

UT4

Optimización y  
Documentación

# 1. INTRODUCCIÓN

- ◉ El objetivo de esta unidad de trabajo es manejar tres tipos de herramientas básicas para cualquier programador:
  - **Herramientas de control de versiones:** mantiene un historial de todos los cambios en un proyecto y quién los ha realizado. Muy importante en equipos de software.
  - **Herramientas para documentar los programas:** generan automáticamente documentación que informa de lo que hace cada clase y cada método.
  - **Herramientas de refactorización:** permiten simplificar el código de los programas, favoreciendo su lectura, entendimiento y fácil mantenimiento.

## 2. DOCUMENTACIÓN

- ◉ Documentación: texto escrito que acompaña a los proyectos de software.
- ◉ La documentación es un requisito importante en un proyecto comercial, el cliente siempre va a solicitar que se documenten las distintas fases del proyecto.
- ◉ Tipos de documentación
  - Documentación de las especificaciones.
  - Documentación del diseño: diagramas de flujo, diagramas de clases, diagramas E/R, etc.
  - Documentación del código fuente.
  - Documentación de usuario final: documentación que se entrega a los usuarios. Se explica cómo utilizar las aplicaciones del proyecto.

## 2.1 DOCUMENTACIÓN DEL CÓDIGO FUENTE

- ◉ Un programa bien documentado es mucho más fácil de depurar y añadirle nuevas funcionalidades.
- ◉ Generalmente los programas serán modificados en el futuro, bien por el autor del programa o por otro programador del equipo.
- ◉ Por esta razón, es necesario tener bien documentado el código, para poder hacer modificaciones de forma sencilla.

## 2.1 DOCUMENTACIÓN DEL CÓDIGO FUENTE

- ◉ En Java se documenta el código fuente de dos formas:
  - Comentarios "internos" con los caracteres `//` seguidos del comentario, o `/* .... */` con el comentario en el lugar de los puntos suspensivos.
  - Para explicar qué hace una pieza de código en Java se utilizan los **comentarios JavaDoc**, y se escriben comenzando por `/**` y terminando con `*/`

## 2.1.1 ¿POR QUÉ ES ÚTIL JAVADOC?

- ◉ El mayor problema a la hora de documentar el código es el mantenimiento de esa documentación.
- ◉ Si la documentación y el código están separados, cambiar la documentación cada vez que se cambia el código se convierte en un problema.
- ◉ Solución: unir el código a la documentación, es decir, poner todo en el mismo archivo.
- ◉ Para ello, hay que usar:
  - Sintaxis propia de JavaDoc
  - Herramienta para extraer los comentarios y exportarlos

## 2.1.2 QUÉ ES JAVADOC

- ◉ La herramienta para extraer los comentarios se denomina *javadoc*.
- ◉ Javadoc es una herramienta para generar documentación de código Java a partir de comentarios (escritos según ciertas reglas) que se intercalan en el código fuente.
- ◉ La documentación se genera en formato HTML
- ◉ Un ejemplo de documentación generada con javadoc es la especificación de la API de Java.

## 2.1.2.1 SINTAXIS DE JAVADOC

- ◉ Hay tres "tipos" de documentación Javadoc:
  - Una clase
  - Una variable
  - Un método.
- ◉ Todas aparecen **justo antes** de la documentación de la clase, variable o método
- ◉ Un error muy habitual es el siguiente:

```
/**  
 * Este es el comentario para la documentación para la clase Whatever.  
 */  
import com.sun;  
public class Whatever  
{ ...
```

Javadoc ignora el comentario y no genera documentación para la clase



## 2.1.2.1 SINTAXIS DE JAVADOC

- ◉ Un comentario javadoc está compuesto por:
  - **Una descripción principal:** empieza después del marcador de principio de comentario `/**` y sigue hasta la sección de etiquetas. No puede continuar una vez iniciada la sección de etiquetas.

```
/**
 * Al usar este estilo de comentario estoy consiguiendo que cuando
 * lo pase por javadoc, esta herramienta copie este texto a una página
 * html.
 * Puedes ver que también admite <b>etiquetas HTML</b>
 * y ésto último que acabo de escribir aparecerá en negrita.
 */
```

- **Una sección de etiquetas (tags):** el principio de la sección de etiquetas está marcado por el primer carácter `@`.

## 2.1.2.2 ETIQUETAS JAVADOC

- ◉ Vamos a ver las etiquetas Javadoc más utilizadas:
  - **@author** *nombre* (desde la versión 1.0 del JDK/SDK)
    - Indica el autor de la clase en el argumento *nombre*.
    - Podemos usar varias etiquetas autor o bien podemos ponerlos a todos en una sola etiqueta. En este último caso, Javadoc inserta una (,) y un espacio entre los diferentes nombres.

## 2.1.2.2 ETIQUETAS JAVADOC

- **@deprecated** *comentario* (desde la versión 1.0 del JDK / SDK)

Indicando que ese método/clase no debería volver a usarse por estar "desfasado", aunque siga perteneciendo al SDK.

Es sólo una advertencia que damos a los usuarios de nuestra API, igual que las distribuciones de Sun hacen con nosotros. Suele ser un paso previo a la desaparición del método/clase.

## 2.1.2.2 ETIQUETAS JAVADOC

- **@see** *referencia* (desde la versión 1.0 del JDK/SDK)

Añade un cabecero "See Also" con un enlace o texto que apunta a una referencia. Puede aparecer varias etiquetas de este tipo en el mismo cabecero.

Se puede crear de tres maneras diferentes:

- **@see String:** no se genera ningún enlace. Ejemplo:  
`@see "The Java Programming Language"`
- **@see <a href="URL#value">label</a>:** Añade un enlace definido por URL#value. Ejemplo:  
`@see <a href="spec.html#section">Java Spec</a>`
- **@see package.class#member texto:** Añade un enlace, con el texto visible *texto*, que apunta a la documentación de la entidad especificada: paquete, clase, método o un campo (*texto* es opcional, para representarlo por un nombre corto)

## 2.1.2.2 ETIQUETAS JAVADOC

### ◉ Ejemplos

```
See also:
@see java.lang.String           // String
@see java.lang.String The String class // The String class
@see String                     // String

@see Character#MAX_RADIX        // Character.MAX_RADIX
@see <a href="spec.html">Java Spec</a> // Java Spec
@see "The Java Programming Language" // "The Java Programming Language"
```

## 2.1.2.2 ETIQUETAS JAVADOC

- `{@link nombre etiqueta}` (versión  $\geq 1.2$  del JDK/SDK)

Misma sintaxis que la etiqueta `@see`, pero genera un enlace autocontenido en vez de colocar el enlace en la sección "See Also".

Ejemplo: `@deprecated Desde JDK1.1 Usar el método {@link #getComponentAt(int,int) getComponentAt}`

Resultado: Usar el método `getComponentAt`

## 2.1.2.2 ETIQUETAS JAVADOC

- **@since texto** (desde la versión 1.1 del JDK/SDK)

Indica cuándo se creó este paquete, clase o método. Normalmente se pone la versión de nuestra API en que se incluyó, así en posteriores versiones sabremos a qué revisión pertenece o en qué revisión se añadió.

Ejemplo:

*@since JDK1.1*

## 2.1.2.2 ETIQUETAS JAVADOC

- **@version** *version* (desde la versión 1.0 del JDK/SDK)

Añade un cabecero a la documentación generada con la versión de esta clase. Por versión, normalmente nos referimos a la versión del software que contiene esta clase o miembro.

- **@throws** *nombre-clase descripcion* (desde la versión 1.2 del JDK/SDK)
- **@exception** *nombre-clase descripción* (desde la versión 1.0 del JDK/SDK)

En ambos casos, se añade una cabecera "Throws" a la documentación generada con el nombre de la excepción que puede ser lanzada y una descripción de por qué se lanza.



## 2.1.2.2 ETIQUETAS JAVADOC

- **@param** *parámetro descripción* (versión 1.0 del JDK/SDK)

Añade un parámetro y su descripción a la sección "Parameters" de la documentación. Para cada método emplearemos tantas etiquetas de este estilo como parámetros de entrada tenga dicho método.

- **@return** *descripción* (versión 1.0 del JDK/SDK)

Añade a la sección "Returns" de la documentación HTML que va a generar la descripción del tipo que devuelve el método.

## 2.1.2.2 ETIQUETAS JAVADOC

- ◉ Ejemplo: Si suponemos que tenemos el método *public String concatena(String s1, String s2, String s3)* deberíamos comentar nuestro código fuente de la siguiente manera:

```
/*  
 * ....  
 * @param s1 Texto que ocupa la cabecera  
 * @param s2 Texto que ocupa el cuerpo de la string a crear  
 * @param s3 Texto que ocupa la parte final  
 * @return La cadena conformada de tipo String.  
 */
```

## 2.1.2.2.1.1 COMENTARIOS DE CLASES E INTERFACES

- ◉ Disponemos de las siguientes etiquetas: @see, {@link}, @since, @deprecated, @author, @version.
- ◉ Veámos un ejemplo del comentario de una clase:

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *     Window win = new Window(parent);
 *     win.show();
 * </pre>
 *
 * @author Sami Shaio
 * @version %I%, %G%
 * @see java.awt.BaseWindow
 * @see java.awt.Button
 */
class Window extends BaseWindow {
    ...
}
```

## 2.1.2.2.1.2 COMENTARIO DE VARIABLES

- ◉ Las etiquetas que podemos utilizar en esta ocasión son: `@see`, `{@link}`, `@since`, `@deprecated`, `@serial`, `@serialField`.
- ◉ Un ejemplo:

```
/**  
 * The X-coordinate of the component.  
 *  
 * @see #getLocation()  
 */  
int x = 1263732;
```

## 2.1.2.2.1.2 COMENTARIO DE MÉTODOS Y CONSTRUCTORES

- ◉ En este caso disponemos de: `@see`, `{@link}`, `@since`, `@deprecated`, `@param`, `@return`, `@throws` (`@exception`), `@serialData`.
- ◉ Un ejemplo:

```
/**
 * Returns the character at the specified index. An index
 * ranges from <code>0</code> to <code>length() - 1</code>.
 *
 * @param    index    the index of the desired character.
 * @return   the desired character.
 * @exception StringIndexOutOfBoundsException if the index is not in the range
 * to <code>length()-1</code>.
 * @see      java.lang.Character#charValue()
 */
public char charAt(int index) {
    ...
}
```

# EJEMPLO

```
/**
 * <h2>Clase Empleado, se utiliza para crear y leer empleados de una BD </h2>
 * Busca información de javadoc en <a href="http://google.com">GOOGLE</a>
 *
 * @see <a href="http://www.google.com">Google</a>
 * @author Ana
 * @version 1.0
 * @since 01-01-2019
 */
public class Empleado {

    //Atributos

    /**
     * Nombre del empleado
     */
    private String nombre;

    /**
     * Apellido del empleado
     */
    private String apellido;

    /**
     * Edad del empleado
     */
    private int edad;

    /**
     * Salario del empleado
     */
    private double salario;
```

# EJEMPLO

```
//Constructores
/**
 * Constructor por defecto
 */
public Empleado() {
    this ("", "", 0, 0);
}

/**
 * Constructor con 4 parametros
 * @param nombre nombre del empleado
 * @param apellido nombre del empleado
 * @param edad edad del empleado
 * @param salario salario del empleado
 */
public Empleado(String nombre, String apellido, int edad, double salario){
    this.nombre=nombre;
    this.apellido=apellido;
    this.edad=edad;
    this.salario=salario;
}
```

# EJEMPLO

```
//Metodos publicos

/**
 * Suma un plus al salario del empleado si el empleado tiene mas de 40 años
 * @param sueldoPlus
 * @return <ul>
 *     <li>true: se suma el plus al sueldo</li>
 *     <li>false: no se suma el plus al sueldo</li>
 * </ul>
 */
public boolean plus (double sueldoPlus){
    boolean aumento=false;
    if (edad>40 && compruebaNombre()){
        salario+=sueldoPlus;
        aumento=true;
    }
    return aumento;
}

//Metodos privados
/**
 * Comprueba que el nombre no este vacio
 * @return <ul>
 *     <li>true: el nombre es una cadena vacia</li>
 *     <li>false: el nombre no es una cadena vacia</li>
 * </ul>
 */
private boolean compruebaNombre() {
    if(nombre.equals("")) {
        return false;
    }
    return true;
}
}
```

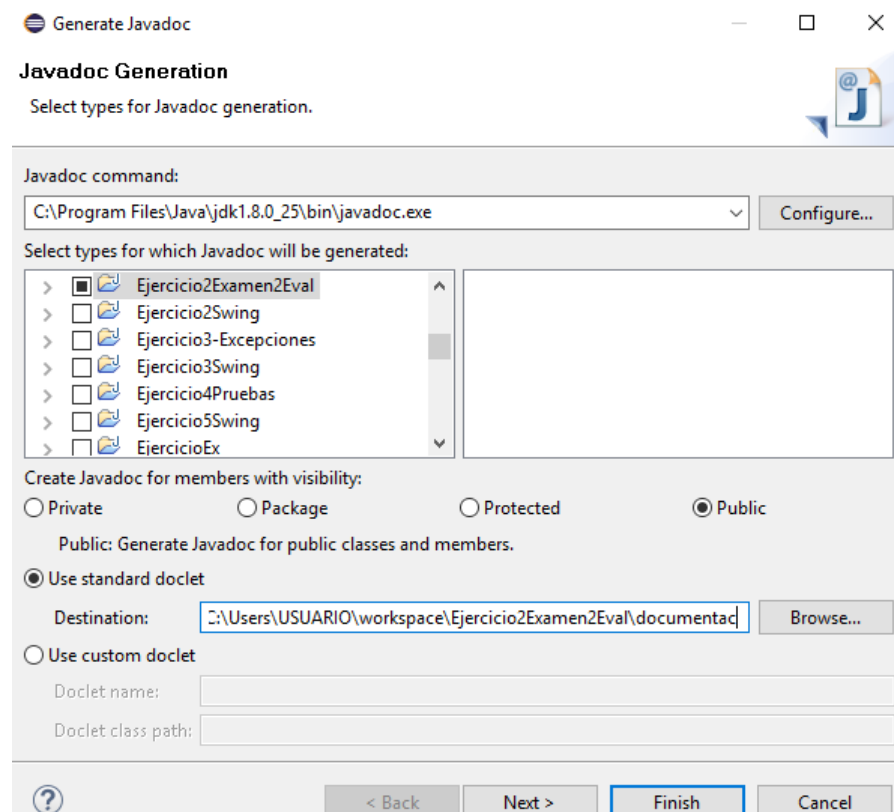


## 2.1.3 GENERAR LA DOCUMENTACIÓN

- ◉ Para ejecutar Javadoc desde Eclipse, se abre el menú Project y se elige *Generate Javadoc*.
- ◉ En la ventana que se muestra pedirá la siguiente configuración:
  - **Javadoc command:** se tiene que indicar dónde se encuentra el archivo ejecutable de Javadoc, el javadoc.exe (está en la carpeta donde se ha instalado el jdk, dentro de bin).
  - En los dos cuadros inferiores se elegirá el proyecto y las clases a documentar.
  - Se selecciona la visibilidad de los elementos que se van a documentar. Si se selecciona Private se documentarán todos los miembros públicos y protegidos.

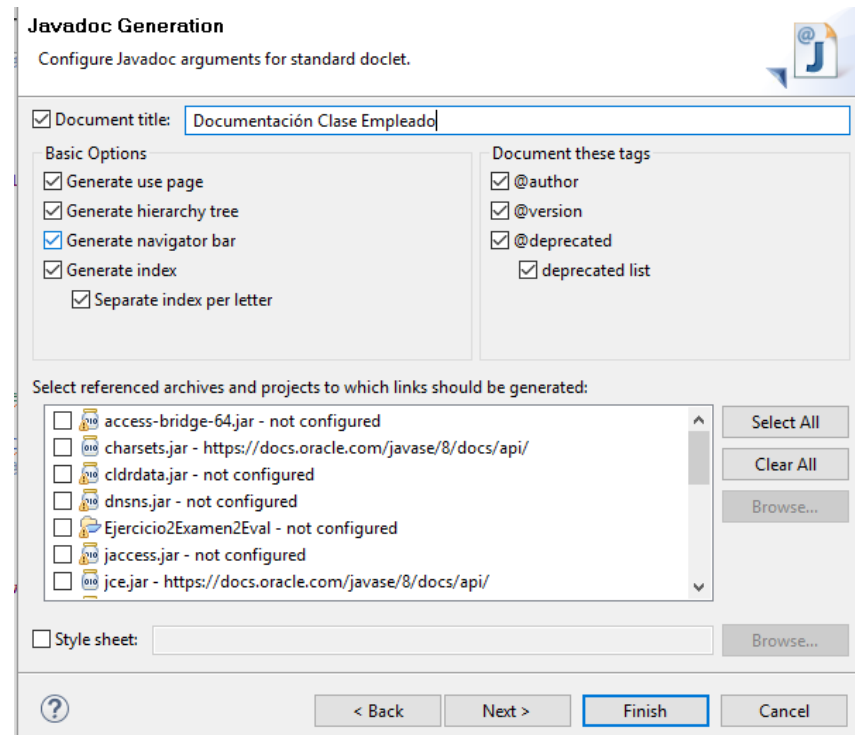
## 2.1.3 GENERAR LA DOCUMENTACIÓN

- Se indica la carpeta de destino donde se almacenará el código HTML (conviene poner un nombre diferente al que viene por defecto pues con doc hay veces que no se cargan los estilos)



## 2.1.3 GENERAR LA DOCUMENTACIÓN

- Se pulsa **Next** y en la siguiente ventana se indica el título del documento HTML que se genera.
- De las opciones, como mínimo se elige la barra de navegación y el índice.



## 2.1.3 GENERAR LA DOCUMENTACIÓN

- ◉ Al pulsar ***Finish*** se genera la documentación.

### **Class Empleado**

java.lang.Object  
modelo.Empleado

---

```
public class Empleado  
extends java.lang.Object
```

**Clase Empleado, se utiliza para crear y leer empleados de una BD**

Busca información de javadoc en [GOOGLE](#)

Since:

01-01-2019

Version:

1.0

Author:

Ana

See Also:

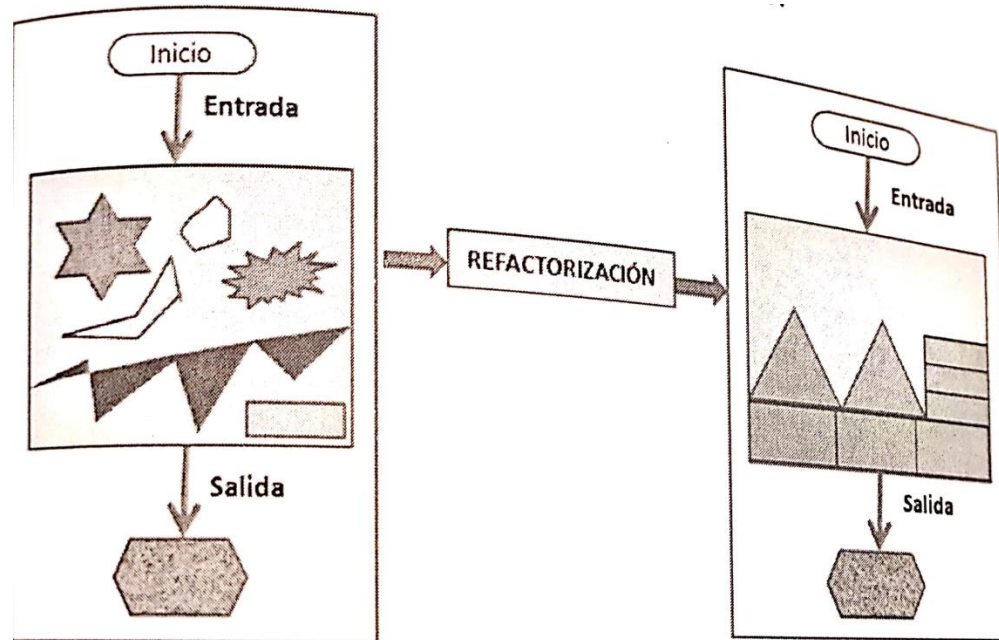
[Google](#)

# ACTIVIDAD

- ◉ Realiza los ejercicios 1 y 2 de la Hoja de Ejercicios

### 3. REFACTORIZACIÓN

- ◉ Técnica de la ingeniería de software que permite la optimización de código ya existente
- ◉ Consiste en hacer cambios internos, sin que se aprecien de forma externa



# 3. REFACTORIZACIÓN

- ◉ ¿Qué hace la refactorización?
  - Limpia el código, mejorando la consistencia y la claridad
  - Mantiene el código, no corrige errores ni añade funciones
  - Facilita la realización de cambios en el futuro
  - Se obtiene un código limpio y altamente modularizado.

## 3.1 ¿CUÁNDO REFACTORIZAR?

- ◉ La refactorización se debe hacer idealmente mientras se desarrolla una aplicación.
- ◉ Síntomas que indican la necesidad de refactorizar:
  - **Código duplicado:** principal razón para refactorizar. Si se detecta el mismo código en más de un lugar, se debe buscar la forma de extraerlo y unificarlo.
  - **Métodos muy largos:** cuanto más largo es un método más difícil es de entender, y normalmente está realizando tareas que deberían hacer otros. Se deben **identificar y descomponer el método en otros más pequeños**. En POO cuanto más corto es un método, más fácil es reutilizarlo.



## 3.1 ¿CUÁNDO REFACTORIZAR?

- **Clases muy grandes:** el mismo caso que con los métodos. Se debe ajustar bien las clases para que realicen solamente las tareas de su ámbito.
- **Lista de parámetros extensa:** tener demasiados parámetros puede indicar:
  - Problema de encapsulación de datos
  - Necesidad de crear una clase con varios de esos parámetros → pasar solamente ese objeto como argumento.
- **Envidia de funcionalidad:** si tenemos un método que utiliza más cantidad de elementos de otra clase que de la suya. Se suele resolver el problema pasando el método a otra clase.

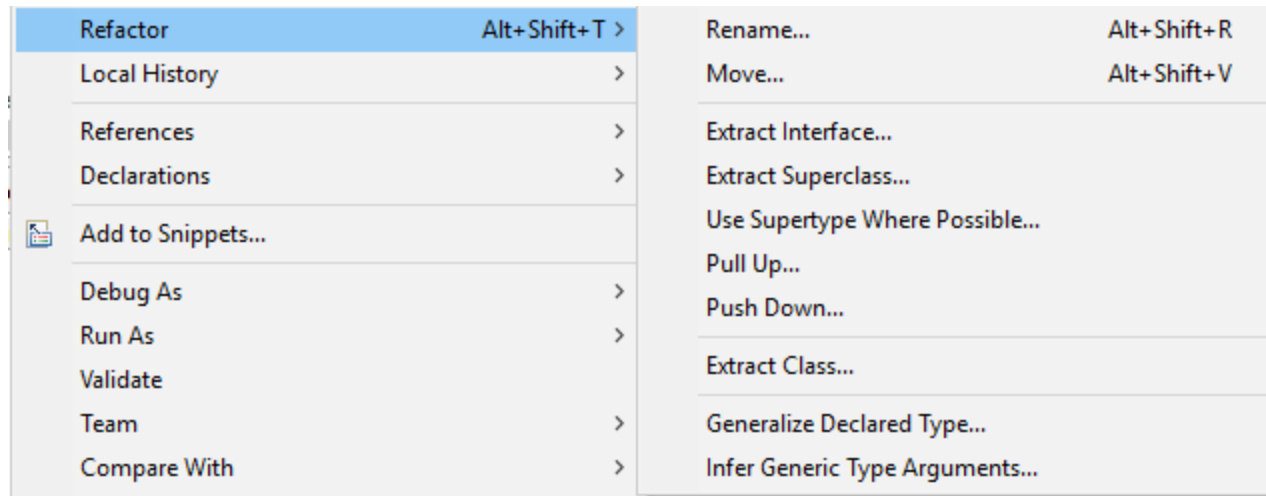
## 3.1 ¿CUÁNDO REFACTORIZAR?

- **Clase de sólo datos:** clase que sólo tiene atributos y métodos de acceso a ellos (get y set). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.
- **Legado rechazado:** subclases que utilizan sólo unas pocas características de sus superclases → Si las subclases no necesitan lo que heredan, hay que replantearse la jerarquía de clases creada.

## 3.2. REFACTORIZACIÓN EN ECLIPSE


- ◉ Eclipse tiene varios métodos de *refactoring*.
- ◉ Dependiendo de donde usemos la refactorización aparece un menú u otro con las opciones de refactorización.
- ◉ Utilizaremos la opción **Refactor** del menú contextual.

### CLASE




## 3.2. REFACTORIZACIÓN EN ECLIPSE

### MÉTODO

Refactor	Alt+Shift+T >	Move...	Alt+Shift+V
Local History	>	Change Method Signature...	Alt+Shift+C
References	>	Extract Interface...	
Declarations	>	Extract Superclass...	
 Add to Snippets...		Use Supertype Where Possible...	
Debug As	>	Pull Up...	
Run As	>	Push Down...	
Validate		Extract Class...	
Team	>	Introduce Parameter Object...	
Compare With	>	Infer Generic Type Arguments...	
Replace With	>		

### ATRIBUTO

Refactor	Alt+Shift+T >	Rename...	Alt+Shift+R
Local History	>	Move...	Alt+Shift+V
References	>	Extract Interface...	
Declarations	>	Extract Superclass...	
 Add to Snippets...		Use Supertype Where Possible...	
Debug As	>	Pull Up...	
Run As	>	Push Down...	
Validate		Extract Class...	
Team	>	Generalize Declared Type...	
Compare With	>	Infer Generic Type Arguments...	

## 3.2.1 MÉTODOS DE REFACTORIZACIÓN

- ◉ Son los que aparecen en los submenús anteriores:
  - **Rename:** Cambia el nombre de casi cualquier identificador Java (variables, clases, métodos, ...) Se modifican todas las referencias a ese identificador.
  - **Move:** mueve una clase de un paquete a otro, se mueve el archivo java a la carpeta y cambian todas la referencias. (también se puede arrastrar y soltar para refactorización automática).
  - **Extract Constant:** convierte un número o cadena literal en una constante. Al hacer la refactorización se mostrará dónde se van a producir los cambios y se puede visualizar el estado antes y después de refactorizar.

## 3.2.1 MÉTODOS DE REFACTORIZACIÓN

- **Extract local variable:** asignar una expresión a variable local. Tras la refactorización, cualquier referencia a la expresión en el ámbito local se sustituye por la variable. La misma expresión en otro método no se modifica.
- **Convert Local Variable to Field:** convierte una variable local en un atributo privado de la clase. Tras la refactorización, todos los usos de la variable local se sustituyen por ese atributo.

# ACTIVIDAD

- ◉ Abre el proyecto ***FicheroAleatorioVentana***.
- ◉ Realiza los siguientes cambios en la clase ***VentanaDepart.java***.
  - Extrae una variable local llamada existeDepart de la cadena: “DEPARTAMENTO EXISTE”.
  - Extrae una constante llamada NOEXISTEDEPART de la cadena: “DEPARTAMENTO NO EXISTE”.
  - Convierte la variable local creada en un atributo de la clase.
  - Extrae una variable local llamada deparError de la cadena: “DEPARTAMENTO ERRÓNEO”. ¿Qué diferencia hay con existeDepart? ¿Por qué?
  - Convierte dicha variable en un atributo de la clase.

## 3.2.1 MÉTODOS DE REFACTORIZACIÓN

- **Extract Method:** nos permite seleccionar un bloque de código y convertirlo en un método. El bloque de código no debe dejar llaves abiertas. Eclipse ajustará automáticamente los parámetros y el retorno de la función. Esto es muy útil para utilizarlo cuando se crean métodos muy largos, que se podrán dividir en varios métodos. También es muy útil extraer un método cuando se tiene un grupo de instrucciones que se repiten varias veces.



# ACTIVIDAD

- ◉ Dentro del método `actionPerformed(ActionEvent e)`, extrae varios métodos, uno para cada una de estas operaciones: alta, consultar, borrar y modificar.
- ◉ Extraer los métodos de las instrucciones que van dentro de los distintos `if (e.getSource())` que preguntan por alta, consu, borra y modif.
- ◉ Llamarlos **altadepart**, **consuldepart**, **borradepart** y **modifdepart**.
- ◉ **Repaso**: comenta con Javadoc los métodos consultar, visualiza, borrar, modificar y grabar de la parte inferior de la clase *FicheroAleatorioVentana*

## 3.2.1 MÉTODOS DE REFACTORIZACIÓN

- **Change Method Signature:** este método permite cambiar la cabecera de un método. Es decir, el nombre del método y los parámetros que tiene. Automáticamente se actualizan todas las llamadas al método en el proyecto.
- **OJO!** Si al refactorizar cambiamos el tipo de dato de retorno del método, aparecerán errores de compilación, por lo que debemos modificarlo manualmente.

# ACTIVIDAD

- ◉ Prueba a cambiar la cabecera del método `altadepart`.
- ◉ Añade un parámetro `String`, valor por defecto “PRUEBA”.
- ◉ Cambia el tipo de dato devuelto haciendo que retorne un entero.
- ◉ Comprueba y corrige los errores de retorno.
- ◉ Fíjate que la llamada al método ha cambiado automáticamente con el valor “PRUEBA” como parámetro.

## 3.2.1 MÉTODOS DE REFACTORIZACIÓN

- **Inline:** introduce la referencia a una variable/método en la línea en la que se utiliza. Se consigue así una única línea de código.

- **Ejemplo:**

```
File fichero = new File("AleatorioDep.dat");  
RandomAccessFile file = new RandomAccessFile(fichero, "r");
```

- Posicionamos el cursor en la referencia al método o variable (en este caso la variable fichero). Seleccionamos la opción Inline y el resultado es:

```
RandomAccessFile file = new RandomAccessFile(new File("AleatorioDep.dat"), "r");
```

## 3.2.1 MÉTODOS DE REFACTORIZACIÓN

- **Member Type to Top Level:** convierte una clase anidada en una clase de nivel superior con su propio archivo de java:
  - Si la clase es estática, la refactorización es inmediata.
  - Si no es estática, nos pide un nombre para declarar el nombre de la clase que mantendrá la referencia con la clase inicial.

# ACTIVIDAD

- ⦿ Crea esta clase anidada dentro de la clase FicheroAleatorioVentana:

```
class claseAnidada{  
    void entrada(){  
        System.out.println("Método de entrada");  
    }  
  
    String salida(int d){  
        System.out.println("Salida.");  
        return "Salida el " + d;  
    }  
}
```

- ⦿ Crea un objeto de esta clase y llama a los métodos dentro del método **verporconsola**. Añade este código en dicho método:

```
claseAnidada ej= new claseAnidada();  
ej.entrada();  
System.out.println("Llamo a salida " + ej.salida(10));
```

# ACTIVIDAD

- ◉ Prueba la ejecución (pulsas el botón ***Ver por consola*** de la ventana de ejecución).
- ◉ Convertir la clase anidada en una de nivel superior con su archivo asociado (Move Type to new file). Observa que se ha creado una nueva clase con su fichero java.
- ◉ Prueba de nuevo la ejecución.

## 3.2.1 MÉTODOS DE REFACTORIZACIÓN

- **Extract Interface:** este método de refactorización nos permite escoger los métodos de una clase para crear una Interface.



# ACTIVIDAD

- ◉ Crea la interface *IntVentanaDepart* que contenga los métodos **altadepart**, **consuldepart**, **borradepart** y **modifdepart**.
- ◉ Observa en el Package Explorer como se visualiza la clase creada, observa también el cambio producido en la declaración de la clase *VentanaDepart*.

## 3.2.1 MÉTODOS DE REFACTORIZACIÓN

- **Extract Superclass:** este método permite extraer una superclase. Si la clase ya utilizaba una superclase, la recién creada pasará a ser su superclase. Se pueden seleccionar los métodos y atributos que formarán parte de la superclase.

# ACTIVIDAD

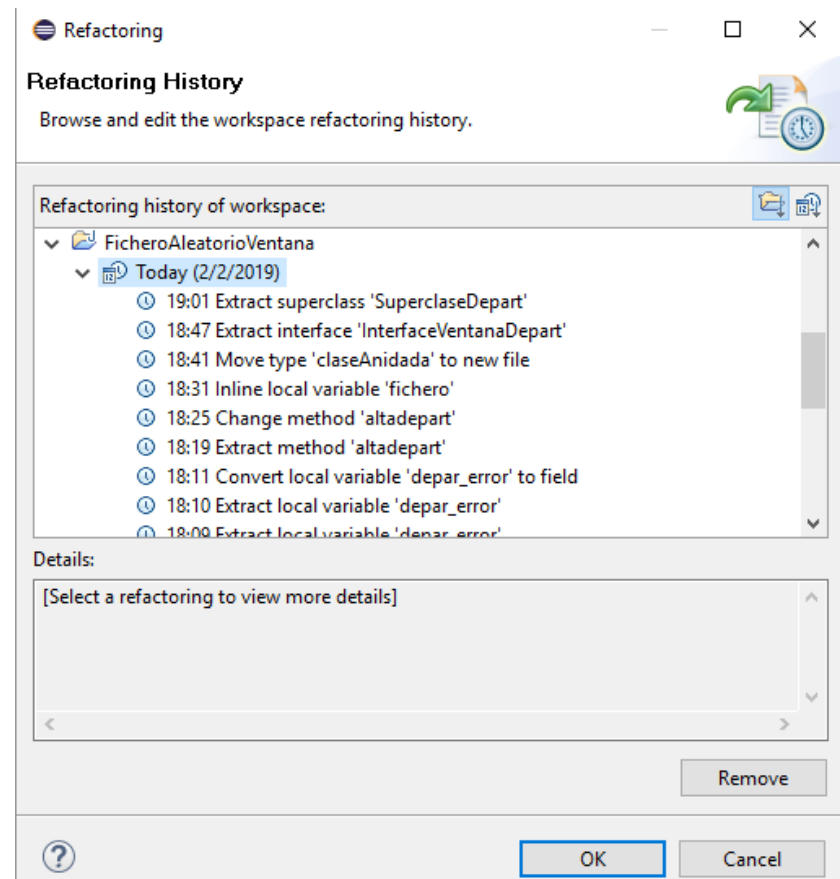
- ◉ A partir de la clase **VentanaDepart** crea la superclase ***SuperclaseDepart*** que incluya sólo los métodos de grabar y visualizar.
- ◉ Observa la clase generada y los constructores generados. Observar que la declaración de la clase ***VentanaDepart*** ha cambiado, ahora es ***extends SuperclaseDepart***.
- ◉ Observa los errores generados que se producen por hacer referencias a campos de la clase original (nombre y loc).
- ◉ Soluciona los errores moviendo las declaraciones de los campos a la superclase.

## 3.2.2 OTRAS OPERACIONES DE REFACTORIZACIÓN

- Para ver el histórico de refactorizaciones sobre un proyecto:

- Se abre el menú **Refactor/History**.

- En la ventana emergente se observan los cambios realizados y el detalle de los cambios.



## 3.2.2 OTRAS OPERACIONES DE REFACTORIZACIÓN

- ◉ Se puede elegir uno de los cambios, pulsar el botón Remove si se desea borrar del histórico de refactorización.
- ◉ Eclipse también permite crear un Script con todos los cambios realizados en la refactorización y guardarlo en un fichero XML (*Refactor/Create Script*).

# ACTIVIDAD

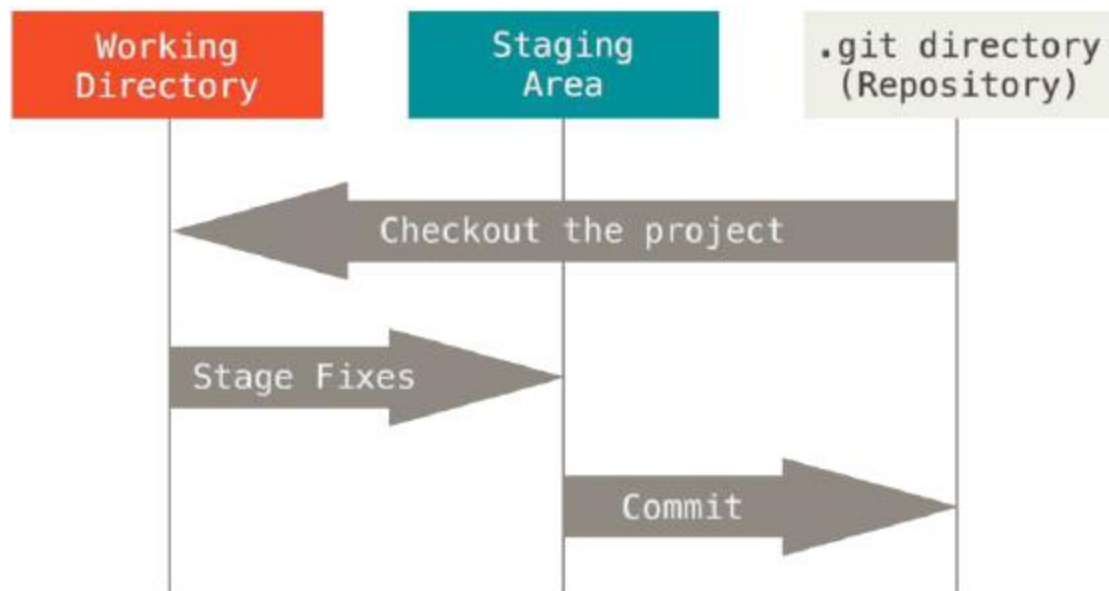
- ◉ Realiza el ejercicio 3 de la Hoja de Ejercicios

## 4. CONTROL DE VERSIONES CON GIT

*Un **Sistema de Control de Versiones** (SCV) es una aplicación que permite gestionar los cambios que se realizan sobre los elementos de un proyecto o repositorio, guardando así versiones del mismo en todas sus fases de desarrollo.*

## 4.1 INTRODUCCIÓN

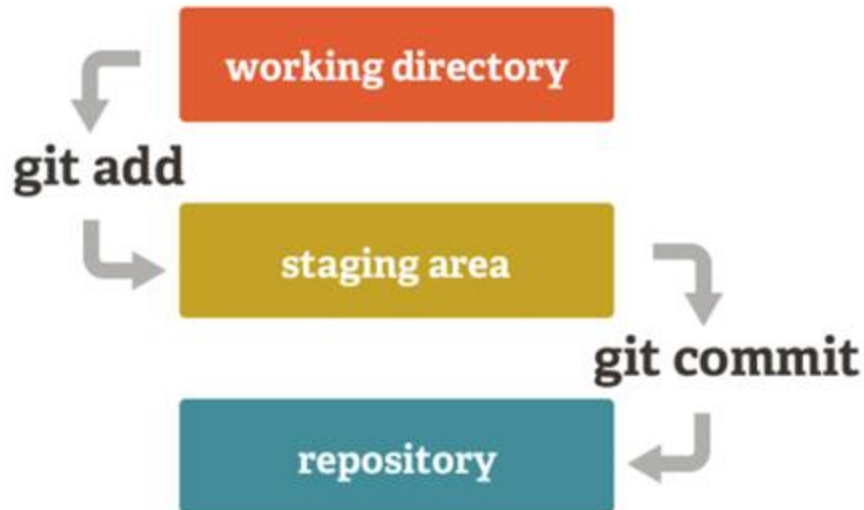
- ◉ Sistema de control de versiones de código abierto
- ◉ Creado por Linus Torvalds (padre de Linux)
- ◉ Estructura básica de un repositorio GIT:
  - Directorio Git (.git)
  - Directorio de trabajo (Working directory)
  - Área de montaje (Staging Area)





## 4.2 FLUJO DE TRABAJO EN GIT

1. Modificar una serie de archivos en *Working directory*.
2. Añadir los archivos al *Staging Area*.
3. Confirmar cambios y crear una instantánea
4. Enviar cambios del repositorio a un repositorio remoto



## 4.3 FORMAS DE UTILIZAR GIT

- ◉ Consola de comandos:  **git**

- ◉ Git online:

- Github 

-  **Bitbucket**

- ◉ Git integrado en un IDE:

- Eclipse
  - Visual Studio Code
  - ...

## 4.3.1 CONSOLA DE COMANDOS

### ◉ Instalación de GIT (consola de comandos): git

- En Windows es muy sencillo, el único paso importante es seleccionar un editor de texto adecuado (Notepad++, Sublime)
- En Linux todavía más sencillo. Para instalar todos sus componentes:
  - `sudo apt-get install git-all`
- Para crear un repositorio GIT, basta con situarse en la carpeta de un proyecto y utilizar el siguiente comando:
  - `git init`
  - En Windows: clic derecho para abrir la consola GIT en una carpeta (Git Bash Here)

## 4.3.1 CONSOLA DE COMANDOS

### ◉ Comandos básicos en local

Comando	Resultado
git add <archivo>	Añade un fichero a <i>staging</i>
<b>git add .</b>	<b>Añade todos los ficheros a <i>staging</i></b>
git rm <archivo>	Quita un fichero de <i>staging</i>
<b>git commit -m "DESCRIPCION"</b>	Envía <i>staging</i> → <i>repository</i>
git commit --amend	Modificar la descripción de un <i>commit</i>
git reset <fichero>	Retorno al estado del fichero en <i>staging</i>
<b>git reset --hard HEAD</b>	Retorno al <i>commit</i> inmediatamente anterior
git reset --hard HEAD~N	Elimina los N <i>commits</i> anteriores
<b>git reset --hard &lt;id-commit&gt;</b>	Retorno a un <i>commit</i> anterior por su id

## 4.3.1 CONSOLA DE COMANDOS

### ◉ Comandos básicos en local

- Los siguientes comandos son útiles para obtener información

Comando	Resultado
<code>git log --oneline</code>	Listar todos los <i>commit</i>
<code>git status -s</code>	Comprobar estado del directorio
<code>git diff &lt;fichero&gt;</code>	Ver diferencias con el fichero en <i>staging</i>

# ACTIVIDAD

- ◉ Se trata de realizar las siguientes tareas ...
  - Creación y actualización de repositorios
  - Historial de cambios
  - Deshacer cambios
- ◉ ... de esta página: Ejercicios resueltos
- ◉ Las haremos en una máquina virtual Debian (Linux)

# ACTIVIDAD

- ◉ Realiza los ejercicios 4 y 5 de la Hoja de Ejercicios

## 4.3.2 REPOSITARIOS REMOTOS

- ◉ Normalmente vamos a utilizar repositorios remotos para ampliar las posibilidades
- ◉ Vamos a empezar con GitHub
- ◉ Lo primero es crear una cuenta con nuestra cuenta de EducaMadrid
- ◉ Para crear un repositorio, arriba a la izquierda

Recent Repositories

 New

Find a repository...




## 4.3.2 REPOSITARIOS REMOTOS

- ◉ En la siguiente pantalla podemos asignar un nombre al proyecto, una descripción y public/private
- ◉ Se puede también añadir un **README** y un **.gitignore**
- ◉ Una vez creado el repositorio, puedo subir mi repositorio local con los comandos que me indican

...or push an existing repository from the command line

```
git remote add origin https://github.com/iferper/Ejercicio4.git
git branch -M main
git push -u origin main
```

- ◉ Al hacer el push pide **user** y password  **Ejercicio4** Private
  - Esto en Windows debería funcionar (genera credenciales)
  - En Linux suele ser necesario crear un token

## 4.3.2.1 PERSONAL ACCESS TOKEN

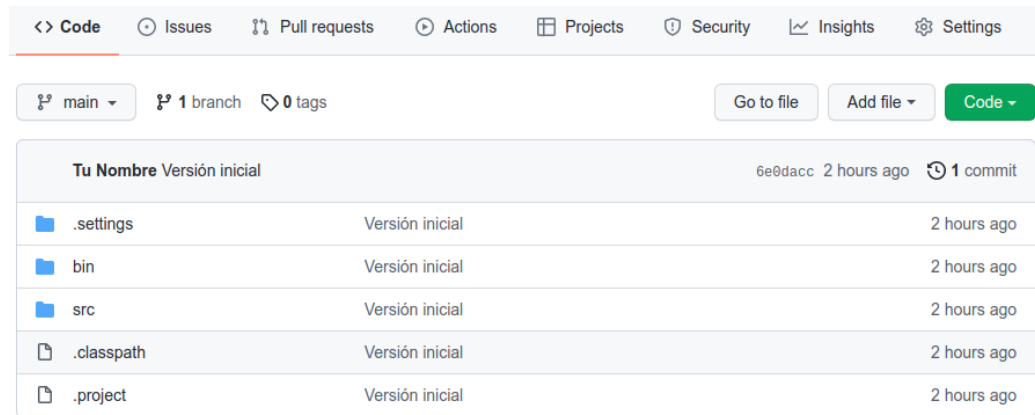
- Para hacer un push a un repositorio en Github, lo mejor es utilizar un Personal access token:
  - En Github: *Settings>Developer Settings>Personal access tokens>Generate new token*
  - Como *scopes* podemos elegir repo
  - Una vez creado lo copiamos y utilizamos este token como password junto al nombre de usuario.

```
madrid@Dpto-Inf-Imp:~/eclipse-workspace/Ejercicio4$ git push -u origin main
Username for 'https://github.com': iferper
Password for 'https://iferper@github.com': 
```

- Si se utilizan varios perfiles diferentes en el mismo PC, puede ser necesario borrar las credenciales:
  - En Windows, desde *Administrador de credenciales, Credenciales de Windows* y eliminando la correspondiente credencial de github.

## 4.3.2 REPOSITARIOS REMOTOS

- Después de hacer el push ya puedo ver mi código en Github

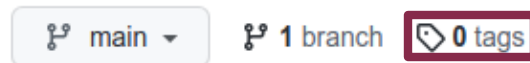


- Y en local podemos ver un cambio

```
madrid@Dpto-Inf-Imp:~/eclipse-workspace/Ejercicio4$ git log --oneline
6e0dacc (HEAD -> main, origin/main) Versión inicial
```

## 4.3.3 TAGS

- ◉ Un **tag** no es más que una etiqueta que marca el fin de una versión
- ◉ Permite organizar los proyectos de forma más sencilla y que otras personas puedan descargarlos
- ◉ Para crear un tag pulsamos tags en GitHub



- ◉ Indicamos un título y una descripción y listo
- ◉ Normalmente se hace desde consola con el comando `git tag`

## 4.3.2 REPOSITARIOS REMOTOS

### ◉ Comandos básicos en remoto

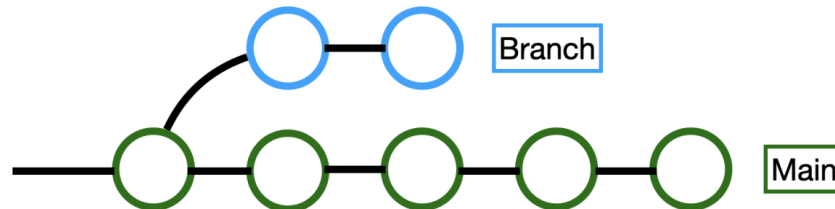
Comando	Resultado
<b>git clone &lt;url&gt;</b>	<b>Crear copia de repositorio remoto</b>
<b>git push origin master</b>	<b>Llevar un commit de local a remoto</b>
<b>git tag &lt;nombre&gt; -m "descripción"</b>	<b>Crear una etiqueta</b>
<b>git push --tags</b>	<b>Enviar tags al repositorio remoto</b>
<b>git fetch origin</b>	<b>Preparar los cambios disponibles</b>
<b>git merge origin/master</b>	<b>Descargar los cambios anteriores</b>
<b>git pull</b>	<b>Comprobar y descargar los cambios del repositorio remoto al repositorio local (fetch + merge)</b>

# ACTIVIDAD

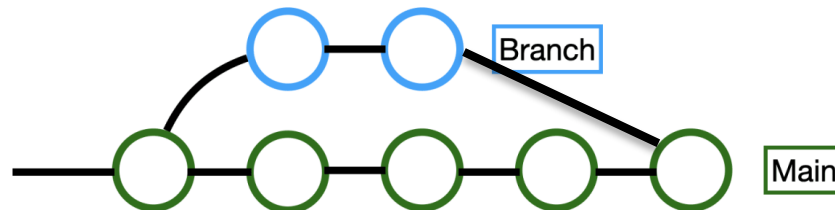
- ◉ Sube el proyecto del ejercicio 4 (aula virtual) a Github y prueba a hacer varias modificaciones
  - Haz modificaciones directamente en remoto y luego actualiza los cambios con git pull
  - Crea un tag desde consola y compruébalo en GitHub
- ◉ **Ampliación:** investiga cómo realizar lo mismo en BitBucket y busca cuáles son las diferencias con GitHub
- ◉ **Avanzado:** investiga sobre cómo usar el fichero .gitignore
  - Aplícalo para no incluir en el proyecto los ficheros de la carpeta bin (es habitual en uso de repositorios no guardar binarios)

## 4.3.3 RAMAS (BRANCHES)

- Una rama es una desviación del proyecto para hacer versiones paralelas



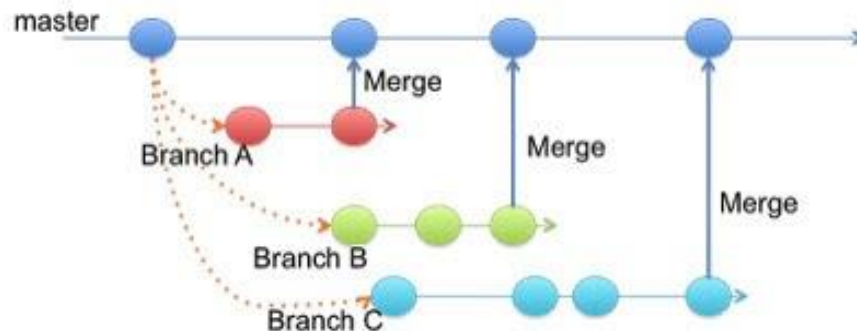
- Cuando la versión está lista para añadirse al proyecto, se fusiona (merge) para unir los cambios



- Si se trata de archivos (clases) paralelas es muy sencillo, GIT junta todo y listo

## 4.3.3 RAMAS (BRANCHES)

- ◉ El problema surge cuando hay secciones diferentes en ambas versiones
- ◉ En estos casos hay que solucionar los conflictos manualmente haciendo cambios en los ficheros
  - GIT muestra donde está el conflicto con elementos visuales
  - En IDEs (Eclipse, Visual Studio Code) suele ser más sencillo y mediante un botón se elige la versión adecuada
- ◉ Un repositorio admite más ramas si son necesarias





## 4.3.3 RAMAS (BRANCHES)

### ◉ Comandos básicos con ramas

Comando	Resultado
<b>git branch &lt;nombre-rama&gt;</b>	<b>Crear una rama</b>
<b>git branch -d &lt;nombre-rama&gt;</b>	<b>Borrar rama</b>
<b>git branch</b>	<b>Consultar en qué rama estoy</b>
<b>git checkout &lt;nombre-rama&gt;</b>	<b>Moverse a otra rama</b>
<b>git merge &lt;nombre-rama&gt;</b>	<b>Fusionar una rama con la rama master</b>

# ACTIVIDAD

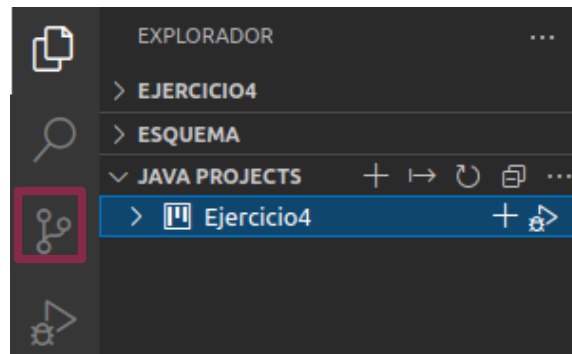
- ◉ Realiza el ejercicio 6 de la Hoja de Ejercicios
- ◉ **Refuerzo:**
  - Realiza los apartados de Ramas y Repositorios remotos del [siguiente enlace](#)
  - Visualiza los vídeos del curso de Píldoras Informáticas (hasta el 5) y sigue los pasos en tu PC

## 4.3.4 GIT INTEGRADO EN IDE

- ◉ Los IDE como VSC facilitan la labor de utilizar GIT
- ◉ Para trabajar con VSC abrimos la carpeta del proyecto
- ◉ Si el proyecto ya tenía código, VSC nos ofrece instalar extensiones para Java, que es recomendable
  - Esto me crea un espacio de trabajo *JAVA PROJECTS* y me permite ejecutar programas

## 4.3.4 GIT INTEGRADO EN IDE

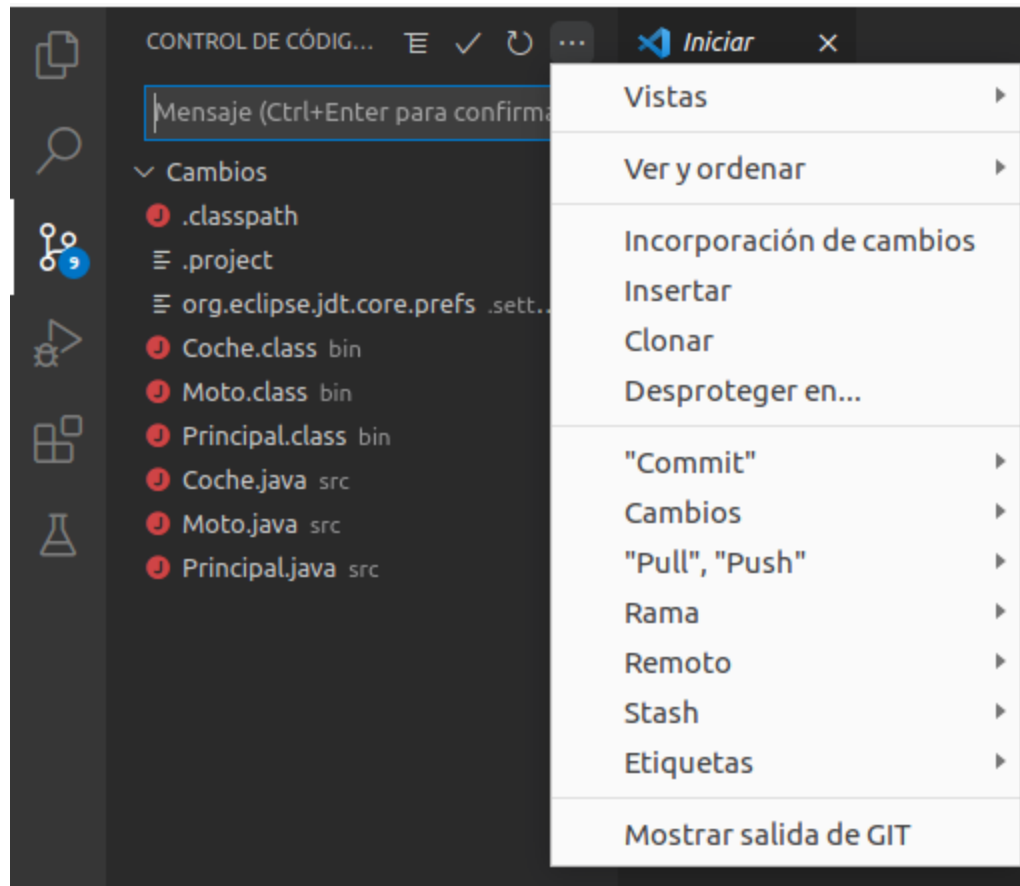
- En el botón indicado se gestiona GIT de forma gráfica



- A partir de aquí podemos crear un repositorio y ejecutar gráficamente las acciones que ya conocemos
- También se puede acceder a la consola mediante Ver>Terminal

## 4.3.4 GIT INTEGRADO EN IDE

- Haciendo clic en los 3 puntos accedemos al menú GIT



## 4.3.4 GIT INTEGRADO EN IDE

- ◉ Una vez que tenemos un repositorio veremos que VSC nos marca los cambios en cada fichero como:
  - **U** : sin seguimiento
  - **A** : añadido
  - **M** : modificado
  - Si no hay ninguna marca, es que el fichero está committed
- ◉ También se puede publicar ramas directamente a GitHub iniciando sesión

## 4.3.4.1 RAMAS EN VSC

- ◉ La ventaja más importante de usar GIT con VSC aparece al utilizar ramas
- ◉ Se puede ver en qué rama estamos en la parte inferior izquierda:



- ◉ Para hacer *merge* podemos usar el menú de GIT (Rama>Fusionar rama)

## 4.3.4.1 RAMAS EN VSC

- ◉ Cuando surge algún conflicto VSC lo resuelve en función del caso:
  - Cuando se trata de código añadido que no interfiere, normalmente combina los cambios por defecto.
    - En este caso igualmente tenemos que hacer un commit aceptando los cambios
  - Cuando el código es diferente y hay que elegir me muestra algo similar a lo siguiente:

```
public void printCoche(){
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<<< HEAD (Current Change)
    System.out.println("El cocherito leré");
=====
    System.out.println("Aquí escribo otra cosa");
>>>>>> prueba (Incoming Change)
}
```



## 4.3.4.1 RAMAS EN VSC

- ◉ En este último caso podemos elegir entre:
  - **Accept Current Change:** acepta cambios de master
  - **Accept Incoming Change:** acepta cambios de la nueva rama
  - **Accept Both Changes:** acepta ambos cambios y los combina
  - **Compare Changes:** muestra los cambios con detalle
- ◉ En cualquier caso, debemos hacer commit de los cambios para finalizar la fusión (*merge*).

# ACTIVIDAD

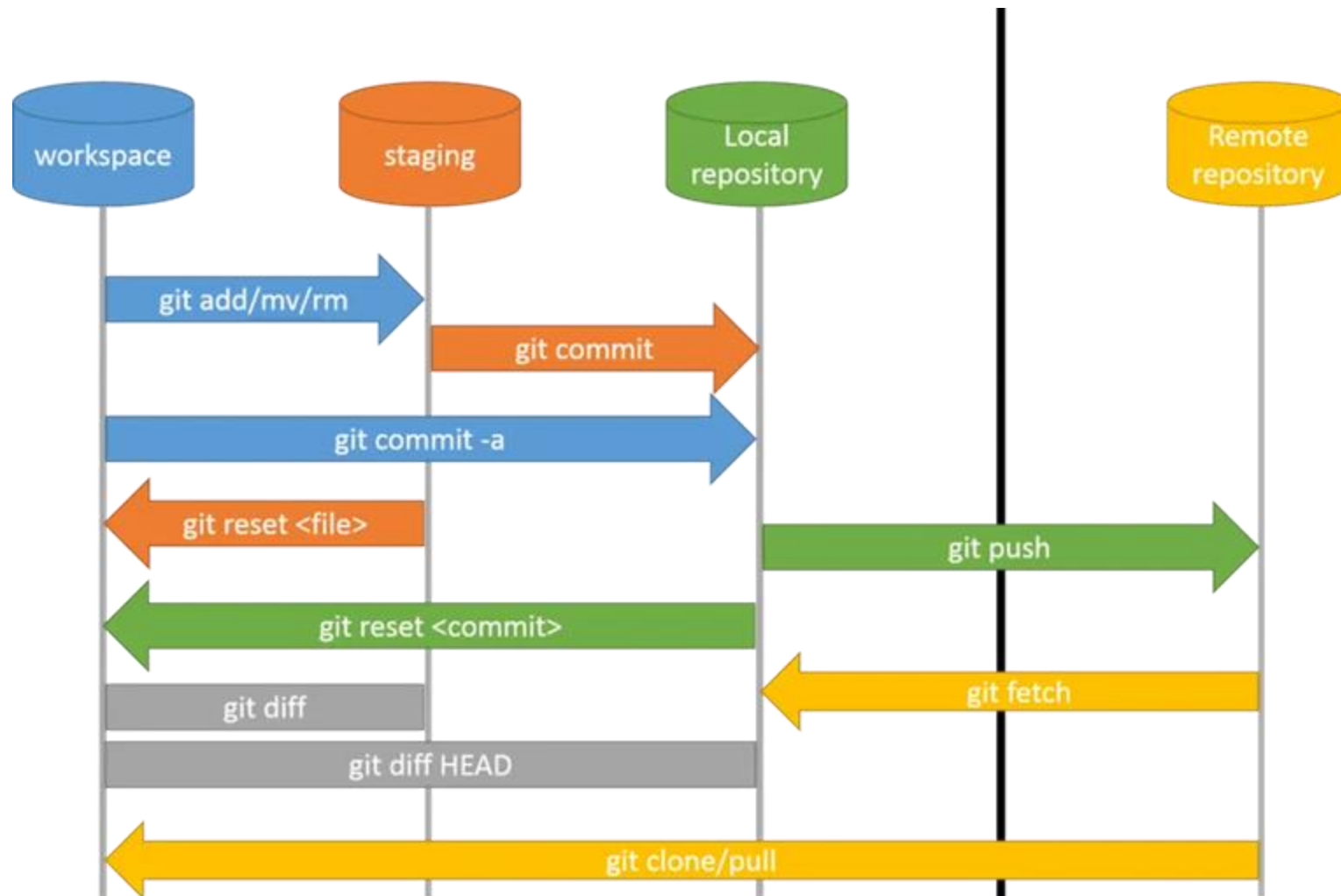
- ◉ Realiza el ejercicio 6 de la Hoja de Ejercicios utilizando VSC
- ◉ A continuación, añade un método en la clase Coche desde la rama principal y haz commit.
- ◉ Crea una rama prueba y modifica ese método para forzar a que ocurra un conflicto. Haz commit.
- ◉ Cámbiate a la rama master y fusiona las ramas. Soluciona el conflicto.

# ACTIVIDAD

## ◉ Refuerzo:

- Visualiza los vídeos del curso de Píldoras Informáticas (6-9) y sigue los pasos en tu PC

## 4.5 RESUMEN DE COMANDOS GIT



# ENLACES DE INTERÉS

- ◉ [GIT - la guía sencilla](#)
- ◉ [Juego para aprender GIT](#)
- ◉ [Curso píldoras informáticas](#)