Curso de Java orientado a Android – Parte 1

Curso de Java orientado a Android

Introducción

Este es el primero de cuatro posts de introducción al desarrollo en Java. Cabe decir que este tutorial de Java va estar encarado al desarrollo par Android. Es decir, se van a omitir partes que no sean interesantes para el posterior desarrollo de Aplicaciones Android.

- Lenguaje de programación Java
- Java Virtual Machine
- JDK y JRE
- Configuración del equipo para programar en Java
- Hello World en Java
    - Utilizando un editor de texto
    - Utilizando un IDE
- Tipos de datos primitivos
- Utilización de nombres

- Comentarios
    - Comentarios para documentación
- Arrays
- Control Flow
    - If/Else and If/Else If
    - Switch statement
    - While loop
    - For Loop

Lenguaje de programación Java

El lenguaje de programación Java fue desarrollado originalmente por Sun Microsystems (Actualmente de Orace) y liberado en 1995. Las aplicaciones Java están típicamente compiladas a byte codes (archivo de clase) que puede funcionar en cualquier máquina virtual de Java (Java Virtual Machine) independientemente del sistema operativo. Java es un lenguaje orientado a objetos. Fue diseñado para permitir a los desarrolladores «escribir una vez y ejecutar en cualquier lugar» (Write Once Run Anywhere), lo que significa que el código que se ejecuta en una plataforma no necesita ser recompilado para funcionar en otra.

Java Virtual Machine

La máquina virtual de Java (JVM) es el componente del framework de Java que ejecuta el código compilado. Cuando un desarrollador compila un archivo Java, el compilador de genera una archivo de bytecode con extensión .class. El bytecode de Java es un lenguaje intermedio generado por el compilador y ejecutado únicamente en una JVM.

JDK y JRE

Para poder iniciar la programación en Java un desarrollador necesita dos componentes principales:

- Java Development Kit.

  El JDK proporciona un compilador de Java, además de otras herramientas. Estas herramientas permiten a un programar código Java y compilarlo a un archivo de bytecodes ejecutable en una JVM. El programa que compila el código se llama javac.En resumen, para empezar a escribir programas Java y compilarlos necesitamos el JDK.

- Java Runtime Environment

  El JRE es el entorno de ejecución para programas Java. Estos programas se compilan en un formato binario portátil (archivos .class) por el compilador. En resumen, para ejecutar programas compilados en Java necesitamos el JRE.

Configuración del equipo para programar en Java

Instalaremos la version 7 de OpenJdk (versión libre de la plataforma de desarrollo Java) y seguidamente comprobaremos que se ha instalado ejecutando un comando para obtener la versión instalada en nuestro equipo:

```
$ sudo apt-get install openjdk-7-jdk
$ java -version
java version "1.7.0_75"
OpenJDK Runtime Environment (IcedTea 2.5.4) (7u75-2.5.4-1~utopic1)
OpenJDK 64-Bit Server VM (build 24.75-b04, mixed mode)
```

Hello World en Java

Vamos a crear el programa que por excelencia es el primer paso para cualquier programador en un nuevo lenguaje. Para ello utilizaremos *pluma* y *Eclipse.*

Utilizando un editor de texto (pluma)

Ejecutaremos el siguiente comando

```
$ pluma HelloWorldAdictos.java
```

Escribimos el siguiente código y guardamos el fichero:

```
public class HelloWorldAdictos {
    public static void main(String[] args) {
        System.out.println("Hello World Adictos!");
    }
}
```

Seguidamente ejecutamos el siguiente comando para compliar:

```
$ javac HelloWorldAdictos.java
```

Y finalmente ejecutamos nuestro programa:

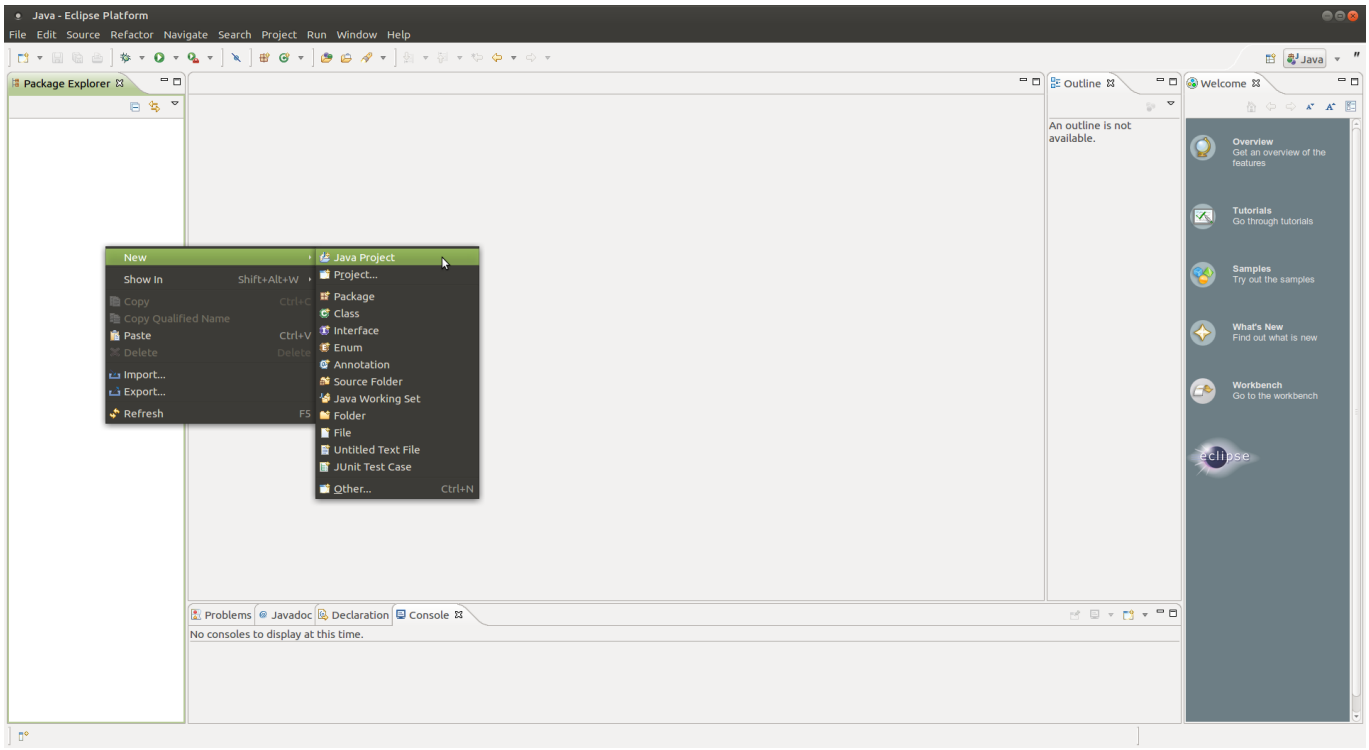
```
$ java HelloWorldAdictos
Hello World Adictos!
```

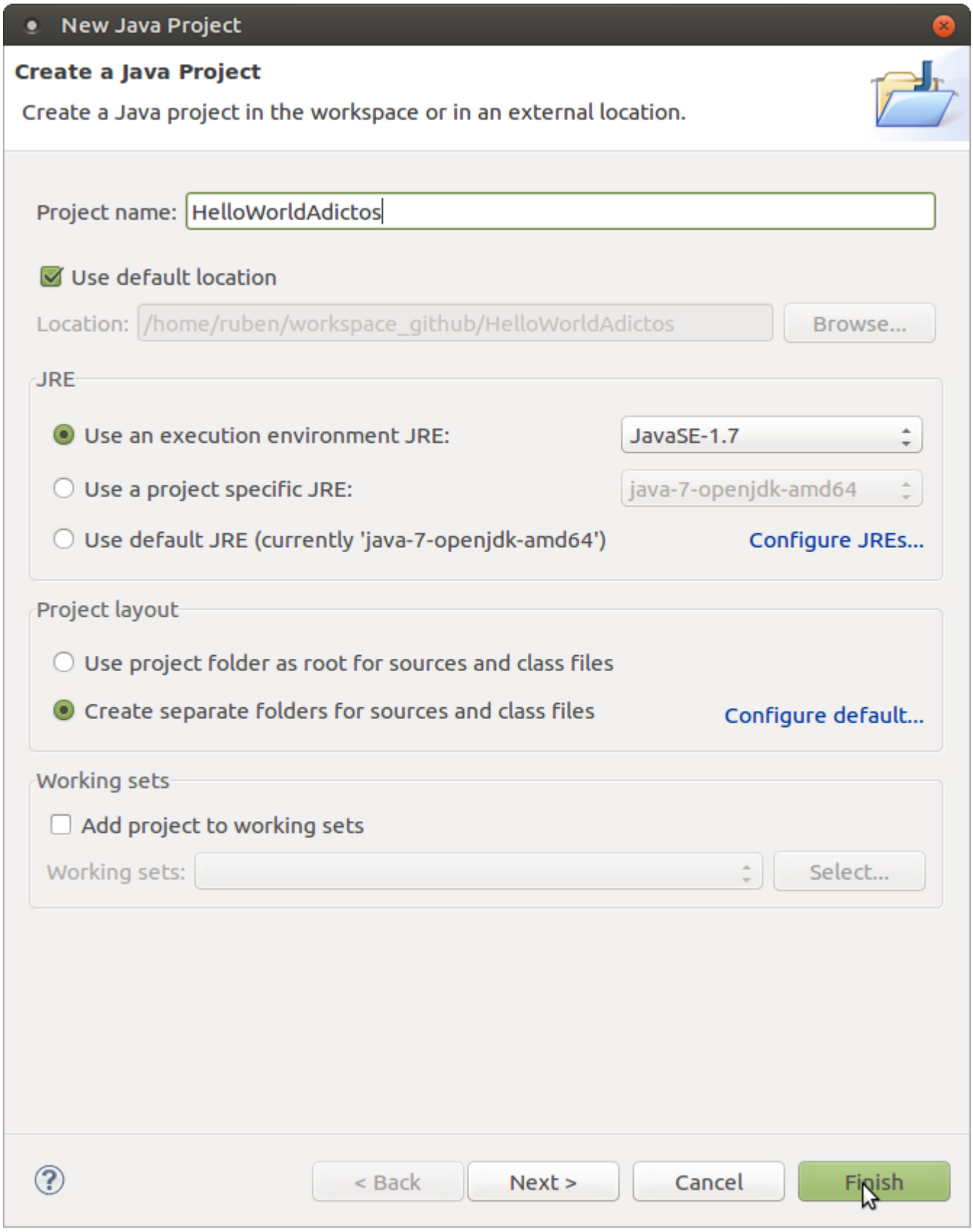
Utilizando un IDE (Eclipse)

Un IDE (Integrated Development Enviroment) es una aplicación que ofrece un conjunto de herramientas para facilitar las tareas de desarrollo a los programadores. En los siguiente pasos instalaremos eclipse, crearemos un proyecto HelloWorldAdictos y lo ejecutaremos. Primeramente instalaremos Eclipse:

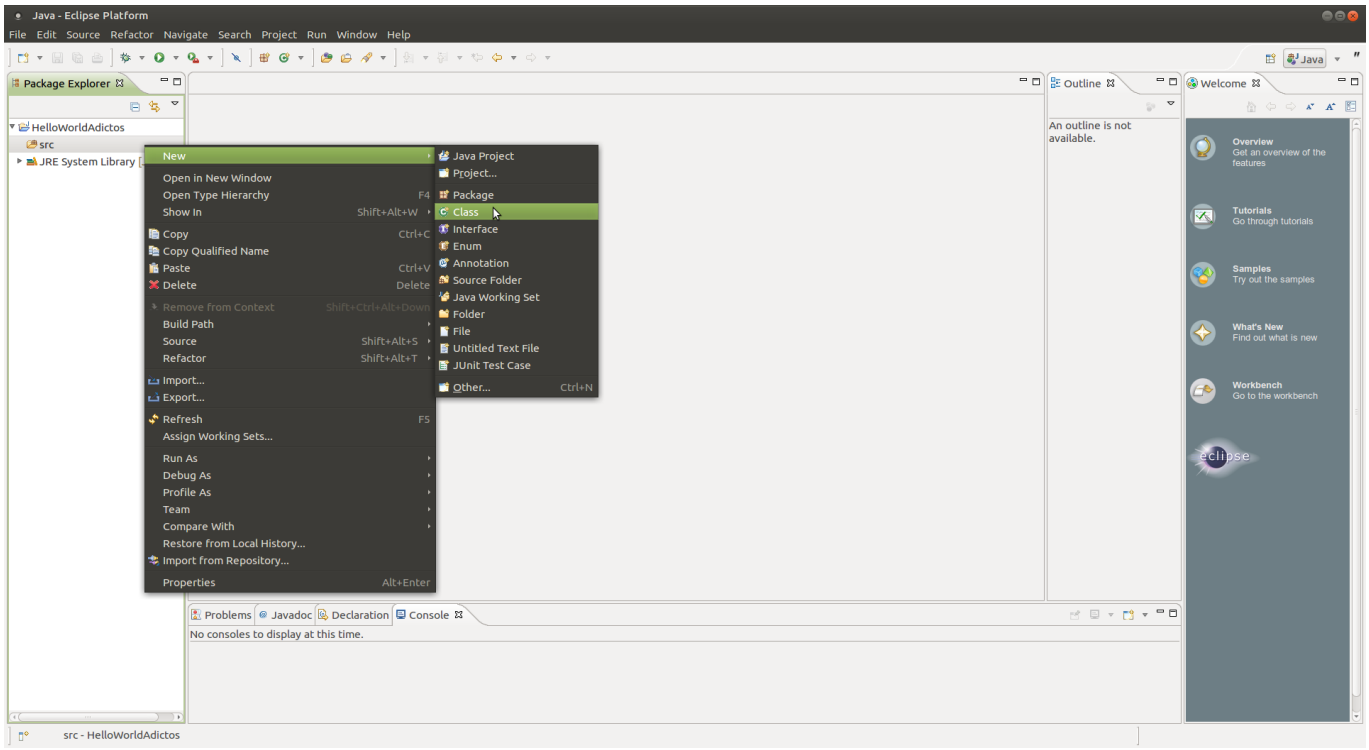
```
$ sudo apt-get install eclipse
```

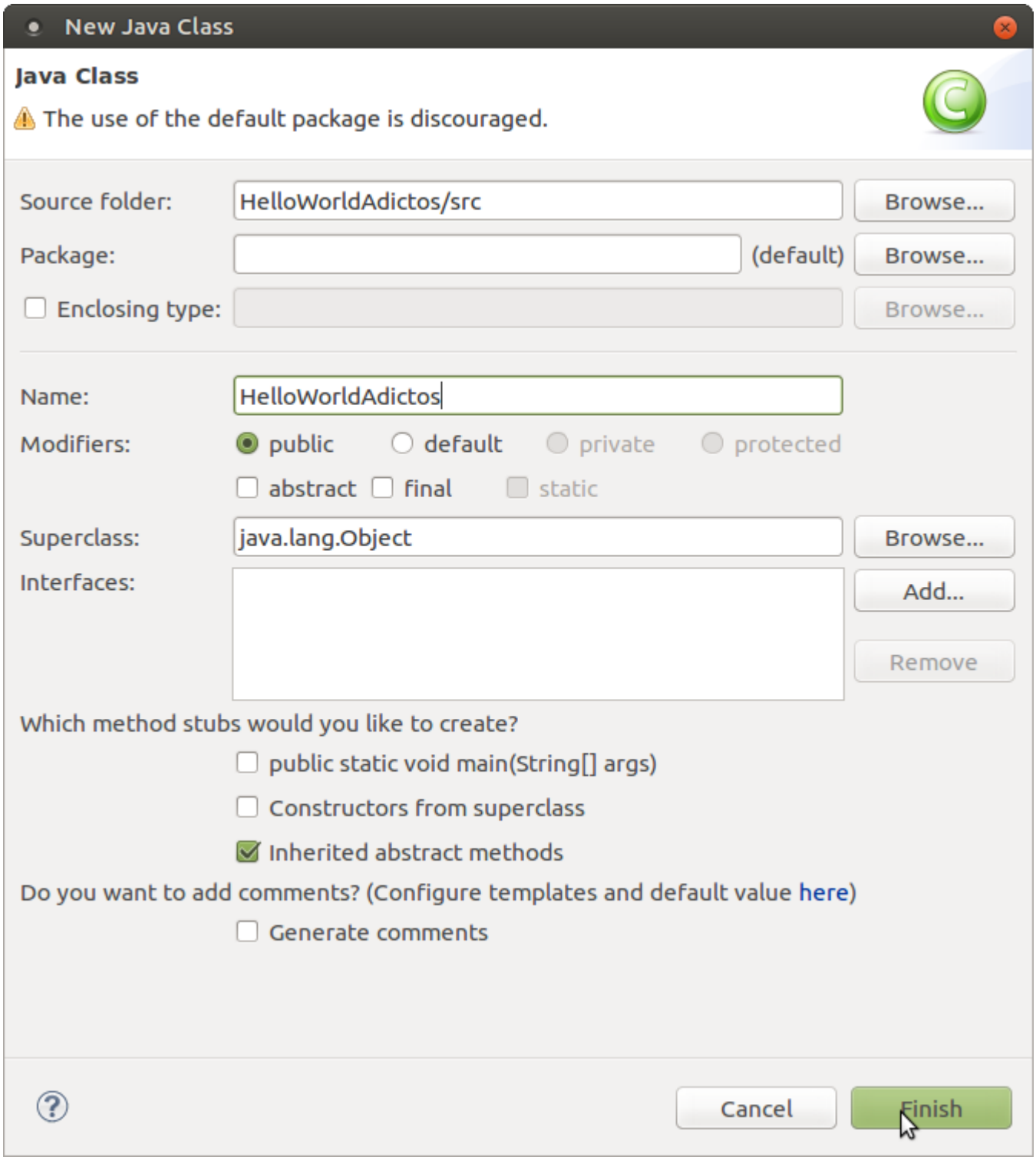
Una vez abierto eclipse creamos un nuevo proyecto java


Java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explorer
Outline
An outline is not available.
Welcome
Overview
Get an overview of the features
Tutorials
Go through tutorials
Samples
Try out the samples
What's New
Find out what is new
Workbench
Go to the workbench
eclipse
New
Show In Shift+Alt+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Import...
Export...
Refresh F5
Java Project
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
JUnit Test Case
Other... Ctrl+N
Problems
Javadoc
Declaration
Console
No consoles to display at this time.

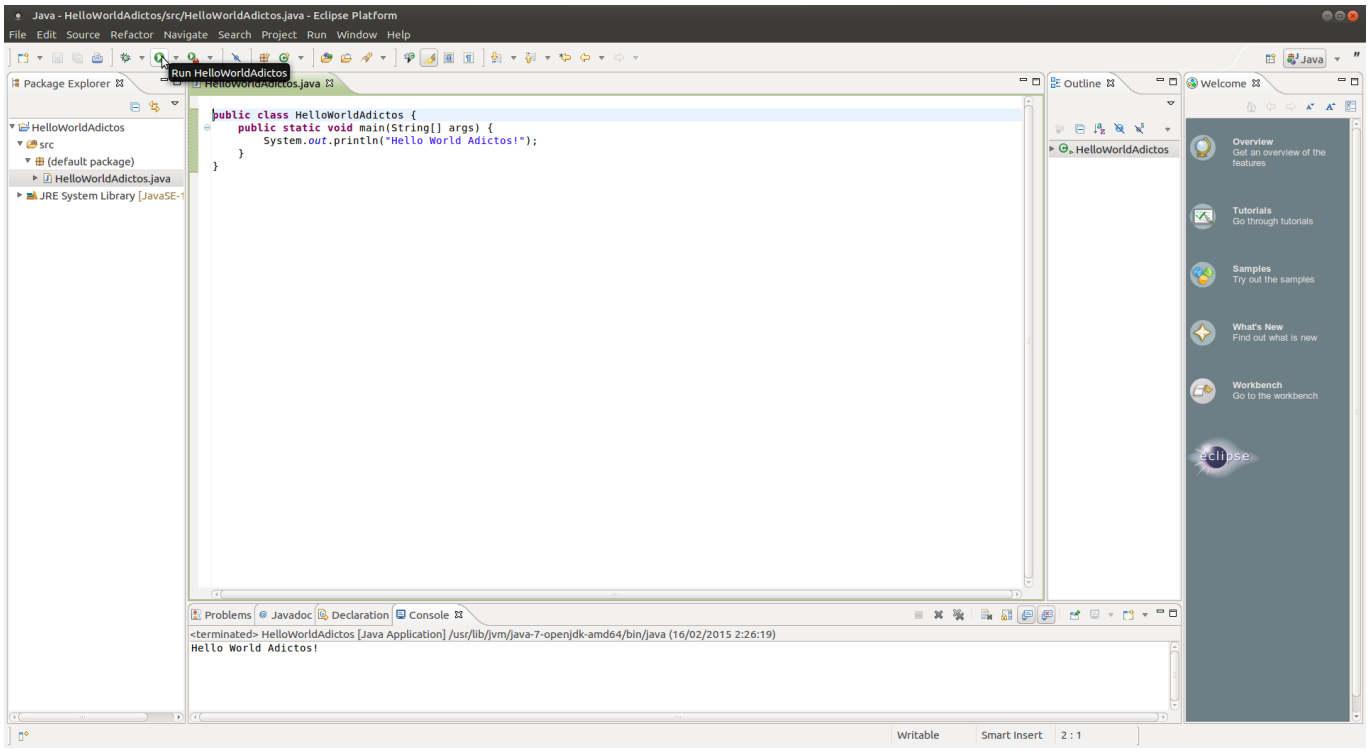


Llegados a este punto deberemos crear una clase en «src»


Java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explorer
HelloWorldAdictos
src
JRE System Library
New
Open in New Window
Open Type Hierarchy F4
Show In Shift+Alt+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Shift+Ctrl+Alt+Down
Build Path
Source Shift+Alt+S
Refactor Shift+Alt+T
Import...
Export...
Refresh F5
Assign Working Sets...
Run As
Debug As
Profile As
Team
Compare With
Restore from Local History...
Import from Repository...
Properties Alt+Enter
Java Project
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
JUnit Test Case
Other... Ctrl+N
Outline
An outline is not available.
Welcome
Overview
Get an overview of the features
Tutorials
Go through tutorials
Samples
Try out the samples
What's New
Find out what is new
Workbench
Go to the workbench
eclipse
Problems Javadoc Declaration Console
No consoles to display at this time.
src - HelloWorldAdictos



Copiamos nuestro código y ejecutamos nuestro «Hola mundo» clicando en «Run». Finalmente en la consola podemos ver el resultado de la ejecución de nuestro programa



Tipos de datos primitivos

Todas las variables en Java deben ser declaradas antes de que puedan ser utilizadas. Esto se realiza especificando el tipo de de la variable y el nombre de la variable:

float testVar = 1;

Java soporta ocho tipos de datos primitivos diferentes:

- byte: El tipo de datos byte es un entero de 8 bits. El intervalo de valores va desde -2^7 hasta 2^7 -1 (-128 a 127)
- short: El tipo de datos a corto es un entero de 16 bits. El intervalo de valores va desde -2^{15} hasta 2^{15}-1 (-32768 a 32767)
- int: El tipo de datos int es un entero de 32 bits. Tiene un valor máximo de 2147483647. El intervalo de valores va desde -2^{31} hasta 2^{31}-1 (-2147483648 a 2147483647)
- long: El tipo de datos de largo es un entero de 64 bits. El intervalo de valores va desde -2^{63} hasta 2^{63}-1 (-9223372036854775808 a 9223372036854775807)

- float: El tipo de datos float es un punto flotante de 32 bits de precisión simple (por ejemplo: **float a=12.5f;**).
- doble: El tipo de datos doble es un punto flotante de 64 bits de doble precisión (por ejemplo **double PI=3.14159**).
- boolean: El tipo de datos booleano tiene sólo dos posibles valores: true y false (por ejemplo: **boolean isEnabled=true;**).
- char: El tipo de datos char es un solo carácter Unicode de 16 bits. El intervalo de valores va desde 0 hasta 65535 (valores de 16-bits sin signo).

Tipos especiales de caracteres:

- retorno de carro: **\r**
- tabulador horizontal **\t**
- nueva línea **\n**
- barra invertida ****

Utilización de nombres

Java tiene las siguientes reglas y convenciones para utilizar nombre en el código:

- Los nombres de las clases se deben capitalizar (**UserContract**)
- Los nombres de variables distinguen entre mayúsculas y minúsculas (**newUser**)
- Por convención, se deben asignar los nombres de variables utilizando «camel case». Es decir, si el nombre consta de una sola palabra todas la letras serán minúsculas (**phone**). Si consiste en más de una palabra,la primera letra de cada palabra subsiguiente se escribe con mayúscula. (**phoneNumber**)
- También por convención, las constantes son capitalizados y contienen subrayado. (**PI**)

Hay nombres reservados para el lenguaje que no se pueden utilizar:

- Tipos de datos: **boolean, float, double, int, char**
- Sentencias condicionales: **if, else, switch**
- Sentencias iterativas: **for, do, while, continue**
- Tratamiento de las excepciones: **try, catch, finally, throw**
- Estructura de datos: **class, interface, implements, extends**
- Modificadores y control de acceso: **public, private, protected, transient**
- Otras: **super**, **null, this**.

Comentarios

Existen dos tipos de comentarios comentarios:

- De línia //

```
//line comment
```

- Y bloques que comienzan con /* y terminan con */

```
/*first line comment
  second line comment
  third line comment*/
```

Comentarios para la documentación

JDK proporciona javadoc, una herramienta para generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. Para utilizar correctamente javadoc debemos seguir unas normas de documentación:

- Los comentarios deben empezar con /** y terminar con */.
- Se pueden incorporar comentarios a nivel de clase, a nivel de variable y a nivel de método.
- Se genera la documentación para miembros public y protected.
- Se pueden usar tags para documentar ciertos aspectos concretos como listas de parámetros o valores devueltos.

Tipo de tag	Formato	Descripción
Todos	@see	Permite crear una referencia a la documentación de otra clase o método.
Clases	@version	Comentario con datos indicativos del número de versión.
Clases	@author	Nombre del autor.
Clases	@since	Fecha desde la que está presente la clase.
Métodos	@param	Parámetros que recibe el método.
Métodos	@return	Significado del dato devuelto por el método
Métodos	@throws	Comentario sobre las excepciones que lanza.
Métodos	@deprecated	Indicación de que el método es obsoleto.

Ejemplo de clase comentada

```
/** Un programa Java simple.
  * Muestra por pantalla el primer digito de version android y
sus nombres.
  * @author Ruben
  * @version 1
  */
public class TestWhile {
        /** Unico punto de entrada.
          * @param args Array de Strings.
          * @return No retorna ningun valor.
          * @throws No dispara ninguna excepcion.
          */
        public static void main(String[] args) {
                String[] androidVersionsNames = new String[5];
                int counter = 0;
                androidVersionsNames[0] = "Donut";
                 androidVersionsNames[1] = "Eclair - Froyo -
```

```
Gingerbread";
                androidVersionsNames[2] = "Honeycomb";
                 androidVersionsNames[3] = "Ice Cream Sandwich
- Jelly Bean - KitKat";
                androidVersionsNames[4] = "Lollipop";
                while (counter <= 4){
                        System.out.println("Version number " +
(counter+1)   +   "   --   Version   name   "   +
androidVersionsNames[counter]);
                        counter++;
                }
        }
}
```

Con el siguiente comando generaremos la documentación en html para la clase TestWhile en el directorio ~/Documents/TestWhileDoc

```
$ javadoc -d  ~/Documents/TestWhileDoc TestWhile.java
```

Arrays

Los arrays son recipientes que contienen un número fijo de valores de un tipo determinado. La longitud de un array es fija y se declara cuando se crea. Por ejemplo, así declaramos un array de integers de 6 elementos:

int [] customArray = new int [5];

Cada contenedor de un array se llama un elemento, y a cada elemento se accede por su índice numérico. El índice de numeración en arrays comienza por 0. Por ejemplo, para acceder al indice 5 utilizaremos el número 4. Se puede asignar un valor a un elemento del array con la siguiente sintaxis:

```
public class TestAarray {
        public static void main(String[] args) {
                int[] customArray;
                customArray = new int[5];
```

```
                // Initialize elements
                customArray[0] = 1;
                customArray[1] = 2;
                customArray[2] = 3;
                customArray[3] = 4;
                customArray[4] = 5;
                  System.out.println("Value at index 0: " +
customArray[0]);
                  System.out.println("Value at index 1: " +
customArray[1]);
                  System.out.println("Value at index 2: " +
customArray[2]);
                  System.out.println("Value at index 3: " +
customArray[3]);
                  System.out.println("Value at index 4: " +
customArray[4]);
        }
}
```

Control Flow

Los bloques de código en Java se ejecutan secuencialmente -de arriba a abajo- en el orden en que aparecen. Sin embargo, un programador puede controlar el flujo de ejecución utilizando condicionales, loops y branches. En esta sección se describe el uso de declaraciones de condicionales (**if /else**, **if /else if** y **switch**), bucles (**for**, **while**), y las expresiones de branch (**break**, **continue**, **return**).

If/Else y If/Else If

If/else ejecuta un bloque de codigo si una condición se evalua como true, en caso contrario se ejecuta otro bloque. La estructura es la siguiente:

```
if (someExpression)
//some code
else
//some code
```

También se pueden encadenar evaluaciones

```
if (someExpression)
//some code
else if (someExpression)
//some code
```

En el siguiente código mediante condicionales if evaluamos el número de versión de Android introducida por el usuario y posteriormente se muestra su nombre

```
import java.util.Scanner;


public class TestIf {

        public static void main(String[] args) {
                int androidVersionNumber = 1;
                String androidVersion="";
                Scanner in;
                in = new Scanner(System.in);
                System.out.println("Enter Android first number
version");
                androidVersionNumber = in.nextInt();
                if (androidVersionNumber == 1) {
                        androidVersion = "Donut";
                }else if(androidVersionNumber == 2){
                          androidVersion = "Eclair - Froyo -
Gingerbread";
                }else if(androidVersionNumber == 3){
                        androidVersion = "Honeycomb";
                }else if(androidVersionNumber == 4){
                        androidVersion = "Ice Cream Sandwich -
Jelly Bean - KitKat";
                }else if(androidVersionNumber == 5){
                        androidVersion = "Lollipop";
                }else{
                          androidVersion = "Invalid number
version";
                }
                    System.out.println("Version -> " +
androidVersion);
```

```
        }
}
```

Switch

La sentencia switch puede tener un número de posibles rutas de ejecución. Switch trabaja con los tipos de datos primitivos byte, short, char e int. En el siguiente código, dependiendo del número introducido por el usuario, se ejecuta una ruta en concreto mediante la cual finalmente se muestra el nombre de una versión de Android

```
import java.util.Scanner;


public class TestSwitch {

        public static void main(String[] args) {
                int androidVersionNumber = 1;
                String androidVersion="";
                Scanner in;
                in = new Scanner(System.in);
                System.out.println("Enter Android first number
version");
                androidVersionNumber = in.nextInt();
                switch (androidVersionNumber) {
                        case 1:
                                androidVersion = "Donut";
                                break;
                        case 2:
                                 androidVersion = "Eclair -
Froyo - Gingerbread";
                                break;
                        case 3:
                                androidVersion = "Honeycomb";
                                break;
                        case 4:
                                 androidVersion = "Ice Cream
Sandwich - Jelly Bean - KitKat";
                                break;
```

```
                        case 5:
                                androidVersion = "Lollipop";
                                break;
                        default:
                                  androidVersion = "Invalid
number version";
                break;
                }
                    System.out.println("Version -> " +
androidVersion);
        }
}
```

While Loop

Una sentencia de bucle while ejecuta continuamente un bloque de código mientras una condición es evaluada como true. Su sintaxis se puede expresar como:

```
while (expresión) {
statement (s)
}
```

El siguiente código itera sobre todas las posiciones del array androidVersionsNames (este contiene el nombre de las versiones de Android). En cada iteración del bucle se comprueba e incrementa la variable counter

```
public class TestWhile {

        public static void main(String[] args) {
                String[] androidVersionsNames = new String[5];
                int counter = 0;
                androidVersionsNames[0] = "Donut";
                 androidVersionsNames[1] = "Eclair - Froyo -
Gingerbread";
                androidVersionsNames[2] = "Honeycomb";
                 androidVersionsNames[3] = "Ice Cream Sandwich
- Jelly Bean - KitKat";
                androidVersionsNames[4] = "Lollipop";
```

```
                while (counter <= 4){
                        System.out.println("Version name " +
androidVersionsNames[counter]);
                       counter++;
                }
        }
}
```

For Loop

La sentencia proporciona una forma compacta para iterar sobre un rango de valores. Se ejecuta en bucle varias veces hasta que una condición se cumple. La forma general de la sentencia for se puede expresar de la siguiente manera:

```
for (inicialización; terminación condición; incremento) {
statement (s)
}
```

El siguiente código ejecuta un bucle for que itera por todas los posiciones del array androidVersionsNames (este contiene el nombre de las versiones de Android)

```
public class TestFor {
        public static void main(String[] args) {

                String[] androidVersionsNames = new String[5];
                androidVersionsNames[0] = "Donut";
                 androidVersionsNames[1] = "Eclair - Froyo -
Gingerbread";
                androidVersionsNames[2] = "Honeycomb";
                 androidVersionsNames[3] = "Ice Cream Sandwich
- Jelly Bean - KitKat";
                androidVersionsNames[4] = "Lollipop";
                for (int counter= 0;counter <= 4; counter++){
                        System.out.println("Version name " +
androidVersionsNames[counter]);
                }
        }
}
```

Observaciones
Bien ya tenemos una primera aproximación al lenguaje Java y sus principales características. Volvemos a repetir este no es un tutorial genérico, es un tutorial especifico encarado a desarrollo de aplicaciones Android. Por eso se obviarán muchas partes como por ejemplo Threads, entornos gráficos (awt)…

Ruben.

Tutorial básico de GIT – Parte 3

Introducción

En este cuarto, y último post, vamos a trabajar la operación merge en base a los conceptos introducidos en el anterior post. Crearemos un un conflicto manualmente y seguidamente lo resolveremos.

- Crear un conflicto
- Resolver un conflicto

Crear un conflicto

En esta sección, vamos a aprender a resolver un conflicto. Para ello vamos a crear manualmente un conflicto utilizando nuestros dos repositorios existentes «tutorial» y «tutorial2».

Trabajando en tutorial

En primer lugar, abriremos el archivo «sample.txt» en el directorio «tutorial». Añadiremos texto al fichero y haremos un commit.

```
$ echo "Each language has its purpose, but do not program in COBOL if you can avoid it." >> sample.txt
$ git add sample.txt
$ git commit -m "added new programming advice"
[master ef95d28] added new programming advice
 1 file changed, 1 insertion(+)
```

Trabajando en tutorial2

A continuación, abriremos el archivo «sample.txt» en el directorio «tutorial2». Añadiremos texto al fichero y haremos un commit.

```
$ echo "The spirit and intent of the program should be retained throughout." >> sample.txt
$ git add sample.txt
$ git commit -m "added new coding advice"
[master f4ddc9c] added new coding advice
 1 file changed, 1 insertion(+)
```

Ahora haremos un push para subir los cambios de «tutorial2» al repositorio remoto.

```
$ git push
Username:
Password:
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 356 bytes | 0 bytes/s, done.
```

```
Total 3 (delta 0), reused 0 (delta 0)
To https://gitlab.com/nebur/tutorial.git
   178cb16..f4ddc9c  master -> master
```

En nuestro repositorio remoto actual, el archivo «sample.txt» contiene la tercera línea «The spirit and intent of the program should be retained throughout.» y se ha realizado el commit al histórico.

Trabajando en tutorial

Ahora, vamos a hacer un push del commit realizado en nuestro repositorio «tutorial» al repositorio remoto.

```
$ git push
Username:
Password:
To https://gitlab.com/nebur/tutorial.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://gitlab.com/nebur/tutorial.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Como podemos observar, Git retorna un conflicto y rechaza el push.

Resolver un conflicto

Con el fin de realizar el push del cambio en «tutorial» al repositorio remoto, vamos a tener que resolver el conflicto manualmente. Vamos a ejecutar un pull para adquirir los cambios más recientes desdel repositorio remoto.

Trabajando en tutorial

Ejecutaremos el siguiente comando

```
$ git pull origin master
```

```
Username:
Password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://gitlab.com/nebur/tutorial
 * branch            master     -> FETCH_HEAD
   178cb16..f4ddc9c  master     -> origin/master
Auto-merging sample.txt
CONFLICT (content): Merge conflict in sample.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Debería aparecer un mensaje advirtiéndonos de un conflicto de merge.

En abrir «sample.txt», podremos ver una sección del archivo delimitada por marcadores que ha añadido Git para indicarnos donde se encuentra el conflicto.

```
$ cat sample.txt
After three days without programming, life becomes meaningless.
new line text
<<<<<<< HEAD Each language has its purpose, but do not program in COBOL if you can avoid it. ======= The spirit and intent of the program should be retained throughout. >>>>>>> f4ddc9c9a3d3329f8ad799ae582adf9efe631211
```

Vamos a resolver el conflicto aceptando los cambios y quitando el marcador.

```
$ cat sample.txt
After three days without programming, life becomes meaningless.
```

```
new line text
Each language has its purpose, but do not program in COBOL if you can avoid it.
The spirit and intent of the program should be retained throughout.
```

Una vez editado el fichero podremos proceder a realizar el commit

```
$ git add sample.txt
$ git commit -m "merge"
[master 4351226] merge
```

Ahora estamos al día con los últimos cambios del repositorio remoto.
Podemos comprobar la integridad del histórico del repositorio usando el comando «log». La opción –graph mostrará el histórico del branch en un formato gráfico y la opción –oneline tratará de compactar el mensaje de salida.

```
$ git log --graph --oneline
*   4351226 merge
|\
| * f4ddc9c added new coding advice
* | ef95d28 added new programming advice
|/
* 178cb16 new line appened
* 0742011 first commit
```

Esto indica que las dos historias se han fusionado de manera segura con un nuevo commit del merge realizado a mano.

Podemos hacer un push con la seguridad seguridad de que este cambio no provocará un conflicto en el repositorio remoto.

Observaciones

Y con este último post damos por finalizada esta serie de

tutoriales sobre Git. De una forma básica y cercana hemos visto como trabajar con este sistema de control de versiones. No hemos mostrado todo su potencial, pero si introducido conceptos básico y las herramientas para empezar a trabajar con él. En una nueva serie profundizarmos en conceptos mucho más potente, por ahora podemos empezar experimentar con el.

Ruben.

Tutorial básico de GIT – Parte 2

Introducción

En este tercer post vamos a trabajar la comparación con un repositorio remoto e introduciremos el concepto merge. Como servicio de repositorio Git remoto usaremos GitLab por algunas razones. Principalmente porqué es un servicio liviano, sin añadidos, simple, con gestión de reportes de errores, wiki y que nos permite realizar repositorios privados. Es simple (en comparación con otros servicios mas potentes como GitHub) y nos permite realizar repositorios privados, es decir perfecto para empezar a hacer pruebas y que estas no sean visible a

toda la comunidad (pues es innecesario)

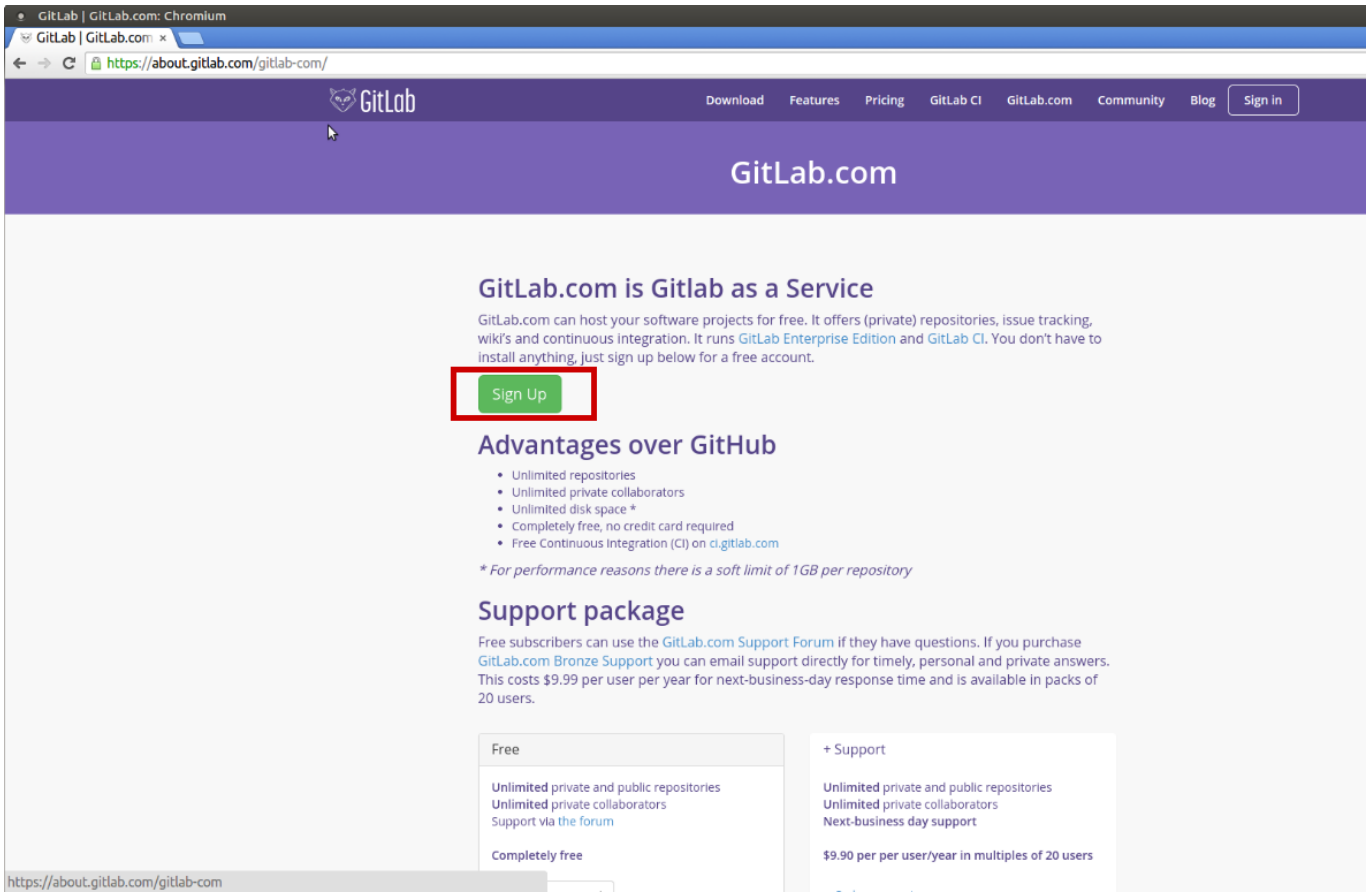
Vamos a tratar:

- Compartir un repositorio
  - Crear un repositorio remoto en GitLab
  - Hacer push a un repositorio remoto
  - Clonar un repositorio remoto
  - Realizar un push des de un repositorio clonado
  - Hacer pull des de un repositorio
- Merge de histórico
  - Hacer merge de un histórico
  - Resolver un conflicto

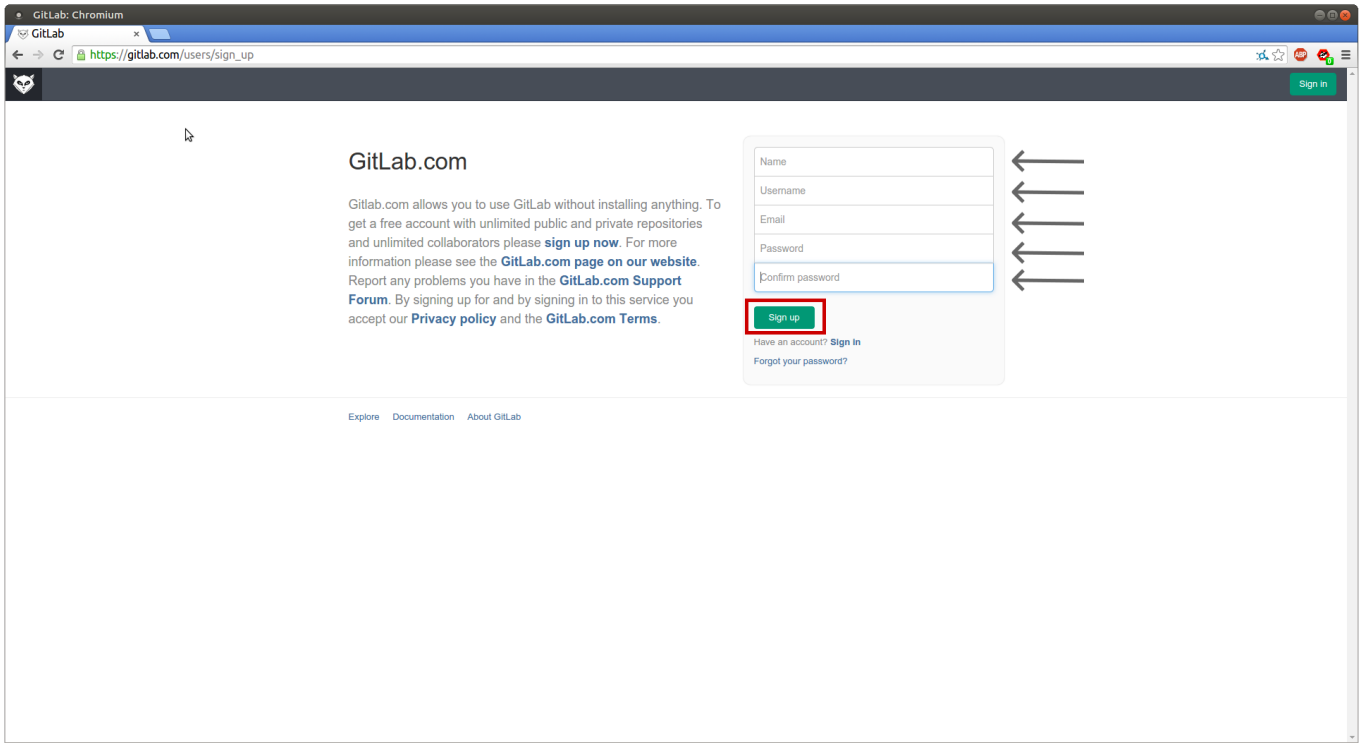
Compartir un repositorio

Crear un repositorio remoto en GitLab

Como hemos comentado antes crearemos el repositorio en GitLab. Primera mente accederemos a la web del servicio y crearemos un usuario des de la siguiente url: https://about.gitlab.com/gitlab-com/

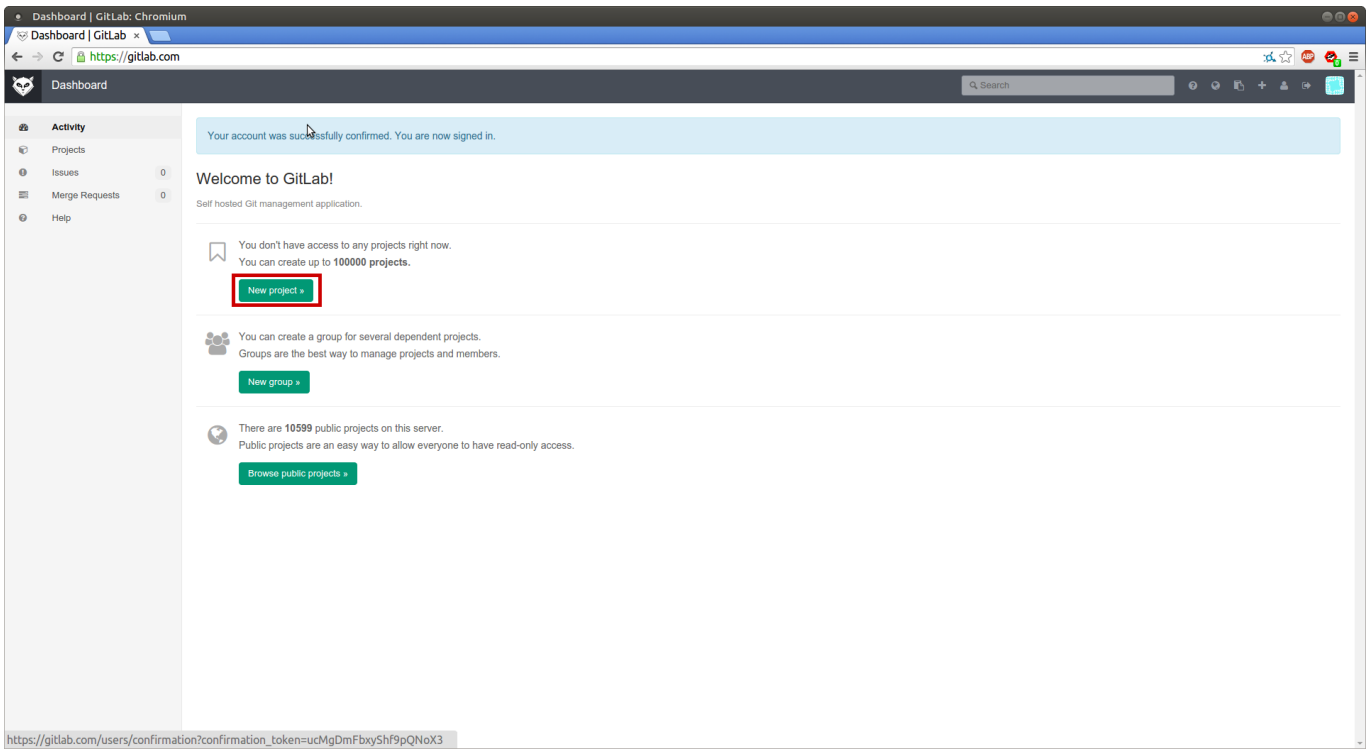


Rellenamos los campos necesarios para crear nuestra cuenta:

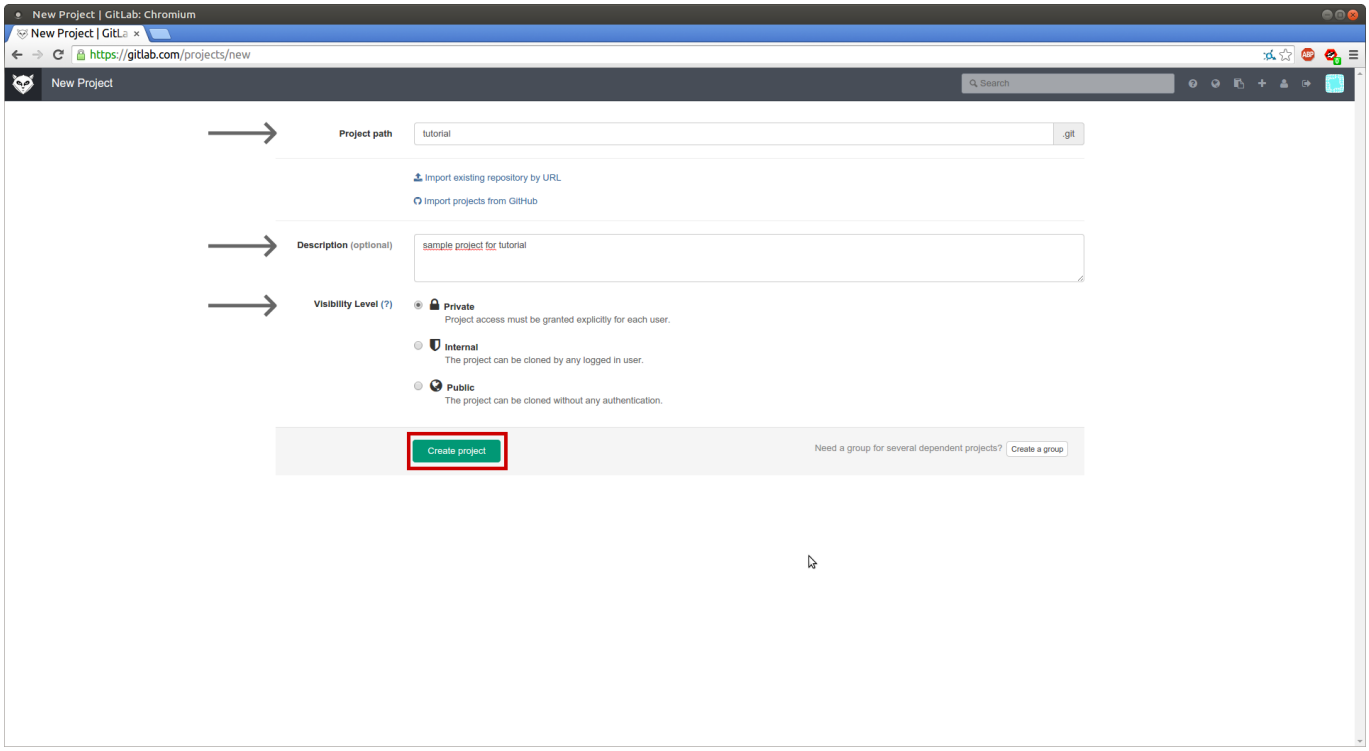


Una vez creada la cuenta entraremos con nuestro usuario y

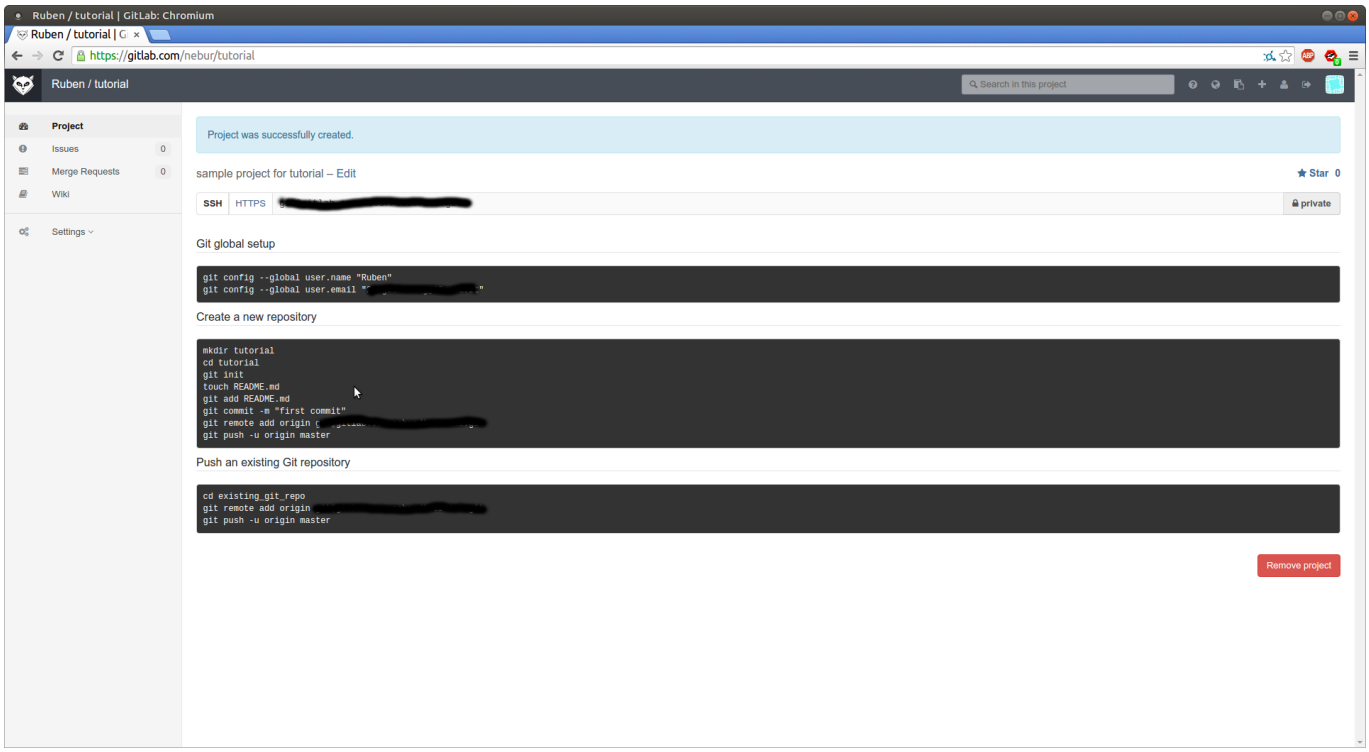
crearemos un proyecto nuevo. El proyecto será privado, puesto que no tienen ningún interés para la comunidad un repositorio donde se hacen pruebas:



Rellenamos los campos necesarios para crear nuestra nuestro repositorio como se muestra a continuación:



Finalmente podemos observar que nuestro repositorio se ha creado correctamente:



Hacer push a un repositorio remoto

Haremos un push del repositorio local «tutorial» que hemos creado anteriormente en la primera parte del tutorial básico de Git.

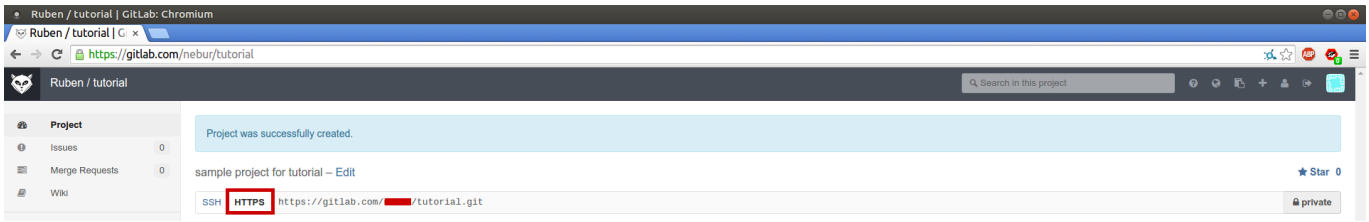
Podemos utilizar un alias o apodo para un repositorio remoto. Esto es útil ya que no necesitamos recordar la larga dirección del repositorio remoto cada vez que tenemos la intención de hacer un push. En este tutorial, vamos a registrar un nombre de repositorio remoto como «origin».

Para añadir un repositorio remoto, utilizaremos el comando «remote». <name> se utiliza como un alias de un repositorio remoto, seguido de <url> con la URL del repositorio remoto.

$ git remote add

Ejecutamos el comando utilizando la url del repositorio repoto que hemos creados anteriormente. El nuevo repositorio remoto

tendrá el alias «origin»




```
$ git remote add origin https://gitlab.com/[user_name]/tutorial.git
```

El repositorio remoto llamado «origin» se utiliza por defecto si se omite el nombre remoto al hacer push/pull. Esto se debe a «origin» es comúnmente usado como un nombre remoto por convención.

Para hacer un push de nuestros cambios al repositorio remoto, utilizaremos el comando «push». Asignaremos la dirección en <repository> y el nombre del branch en <refspec>, al cual queremos hacer el push. Hablaremos de lo branchs en Git en el tutorial avanzado de Git más adelante.

```
$ git push  ...
```

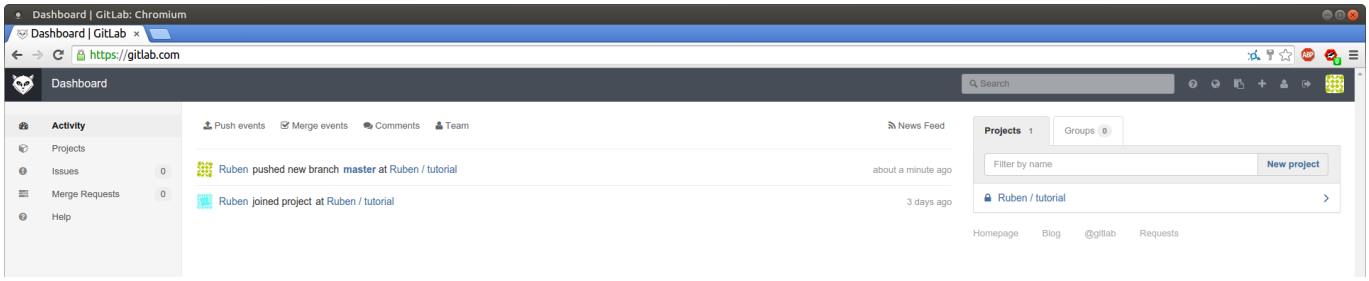
Ejecutaremos el siguiente comando para insertar un commit al repositorio remoto «origin». Si especifica la opción -u al ejecutar el comando, se puede omitir el nombre del branch la próxima vez que se hagamos un push al repositorio remoto. Cuando empujas a un remoto vacante sin embargo, se debe especificar el repositorio remoto y nombre de la rama.

Cuando se nos pregunte por el nombre de usuario y contraseña, introduzca los creados en GitLab.

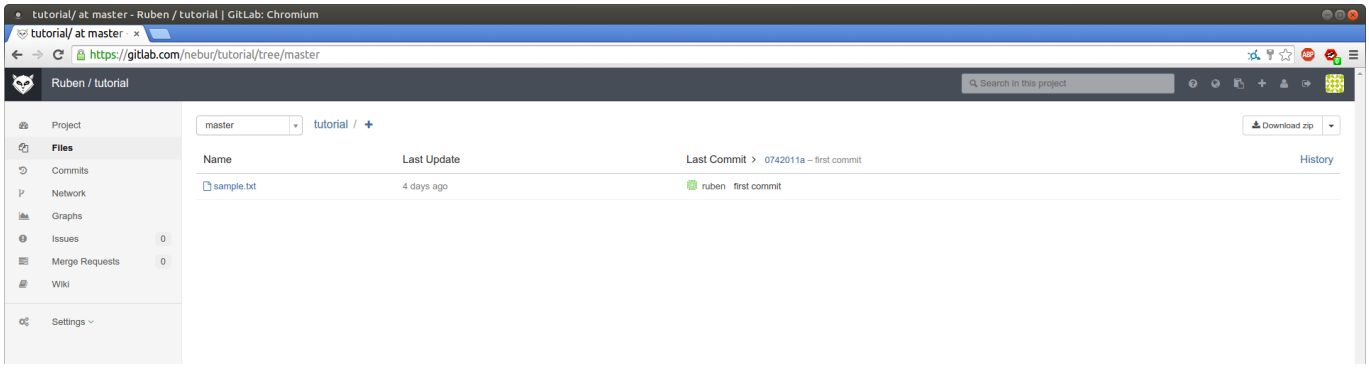
```
$ git push -u origin master
Username:
Password:
Counting objects: 3, done.
Writing objects: 100% (3/3), 245 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://monkey.backlogtool.com/git/BLGGIT/tutorial.git
```

```
 * [new branch]      master -> master
```

Abra la página de Git en Cartera. Usted encontrará que una nueva actualización que corresponde a su empuje al repositorio remoto se ha incluido en las actualizaciones recientes.



El archivo que hemos subido mediante push también se ha añadido en la lista de archivos del repositorio remoto y podemos observar el comentario.



Clonar un repositorio remoto

Realizaremos una copia de un repositorio remoto para poder empezar a trabajar con él en el equipo local.

Ahora, vamos a asumir el papel de otro miembro en el equipo y clonar el repositorio remoto existente en otro directorio llamado «tutorial 2».

Para ello utilizaremos el comando «clon» para copiar un repositorio remoto como se muestra en el siguiente ejemplo. Substitiremos <repository> con la URL del repositorio remoto y <directory> con el nombre del nuevo directorio en el que se descargaran los contenidos remotos.

```
$ git clone <repository> <directory>
```

Ejecutando el siguiente comando, el repositorio remoto se copiará en el directorio tutorial2.

```
$ git clone  https://gitlab.com/<user_name>/tutorial.git tutorial2
Cloning into 'tutorial2'...
Username:
Password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

Para comprobar que la clonación se ha ejecutado correctamente, ejecutaremos el siguiente comando para comprobar el contenido de sample.txt del directorio clonado «tutorial2».

```
$ cat sample.txt
After three days without programming, life becomes meaningless.
```

Realizar un push des de un repositorio clonado

Vamos a realizar un push des de un reporitorio clonado. Para ello trabajaremos en el repositorio clonado **tutorial2**

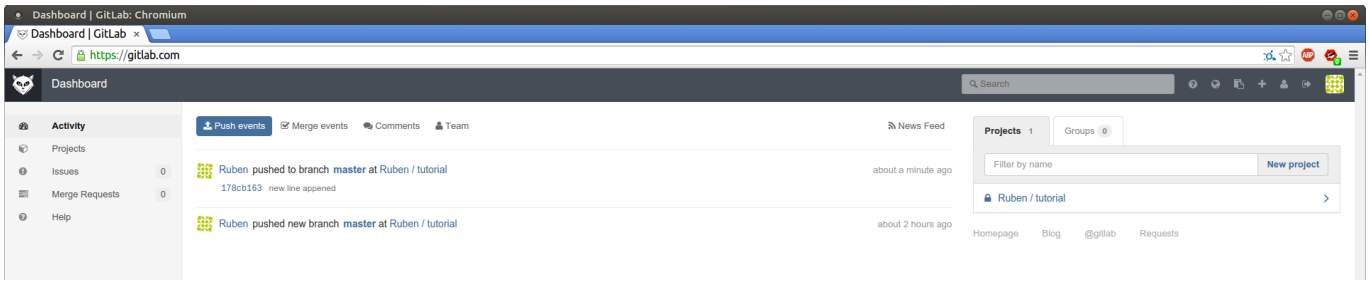
Primeramente añadiremos texto al fichero sample.txt

```
echo "new line text" >> sample.txt
$ git add sample.txt
$ git commit -m "new line appened"
[master 178cb16] new line appened
 1 file changed, 1 insertion(+)
```

Ahora realizaremos el push de este nuevo commit al repositorio remoto. Podemos omitir el repositorio y el branch cuando hacemos un push en el directorio de un repositorio clonado.

```
$git push
Username:
Password:
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 282 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://gitlab.com/nebur/tutorial.git
   0742011..178cb16  master -> master
```

Podemos observar el nuevo push en GitLab. Podremos verlo en el Dashboard.



Hacer pull des de un repositorio

En esta sección, vamos a hacer un pull del último cambio del repositorio remoto a nuestro repositorio local (**en el directorio tutorial**).

Ahora que nuestro repositorio remoto está actualizado con los cambios de «tutorial2», vamos a hacer un pull para obtener el cambio y sincronizar nuestro repositorio inicial en el directorio «tutorial».

Para ejecutar un pull, utilizaremos el comando «pull». Si no se incluye el nombre del repositorio, el pull se hará en el repositorio con alias «origin».

```
$ git pull  ...
```

Ejecutaremos el siguiente comando

```
$ git pull origin master
```

```
Username:
Password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://gitlab.com/nebur/tutorial
 * branch            master     -> FETCH_HEAD
   0742011..178cb16  master     -> origin/master
Updating 0742011..178cb16
Fast-forward
 sample.txt | 1 +
 1 file changed, 1 insertion(+)
```

Ahora vamos a comprobar que el historico esta actualizado con el comando log

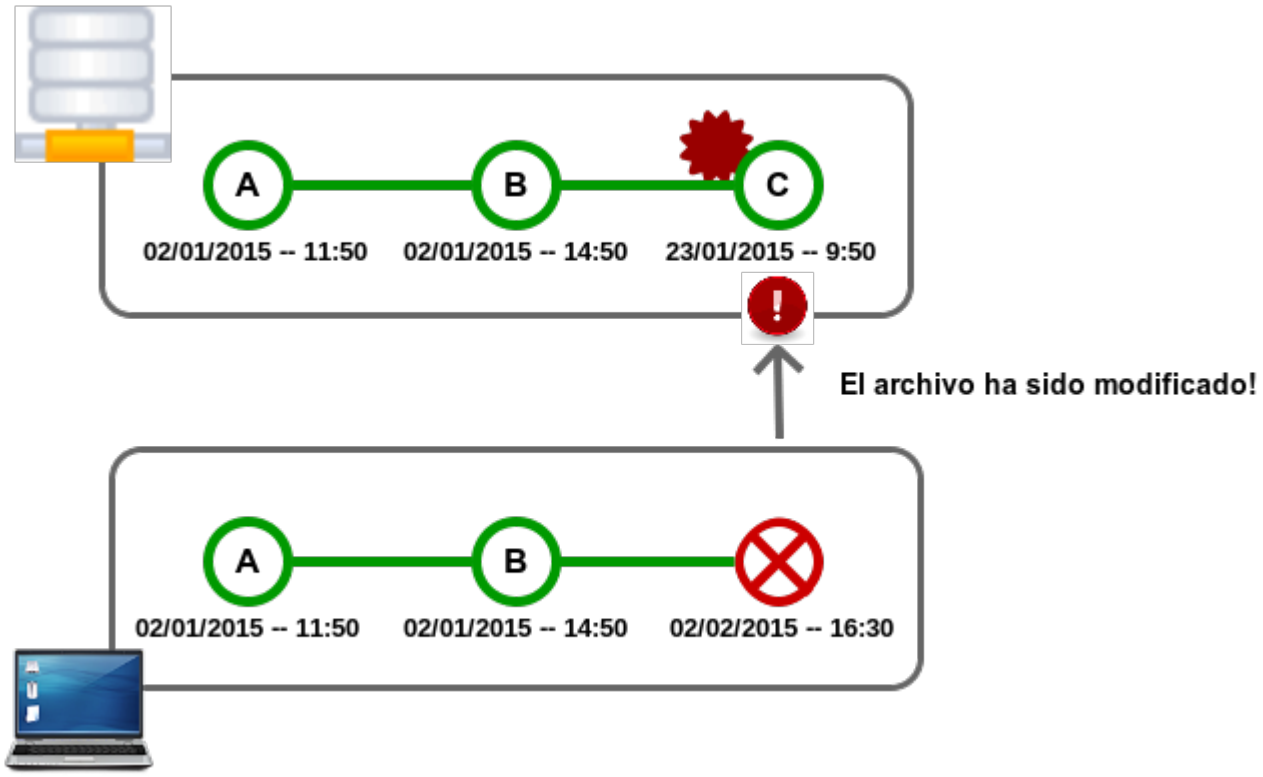
```
$ git log
commit 178cb1635855ea757a3bf6ed748f0096cdc1f6de
Author: ruben <ruben11set@hotmail.com>
Date:   Tue Feb 3 21:13:52 2015 +0100

    new line appened

commit 0742011aaf70e0e3a611fb22500c73d633f755c1
Author: ruben <ruben11set@hotmail.com>
Date:   Fri Jan 30 18:08:42 2015 +0100

    first commit
```

El nuevo commit que hemos añadido en «tutorial2» ahora aparece en la lista de registro de la histórico del repositorio «tutorial». Con el siguiente comando comprobaremos el contenido del fichero sample.txt

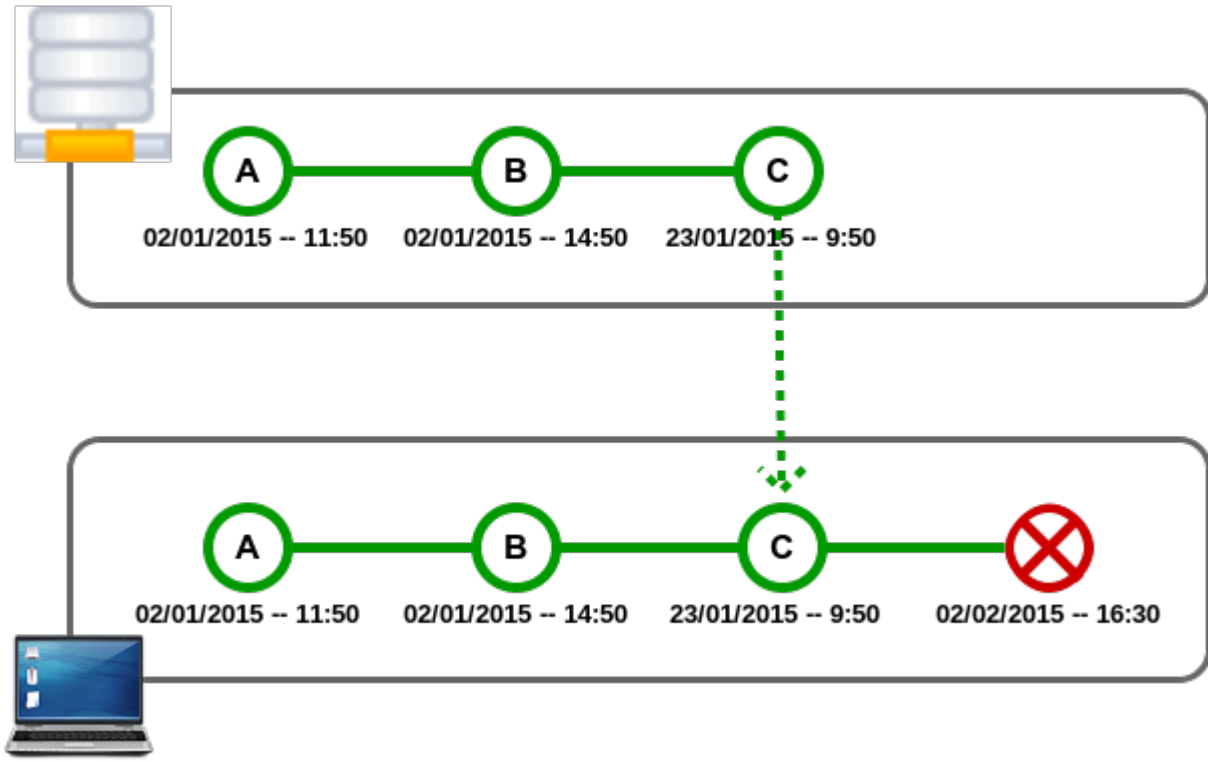
```
$ cat sample.txt
After three days without programming, life becomes meaningless.
new line text
```

Merge de histórico

Hacer merge de un histórico



Un push puede ser rechazado si nuestro repositorio local no está actualizado, posiblemente porque hay algunos cambios añadidos por otros en el repositorio remoto que aún no tenemos en nuestro repositorio local todavía.



Si ese es el caso, hacer un «merge» deberemos obtener el último cambio del repositorio remoto antes de hacer un push. Git de esta manera esto para asegurarse de que los cambios realizados por otros miembros quedan retenidos en el histórico (Commit C en la figura anterior).

Durante un «merge», Git intentará aplicar automáticamente los cambios de la historia y los combina con la rama actual. Sin embargo, si hay un conflicto debido a los cambios, Git retornará un error. En este caso se nos indicará que debemos resolver el conflicto de forma manual.

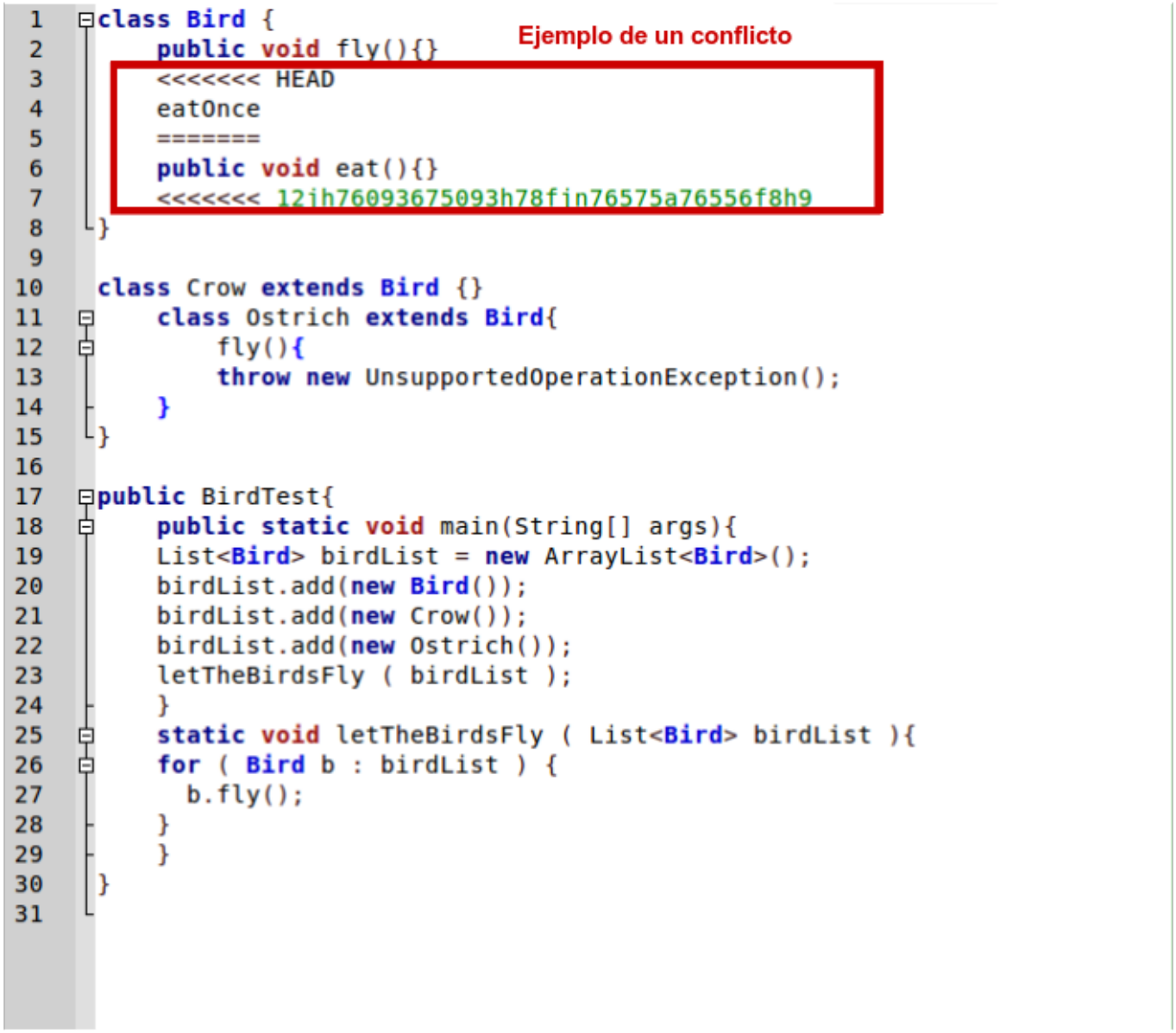
Resolver un conflicto

Como se describe en el apartado anterior, Git intentará aplicar automáticamente los cambios que enlazará con un historial de cambios existente cuando se ejecuta «merge».

Aveces un «merge» puede fallar y puede suceder cuando hay un conflicto. Si dos o más miembros hacen cambios en la misma

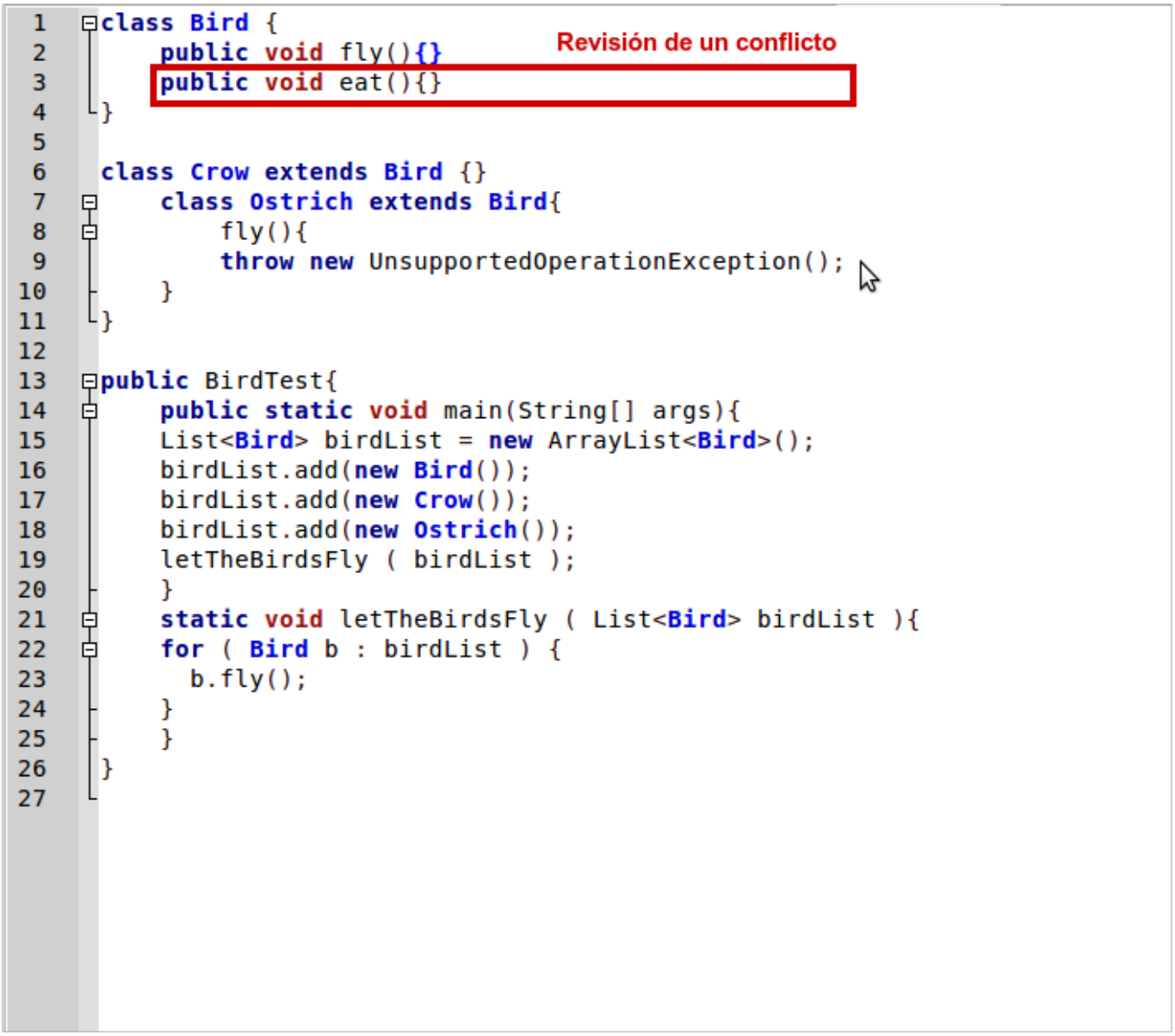
parte de un archivo en las dos branchs (remota y local en este caso) que está siendo fusionadas, Git no será capaz de hacerlo automáticamente y retornará un conflicto de combinación.

Cuando esto suceda, Git agregará algunos marcadores de resolución de conflictos al archivo. Los marcadores actúan como un indicador para ayudarnos a entender las secciones en el contenido del archivo en conflicto que debemos resolver manualmente.



Todo lo anterior «=====» es su contenido local, y todo lo que se encuentra a continuación se trata de la rama remota.

Podemos resolver las partes en conflicto tal y como se muestra a continuación. Ahora ya está listo para proceder con la creación de un «merge commit».



Observaciones

En este tercer post hemos visto como trabajar con un repositorio remoto y hemos introducido el concepto merge. En el próximo, y último post de esta serie, mostraremos paso a paso como trabajar con merge.

Ruben.

Tutorial básico de GIT – Parte 1

Introducción

En este segundo post vamos a preparar el entorno para poder trabajar y realizar algunas operaciones básica. También introduciremos algunos conceptos nuevos para trabajar con repositorios remotos, los cuales profundizaremos en el próximo post

Vamos a tratar:

- Operaciones básicas
  - Instalando Git
  - Configuraciones por defecto
  - Crear un nuevo repositorio
  - Hacer commit con un fichero
- Trabajar con un repositorio remoto

- Hacer push a un repositorio remoto
- Hacer clone de un repositorio remoto
- Hacer pull de un repositorio remoto

Operaciones básica

Instalando Git

Bien antes de empezar a trabajar con Git deberemos preparar el entorno de trabajo. Esto es realmente fácil simplemente lo instalamos en GNU\Linux haciendo uso del sistema de paquetes

Para sistema basados en rpm como Fedora:

```
$ yum install git
```

Para sistemas basados en deb como Debian:

```
$ sudo apt-get install git
```

Acabada la instalación y a modo de prueba podemos ejecutar el siguiente comando que nos devolverá la versión instalada:

```
git --version
git version 2.1.0
```

Configuraciones por defecto

Ahora, vamos a configurar el nombre de usuario por defecto y dirección de correo electrónico para que Git identifique la persona que commitea los cambios. Esta configuración sólo hay que hacer una vez.

La configuración de la consola Git se guarda en el archivo .gitconfig en el directorio home del usuario. Se puede editar manualmente el archivo, pero en este tutorial vamos a utilizar el comando «config».

```
$ git config --global user.name "<Username>"
$ git config --global user.email "<Email address>"
```

Configuramos la salida de color de Git

```
$ git config --global color.ui auto
```

También puede configurar los alias para los comandos de Git. Por ejemplo, se puede abreviar «checkout» para «co» y usarlo para ejecutar el comando.

```
$ git config --global alias.co checkout
```

Crear un nuevo repositorio

Vamos a empezar por la creación de un nuevo repositorio local. Nuestro objetivo es crear un directorio de pruebas y ponerlo bajo control de versión con Git. Utilizaremos este directorio lo largo del tutorial.

Crearemos un directorio tutorial en cualquier lugar de nuestro equipo. Después accederemos al directorio y utilizaremos el comando «init» para convertir ese directorio en un repositorio Git local.

```
$ git init
```

Mediante los siguientes comandos crearemos el nuevo directorio tutorial en un repositorio Git.

```
$ mkdir ~/tutorial
$ cd ~/tutorial
$ git init
Initialized empty Git repository in /home/yourname/tutorial/.git/
```

Hacer commit de un fichero

En el directorio del tutorial que hemos creado anteriormente, vamos a añadir un nuevo archivo y lo registraremos en el

repositorio.

Crea el archivo «sample.txt» en ese directorio con un texto cualquiera, por ejemplo:

```
echo "After three days without programming, life becomes meaningless." > sample.txt
```

Podemos usar el comando «status» para confirmar el estado de nuestro «working tree» e «index» en Git.

```
$ git status
```

Obteniendo el siguiente resultado

```
On branch master

Initial commit

Untracked files:
  (use "git add ..." to include in what will be committed)

        sample.txt

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

Como podemos deducir de respuesta de estado, «sample.txt» actualmente no se está bajo seguimiento. Tendremos que añadir «sample.txt» en el «index» primeramente antes de poder registrar y trabajar con sus cambios.

Para hacerlo simplemente utilizaremos el comando «add», seguido por el que queremos añadir al «index». Si queremos añadir varios archivos al «index», podemos hacerlo separándolos con espacios.

```
$ git add <file1> <file2>
```

Si queremos añadir todos los ficheros de un directorio al «index» simplemente utilizaremos «.»

```
$ git add .
```

Ahora, vamos a comprobar que «sample.txt» se ha añadido con éxito al «index».

```
$ git add sample.txt
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached ..." to unstage)

        new file:   sample.txt
```

Ahora que «sample.txt» se ha añadido al «index», podemos hacer un commit del archivo. Utilizaremos el comando «commit» tal y como se muestra a continuación.

```
$ git commit -m ""
```

Despues de ejecutar el commit comprobaremos el estado.

```
$ git commit -m "first commit"
[master (root-commit) 0742011] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 sample.txt

$ git status
On branch master
nothing to commit, working directory clean
```

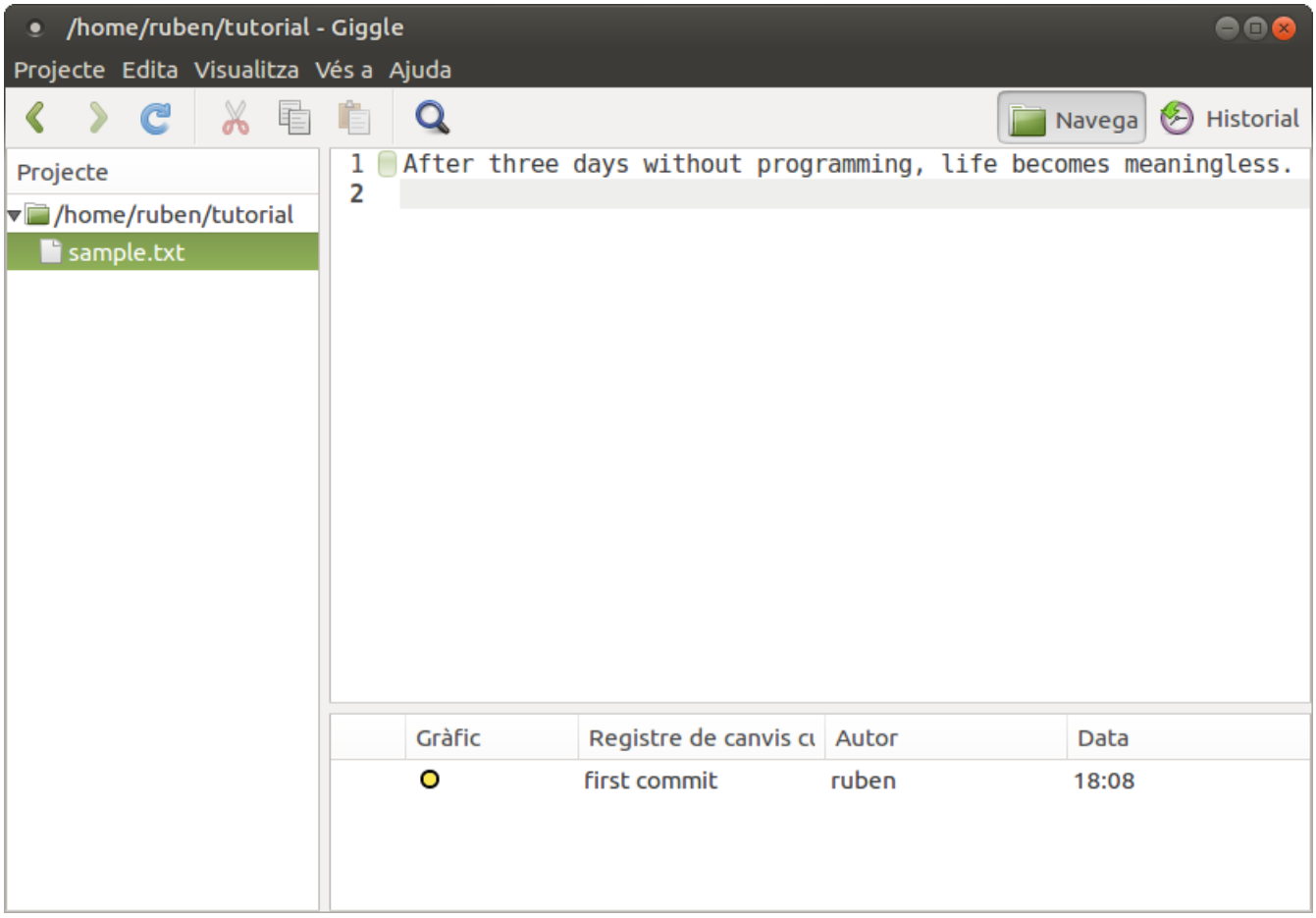
El resultado del comando «status» nos indica que no hay nuevos cambios a ser comiteados.

Podemos ver lo que se acaba de agregar mediante commit en el registro histórico del repositorio con el comando «log».

```
$ git log
commit 0742011aaf70e0e3a611fb22500c73d633f755c1
Author: someone <someone@gmail.com>
Date:   Fri Jan 30 18:08:42 2015 +0100
```

```
    first commit
```

Como referencia diremos que hay varios entorno gráficos que permiten realizar estas tareas. Por ejemplo tenemos Giggle



que podemos instalar fácilmente mediante los siguientes comando:

```
sudo apt-get install giggle giggle-terminal-view-plugin giggle-personal-details-plugin
```

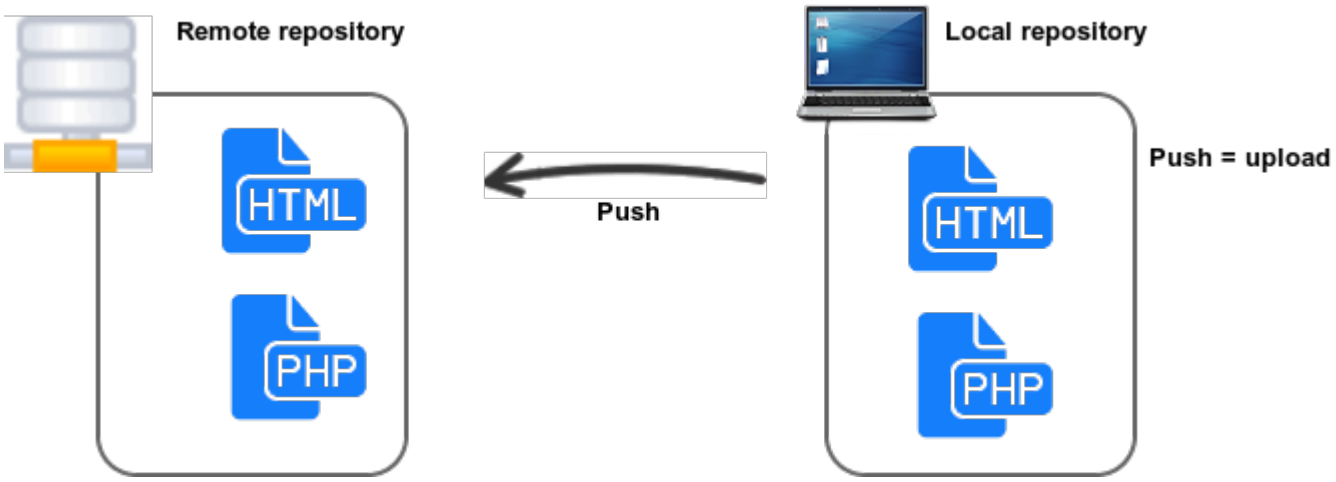
Trabajar con un repositorio remoto

Hacer push a un repositorio remoto

Hasta ahora, sólo hemos trabajado con repositorios locales. A partir de ahora trabajaremos con repositorios remotos que nos permitirán que compartir nuestros cambios con otros miembros

del equipo.

Para empezar a compartir nuestros cambios, tenemos que subirlos a un repositorio remoto con el comando «push». Esto hará que el repositorio remoto se actualice y sincronice con nuestro repositorio local.



Hacer clone de un repositorio remoto

Si ya existe un repositorio remoto, se puede recuperar una copia y guardarla en tu máquina local para y empezar a trabajar.

Se utiliza el comando «clon» para copiar un repositorio remoto. Por defecto, el comando «clon» sería configurar automáticamente una rama master en local a partir de la rama master remota des de la cual se ha realizado el clonado.

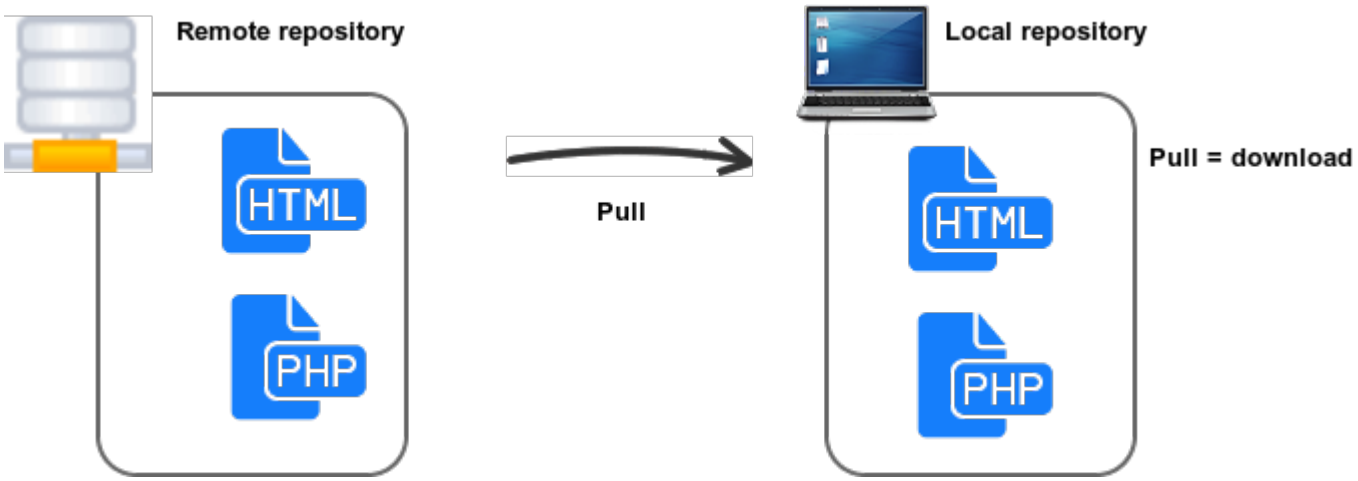
Un repositorio clonado tendrá el mismo registro histórico que el repositorio remoto. Se puede consultar y revertir cualquier de los commits en el repositorio local.

Hacer pull de un repositorio remoto

Cuando se realiza un «push» de los cambios realizados al repositorio remoto compartido, el repositorio local se queda

desincronizado respecto a la última versión remota. Mediante Git, es bastante fácil de sincronizar el repositorio local con el repositorio remoto que ha sido actualizado.

Mediante una operación «pull» se pueden recuperar los cambios que se encuentran en el repositorio remotol. Cuando se ejecuta un «pull», la última revisión se descarga desde el repositorio remoto y se importa al repositorio local.



Observaciones

En este segundo post hemos visto como crear un repositorio trabajar con el, hacer un commit… Todo ello en local. Hemos introducido los conceptos necesarios para trabajar con un repositorio remoto, en el próximo post mostraremos paso a paso como hacerlo.

Ruben.

Tutorial básico de GIT – Introducción

Introducción

Bien este va a ser el primero de un seguido de posts sobre GIT. A través de ellos aprenderemos el uso de Git e instalación.

Este extendido sistema de versiones nos permite trabajar conjuntamente en equipo sobre un mismo programa y guardar revisiones sobre nuestro código, siendo después fácilmente recuperables. Git fue creado por Linus Torvalds (el creador del Kernel de Linux)

En este primer post trataremos:

- ¿Qué es Git?
- Principios de Git
- Beneficios de Git
- Diferencias con otros sistemas de control de versiones
- Repositorios
- Working Tree i Index

¿Qué es Git?

Git es un sistema de control de versiones distribuido (Version

Control System») o herramienta de gestión de código («Code Management tool «). Creado por Lnuis Torval y actualmente utilizado por el equipo de desarrollo del Kernel de GNU\Linux.

Utilizando Git fácilmente puedes revisar el histórico de los códigos fuentes de tu aplicativo y revertir cambios volviendo una versión anterior o comprobar diferencias entre versiones de tus ficheros.

Si la última versión de un archivo se encuentra en un repositorio compartido, Git evitará una sobrescritura no intencionada por lo que mantienen una versión anterior del archivo.

Principios de Git

El código fuente se guarda en un «working directory», donde tienes ficheros «tracked» (ficheros que estan controlados por versiones) y ficheros «untracked» (exluidos del control de versiones, por ejemplo ficheros class generados a partir del código fuente que no son necesarios).

La línea de tiempo de desarrollo se compone de las revisiones («commits» en terminología de Git) del código fuente. Cada commit tiene conocimiento de sus predecesores. Mediante la comparación de un commit con su padre (es decir, la comparación de una revisión a la anterior), uno puede ver los cambios introducidos por el commit hijo.

Un commit contiene los nombres y contenidos de los archivos bajo control de versión, información sobre el autor y el committer (quien realizo el commit y que no tiene por qué ser el autor,), el momento en que se hizo el commit y un mensaje en el se deben indicar cuáles son las razones para los cambios realizados.

Para hacer un commit, primero se debe indicar a Git qué

cambios deben formar parte del nuevo commit mediante un comando add y luego de realizar el commit en sí, seguidamente abrirá un editor en el que podremos escribir un mensaje para indicar la razón por la cual se hicieron dichos cambios.

Beneficios de Git

Git no sólo permite realizar un histórico de tus cambios, permite realizar un seguimiento fácil del desarrollo de otros programadores e integrar esos cambios («merge» en terminología Git).

Aunque nos encontremos en un punto en que nuestro programa ha dejado de funcionar por completo mediante Git podemos saber fácilmente que cambios hemos hecho respecto al commit en el que trabajamos.

Otras ventajas:

- Estudiar el historico de un proyecto para entender no sólo lo que hicieron los desarrolladores, sino también por qué al leer sus mensajes en los commits (es decir, el registro de los cambios).
- Recuperar cualquier revisión anterior (es decir, «regression»), por ejemplo, cuando la versión más reciente contiene bugs que no están presentes en una versión anterior.
- Encontrar fácilmente el commit que introdujo los errores para realizar la regresión.
- Asegurarse de que el código y su histórico nunca se perderán, incluso si nuestro disco duro muere. Cualquier persona con un «clon» del repositorio tiene una copia de toda la historia.
- El trabajo en múltiples funcionalidades o corrección de errores, organizándolas fácilmente y cambiando entre ellas utilizando branches.

El único aspecto negativo de Git es el tiempo de aprendizaje. Una vez se llega a dominar, Git es una gran herramienta para cualquier proyecto de desarrollo y su equipo.

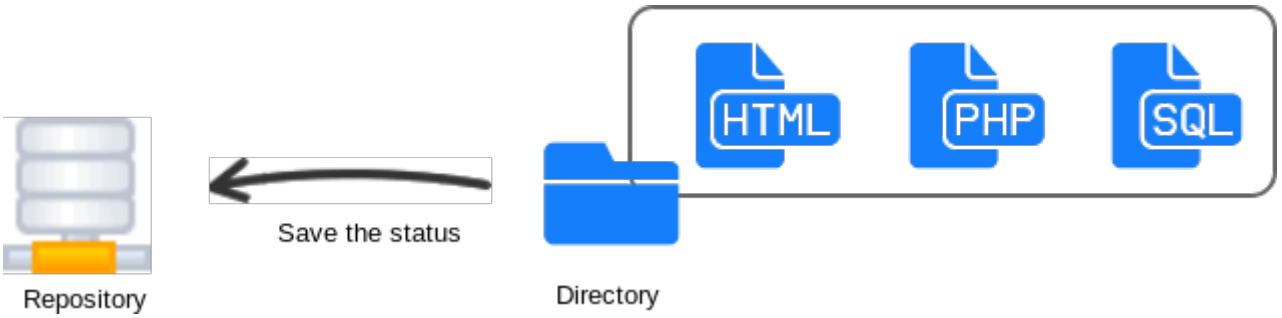
Diferencias con otros sistemas de control de versiones

En comparación con otros sistemas, como CVS o Subversion, nos encontramos con que:

- En Git, cada repositorio es local. Para publicar cambios, es necesario tener un repositorio remoto, también, y hacer un «push» para guardar los cambios allí.
- En Git, ramas son de fáciles de usar y rápidas.
- En Git, imbécil agregar contenido, no los archivos. En otras palabras, cuando el fichero README que ya se realiza un seguimiento, git add README dirá Git que desea que los cambios a los ficheros de ser parte de la próxima confirmación.
- En Git, nunca, nunca intenta integrar los cambios remotos en un estado comprometido. En otras palabras, si usted tiene cambios no confirmados, siempre cometerlos antes de llamar git fetch origen; git merge origin / master.

Repositorios

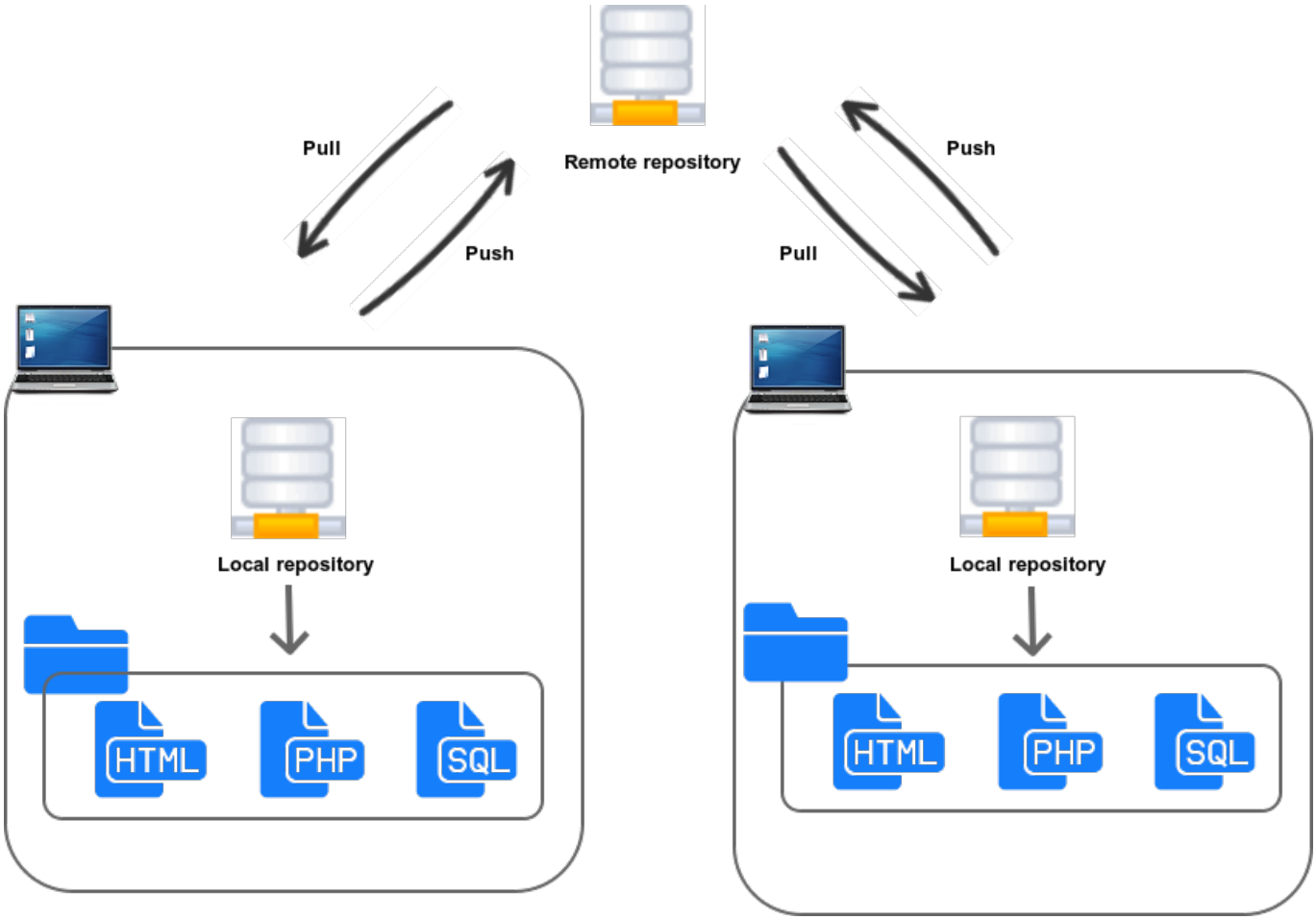
Cuando secrea un repositorio de Git con directorios y ficheros, se puedee guardar sus cambios e el historico y revisar sus estados y versiones.



Existen dos tipos de repositorios:

- Remote repository: Repositorio que reside en un servidor remoto, el cual se comparte con varios miembros del equipo.
- Local repository: Repositorio que reside en un equipo local para uso individual.

Se pueden usar todas las funcionalidades de control de versiones de Git en tu repositorio local (revertir cambios, seguimiento de los cambios, etc.). Sin embargo, cuando se trata de compartir cambios o obtener cambios de miembros de del equipo de desarrollo un repositorio remoto viene muy bien.

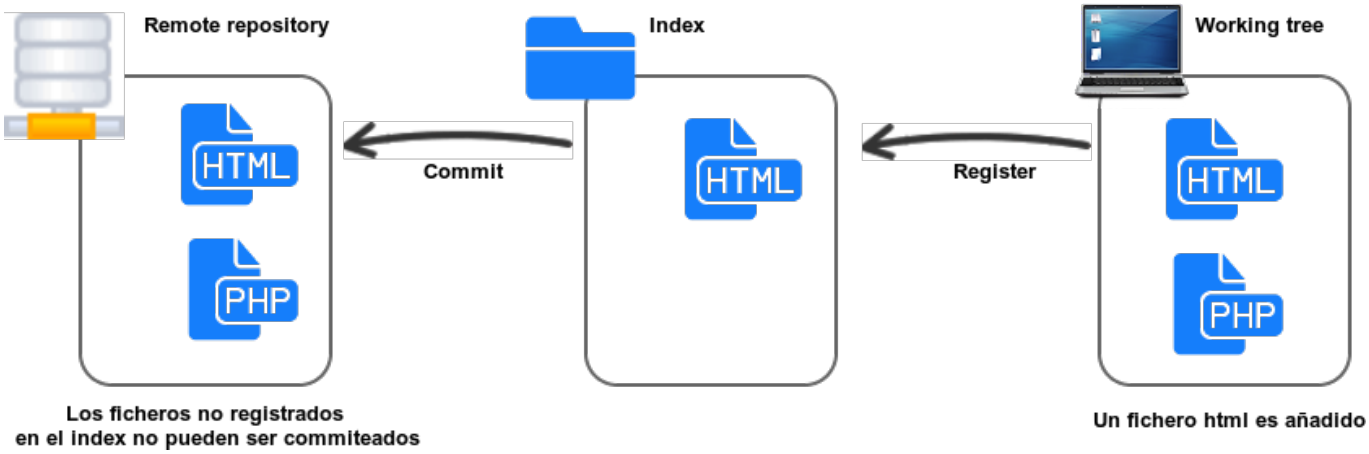


Hay dos formas de crear un repositorio local:

- Puede crear un nuevo repositorio desde cero
- Mediante la clonación obtener un repositorio remoto existente en en nuestro equipo local.

Working Tree y Index

Un «working tree» es un conjunto de archivos con los cuales estas trabajando. Un «index» es un área de ensayo donde se preparan nuevos commits. Actúa como interfaz entre un repositorio y un «working tree».



Los cambios realizados en el «working tree» no serán commiteados directamente al repositorio. Necesitan ser movidos al «index» primero. Todos los cambios que esten en el «index» serán los que realmente se commitearan al repositorio.

Un «index» permite un mejor control sobre qué archivos deben ser incluidos y permite hacer un commit de una parte especifica de un archivo que se encuentra en el «index» al repositorio.

Observaciones

Con está introducción podemos tener un acercamiento a la importancia de los sistemas de versiones y a la versatilidad de Git en concreto. Comenzamos una serie de posts para profundizar en la utilización y explotación de este sistema para mostrar también como puede ayudarnos a trabajar colaborativamente en un proyecto.

Ruben.

Pincipios de programación orientada a objetos

Introducción

La programación orientada a objetos entiende los programas como un conjunto organizado de objetos cooperativos. Cada objeto representa una instancia de una clase. Muchas de estas clases se relacionan mediante herencia o composición. Podemos decir que se focaliza la programación en el tratamiento de la información más que el la algorítmica y la información es parte privada de cada objeto. Solo un objeto sabe como operar con su propia información, lo cual protege la información de cambios indeseados.

Los principios que vamos a tratar son:

- Abstraction
- Encapsulation
- Inheritance
- Composition
- Modularity
- Polymophism

Abstraction

La abstracción denota las características esenciales de un objeto, que lo distinguen de otros objetos. Se centra en las características y operaciones esenciales, haciendo una interpretación codificada del problema que queremos resolver.

Por ejemplo, un coche puede ser visto como un conjunto que funciona. Su abstracción nos llevara a sus características

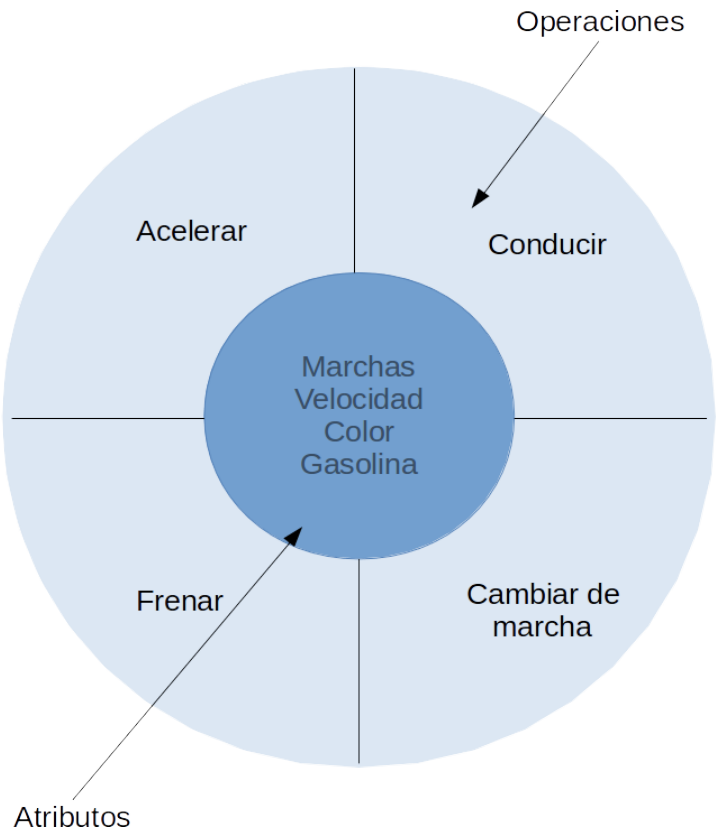
(niveles de gasolina, neumáticos, velocidad, frenos) y a un conjunto de operaciones (acelerar, frenar, repostar). Las operaciones pueden afectar a características (acelerar afectara al nivel de gasolina, a los neumáticos, la velocidad…).

Encapsulation

Consiste en separar la parte contractual de una abstracción y su implementación. Es decir, el proceso de compartimentar los elementos de una abstracción que constituye su estructura y comportamiento.

En la encapsulación, los elementos extraídos de la abstracción son compartimentados de tal manera que no todos serán accesibles des de fuera. Estos elementos pueden ser operaciones o atributos y estos son enlazados de tal manera que ciertos elementos no son accesibles des de fuera (normalmente atributos). De esta manera en vez de dar acceso a atributos directamente se hace des de operaciones donde se controla el acceso.

Cada objeto tiene su propio funcionamiento. Pero este no es visible a los usuarios. Tampoco es necesario y así evitamos problemas (posibles accesos a información sensible). Esto produce que los usuarios no entren en contacto con la implementación, esto permite modificarla sin ningún problema siempre y cuando se mantengan las operaciones intactas.

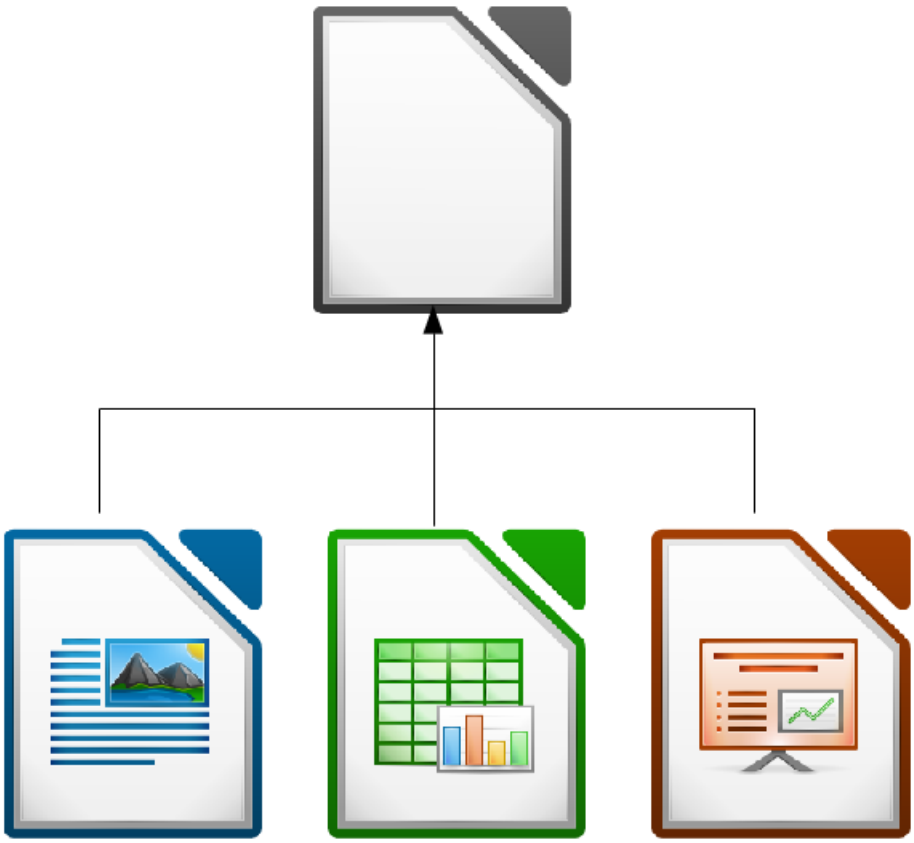


Inheritance

La herencia es la propiedad a través de la cual un objeto «hereda» las funcionalidades y estructura de otro.

Ciertos objetos tiene una misma estructura. En estos casos en vez de implementar esta misma estructura repetidamente, se puede representar en un solo objetos y los demás objetos puede «heredarlo» . Esto es posible cuando se ha creado una relación entre clases. Cuando se crean objetos de la clase el objeto hijo hereda las característica y funcionalidades del objeto padre.

En el siguiente esquema tenemos una clase documento. La clase documento contendrá la cabecera de los documentos y diferentes propiedades comunes de texto. Los documentos de texto, presentación y de hoja de calculo heredarán estas propiedades. De esta manera evitaremos repetir código inútilmente y lo reutilizaremos. Este tipo de relación de herencia se conoce como «is a».



Composition

La composición es una propiedad que permite a un objeto ser compuesto dentro de otro.

La composición facilita la reutilización de código. Se logra mediante la composición de un objeto dentro de otro. Por ejemplo, un objeto que representa un coche va a se compuesto por un objeto motor. Esta relación da lugar a objetos complejos y es vital para aplicaciones de gran tamaño.

Este tipo de relación se conoce como «has a» o «part of». Y existen dos tipos:

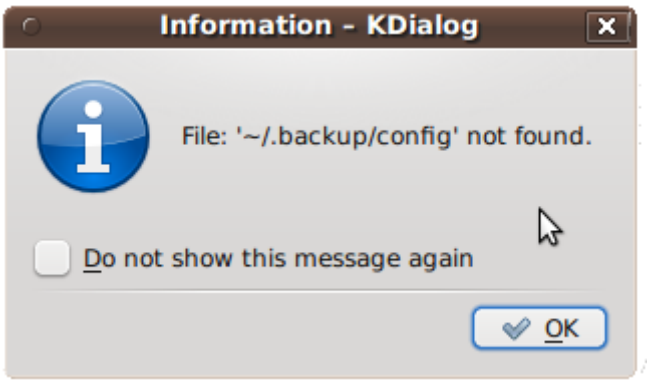
1. Aggregation

Se refiere a una relación débil entre el objeto exterior y el objeto interior. Donde este no depende de la vida útil del objeto exterior. En cuanto a código se refiere esto implica que el objeto exterior debe tener una referencia o puntero al

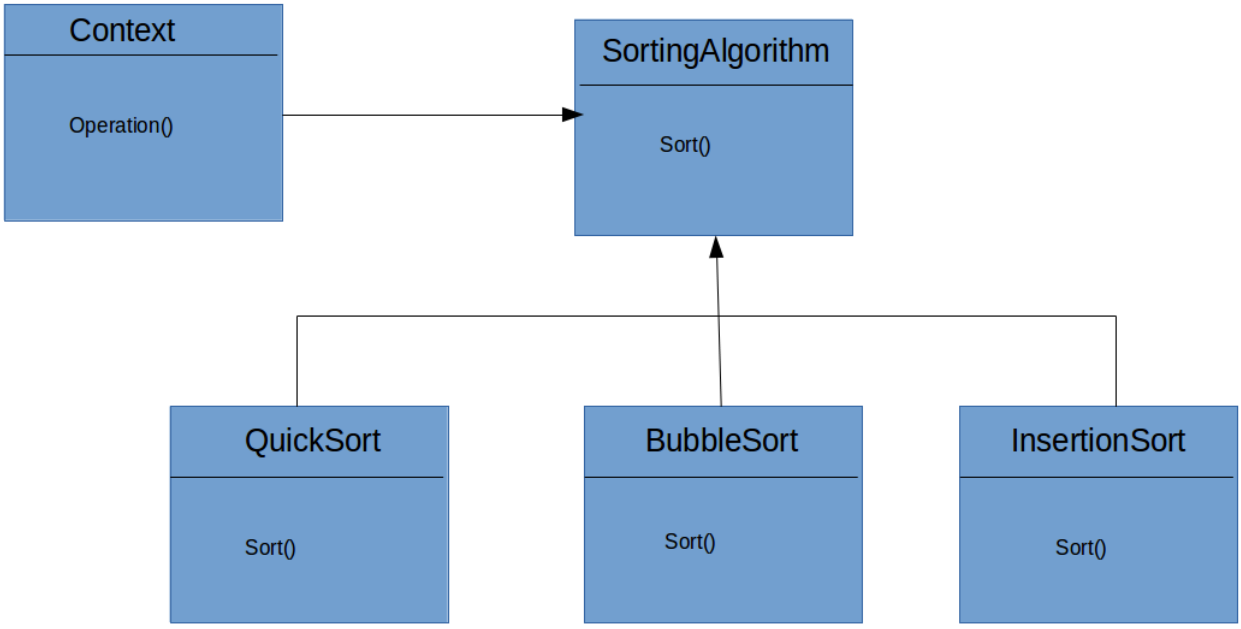
objeto interior. Por ejemplo, una ventana puede utilizar el cursor, pero cuando la ventana se destruye el cursor no.

1. Containment

Se refiere a una relación fuerte entre el objeto exterior y el objeto interior. En este tipo de relación (por ejemplo, en C++ una clase contiene el objeto de otra), la destrucción del objeto exterior implica la destrucción del objeto interior. Por ejemplo, cuando se destruye la ventana todos los controles de que dispone son destruidos (botones, label, edit boxes…)



A diferencia de la herencia, la composición es una relación dinámica porque se crea en tiempo de ejecución. Esto se consigue mediante punteros y referencias. En la figura siguiente, es posible para la clase *Context* utilizar cualquier tipo de algoritmo de ordenación en tiempo de ejecución.



Dada la naturaleza dinámica de la composición es mas flexible que la herencia. Las relaciones entre objetos se pueden crear o romper en tiempo de ejecución.

Modularity

Normalmente, en especial en grandes aplicaciones, tenemos que trabajar con un gran nombre de clases y otros útiles como

parte del código fuente. Desarrollar el código fuente como una única unidad es una tarea complicada i difícil.

por lo tanto, para reducir la complejidad, el código fuente es dividido en módulos pequeños e independientes. Normalmente estos son agrupados de forma lógica (conjuntos de procedimiento o bien la información con la que trabajan), así mantenemos grupos de código ordenados y estructurados.

El objetivo de la modularidad es descomponer el código en pequeñas unidades que faciliten el diseño, desarrollo y test de módulos individuales. Así podemos modificar un módulo sin tener conocimiento de los demás o bien que estos se vean afectados.

En cuanto a mantenimiento se refiere la modularidad es una gran herramienta. Sabes donde buscar exactamente cuando surge un error puesto que todo esta debidamente ordenado y compartimentado. Y una vez solucionado el problema no es necesario realizar tests de todo el código, solo de la parte afectada.

Polymophism

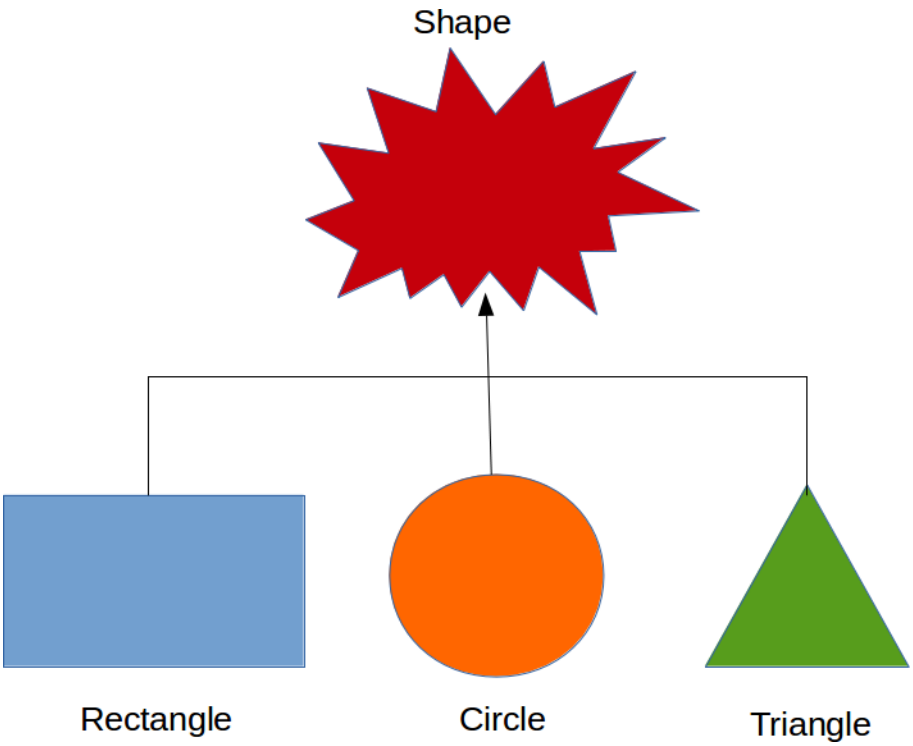
Es una característica de la programación orientada a objetos que denota la posibilidad de asignar un significado diferente para un símbolo particular o «operador» en diferentes contextos.

Un nombre puede referenciar a objetos de diferentes tipos que estn relacionados por características similares. Este nombre debe responder a un conjunto de operaciones y cada operación debe ejecutarse para el tipo adecuado.

En resumen el polimorfismo es la habilidad que tiene diferentes objetos para responder (a su manera) al mismo mensaje

En los lenguajes orientados a objetos el polimorfismo se consigue referenciando (o mediante punteros) al objeto de una subclase por el objeto padre. Las operaciones serán llamadas en la clase base del objeto, pero sern ejecutadas en el objeto correcto automaticamente

En este ejemplo el cliente llama a la función draw() en el objeto padre. La figura correcta se dibuja automáticamente en base a lo que el objeto principal se referencia (podría ser cualquier subclase de Shape)



Hay dos tipos de polimorfismo:

- Static (tiempo de compilación)
- Dynamic (run time)

El **polimorfismo estático** denota que la información requerida esta disponible en tiempo de ejecución. Por lo tanto, las llamadas funciones se pueden resolver en tiempo de compilación. La función exacta a llamar es determinada por el número de parámetros.

El polimorfismo estático se realiza mediante la sobre carga de funciones, operadores de sobrecarga o incluso templates (en c++). Siempre es mas rápido que el dinámico porqué el coste en tiempo de ejecución para resolver que función debe ser llamada se evita.

El **polimorfismo dinámico** denota que la información requerida para llamar la función no se conoce hasta que nos encontramos en tiempo de ejecución. esto se consigue mediante herencia o funciones virtuales.

Si una clase necesita redefinir una función en concreto definido en la clase base, el método es definido como virtual en la clase base y redefinido, para ajustarse a sus necesidades, en la clase derivada. La función virtual en la clase base básicamente define la interficie de la función. Cada clase derivada de ola clase base con las funciones virtuales puede redefinir las funciones con su propia implementación.

Una referencia a una clase base (puntero en C++) se puede usar para apuntar a un objeto de cualquier clase, el método es declarado como virtual en la clase base y redefinido según necesidad. En la clase base se define la interficie de la función. Cada clase derivada de la clase base con métodos virtuales puede redefinir los métodos con su propia implementación.

Dado que la llamada se resuelve en tiempo de ejecución, resultados polimorfismo dinámicos en poco más lenta ejecución del programa.

Observaciones

Espero que este post resulte útil. Es una primera aproximación

a la programación orientada a objetos. En los próximos días introduciremos los principios de diseño de clases. Cabe tener en cuenta que todo esto no es un dogma, mas bien una guía muy útil a la hora de programar.

Ruben.

Foremost – Recuperando archivos php borrados en servidor

Introducción

Bien en este post vamos a mostrar como recuperar archivos php borrados accidentalmente (situación a la que no se debe llegar, puesto que se tiene que contar con un sistema de backup). Para ello utilizaremos Foremost. Lo mas importante llegados a este punto es no usar dicho almacenamiento!!!! Los datos continúan estando en el disco a menos que grabemos un nuevo dato en el mismo sector, en ese caso no habrá posible recuperación. Tampoco podemos tomar esto como la solución a todos nuestros males, el uso de esta herramienta no es garantía de recuperar nuestros datos.

Foremost

Es un programa para hacer carving (rescate selectivo de ficheros).

¿Cómo funciona? Foremost trabaja con imágenes generadas con dd o particiones directamente, y se basa en el análisis de

encabezados y footers de los archivos para ‘extraer’ lo que se pueda salvar. Esto se realiza mediante el encabezado hexadecimal de un fichero, por ejemplo:

- jpg
- gif
- png
- bmp
- avi
- exe
- mpg
- wav
- riff
- wmv
- mov
- pdf
- ole (PowerPoint, Word, Excel, Access y StarWriter)
- doc
- zip
- rar
- htm
- cpp

Instalando Foremost

sudo apt-get install foremost

Configurando Foremost para recuperar archivos php

Vamos a explicar poco la estructura del archivo de configuración. Este consta de líneas en las que se especifican búsquedas

Cada linea tiene la siguiente estructura:

- Extensión del archivo (php, cpp) que se quiera buscar
- Definir si se debe hacerse búsqueda case-sensitive «y» o no «n»

- Tamaño máximo del archivo.
- Encabezado: cadena de texto a buscar en los encabezados de los archivos; puede ser especificado en texto o hexadecimal.
- Footer: cadena de texto a buscar al final de los archivos; puede ser especificado en texto o hexadecimal.

Bien deberemos editar el archivo de configuración

sudo vi /etc/foremost.conf

y añadir la siguiente línea

php y 100000 \x3C\x3F\x70\x68\x70/

Bien para empezar con la recuperación ejecutaremos el siguiente comando:

foremost -t php -i /dev/sda1 -o /dev/sda2/recover/

Teniendo en cuenta que:

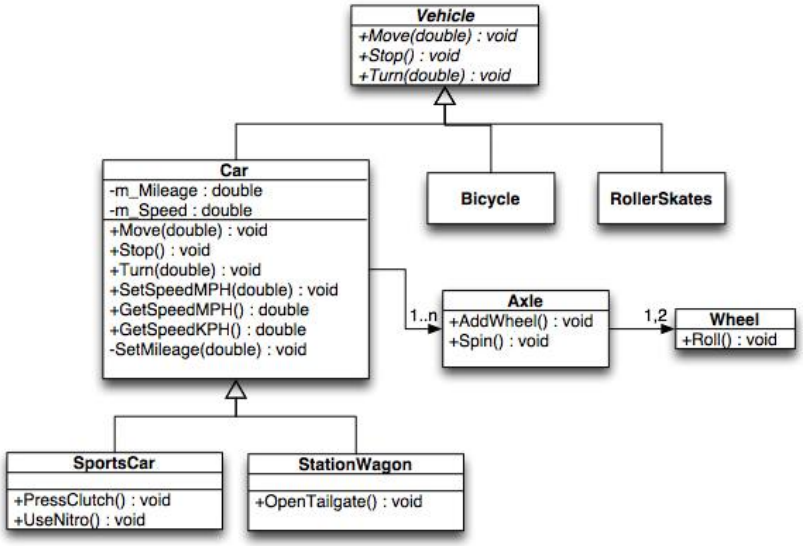
- -t especifica el tipo de archivo a buscar (si no se usa por defecto busca todos los que esten configurados en foremost.conf)
- Deberemos substituir /dev/sda1 por la ruta al disco del cual se quiera recuperar datos
- Deberemos modificar /dev/sda2/recover/ por la ruta donde queremos guardar los archivos recuperados (esta carpeta debe estar vacía)
- Los archivos recuperados no conservaran el nombre original. Nos encontraremos con archivos tipo 6856758.php
- Se debe tener mucha paciencia este proceso no durará segundos

Observaciones
Espero que esto pueda servir de ayuda. Normalmente esto ocurre cuando se no se tiene un sistema de backups. Se debe tener cuidado, hacer periódicamente backups incrementales puede evitar encontrarnos en la tediosa situación de utilizar este

tipo de programas. Que si, deben estar, se agradecen y mucho, pero no son garantía de recuperación y entramos en un terreno delicado nunca es bueno.

Ruben.

Principios del diseño de clases



Introducción

Los principios de diseño nos ayudan a exprimir todo el potencial de la programación orientada a objetos. Permitiéndonos programar software flexible, extensible y escalable. Estos principios suelen conocerse con el acrónimo de SOLID, que se refiere a cada una de las iniciales de estos principios:

- Single Responsabilty
- Open-Closed
- Liskov Substitution
- Inteface Segregation
- Dependency Inversión

Single Responsability

Nunca debe haber más de una razón para un cambiar una clase.

La responsabilidad de una clase es definida como una razón para cambiarla, porque una responsabilidad implica implementación. Y las implementaciones pueden cambiar en el futuro. Si una clase tiene más de una razón para cambiar entonces tiene más de una responsabilidad.

Vamos a ver la siguiente clase que representa el jugador de un equipo

```
public class Player{

        private String name;
        private int weight;
        private int height;
        private int position;

        public Player(String name, int weight, int height, int position){
      this.name = name;
      this.weight = weight;
      this.height = height;
      this.position = position;
        }

        public int calculateBMI(){
                //Implementation
        }

        public void SaveToFile(){
```

```
                //Implementation
        }

    public void LoadFromFile(){
                //Implementation
        }
}
```

Como podemos observar la clase tiene la responsabilidad de trabajar con la información de un jugador. Pero ademas, también de guardar y cargar dicha información.

Como podemos observar las funciones de guardado y carga pretenden hacerlo des de un fichero. Si mas adelante tuviéramos que hacerlo des de una base de datos nos veriamos obligados a modificar el código, teniendo que retestear todo por la posible introducción de errores. Estos errores afectarían a la clase Player y a las que dependan de esta.

Resulta mucho mas util separar las responsabilidades en dos clases diferentes.

```
public class Player{

        private String name;
        private int weight;
        private int height;
        private int position;

        public Player(String name, int weight, int height, int
position){
      this.name = name;
      this.weight = weight;
      this.height = height;
      this.position = position;
        }

        public int calculateBMI(shapeType){
                //Implementation
        }
}
```

```
public class Db{

        private String playerInfo;

         public Db(String playerInfo, int weight, int height,
int position){
      this.playerInfo = playerInfo;
        }

        public void Save(){
                //Implementation
        }

    public void Load(){
                //Implementation
        }
}
```

Open-Closed Principle

Las entidades de software (clases, módulos, funciones…) deben estar preparadas para ser extensibles pero cerradas a modificaciones.

En el siguiente código podemos ver una clase que dibuja figuras y calcula su área:

```
public enum ShapeType {
    CIRCLE, SQUARE, TRIANGLE
}

public class ManageShape{

        private ShapeType shapeType;
    private int factorResize;

        public ManageShape(ShapeType shapeType){
      this.shapeType = shapeType;
        }
```

```
        void calculateArea(shapeType){
                switch(shapeType){
                        case shapeType.CIRCLE:
                                //implementation
                                break;
                        case shapeType.TRIANGLE:
                                //implementation
                                break;
                        case shapeType.SQUARE:
                                //implementation
                                break;
                        default:
                                //implementation
                                break;
                }
        }

        void drawShape(shapeType){
                switch(shapeType){
                        case shapeType.CIRCLE:
                                //implementation
                                break;
                        case shapeType.TRIANGLE:
                                //implementation
                                break;
                        case shapeType.SQUARE:
                                //implementation
                                break;
                        default:
                                //implementation
                                break;
                }
   }
}
```

Podemos encontrarnos con la necesidad de añadir mas figuras en el futuro. Estos cambios pueden introducir errores y causar varios fallos. Nuestro enum deber cambiar también y todos los módulos que depengan de la clase ManageShape con todo lo que ello supone.

El principio «open-closed» nos sugiere programar de la siguiente manera:

- Los módulos están abiertos para su extensión. La funcionalidad puede cambiar añadiendo nuevo código.
- El código existente esta cerrado a modificación y rara vez se tiene que modificar el código existente para cambiar el comportamiento del módulo.

La programación orientada a objetos nos provee de herramientas para implementar el principio «open-close». Una opción is codificar nuestro programa utilizando herencia y polimorfismo. Llevado esto a nuestro código crearemos una estructura en la que cada figura heredara de ManageShape y cada subclase se encargara de mantener su funcionalidad y la enumeración no sera necesaria.

```
public abstract class ManageShape{

    private int factorResize;

        public abstract void calculateArea(){

        }

        public abstract void drawShape(){

   }
}

public class Triangle extends ManageShape{

        void calculateArea(shapeType){

        }

        void drawShape(shapeType){

   }
```

```
}

public class Circle extends ManageShape{

        void calculateArea(shapeType){

        }

        void drawShape(shapeType){

   }
}

public class Square extends ManageShape{

        void calculateArea(shapeType){

        }

        void drawShape(shapeType){

   }
}
```

De esta manera los módulos que necesiten utilizar una figura dependerán de la clase abstracta ManageShape y utilizaran las subclsases o punteros. Si se añaden mas clases de tipo figura el código existente no se tendrá que modificar, no habrá impacto en el resto del código. Otra ventaja es que el código común esta contenido en la clase base y así se evitan duplicidades.

Liskov Substitution

Las funciones que utilizan punteros o referencias a una clases base tienen que poder utilizar objetos de las clases derivadas sin tener conocimiento de ello. Es decir, una clase debe funcionar correctamente cuando se utiliza su sublcase en vez de la clase padre.

Cojamos el siguiente código de ejemplo:

```
class Bird {
        public void fly(){}
        public void eat(){}
}

class Crow extends Bird {}

class Ostrich extends Bird{
}
```

Como podemos obervars la classe Birs tiene los metodos fly() y eat(). Nos encontramos que al crear la clase Ostrich nos sobra el método fly(). Y aquí nos encontramos con el problema al utilizar la clase Ostrich (subclase de Bird) nos encontramos con un método que debería volver una excepció del tipo UnsupportedOperationException().

Una solución sería la siguiente:

```
class Bird{
        void eat();
}

class Crow extends Bird{
        public void fly(){
        }
        @Override
        public void eat() {
                // TODO Auto-generated method stub

        }
}

class Ostrich extends Bird{

        @Override
        public void eat() {
                // TODO Auto-generated method stub

        }
```

```
}
```

Una la clase Bird contiene solo el método eat(), el cual es comun en todas la aves. El método fly() el cual se implementará en la subclase y a su vez Ostrich.

Inteface Segregation

Los clientes no se deben ver forzados a depender de métodos que no utilizan.

Este principio trata de lidiar con interfícies no cohesionadas y excesivamente generalistas.Esto produce que los clientes se vean forzados a depender de métodos que jamas van a utilizar.

Por ejemplo Vamos a poner el caso de una interficie IFile utilizada para operaciones de ficheros. Dos clases heredan de ella TextFile y SocketFile

```
interface IFile {
        void Open();
        void Close();
        void Read();
        void Write();
        void Seek();
        void Position();
}

class TextFile implements IFile {

        public void Open(){
                //Implementation
        }

        public void Close(){
                //Implementation
        }

        public void Read(){
                //Implementation
```

```
        }

        public void Write(){
                //Implementation
        }

        public void Seek(){
                //Implementation
        }

        public void Position(){
                //Implementation
        }

}

class SocketFile implements IFile {

        public void Open(){
                //Implementation
        }

        public void Close(){
                //Implementation
        }

        public void Read(){
                //Implementation
        }

        public void Write(){
                //Implementation
        }

        public void Seek(){
                throw new NotImplementedException();
        }

        public void Position(){
                throw new NotImplementedException();
        }
```

```
}
```

Los métodos seek() y position() no son útiles para SocketFile. The echo está clase no puede implementarlos y por ello se lanza una excepción de tipo NotImplementedException(). Una solución que para nada resulta pulida. La clase puede acceder a los métodos pero para asegurarnos que no va haber ningún problema tenemos que lanzar una excepción.

El principio de segregación de interficies sugiere que una interficie no cohesionadas se deben dividir en interficies pequeñas, para que cada una de estas de servicio al cliente correspondiente.

Aplicando esto a nuestro caso, la interficie IFile se verá dividida en dos interficies

```
interface IFile {
        void Open();
        void Close();
        void Read();
        void Write();
}

interface IDiskFile implements IFile{
        void Seek();
        void Position();
}
```

La interficie IDiskFile será implementada por la clase TextFile, la cual implementara los metodos de las dos interficies (vease que IDiskFile implementa IFile). I SocketFile implementara solo la interficie IFile y no tendrá que preocuparse de métodos que no tulilizará (seek() y position()).

Dependency Inversión

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Y ambos deben depender de la abstracción. La abstracción no debe depender de los detalles. Los detalles deberían depender de las abstracciones

En los diseños de análisis estructurado, el software que se crea tiende a tener una dependencia en la dirección equivocada. Las reglas de negocio dependen de los detalles de implementación de bajo nivel. Estos causa problemas cuando las reglas de negocio cambian y la implementación de bajo nivel se tiene que modificar. Sino que más bien debería ser al revés; la implementación de bajo nivel debe depender de las reglas de negocio. Por lo tanto, la dependencia tiene que ser invertida; de ahí el nombre de principio de dependencia invertida.

Consideremos el ejemplo de un botón que enciende y apaga una lampara

```
class Lamp {

        public void turnOn(){
                //Implementation
        }

        public void turnOff(){
                //Implementation
        }

}

class Button {
        private Lamp lamp = new Lamp();
        private Boolean pressed = false;

        public void pressButton(){
                if(pressed){
                        lamp.turnOff();
                }else{
                        lamp.turnOn();
                        pressed = !pressed;
```

```
                }
        }
}
```

la clase Lamp tiene la implementación de las funcionalidades de la lampara. Està controlada por la clase Button que llama a los métodos turnOn() y turnOff(). Esto es un ejemplo de un diseño estructurado. La logica de control esta en Button y la implementación de bajo nivel en Lamp. De esta manera la clase Button contiene la implementación de alto nivel (logica) y Lamp contiene la implementación de bajo nivel.

Es bastante obvio que la dirección de la dependencia es de Button a Lamp. Si la implementación de Lamp se tiene que modificar, también afectará a la clase Button. Esto significa que la lógica de negocio se ve afectada por la implementación de bajo nivel. Este ejemplo muestra un código muy pequeño pero en una aplicación grande puede generar una modificación en cascada de las clases de alto nivel.

Idealmente, las lógica de negocio define cómo se debe hacer la implementación de bajo nivel. Así, los módulos de bajo nivel idealmente dependen de la lógica de negocio y operan de acuerdo a ellos.Dado esto debemos invertir la dependencia de la clase Button en la clase Lamp.

La solución es la interficie Switchable que contiene el protocolo que todos los objetos deben seguir para controlar la clase Button. La clase Button no interactuará con Switchable en vez de con la clase Lamp.

```
interface Switchable {
        void turnOn();
        void turnOff();
}

class Lamp implements Switchable{
```

```
        public void turnOn(){
                //Implementation
        }

        public void turnOff(){
                //Implementation
        }
}

class Television implements Switchable{

        public void turnOn(){
                //Implementation
        }

        public void turnOff(){
                //Implementation
        }
}

class Button {
        private Switchable switchable = new Lamp();
        private Boolean pressed = false;

        public Button(Switchable s){
                this.switchable = s;
        }

        public void pressButton(){
                if(pressed){
                        this.switchable.turnOff();
                }else{
                        this.switchable.turnOn();
                        pressed = !pressed;
                }
        }
}
```

Ahora, es posible añadir mas objetos que se puedan encender y apagar a traves de Button sin que este tenga que ser modificado. La dependencia se ha invertido, los módulos de

bajo nivel dependen de los módulos de alto nivel. Esto crea una estructura flexible facil de mantener y a la cual se pueden añadir nuevas funcionalidades

Observaciones

En esta segunda aproximación hemos definido los principios del diseño de clases. Este conjunto de reglas nos ayudará a mantener un código limpio, ordenado, fácil de mantener y robusto. Todo ello enfocado a facilitar la nueva entrada de funcionalidades a nuestra aplicación y la corrección de errores.

Ruben.

Mundos tridimensionales

Introducción

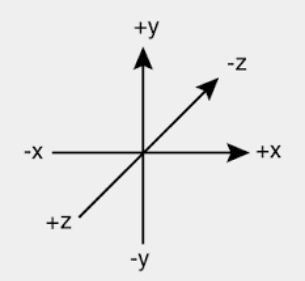
En este post voy a introducir algunos conceptos básicos sobre los mundos tridimensionales. Vamos a enfocarlo des del punto de vista de los motores gráficos para juegos.
Los principales temas que vamos a tratar son:

- Coordenadas
- Espacio local (local space) y mundos (world space)

- Vectores
- Camaras
    - Projection mode (3D vs 2D)
- Plygons, edges, vertices and meshes
- Materials, textures and shader
- Rigidbody dynamics
    - Detección de colisiones

Coordenadas

En los mundos 3D se definen 3 coordenadas:



- Z-axis -> profundidad
- Y-axis -> altura
- X-axis -> anchura

Se representan de la siguiente manera (x,y,zy):

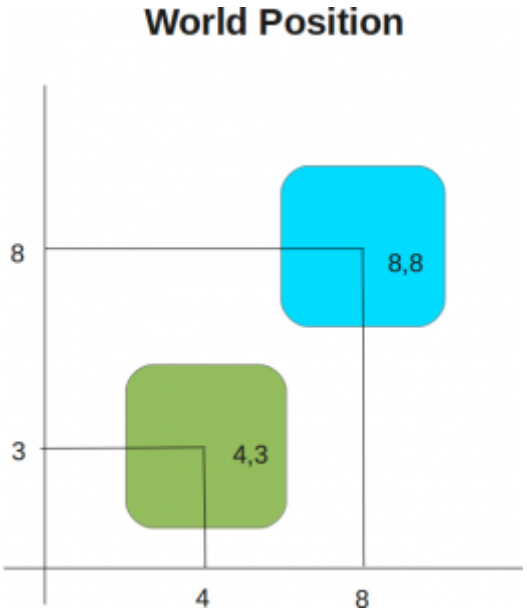
Local space (espacio local) y world space (mundo)

Es muy importante tener claras las diferencias entre local space y world space.

World space

En cualquier entorno 3D el world space será infinito por lo cual para poder referenciar la posición de los objetos deberemos marcar un punto llamado ‘origin’ o ‘world zero’. Este punto será la representación de la posición (0,0,0) .

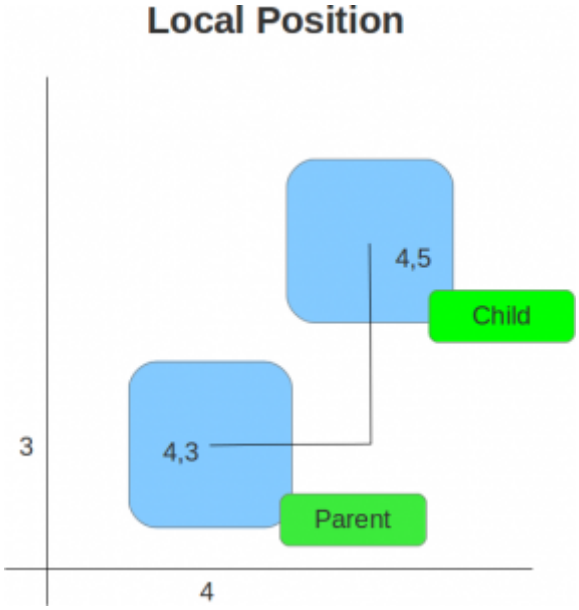
Como podemos ver en esta imagen los dos objetos tienen sus respectivas posiciones en verso el punto (0,0), esto es igualmente aplicable a escenarios 3D.



Local space

Los objetos en un escenario 3D están posicionados en relación al world zero. Pero, para hacer las cosas un poco más fáciles usamos local space (o también llamado object space) para definir la posición de un objeto respecto de otro. Esta relación se conoce como parent-child. Local space asume que cada objeto tiene su posición (0,0,0), desde este punto, normalmente el centro del objeto, se expandirá su volumen. La relación parent-child nos permitirá comparar las posiciones de diferentes objetos entre ellos a través de las relaciones.

Como podemos ver en esta imagen el segundo objeto (child) con posición (4,5) esta situado respecto al objecto (parent) con posición (4,3), esto es igualmente aplicable a escenarios 3D.



Posicionamiento y diseño de assets

Es importante tener en cuenta que si diseñamos assets con alguna herramienta de modelado (por ejemplo blender) tenemos que asegurarnos que la posición en la que creamos el modelo sea la (0,0,0).

Vectores

Los vectores simplemente son lineas dibujada en un escenario 3D con una longitud y dirección. Se pueden mover en el world space sin que estos sufran modificaciones. Son muy útiles en el diseño de juegos 3D, nos permiten:

- Calcular distancias entre objetos
- Ángulos de posición entre ellos
- La dirección de estos

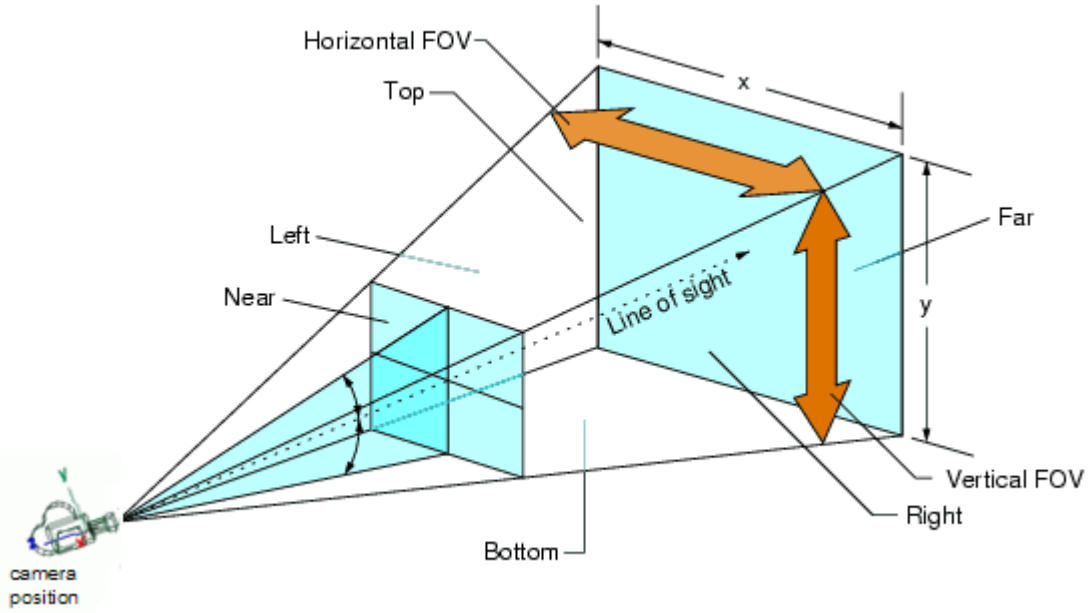
Cámaras

Las cámaras son esenciales en mundos 3D, ya que actúan como el punto de visión. Pueden ser posicionadas en cualquier punto del mundo, animadas, enlazadas a un objeto u objetos que forman parte de un escenario. Varias cámaras pueden existir en una escena particular, pero se asume que una siempre será la «main camera» (cámara principal) que renderizará lo que el

usuario puede visualizar.

Projection mode (3D vs 2D)

El Projection mode (modo de proyección) de estados cámara se representa en 3D (perspectiva) o 2D (ortográfica). Por lo general, las cámaras están en el modo perspectiva de proyección, y como tal, tiene un campo de visión en forma de pirámide (FOV).



Se puede utilizar en una cámara principal rectangular para juegos 2D o como cámara secundaria en juegos 3D que sirve para hacer Heads Up Display (HUD) elementos como un mapa o una barra de energía. En los motores de juego los efectos como iluminación, desenfoques de movimiento y otros se aplican a la cámara para ayudar a crear una experiencia de juego más realista al ojo humano. Los juegos 3D más modernos utilizan varias cámaras para mostrar las partes del mundo.

Plygons, edges, vertices and meshes

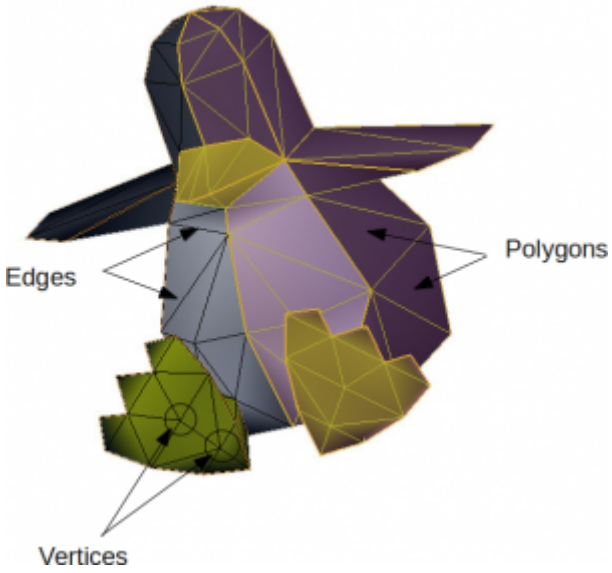
Las formas 3D estan compuestas de pequeñas formas 2D:

- Polygons: Formas 2D (por ejemplo en Unity son

triángulos)

- Edges: Los triángulos están formados por edges (bordes) conectados entre si.
- Vertices: Punto en el que convergen los edges.
- Meshes: son formas complejas creadas a partir de la interconexión de muchos plygons. Por ejemplo el tux que tenemos bajo estas lineas.

Tux Mesh



Al conocer estos puntos y ploygons, los motores de juego son capaces de hacer cálculos sobre los puntos de impacto, conocidos como colisiones. Se utiliza la detección de colisiones complejas con Mesh Colliders (colisionadores de malla), por ejemplo en juegos de shooters para detectar la ubicación exacta en la que una bala impacta sobre un objeto. Los meshes pueden tener otros usos, por ejemplo se pueden utilizar para especificar una forma del objeto menos detallado, pero más o menos la misma forma. Esto puede ayudar a mejorar el rendimiento del motor de física evitando que este compruebe un mesh muy detallado.

Materials, textures and shader

- Materials: Son un concepto común en los entornos 3D, ya

que establecen la apariencia visual de un modelo 3D (mesh). Desde colores básicos a reflejos en imágenes que representan superficies, los materiales lo manejan todo.

- Textura: una o más imágenes que se pueden aplicar al material para mejorar la apariencia de este.
- Shaders: un material trabaja con un shader, que no es más que un script encargado de estilizar el renderizado.

Cuando se crean texturas con programas de diseño como Photoshop o GIMP, hay que tener en cuenta la resolución. Grandes texturas aseguran un mayor detalle pero es más costoso renderizarlas.

Rigidbody physics

Cuando se trabaja con motores de juego, los motores de física proporcionan la posibilidad de que los objetos en los juegos simulen respuestas similares a las del mundo real.
En los motores de juego, por defecto un objeto no debe verse afectado por la física, porque esto requiere una gran cantidad de potencia de procesamiento, y porque simplemente no hay necesidad de hacerlo. Por ejemplo, en un juego de de carreras en 3D tiene sentido
que los coches esten bajo la influencia del motor de física, pero no la pista o alrededores (árboles, paredes…) que se mantengan estáticos durante la partida.
Los motores de física para juegos utilizan el sistema Rigidbody dynamics para crear movimientos realistas. Esto significa que en lugar de tener objetos estáticos en un entorno 3D, estos pueden tener propiedades tales como la masa, gravedad, velocidad y fricción. A medida que la potencia del hardware y software incrementa, Rigidbody physics es cada vez más usado, ya que ofrece la posibilidad simular entornos 3D más variados y realistas.

Detección de colisiones

La detección de colisiones es la forma en la que analizamos nuestro mundo 3D. Al aplicar un Collider component a un objeto, estamos poniendo una red invisible a su alrededor. Esta red, por lo general, imita su forma y es la encargada de informar de cualquier colisión con otros Colliders, haciendo que el motor de juego de responder en consecuencia.

Observaciones
Bien, este ha sido el primer post. Se han introducido los conocimientos básicos para poder trabajar en un entrono 3D. En el siguiente post haremos una aproximación al motor Unity.

Ruben.

Recuperar Lg Optimus 3D (Unbrick, Unroot, UnRecovery)

Introducción

En este post voy a mostrar como recuperar un Lg Optimus 3D que no arranca. Lo primero es coger un poco de aire. Es importante (aunque sea difícil de conseguir) tranquilizarse un poco para seguir los pasos adecuadamente. Por suerte este dispositivo tiene un modo de recuperación que se ejecuta a muy bajo nivel, así que es muy difíl (que no imposible) dar por perdido por completo el dispositivo. Al acabar el proceso, si todo ha ido adecuadamente, tendremos nuestro Lg funcionando sin acceso a root, con el recovery original y la versión más actualizada de la Rom ofrecida por Lg.

Consideraciones previas:

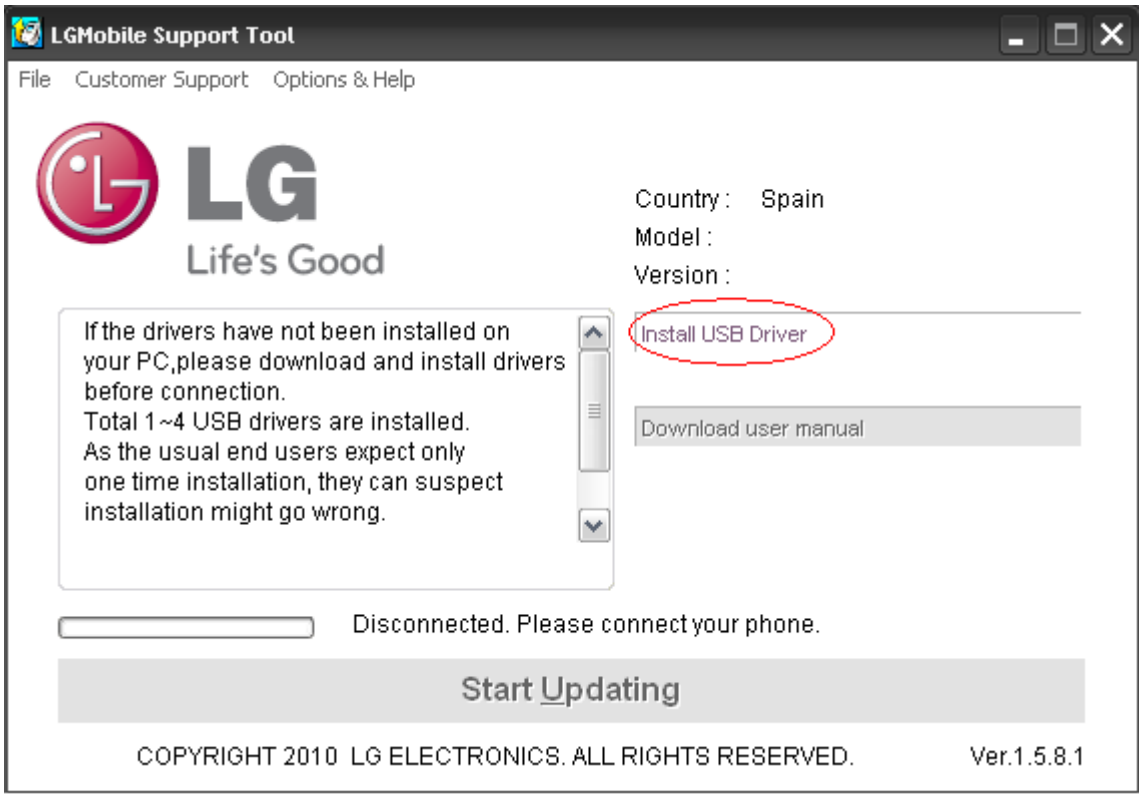
Bien en ningún caso me hago responsable de ningún daño ocasionado a algún terminal. Esta guía y las aplicaciones que se mencionan en ella carecen de cualquier tipo de garantía (un software de LG que decir…). Es decir, bajo vuestra responsabilidad queda lo que hagáis con vuestro terminal.

– Requisitos:

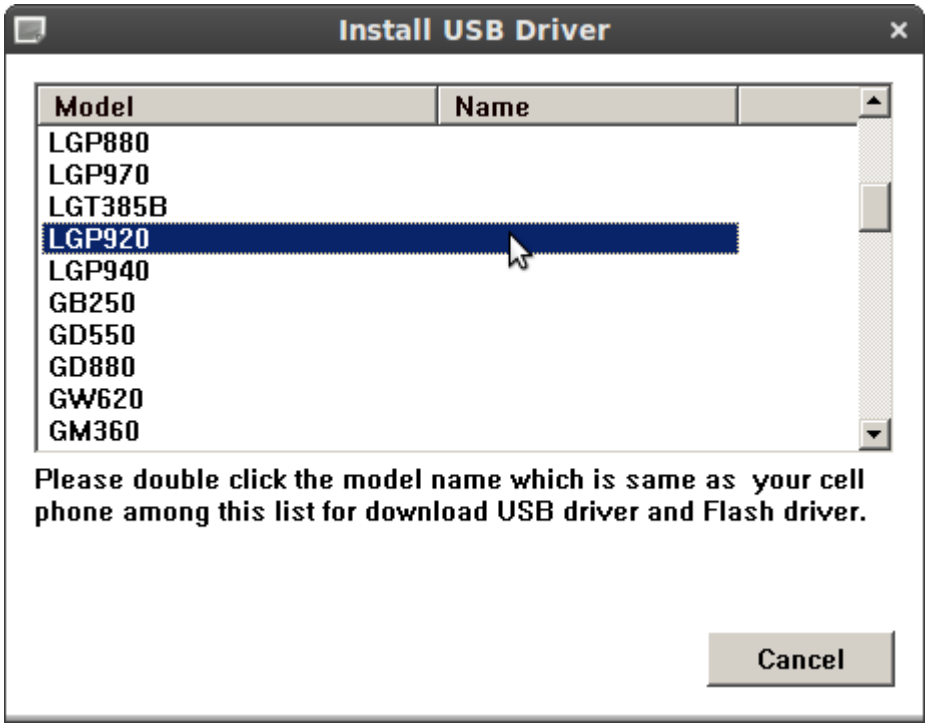
- Herramienta de recuperación oficial -> B2CAppSetup.
- Un pc con Windows. Si lo sé, pero que le haremos la herramienta de recuperación es para windows y el que se encuentre en esta situación no esta para experimentos con wine. También puntualizar que con màquina virtual no funciona (por lo menos con VirtualBox).
- El dispositivo debería tener suficiente batería (en torno un 50%), es un requerimiento difícil pero de lo contrario podríamos quedarnos a media instalación.
- Conexión a internet, la rom oficial que se instalará será bajada de internet.
- Necesitaremos el IMEI y el Numero de serie, podemos encontrarlos debajo de la batería.

Pasos a seguir

- Instalar drivers -> Arrancaremos B2CAppSetup y clicamos en install drivers.



- Seguidamente seleccionamos **LGP-920** e instalamos el driver.

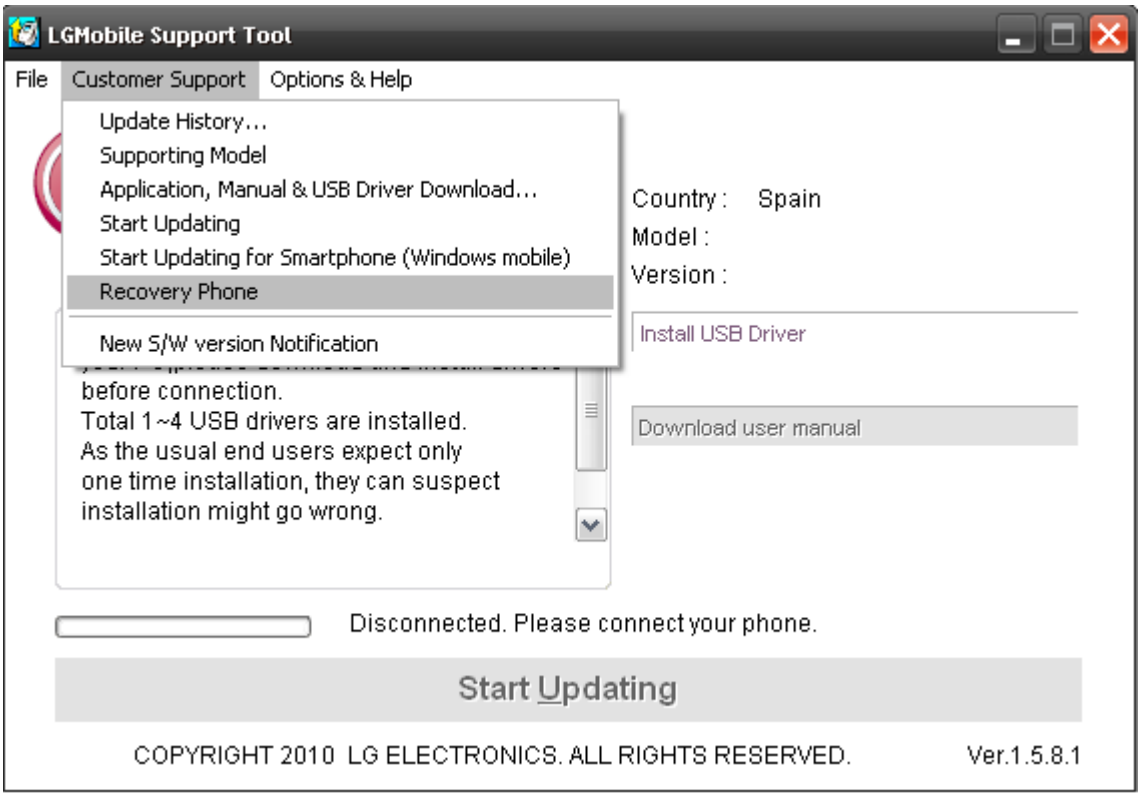


- Arrancar Lg Optimus 3D en modo recuperación

1. *Con el dispositivo desconectado del ordenador le quitamos la bateria y la volvemos a poner.*
2. *Introducimos la batería y conectamos el dispositivo al ordenador. E$l dispositivo arrancara saldrá el logo de LG seguidamente la batería y se apagará.*
3. *Presiona el botón de subir el volumen y el botón de encendido y no los sueltes hasta que salga el logo de LG y se vuelva apagar la pantalla. Con la pantalla apagada vuelve presionar los botones hasta que en la barra de notificaciones de windows veas que se empiezan a instalar los drivers. Puede ser que no te salga el símbolo de la batería, si a la primera no funciona tomatelo con calma y repite los pasos.*
4. *Una vez instalado el driver en el programa de recuperación de LG nos saldrá que el dispositivo está conectado.*

- Recuperación

    1. *Cuando el programa nos haya reconocido el dispositivo debemos selecionar Recovery Phone*



1. *Seguidamente una nueva ventana nos pedirá que introduzcamos el IMEI y el Numero de serie, que previamente hemos apuntado, una vez introducidos clica en el CHECK del IMEI y automáticamente comenzará el proceso de recuperación. Hay que tener en cuenta que se bajará la Rom de internet, esto quiere decir que dependiendo de tu connexión el proceso puede tardar más o menos (unos 30 min.), paciencia.*
2. *Una vez acabada la instalación desconnecta el dispositivo del ordenador, quita la batería, ponla, arranca y listo.*

Observaciones

Espero que esto pueda servir de ayuda. Normalmente esto ocurre cuando se flashean Roms de diferente baseband. Se debe tener cuidado siempre que flasheemos una Rom para evitar estos problemas. No nos veríamos en estas situaciónes (bricks producidos por instalar roms funcionales) si Lg tuviera un poco más en cuenta la calidad de su software…

Ruben.