ESTRUCTURAS DE DATOS:

Tipos de comandos:

- Commit
- Push
- Pull
- Fetch

Una vez guardado el fichero en guardar fichero (en Intellij lo hace automáticamente todo el rato).

1) Commit:

✓ En el git local marca un conjunto de cambios que funcionan o que si son suficientes como para guardarlos como un grupo. Marca una versión nueva del fichero con una nueva función.

✓ Lo manda al git local

2) Push:

✓ Coge las versiones locales y las manda al servidor remoto, y manda solo trabajo acabado no el que está a medias, tienes que hacerlo comic.

✓ El servidor es exterior al local

3) Pull:

✓ Descarga las versiones nuevas que se han subido al servidor remoto por el comando push, las versiones antiguas no se borran pero no se descargan, hay que hacer nuevo comando.

4) Fetch:

✓ Te saltas los comandos anteriores y solo te informa de las versiones que hay, no te permite descargarlo, solo informar de lo que hay y las versiones.

Antes de hacer un push, hay que hacer un pull.

✶ **Merge:**

- ✓ Sirve para unir versiones
- ✓ En la descripción de un cómic se recomienda descripción de máx. 50 caracteres.

✶ Podemos hacer ramas, porque vamos a trabajar en una versión especial del cómic, pudiendo saltar entre las distintas ramas. Sirve para versiones, arreglos, encuentro de errores,…

Paquetes:

- ✓ Pueden privados (protegidos) o públicos.
- ✓ Se declaran de la siguiente manera:

```
Package es.uah.pruebas.el1.eja.e1
Public class    MiClase
protected  int dato; // solo clases de mi paquete pueden acceder
```

```
public class MiSeguridad
//solo acceder a los documentos de mi paquete
@override (sustituyes comportamiento)
Public String to String()
// Son ANOTACIONES que nos vienen dadas, y son para que tengamos
cuidado con esta parte o clase de objeto del código

(Si usamos un abstract no podemos usar un new y se queja)
public class Vehiculo{
 private String matricula;
 public String SetMatricula(){
 return matricula;
}
Public void setMatricula(String m){
 this.matricula = m;
}
}
```

```
public class V.ruedas extends Vehiculo{
 prueba int n.ruedas;
 public int Setuedas (){
 return n.ruedas;
 }
 public void SetRuedas(int n){
```

```
this.n.ruedas = n,
}
}
```

```
public class V.volador extends Vehiculo{}
//clase vacia pero correcta

public class Motocicleta extends V.Ruedas{}
```

En java para evitar la herencia múltiple se usan interfaces:

```
Interface terrestre{ //firma método
 public getEnded();
}
Interface acuático {
 public.get… ();
}

class Anfibio implements terrestre, acuático
 public getEnded(){
}

// Terrestre t1 = new Perro(),
```
