidades al proyecto sin que interfieran con lo desarrollado hasta ahora.

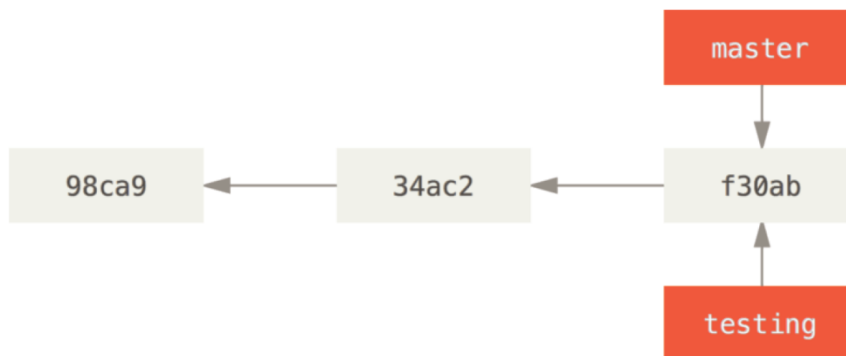
Cuando se termina el desarrollo de las nuevas funcionalidades las ramas se pueden fusionar para incorporar los cambios al proyecto principal.

## Creación de ramas

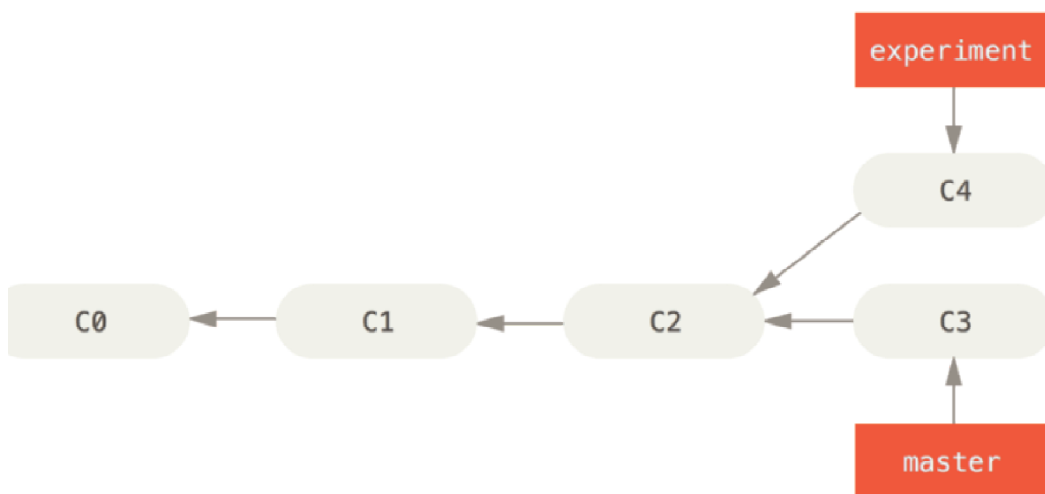
### `git branch`

- `git branch <rama>` crea una nueva rama con el nombre `<rama>` en el repositorio a partir del último commit, es decir, donde apunte HEAD.

Al crear una rama a partir de un commit, el flujo de commits se bifurca en dos de manera que se pueden desarrollar dos versiones del proyecto en paralelo.



## Desarrollo en ramas diferentes



## Listado de ramas

### git log

- **git branch** muestra las ramas activas de un repositorio indicando con \* la rama activa en ese momento.
- **git log --graph --oneline** muestra la historia del repositorio en forma de grafo (--graph) incluyendo todas las ramas (--all).

## Cambio de ramas

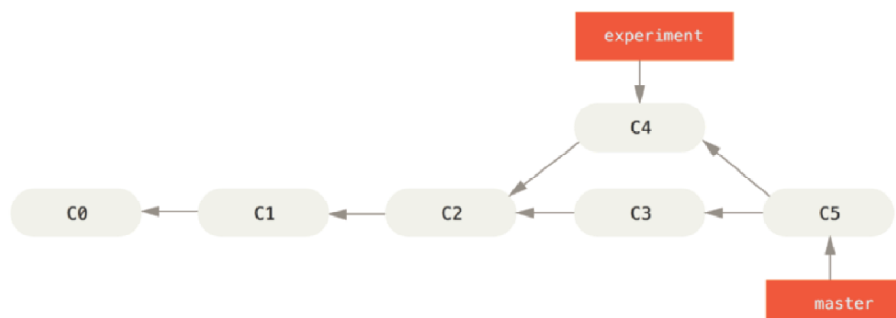
### git checkout

- **git checkout <rama>** actualiza los ficheros del directorio de trabajo a la última versión del repositorio correspondiente a la rama <rama>, y la activa, es decir, HEAD pasa a apuntar al último commit de esta rama.
- **git checkout -b <rama>** crea una nueva rama con el nombre <rama> y la activa, es decir, HEAD pasa a apuntar al último commit de esta rama. Este comando es equivalente aplicar los comandos **git branch <rama>** y después **git checkout <rama>**.

## Fusión de ramas

### git merge

- **git merge <rama>** integra los cambios de la rama <rama> en la rama actual a la que apunta HEAD.



## Eliminación de ramas

**git branch -d**

- **git branch -d <rama>** elimina la rama de nombre <rama> siempre y cuando haya sido fusionada previamente.
- **git branch -D <rama>** elimina la rama de nombre <rama> incluso si no ha sido fusionada. Si la rama no ha sido fusionada previamente se perderán todos los cambios de esa rama.

## Repositorios remotos

La otra característica de Git, que unida a las ramas, facilita la colaboración entre distintos usuarios en un proyecto son los repositorios remotos.

Git permite la creación de una copia del repositorio en un servidor git en internet. La principal ventaja de tener una copia remota del repositorio, a parte de servir como copia de seguridad, es que otros usuarios pueden acceder a ella y hacer también cambios.

Existen muchos proveedores de alojamiento para repositorios Git pero el más usado es GitHub.

## ¿Qué es GitHub?

**GitHub** es el proveedor de alojamiento en la nube para repositorios gestionados con git más usado y el que actualmente tiene alojados más proyectos de desarrollo de software de código abierto en el mundo.

La principal ventaja de GitHub es que permite albergar un número ilimitado de repositorios tanto públicos como privados, y que además ofrece servicios de registro de errores, solicitud de nuevas funcionalidades, gestión de tareas, wikis o publicación de páginas web, para cada proyecto, incluso con el plan básico que es gratuito.

## Copia de repositorios

### git clone

- **git clone <url-repositorio>** crea una copia local del repositorio ubicado en la dirección <url-repositorio>.

A partir de que se hace la copia, los dos repositorios, el original y la copia, son independientes, es decir, cualquier cambio en uno de ellos no se verá reflejado en el otro.

## Añadir un repositorio remoto

### git remote add

- **git remote add <repositorio-remoto> <url>** crea un enlace con el nombre <repositorio-remoto> a un repositorio remoto ubicado en la dirección <url>.

Cuando se añade un repositorio remoto a un repositorio, Git seguirá también los cambios del repositorio remoto de manera que se pueden descargar los cambios del repositorio remoto al local y se pueden subir los cambios del repositorio local al remoto.

## Lista de repositorios remotos

### git remote

- **git remote** muestra un listado con todos los enlaces a repositorios remotos definidos en un repositorio local.
- **git remote -v** muestra además las direcciones url para cada repositorio remoto.

## Subir cambios a un repositorio remoto

### git push

- **git push <remoto> <rama>** sube al repositorio remoto <remoto> los cambios de la rama <rama> en el repositorio local.

## Descargar cambios desde un repositorio remoto

### git pull

- **git pull <remoto> <rama>** descarga los cambios de la rama <rama> del repositorio remoto <remoto> y los integra en la última versión del repositorio local, es decir, en el HEAD.
- **git fetch <remoto>** descarga los cambios del repositorio remoto <remoto> pero no los integra en la última versión del repositorio local.

Sentencia	Acción
mkdir nombreCarpeta	Crear carpeta
cls	Borra pantalla
cd nombreCarpeta	Entrar carpeta
cd ..	Salir carpeta
ls -la ó dir	Muestra contenido carpeta
git config --global user.name usuario	Identificación usuario
git config --global user.email correo	Identificación correo
git config --global user.name	Consulta el usuario
git config --global user.email	Consulta el correo
git config --global color.ui auto	Colorear cambios consola
git config --list	Lista configuración
git init nombreCarpeta	Crea directorio trabajo git
git status	Estado del Repositorio
cat > fichero.txt	Crea fichero.txt (CTRL+D save)
git add .	Pasa ficheros al área preparada
git add fichero	Pasa fichero al área preparada
git commit -m "comentario"	Pasa del área preparada al repositorio
nano nombreFichero.txt	Crea/abre fichero(CTRL+X/Y Intro save)
git diff	Diferencia dir. trab. y el área preparada
git diff --cached	Diferencia área preparada y repositorio
git diff HEAD	Diferencia directorio trabajo y últ commit
git log	Listado de commits realizados
git log --oneline	Listados commit por línea
git log --graph	Listado commits en modo gráfico

Sentencia	Acción
<code>git commit --amend -m "comentario"</code>	Cambia el comentario del último commit
<code>git reset fichero</code>	Elimina fichero del área preparada
<code>git checkout hash -- fichero</code>	Vuelve fich a la versión anterior con hash
<code>git reset --hard HEAD</code>	Deshacer cambios desde el último commit
<code>git show</code>	Mostrar info y dif respecto último commit
<code>git show hash</code>	Mostrar info y dif respecto commit indicado
<code>git branch rama</code>	Crea una nueva rama
<code>git checkout rama</code>	Cambiamos de rama
<code>git checkout -b rama</code>	Creamos rama y cambiamos de rama
<code>git branch</code>	Muestra ramas (activa con *)
<code>git merge rama</code>	Fusiona rama con actual
<code>git branch -d rama</code>	Elimina rama fusionada
<code>git branch -D rama</code>	Elimina rama no fusionada
<code>git remote add nombre url</code>	Crea un enlace nombre a repositorio url
<code>git remote</code>	Listado enlaces repositorios remotos
<code>git remote -v</code>	Listado enlaces repositorios url remotos
<code>git clone url</code>	Crea copia del repositorio de la url
<code>git reset --hard remoto/rama</code>	Descarga rama remota al repositorio local
<code>git push remoto rama</code>	Sube rama local al repositorio remoto

### **Práctica Guiada:**

**Abrir cmd y en el disco c: crear una carpeta con nuestro nombre. Identificarse mediante nombre:**

```
C:\mario
λ git config --global user.name "mario"
```

**Identificarse mediante email:**

```
C:\mario
λ git config --global user.email "mario@formaval.com"
```

**Consultar el nombre de Identificación:**

```
C:\mario
λ git config --global user.name
mario
```

**Consultar el email de Identificación:**

```
C:\mario
λ git config --global user.email
mario@formaval.com
```



**Configurar colores para la consola de salida:**

```
C:\mario
λ git config --global color.ui auto
```

**Listar la Configuración:**

```
C:\mario
λ git config --list
```

**Crear directorio de trabajo git llamado proyecto:**

```
C:\mario
λ git init proyecto
Initialized empty Git repository in C:/mario/proyecto/.git/
```

**Entrar en la carpeta proyecto:**

```
C:\mario
λ cd proyecto
```

**Listar los ficheros del directorio proyecto:**

```
C:\mario\proyecto (master)
λ ls -la
total 4
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ./
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ../
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 .git/
```

**Mostrar el estado del proyecto:**

```
C:\mario\proyecto (master)
λ git status
On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

**Crear un fichero llamado codigo.txt**

```
C:\mario\proyecto (master)
λ cat > codigo.txt
1
2
3
```

**Mostrar el estado del proyecto:**

```
C:\mario\proyecto (master)
λ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    codigo.txt
nothing added to commit but untracked files present (use "git add" to track)
```

**Añadir los ficheros del Directorio de trabajo al Área Preparada:**

```
C:\mario\proyecto (master)
λ git add .
warning: LF will be replaced by CRLF in codigo.txt.
The file will have its original line endings in your working directory
```

**Mostrar el estado del proyecto:**

```
C:\mario\proyecto (master)
λ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   codigo.txt
```

**Crear una versión del proyecto en el repositorio llamado "Desarrollo Código":**

```
C:\mario\proyecto (master)
λ git commit -m "Desarrollo Código"
[master (root-commit) 59eb39c] Desarrollo Código
 1 file changed, 3 insertions(+)
 create mode 100644 codigo.txt
```

**Mostrar el estado del proyecto:**

```
C:\mario\proyecto (master)
λ git status
On branch master
nothing to commit, working tree clean
```

**Editar el fichero código.txt quitando el 3 y añadiendo 4 y 5:**

```
C:\mario\proyecto (master)
λ nano codigo.txt
```

**Mostrar las diferencias entre el directorio de trabajo y el área preparada:**

```
C:\mario\proyecto (master)
λ git diff
warning: LF will be replaced by CRLF in codigo.txt.
The file will have its original line endings in your working directory
diff --git a/codigo.txt b/codigo.txt
index 01e79c3..bea819b 100644
--- a/codigo.txt
+++ b/codigo.txt
@@ -1,3 +1,5 @@
 1
 2
-3
+4
+5
+
```

**Mostrar las diferencias entre el directorio de trabajo y el último commit:**

```
C:\mario\proyecto (master)
λ git diff HEAD
warning: LF will be replaced by CRLF in codigo.txt.
The file will have its original line endings in your working directory
diff --git a/codigo.txt b/codigo.txt
index 01e79c3..bea819b 100644
--- a/codigo.txt
+++ b/codigo.txt
@@ -1,3 +1,5 @@
 1
 2
-3
+4
+5
+
```

**Mostrar las diferencias entre el área preparada y el último commit:**

```
C:\mario\proyecto (master)
λ git diff --cached
```

**Mostrar los commits realizados:**

```
C:\mario\proyecto (master)
λ git log
commit 59eb39c3158f56f06bcd5298bc860cc0430ad6a6 (HEAD -> master)
Author: mario <mario@formaval.com>
Date: Tue Jan 14 06:47:55 2020 +0100
    Desarrollo Código
```

**Añadir el fichero codigo.txt al área preparada:**

```
C:\mario\proyecto (master)
λ git add codigo.txt
warning: LF will be replaced by CRLF in codigo.txt.
The file will have its original line endings in your working directory
```

**Mostrar las diferencias entre el área preparada y el último commit:**

```
C:\mario\proyecto (master)
λ git diff --cached
diff --git a/codigo.txt b/codigo.txt
index 01e79c3..bea819b 100644
--- a/codigo.txt
+++ b/codigo.txt
@@ -1,3 +1,5 @@
 1
 2
-3
+4
+5
+
```

**Crear una versión del proyecto en el repositorio llamado "Desarrollo Código Modificado":**

```
C:\mario\proyecto (master)
λ git commit -m "Desarrollo Código Modificado"
[master a4c546a] Desarrollo Código Modificado
1 file changed, 3 insertions(+), 1 deletion(-)
```

**Mostrar los commits realizados con un commit por línea:**

```
C:\mario\proyecto (master)
λ git log --oneline
a4c546a (HEAD -> master) Desarrollo Código Modificado
59eb39c Desarrollo Código
```

**Mostrar los commits realizados en modo gráfico:**

```
C:\mario\proyecto (master)
λ git log --graph
* commit a4c546abe755d2b9ee804ed8ab03b5873be8dc0c (HEAD -> master)
| Author: mario <mario@formaval.com>
| Date: Tue Jan 14 06:53:27 2020 +0100
|
|     Desarrollo Código Modificado
|
* commit 59eb39c3158f56f06bcd5298bc860cc0430ad6a6
  Author: mario <mario@formaval.com>
  Date: Tue Jan 14 06:47:55 2020 +0100
      Desarrollo Código
```

**Cambiar el comentario identificador del último commit por "Desarrollo Código Alternativo"**

```
C:\mario\proyecto (master)
λ git commit --amend -m "Desarrollo Código Alternativo"
[master 0d67da9] Desarrollo Código Alternativo
Date: Tue Jan 14 06:53:27 2020 +0100
1 file changed, 3 insertions(+), 1 deletion(-)
```

**Crear el fichero nofich.txt con las siguientes líneas:**

```
C:\mario\proyecto (master)
λ cat > nofich.txt
a
b
c
```

**Mostrar los ficheros de la carpeta:**

```
C:\mario\proyecto (master)
λ ls -la
total 6
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:58 ./
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ../
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:56 .git/
-rw-r--r-- 1 Mario G 197121 9 ene. 14 06:48 codigo.txt
-rw-r--r-- 1 Mario G 197121 6 ene. 14 06:58 nofich.txt
```

**Añadir el fichero nofich.txt del directorio de trabajo al área preparada:**

```
C:\mario\proyecto (master)
λ git add nofich.txt
warning: LF will be replaced by CRLF in nofich.txt.
The file will have its original line endings in your working directory
```

**Mostrar el estado del proyecto:**

```
C:\mario\proyecto (master)
λ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   nofich.txt
```

**Eliminar el fichero nofich.txt del área preparada:**

```
C:\mario\proyecto (master)
λ git reset nofich.txt
```

**Mostrar el estado del proyecto:**

```
C:\mario\proyecto (master)
λ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    nofich.txt

nothing added to commit but untracked files present (use "git add" to track)
```

**Añadir el fichero nofich.txt del directorio de trabajo al área preparada:**

```
C:\mario\proyecto (master)
λ git add nofich.txt
warning: LF will be replaced by CRLF in nofich.txt.
The file will have its original line endings in your working directory
```

**Crear una versión del proyecto en el repositorio llamado "Nofich al repositorio":**

```
C:\mario\proyecto (master)
λ git commit -m "Nofich al repositorio"
[master cc91eec] Nofich al repositorio
 1 file changed, 3 insertions(+)
 create mode 100644 nofich.txt
```

**Mostrar los commits realizados:**

```
C:\mario\proyecto (master)
λ git log
commit cc91eec3cded8a559f9ce20d19f9e72086c678dc (HEAD -> master)
Author: mario <mario@formaval.com>
Date:   Tue Jan 14 07:01:16 2020 +0100
    Nofich al repositorio
commit 0d67da96404f8cba3f967e768be45395345f0030
```

```
Author: mario <mario@formaval.com>
Date: Tue Jan 14 06:53:27 2020 +0100
    Desarrollo Codigo Alternativo
```

```
commit 59eb39c3158f56f06bcd5298bc860cc0430ad6a6
Author: mario <mario@formaval.com>
Date: Tue Jan 14 06:47:55 2020 +0100
    Desarrollo Codigo
```

**Modificar el fichero nofich.txt quitando las letras b y c:**

```
C:\mario\proyecto (master)
λ nano nofich.txt
```

**Volver el fichero nofich.txt a la versión del último commit:**

```
C:\mario\proyecto (master)
λ git checkout HEAD -- nofich.txt
```

**Mostrar el fichero nofich.txt y comprobar que vuelven a estar las líneas b y c:**

```
C:\mario\proyecto (master)
λ nano nofich.txt
```

**Muestra Información y diferencias del directorio de trabajo respecto al commit llamado "Desarrollo Código" y con hash 59eb39c3158f56f06bcd5298bc860cc0430ad6a6:**

```
C:\mario\proyecto (master)
λ git show 59eb39c3158f56f06bcd5298bc860cc0430ad6a6
commit 59eb39c3158f56f06bcd5298bc860cc0430ad6a6
Author: mario <mario@formaval.com>
Date: Tue Jan 14 06:47:55 2020 +0100
    Desarrollo Codigo
diff --git a/codigo.txt b/codigo.txt
new file mode 100644
index 0000000..01e79c3
--- /dev/null
+++ b/codigo.txt
@@ -0,0 +1,3 @@
+1
+2
+3
```

**Muestra Información y diferencias del directorio de trabajo respecto al último commit:**

```
C:\mario\proyecto (master)
λ git show
commit cc91eec3cded8a559f9ce20d19f9e72086c678dc (HEAD -> master)
Author: mario <mario@formaval.com>
Date: Tue Jan 14 07:01:16 2020 +0100
    Nofich al repositorio
diff --git a/nofich.txt b/nofich.txt
new file mode 100644
index 0000000..de98044
--- /dev/null
```

```
+++ b/nofich.txt
@@ -0,0 +1,3 @@
+a
+b
+c
```

*Crear una rama llamada basedatos:*

```
C:\mario\proyecto (master)
λ git branch basedatos
```

*Mostrar ramas. Se muestra \* en la rama activa:*

```
C:\mario\proyecto (master)
λ git branch
  basedatos
* master
```

*Cambiar a la rama basedatos:*

```
C:\mario\proyecto (master)
λ git checkout basedatos
Switched to branch 'basedatos'
```

*Crear el fichero datos.txt*

```
C:\mario\proyecto (basedatos)
λ cat > datos.txt
d1
d2
d3
```

*Añadir los ficheros del Directorio de trabajo al Área Preparada:*

```
C:\mario\proyecto (basedatos)
λ git add .
warning: LF will be replaced by CRLF in datos.txt.
The file will have its original line endings in your working directory
```

*Crear una versión en el repositorio llamada "Añade los datos":*

```
C:\mario\proyecto (basedatos)
λ git commit -m "Añade los datos"
[basedatos 918dd5f] Añade los datos
1 file changed, 3 insertions(+)
create mode 100644 datos.txt
```

*Muestra los ficheros de la carpeta:*

```
C:\mario\proyecto (basedatos)
λ ls -la
total 7
drwxr-xr-x 1 Mario G 197121 0 ene. 14 07:07 ./
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ../
drwxr-xr-x 1 Mario G 197121 0 ene. 14 07:08 .git/
-rw-r--r-- 1 Mario G 197121 9 ene. 14 06:48 codigo.txt
-rw-r--r-- 1 Mario G 197121 9 ene. 14 07:07 datos.txt
```

```
-rw-r--r-- 1 Mario G 197121 9 ene. 14 07:02 nofich.txt
```

**Cambiar a la rama master:**

```
C:\mario\proyecto (basedatos)
λ git checkout master
Switched to branch 'master'
```

**Mostrar los ficheros de la carpeta. Comprobar que no aparece el fichero datos.txt:**

```
C:\mario\proyecto (master)
λ ls -la
total 6
drwxr-xr-x 1 Mario G 197121 0 ene. 14 07:08 ./
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ../
drwxr-xr-x 1 Mario G 197121 0 ene. 14 07:08 .git/
-rw-r--r-- 1 Mario G 197121 9 ene. 14 06:48 codigo.txt
-rw-r--r-- 1 Mario G 197121 9 ene. 14 07:02 nofich.txt
```

**Mostrar las ramas:**

```
C:\mario\proyecto (master)
λ git branch
  basedatos
* master
```

**Unir las rama basedatos con master:**

```
C:\mario\proyecto (master)
λ git merge basedatos
Updating cc91eec..918dd5f
Fast-forward
 datos.txt | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 datos.txt
```

**Mostrar las ramas:**

```
C:\mario\proyecto (master)
λ git branch
  basedatos
* master
```

**Mostrar los ficheros de la carpeta:**

```
C:\mario\proyecto (master)
λ ls -la
total 7
drwxr-xr-x 1 Mario G 197121 0 ene. 14 07:10 ./
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ../
drwxr-xr-x 1 Mario G 197121 0 ene. 14 07:10 .git/
-rw-r--r-- 1 Mario G 197121 9 ene. 14 06:48 codigo.txt
-rw-r--r-- 1 Mario G 197121 12 ene. 14 07:10 datos.txt
-rw-r--r-- 1 Mario G 197121 9 ene. 14 07:02 nofich.txt
```

**Eliminar la rama basedatos:**

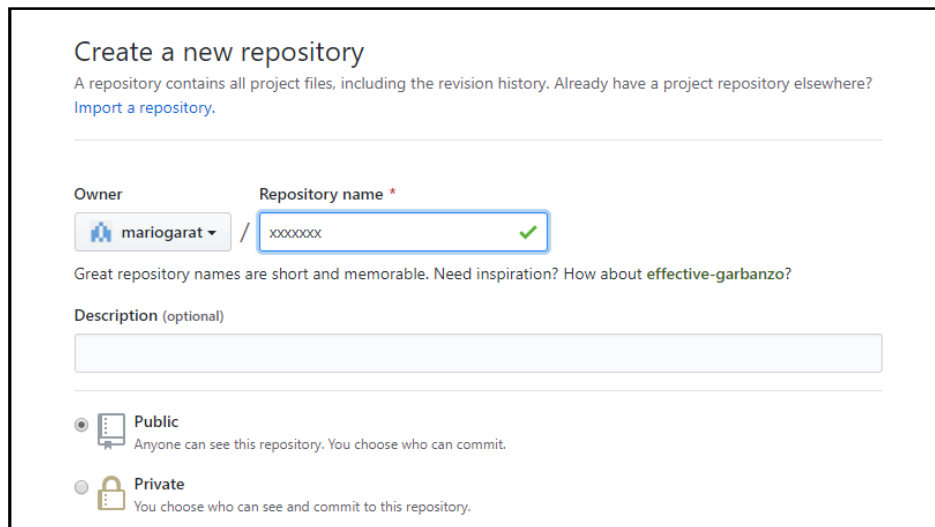


```
C:\mario\proyecto (master)
λ git branch -d basedatos
Deleted branch basedatos (was 918dd5f).
```

**Mostrar las ramas:**

```
C:\mario\proyecto (master)
λ git branch
* master
```

**Creamos una cuenta gratuita GitHub en <https://github.com/>. A continuación creamos un repositorio remoto en la web de GitHub (cuenta: mariogarat / repositorio: proyecto)**



**Crear enlace llamado accesoProyecto al repositorio remoto <https://github.com/mariogarat/proyecto>:**

```
C:\mario\proyecto (master)
λ git remote add accesoProyecto https://github.com/mariogarat/proyecto
```

**Mostrar los enlaces a repositorios remotos:**

```
C:\mario\proyecto (master)
λ git remote
accesoProyecto
```

**Mostrar los enlaces a repositorios remotos y sus url's:**

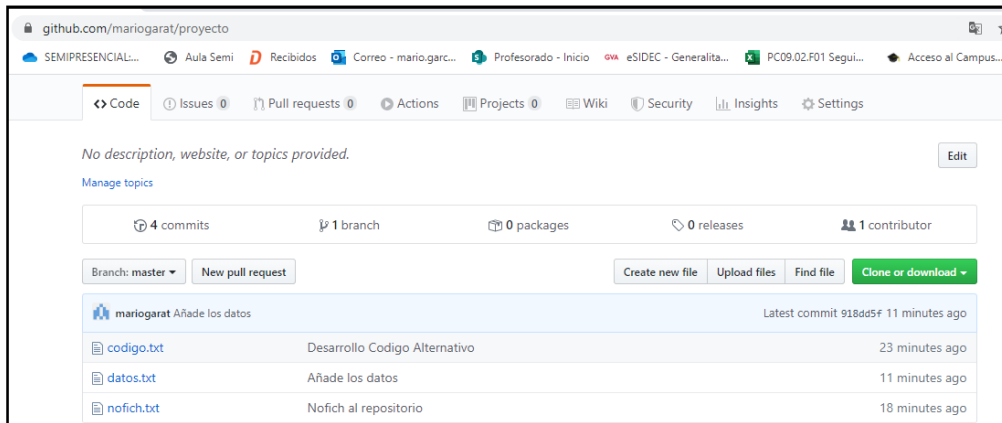
```
C:\mario\proyecto (master)
λ git remote -v
accesoProyecto https://github.com/mariogarat/proyecto (fetch)
accesoProyecto https://github.com/mariogarat/proyecto (push)
```

**Subir los ficheros de la rama local master al repositorio remoto con enlace accesoProyecto:**

```
C:\mario\proyecto (master)
λ git push accesoProyecto master
Username for 'https://github.com': mario@formaval.com
Password for 'https://mario@formaval.com@github.com': xxxxxxxxxxxxxxxxxxxx
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 8 threads
```

```
Compressing objects: 100% (6/6), done.  
Writing objects: 100% (12/12), 981 bytes | 490.00 KiB/s, done.  
Total 12 (delta 0), reused 0 (delta 0)  
To https://github.com/mariogarat/proyecto  
* [new branch]      master -> master
```

**Comprobar que el repositorio remoto contiene los ficheros del directorio de trabajo local:**



**Eliminar el fichero nofich.txt**

```
C:\mario\proyecto (master)  
λ git rm nofich.txt  
rm 'nofich.txt'
```

**Eliminar el fichero datos.txt**

```
C:\mario\proyecto (master)  
λ git rm datos.txt  
rm 'datos.txt'
```

**Mostrar los ficheros de la carpeta:**

```
C:\mario\proyecto (master)  
λ ls -la  
total 5  
drwxr-xr-x 1 Mario G 197121 0 ene. 14 07:22 ./  
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ../  
drwxr-xr-x 1 Mario G 197121 0 ene. 14 07:22 .git/  
-rw-r--r-- 1 Mario G 197121 9 ene. 14 06:48 codigo.txt
```

**Mostrar los enlaces a repositorios remotos:**

```
C:\mario\proyecto (master)  
λ git remote  
accesoProyecto
```

**Mostrar el estado del proyecto:**

```
C:\mario\proyecto (master)  
λ git status  
On branch master  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstage)
```

```
deleted:    datos.txt
deleted:    nofi ch.txt
```

**Añadir los cambios del directorio de trabajo al área preparada:**

```
C:\mario\proyecto (master)
λ git add .
```

**Crear una versión llamada "Borrado nofich y datos" en el repositorio local:**

```
C:\mario\proyecto (master)
λ git commit -m "Borrado nofich y datos"
[master 5df381e] Borrado nofich y datos
 2 files changed, 6 deletions(-)
 delete mode 100644 datos.txt
 delete mode 100644 nofi ch.txt
```

**Mostrar los ficheros de la carpeta:**

```
C:\mario\proyecto (master)
λ ls -la
total 5
drwxr-xr-x 1 Mario G 197121 0 ene. 15 07:02 ./
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ../
drwxr-xr-x 1 Mario G 197121 0 ene. 15 07:17 .git/
-rw-r--r-- 1 Mario G 197121 9 ene. 14 06:48 codigo.txt
```

**Volver el proyecto a la versión del repositorio remoto, con lo que se recuperan los ficheros nofich.txt y datos.txt:**

```
C:\mario\proyecto (master)
λ git reset --hard accesoProyecto/master
HEAD is now at 351d9e9 Vuel ta
```

**Mostrar los ficheros de la carpeta:**

```
C:\mario\proyecto (master)
λ ls -la
total 7
drwxr-xr-x 1 Mario G 197121 0 ene. 15 07:25 ./
drwxr-xr-x 1 Mario G 197121 0 ene. 14 06:44 ../
drwxr-xr-x 1 Mario G 197121 0 ene. 15 07:25 .git/
-rw-r--r-- 1 Mario G 197121 9 ene. 14 06:48 codigo.txt
-rw-r--r-- 1 Mario G 197121 14 ene. 15 07:25 datos.txt
-rw-r--r-- 1 Mario G 197121 11 ene. 15 07:25 nofi ch.txt
```

**Mostrar el estado del proyecto:**

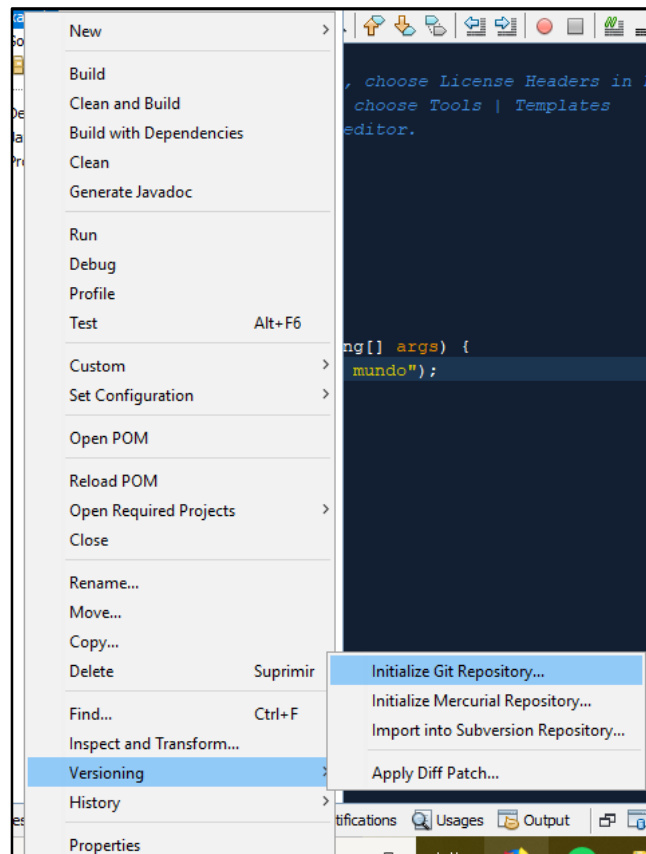
```
C:\mario\proyecto (master)
λ git status
On branch master
nothing to commit, working tree clean
```

## 6.- Git en Netbeans.

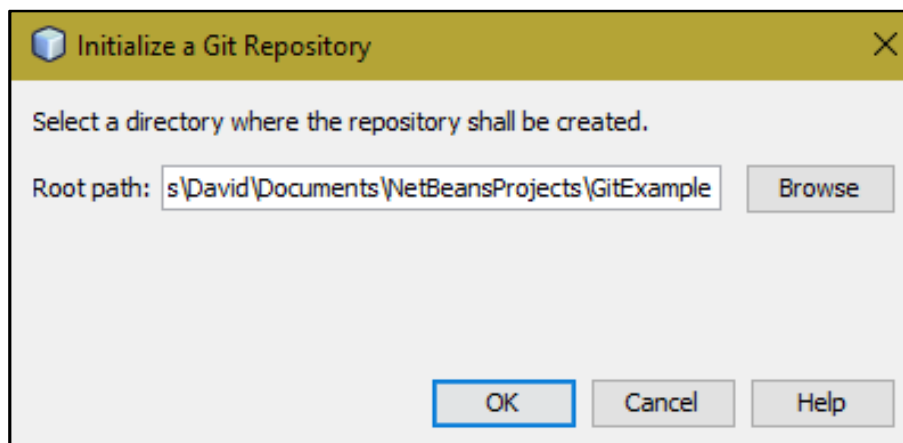
### a) Inicializar Repositorio GIT

El primer paso para hacer uso del sistema de control de versiones Git en NetBeans es inicializar Git, esto se logra vía el menú Versioning al cual accede presionando el botón derecho del ratón en el proyecto.

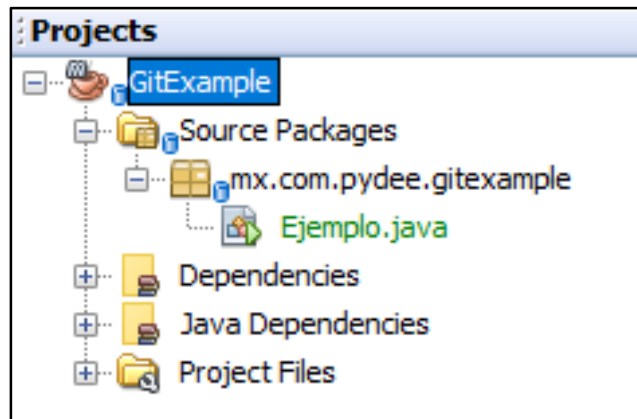
Podemos inicializar un repositorio Git para nuestro proyecto, de ese modo todos los cambios que hagamos podrán guardarse en una forma ordenada y ocupando menos espacio que si hiciéramos una copia del proyecto a cada cambio, esto requiere que indiquemos donde deseamos se cree la carpeta del repositorio, como se ve a continuación.



En general la ubicación por defecto es la ideal, así que recomiendo use esa, que es en la misma carpeta que su proyecto.



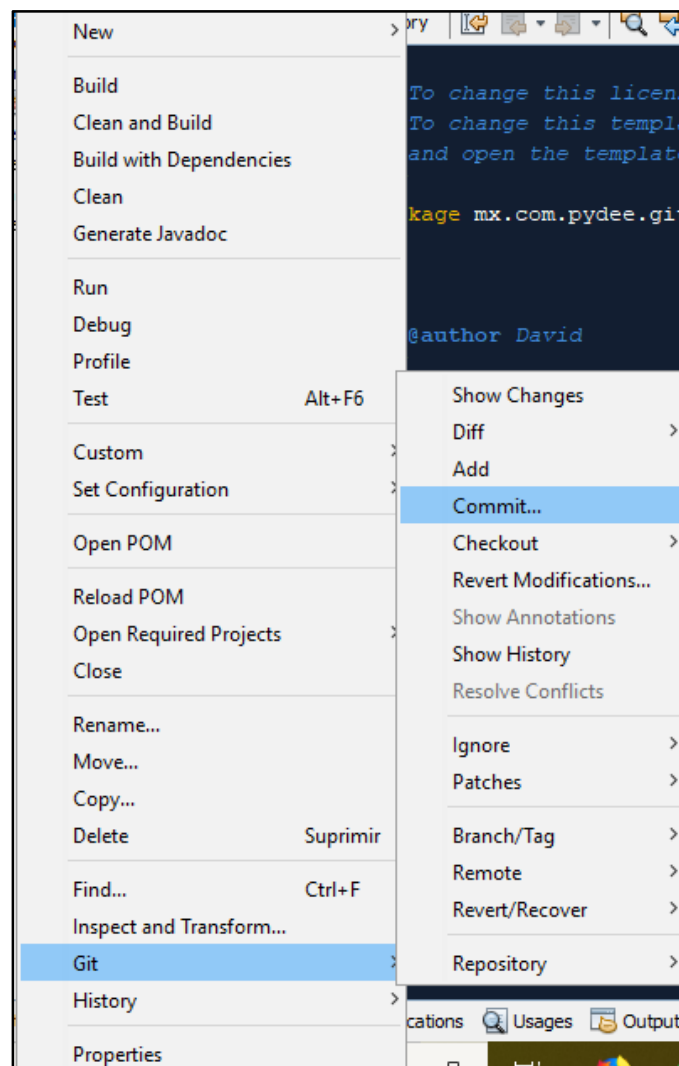
Ahora que ha inicializado el repositorio Git notara que los nombres de los archivos en la barra de proyectos han cambiado de color, pasando a tener texto en color verde.



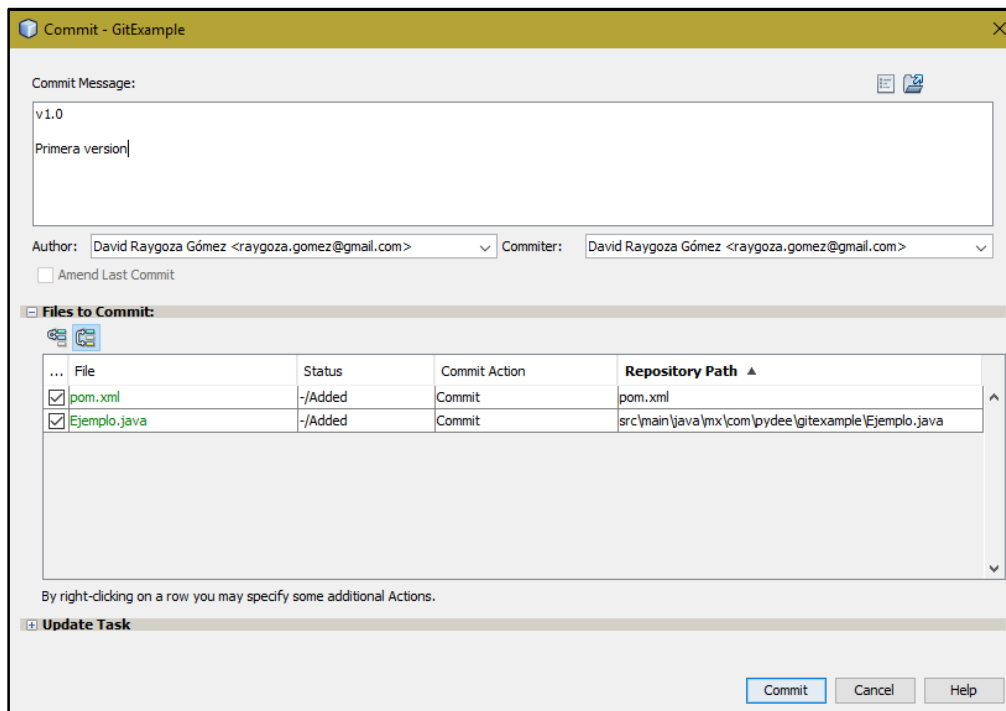
Esta es la forma en que la interfaz de NetBeans le indica que esos archivos se han creado desde la última versión guardada.

#### b) Almacenando Cambios

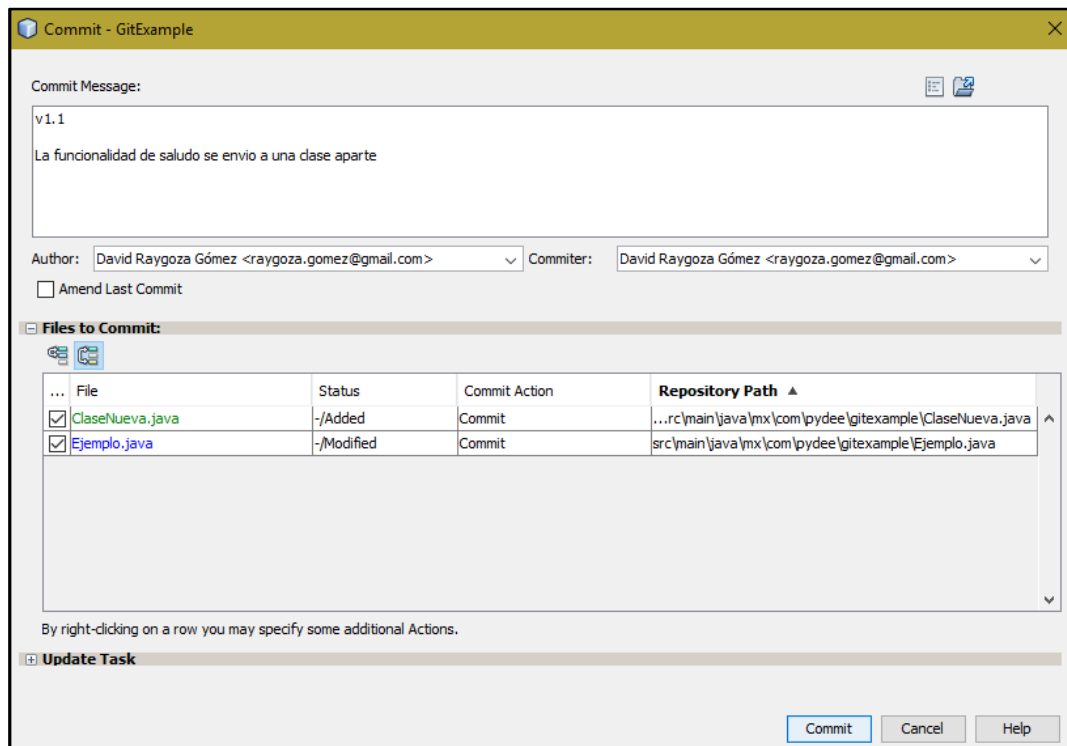
Para crear una nueva versión o Commit basta con ir al menú Git - Commit



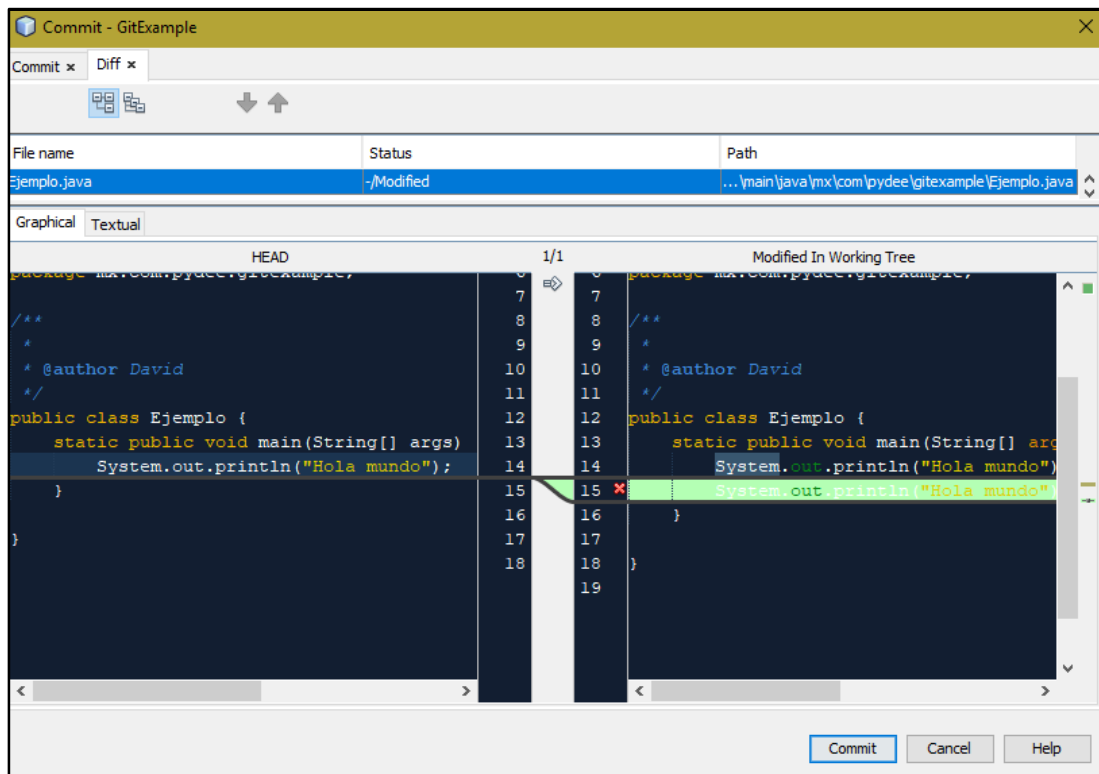
Mostrará una ventana para crear la nueva versión, en la que aparecen los archivos que contienen los cambios desde la última versión, ya sea porque se crearon recientemente, se modificaron o se eliminaron, esto se denota con diferentes colores de texto.



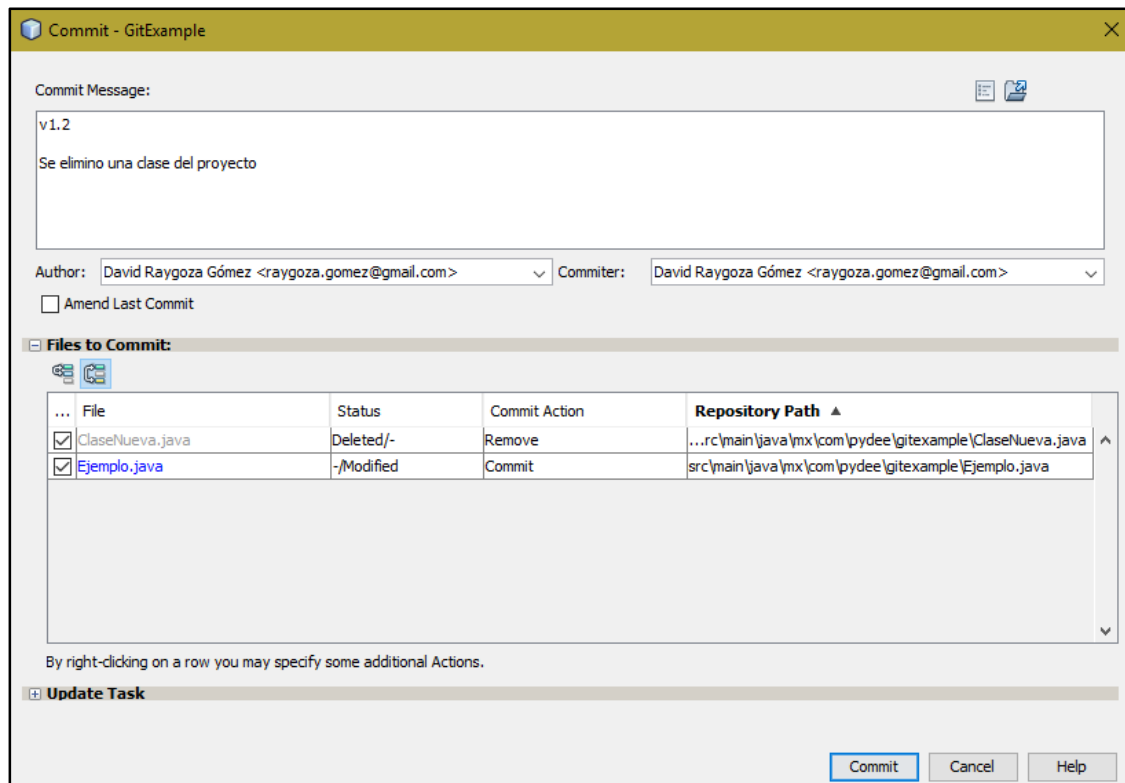
Por ejemplo, si en una nueva versión aparte de crear archivos hubiera modificado archivos se mostrarán con color azul. La columna Status también le indica que acción se realizó sobre los archivos.



Pulsando doble clic sobre el nombre de un archivo podemos comprobar que cambios se realizaron exactamente en el archivo, mostrando una ventana donde se puede comparar la versión del último Commit con el código actual.



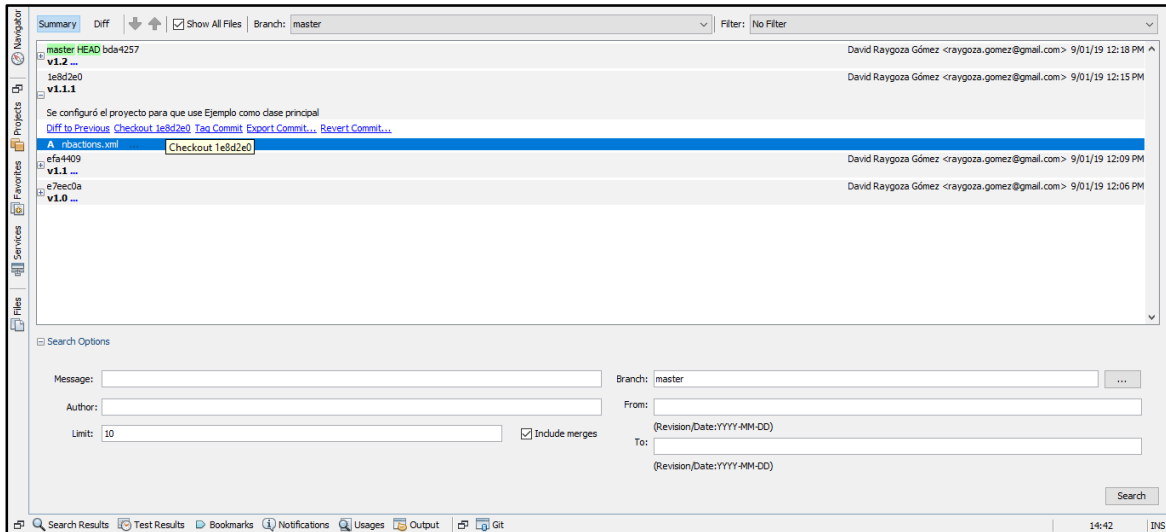
En el caso de que se haya eliminado archivos de su proyecto, estos se mostrarán con texto gris y Deleted en la columna de Status.



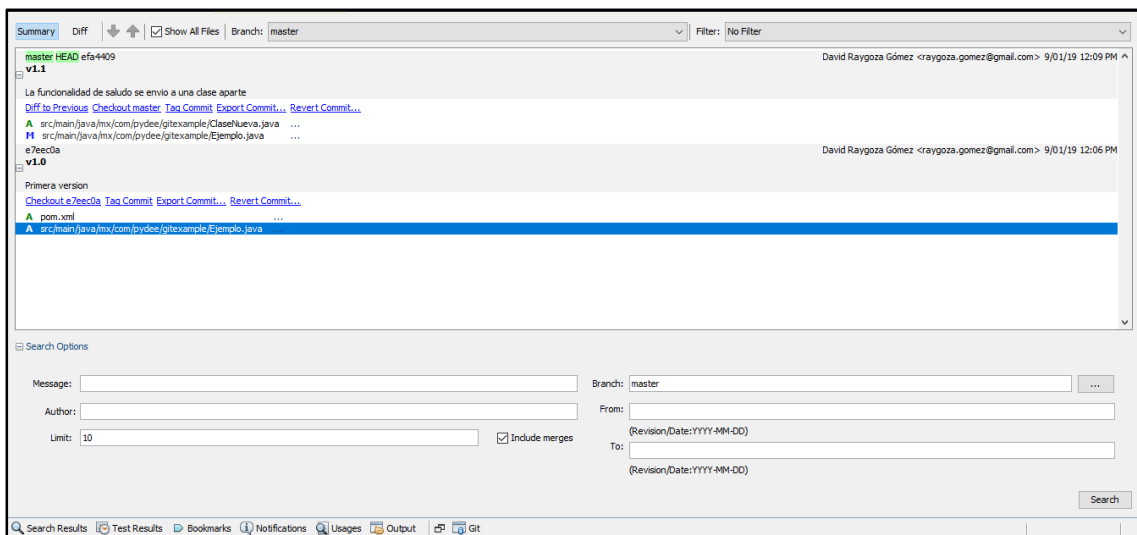
Sólo se pueden crear versiones nuevas si hacen los cambios desde la última versión disponible. Si nos encontramos en una versión anterior debemos regresar a la versión más reciente para crear una nueva versión.

### c) Acceder a versiones anteriores

Si disponemos de diferentes versiones del proyecto, para acceder a las versiones anteriores debemos acceder al menú Git - Show History.



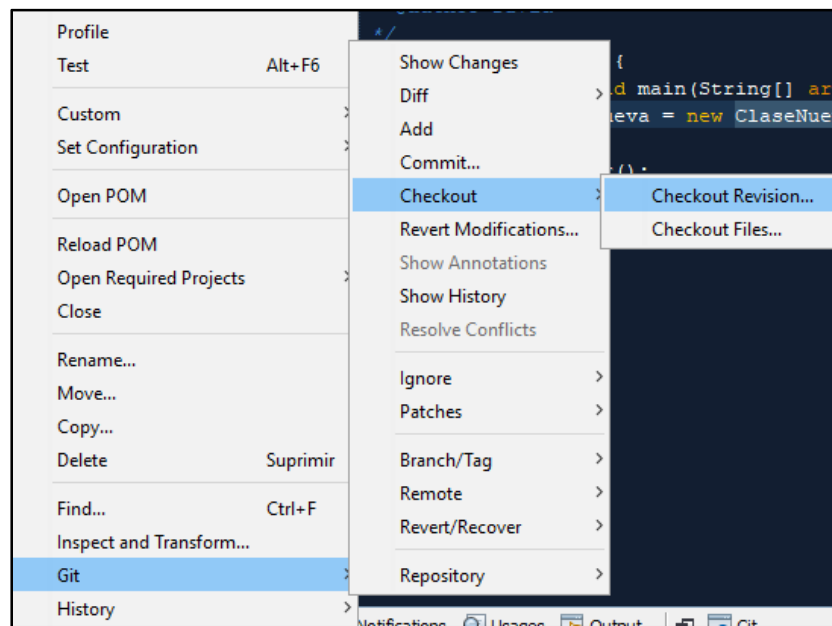
Pulsamos sobre Search para que nos muestre las ultimas 10 versiones. Desde la ventana podemos regresar a versiones anteriores del proyecto pulsando el enlace Checkout. Esto automáticamente cambiara los archivos del proyecto a como se encontraban en esa versión.



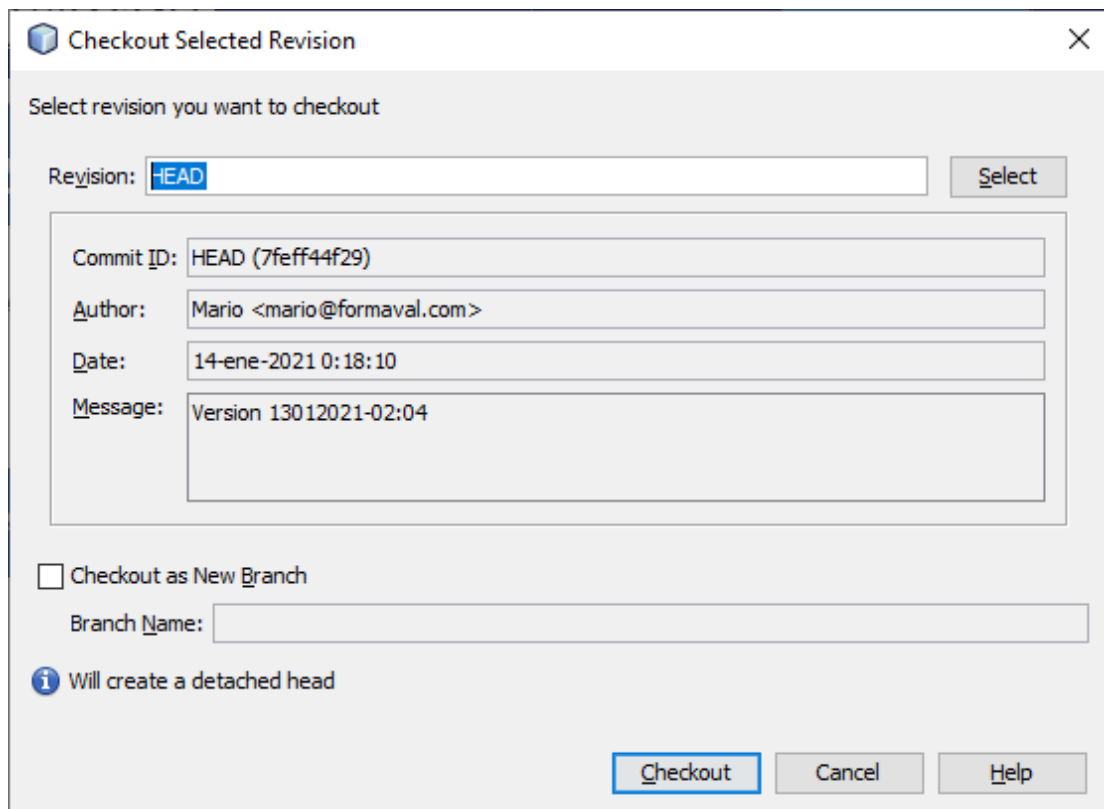
Para volver a una versión más reciente volveremos con Git – Show History a mostrar las versiones del proyecto. Desplegaremos la versión a la que queremos volver y pulsaremos el enlace Checkout.

Si intentamos volver a una versión más reciente, pero no aparecen versiones más recientes a la actual pulsaremos Git – Checkout - Checkout Revision.





Se muestra una ventana con la versión actual y pulsamos Checkout.



A continuación, mostrará una ventana para seleccionar revisiones, mostrando todas las versiones del proyecto. Seleccionamos la versión que queremos revisar y pulsamos Select para ir a esa versión.

