

## **UNIDAD 4.**

# **Introducción a la POO. Desarrollo de clases.**



# Índice

1. Ficha unidad didáctica.....	1
2. Contenidos.....	2
2.1. Paradigma OO.....	2
2.1.1 Fundamentos POO.....	3
2.1.2 Estructura básica de clases.....	6
2.1.3 Uso de clases, objetos y métodos.....	7
2.1.4 Librerías y paquetes.....	8
2.2. Estructura interna de clase.....	14
2.2.1 Constructores.....	15
2.2.1.1 Sobrecarga de constructores.....	16
2.2.2 Encapsulación y visibilidad.....	18
2.2.2.1 Default.....	19
2.2.2.2 Private.....	21
2.2.2.3 Protected.....	22
2.2.2.4 Public.....	22
2.2.3 Definición de atributos.....	23
2.2.3.1 This.....	27
2.2.4 Métodos.....	28
2.2.4.1 Definición de argumentos y parámetros en métodos.....	28
2.2.5 Métodos y atributos estáticos.....	31
2.2.6 Getters and setters.....	33
2.3. Objetos y gestión de memoria.....	35
2.4. Empaquetado de clases. Organización de las clases en paquetes.....	37
2.5. Ficheros jar.....	39
2.6. Módulos.....	41
2.7. Documentación.....	42
3. Actividades y ejercicios.....	53

# 1. Ficha unidad didáctica.

Temporalización: De la semana 12 a la semana 21 del curso (navidades). 42 horas.	
<b>OBJETIVOS DIDÁCTICOS</b>	
<p>OD1: Evaluar las ventajas de la OO.  OD2: Analizar la estructura básica de clases y objetos.  OD3: Utilizar e instanciar objetos a partir de clases.  OD4: Reconocer los diferentes ámbitos de alcance de métodos y atributos.  OD5: Elaborar programas con creación de clases, instanciación y uso de los mismos.  OD6: Analizar y modificar clases con métodos y atributos con diferente visibilidad.  OD7: Comprender la importancia de la protección de datos .  EV2. Concienciar de la importancia de emplear hábitos respetuosos con el medio ambiente y control del gasto energético de las instalaciones informáticas  EV3. Reconocer de la importancia del trabajo en grupo en el ámbito empresarial.  RL1. Utilizar de forma adecuada y ergonómica el mobiliario de oficina, evitando posturas incorrectas que conlleven lesiones.  TIC1. Usar Internet para obtener información técnica.  ID1. Habituar a la lectura de documentación técnica en inglés.</p>	
<b>RESULTADOS DE APRENDIZAJE</b>	RA4
<b>CONTENIDOS</b>	
<p>Paradigma OO.  Estructura básica de clases.  Uso de clases, objetos y métodos.  Paso de parámetros.  Librerías y paquetes.  Estructura interna de clase.  Definición de parámetros en métodos.  Objetos y gestión de memoria.  Empaquetado de clases. Organización de las clases en paquetes.  Documentación.  Creación y uso de constructores avanzados.  Encapsulación y visibilidad.  Análisis de otros lenguajes OO.</p>	
<b>ORIENTACIONES METODOLÓGICAS</b>	
<p>El planteamiento y desarrollo de las actividades potencian el desarrollo de la habilidad de resolver problemas de forma metódica.  Se planifica el incremento gradual de la dificultad de las actividades.  Desarrollar la capacidad de trabajo en grupo a través de actividades que la potencien.  Las actividades de introducción se planifican para descubrir el uso y la tecnologías y despertar el interés por la materia.</p>	
<b>CRITERIO DE EVALUACIÓN</b>	2a, 2b,2c,2d,2e,2f,2g,2h, 2i,4a, 4b, 4c, 4d, 4e, 4f, 4g, 4h, 4i, 4j.

## 2. Contenidos.

### 2.1. Paradigma OO.

El desarrollo utilizando la programación estructurada posee una serie de problemas, en especial en grandes proyectos. Los programas se convierten en un conjunto de funciones que operan con variables y estructuras, muchas veces estas funciones dependen una de otras y un cambio en una estructura o una función puede tener consecuencias difíciles de prever. El mantenimiento y la depuración del código es un 70% del coste del producto, pensar en depurar una llamada a una función que internamente produce 10, 15 o 20 llamadas a otras funciones y que implica 5 o 6 estructuras.

Mucho del código estructurado no es usable en otros proyectos a pesar de ser similar al original y su modificación es compleja, además probar el software es complicado ya que la posibilidades que existen son prácticamente infinitas al poder tomar una gran cantidad de caminos con multitud de parámetros.

Estos problema no implican que un código estructurado con una buena arquitectura no se un buen código, por ejemplo la mayor parte del código del kernel de Linux se encuentra escrito en C estructurado.

Otros de los problemas del paradigma estructurado es la dificultad de plasmar conceptos de la vida real con variables, estructuras y funciones, por ejemplo pensar en la creación de un juego de estrategia en el que sea necesario modelar el comportamiento de cientos de elementos que existen en el juego en un momento dado.

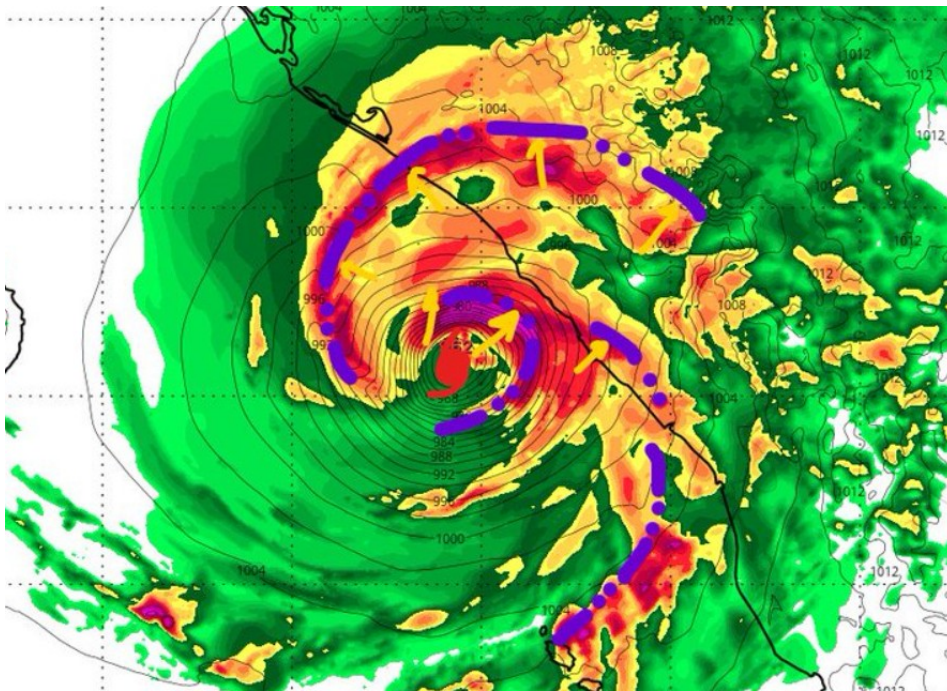
Para intentar solucionar los problemas anteriores como el coste de mantenimiento, hacer fácilmente testeable y confiable, reusable en la medida de lo posible y que acerque el diseño e implementación de software a como piensan las personas en el mundo real se desarrollo el paradigma OO.

Usar un lenguaje OO no implica que se obtenga un buen código que cumpla con los principios de la POO, de igual forma que usar un lenguaje estructurado no implica un mal desarrollo de software. Son simplemente formas de crear y mantener código.

### 2.1.1 Fundamentos POO.

La POO basa en una serie de fundamentos que los lenguajes de programación han de implementar de una u otra forma. Estos son:

**Abstracción:** Permite modelar elementos, relaciones o sistemas del mundo real en software, por ejemplo un simulador del comportamiento de un huracan en función de la temperatura, vientos...previendo su trayectoria y evolución.



**Encapsulación:** Consiste en ocultar detalles de la implementación que no son necesarios conocer, además de proteger los mecanismos internos. Realizando un símil con la vida real, al arrancar un coche no se sabe que se produce internamente para que este arranque, el usuario simplemente ha de conocer que para arrancar se ha de girar la llave o presionar un botón.

**Modularización:** Este principio indica que se ha de dividir en lo posible el sistema o modelo en elementos con una función clara y sencilla, y en la medida de lo posible independiente e intercambiable. Este concepto no es nuevo, en redes se tiene un modelo en capas, cada una de estas con funciones bien definidas. La modularidad ha de cumplir a su vez con otros dos principios:

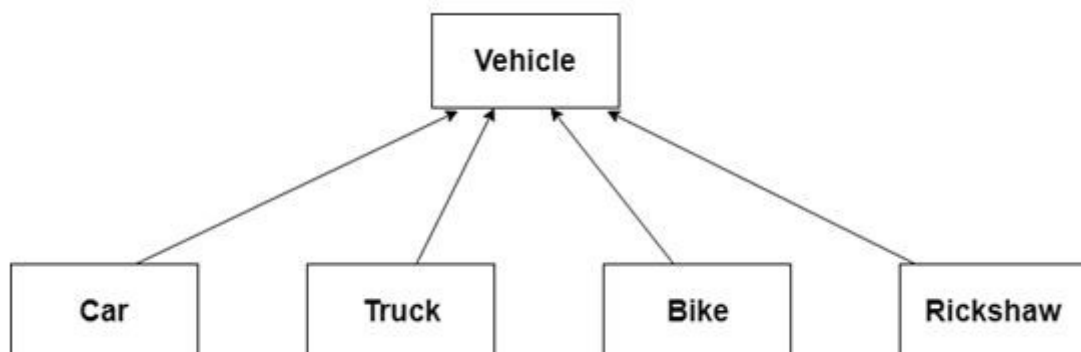
- **Cohesión:** Cada pieza o elemento ha de tener una función clara, sencilla y bien definida, es decir se necesita una alta cohesión.

- **Acoplamiento:** Estos elementos o piezas han de ser lo más independiente posibles del resto de los módulos de forma que sean fácilmente intercambiables y cambios en alguno de ellos no implique cambios o efectos colaterales en otros. Se busca un bajo acoplamiento.



[Vídeo sobre cohesión y acoplamiento.](#)

**Herencia:** La herencia es una de las características más destacadas de los lenguajes orientados a objetos, modela el sistema de forma similar al mundo real, por ejemplo en la vida real se tiene vehículos, estos vehículos se pueden dividir en coches, camiones, motocicletas, bicicletas... Todos estos vehículos poseen algunas características comunes como velocidad, acelerar, girar o número de ruedas y no tendría mucho sentido redefinir o describir una y otra que cada uno de ellos tiene una velocidad. La herencia permite definir unas cualidades y comportamientos comunes en una clase **base o padre** y que el resto de clases relacionadas **hereden** las cualidades y el comportamiento y se puede tratar **de igual forma que la clase padre**.




**Polimorfismo:** La herencia indica que ciertas acciones se heredan de la clase padre a la o las clases hijas, pero que ciertas acciones se denominen igual no implica que **internamente se realicen de la misma forma**, siguiendo con el mismo ejemplo, internamente girar en un coche implica una serie de acciones diferentes a girar en una bicicleta, de forma que se puede hacer la acción en ambos, pero internamente las acciones a realizar son diferentes. Este es el primer modo de polimorfismo en los lenguajes OO, pero no el único, la sobrecarga de métodos permite realizar acciones pasando diferentes número de parámetros, supongamos que se tiene un coche eléctrico moderno, es posible arrancar el coche pulsando un botón o facilitarle una fecha y hora



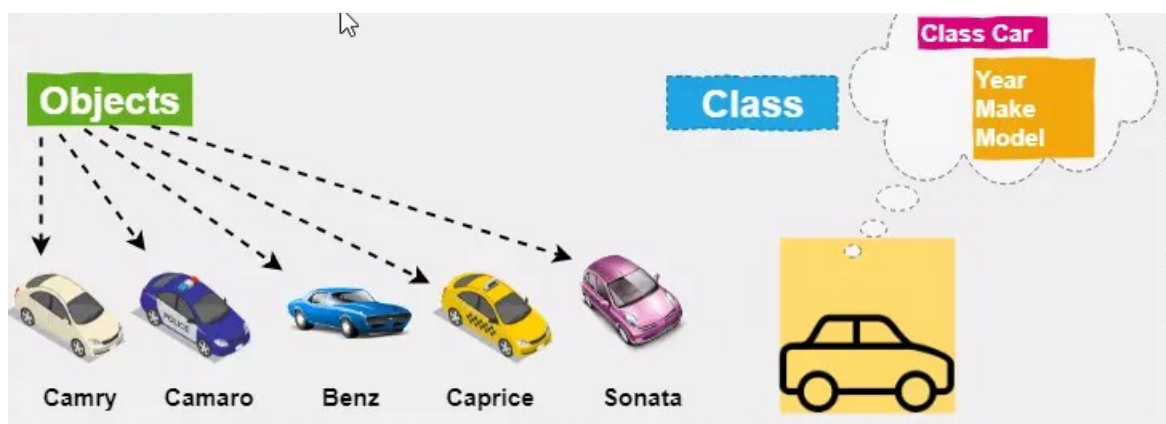
para que arranque, la acción es la misma, pero en el primer caso no es necesario dar más información y en la segunda se le ha de pasar la fecha y hora.


## Introducción a la POO.

Estos fundamentos se basan en unir los datos (variables simples, estructuras) y el comportamiento (funciones) en una única entidad de forma que se pueda operar internamente sin describir los detalles internos (encapsulación), con unas funciones bien definidas (modularidad), que puede representar y modelar la realidad de forma más natural (abstracción) y que soporta herencia de otras entidades (herencia) y comportamiento diferente ante las mismas llamadas (polimorfismo). La entidad se denomina **clase**.

 Los datos pasan a ser atributos dentro de la clase, el comportamiento (funciones) se denominan métodos y quedan asociados a la clase.

Una clase es la definición o declaración de una entidad abstracta, de la misma forma que `int v[]` define un vector, es necesario instancia (crear) e inicializar dicha clase en memoria, cuando esto se realiza se crean **objetos** de ese tipo de clase. En un momento dado es posible tener muchos objetos de una clase en el programa, todos poseen los mismos atributos y los mismo métodos, pero el valor de cada atributo es diferente en cada objeto y al ejecutar los métodos se realiza sobre los datos concretos del objeto. El programa consiste en la comunicación de diferentes objetos a través de paso de mensajes (uso de métodos de las clases).



 Las clases son las definición de abstracciones usando atributos y métodos, pudiendo crear objetos de esa clase y comunicándose entre objetos de igual o

diferente tipo usando paso de mensajes.

## 2.1.2 Estructura básica de clases.

En Java cada clase ha de estar en un fichero con el nombre de la clase (posteriormente se verá que no es exactamente así, aunque puede tener varias clases, solo una será visible desde el exterior).

Una clase básica se define de la siguiente forma:

```
1 public class Cuenta{
2     //definición de atributos
3     double saldo;
4     String propietario;
5     //definición de métodos
6     public void ingreso(double cantidad){
7         saldo=saldo+cantidad;
8     }
9     public void retirada(double cantidad){
10        if(saldo>=cantidad)
11            saldo=saldo-cantidad;
12    }
13    public double getSaldo(){
14        return saldo;
15    }
16    public String getPropietario(){
17        return propietario;
18    }
19    public void setPropietario(String nuevo){
20        propietario=nuevo;
21    }
22 }
```

En la línea 1 se utiliza la palabra reservada **class** para indicar que es una clase y a continuación el **nombre de la clase**, en este caso cuenta.

De la línea 2 a la 4 se definen los **atributos**, en este caso 2, uno de tipo double que es saldo y otro de tipo cadena con el nombre del propietario. Este orden no es obligatorio para el compilador pero en las reglas de estilo indica que se han de definir los atributos al inicio de la clase..



De la línea 5 a la 22 se definen los métodos de la clase, observar que se le pueden pasar parámetros.



En varias partes de la clase aparece la palabra public ¿A qué piensas que se debe?

### 2.1.3 Uso de clases, objetos y métodos.

Una vez definida la clase se han de crear objetos, cada objeto contendrá atributos con nombre y tipos iguales pero con valores diferentes (se encuentran en diferentes áreas de memoria).



**Para acceder a atributos(aunque como se verá más adelante esto rompe el principio de encapsulación) y métodos se utiliza el operador .(punto) al igual que con las estructuras.**

Un ejemplo en el método estático main que crea 2 cuentas diferentes:

```
1.    public static void main(String[] args) {
2.        Cuenta cuenta1;
3.        Cuenta cuenta2;
4.        cuenta1 = new Cuenta();
5.        cuenta2= new Cuenta();
6.        cuenta1.setPropietario("Pedro");
7.        cuenta1.ingreso(1000.0);
8.        cuenta2.setPropietario("Paco");
9.        cuenta2.ingreso(1345.0);
10.       System.out.println("La cuenta de " + cuenta1.getPropietario() + "
    tiene un saldo de" + cuenta1.getSaldo());
11.       System.out.println("La cuenta de " + cuenta2.getPropietario() + "
    tiene un saldo de" + cuenta2.getSaldo());
12.    }
```

Al ejecutar el código anterior la salida es:

```
La cuenta de Pedro tiene un saldo de 1000.0
La cuenta de Paco tiene un saldo de 1345.0
```

Observar como cada una de las variables de tipo Cuenta, que son objetos de la clase Cuenta **poseen cada uno su atributo saldo y propietario** y es posible enviar mensajes entre objetos (métodos en la clase).

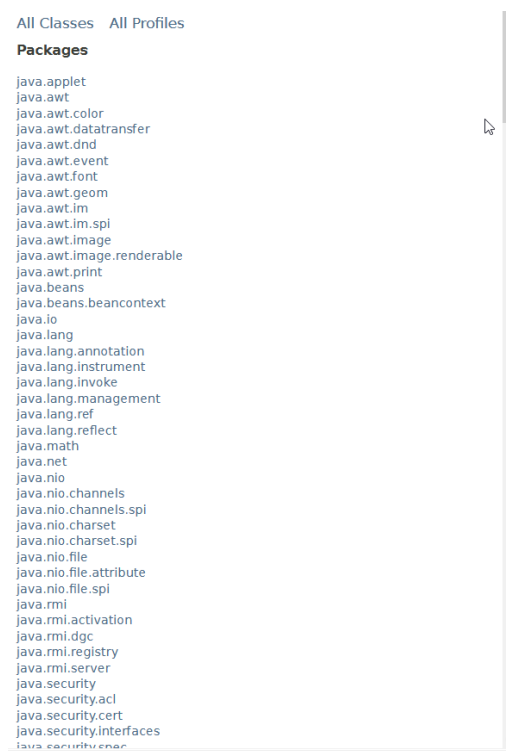
## 2.1.4 Librerías y paquetes.

La programación OO permite la reutilización de código, este código se ordena y clasifica, existiendo dos clasificadores principalmente, la librería y el paquete.

- Librería es la agrupación de código que se puede reemplazar, por ejemplo en Linux al instalar un nuevo programa es necesario instalar librerías de terceros que el programa a usar utiliza, es más utilizado en la programación estructurada. La agrupación de librerías se denomina biblioteca, aunque depende del lenguaje.
- Paquete. Clasifica también el código pero se le añade una estructura lógica que agrupa a diferentes clases relacionadas. Posee una estructura jerárquica en forma de árbol y permite utilizar.

La clasificación entre un concepto y otro es muy difusa, por ejemplo algunos IDE's gestionan agrupaciones de paquetes de código como librerías y/o bibliotecas.

Por defecto J2SE incluye diferentes librerías y paquetes accesibles de forma directa, simplemente se ha de indicar en el código con la palabra reservada **import** y la parte del paquete que se desea utilizar, también es posible usar la ruta completa de la clase dentro del paquete para utilizarlo, aunque este caso **no es el aconsejado**. Algunos de los paquetes incluidos por defecto en JSE 8:



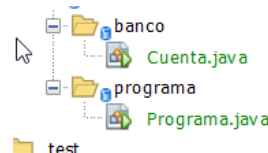
Por ejemplo, el paquete `java.util` posee un conjunto de clases ampliamente utilizadas como

## Scanner o Random.

<b>Random</b>	An instance of this class is used to generate a stream
<b>ResourceBundle</b>	Resource bundles contain locale-specific objects.
<b>ResourceBundle.Control</b>	<b>ResourceBundle.Control</b> defines a set of callback me the bundle loading process.
<b>Scanner</b>	A simple text scanner which can parse primitive types
<b>ServiceLoader&lt;S&gt;</b>	A simple service-provider loading facility.
<b>SimpleTimeZone</b>	<b>SimpleTimeZone</b> is a concrete subclass of <b>TimeZone</b> th
<b>Spliterators</b>	Static classes and methods for operating on or creatin <b>Spliterator.OfLong</b> , and <b>Spliterator.OfDouble</b> .

Estas librerías y/o paquetes se pueden usar de forma sencilla ya que “**se conoce la localización física**” en la que se encuentra, pero en caso de librerías externas es necesario indicar la localización de dichas clases y o paquetes.

Otro caso es cuando se tienen diferentes clases y son necesarias para el programa principal, siguiendo con el ejemplo anterior, se decide separar la clase cuenta, y tenerla en un directorio /banco junto con otras clases y una carpeta denominada programa en la que se encuentra el programa principal:



Al compilar desde línea de comandos indica que no es posible ya que no encuentra la clase cuenta:

```

PS C:\Users\Pedro\Documents\NetBeansProjects\Tema4\src\main\java\programa> javac .\Programa.java
.\Programa.java:3: error: package banco does not exist
import banco.Cuenta;
    ^
.\Programa.java:14: error: cannot find symbol
    Cuenta cuenta1;
    ^
    symbol:   class Cuenta
    location: class Programa
.\Programa.java:15: error: cannot find symbol
    Cuenta cuenta2;
    ^
    symbol:   class Cuenta
    location: class Programa
.\Programa.java:16: error: cannot find symbol
    cuenta1 = new Cuenta();
                ^
    symbol:   class Cuenta
    location: class Programa
.\Programa.java:17: error: cannot find symbol
    cuenta2= new Cuenta();
                ^
    symbol:   class Cuenta
    location: class Programa
5 errors
  
```

Es necesario indicar en que lugar se tiene las clases a generar, el comando será el siguiente, con la opción -d para indicar en que lugar dejar los .class, y mantener la

estructura.

```
javac .\programa\Programa.java .\banco\Cuenta.java -d dist
```

Esta línea compila y enlaza los dos ficheros y el resultado lo deja en la carpeta dist.

Al ejecutar desde la raíz, **indicando el paquete y la clase en la que ha de existir un método estático main**:

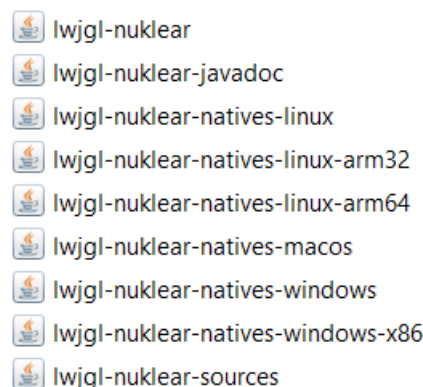
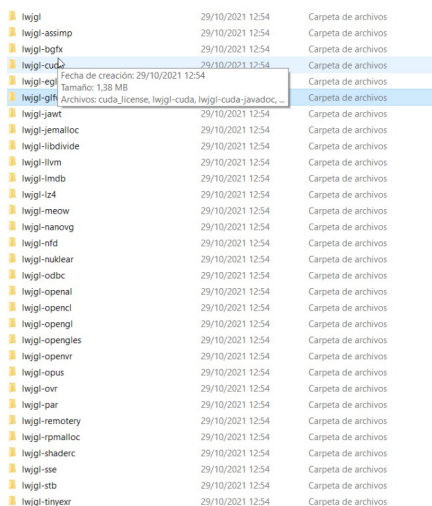
```
java programa.Programa
```

Otro caso posible es la de uso de **librerías de terceros**, estas librerías se distribuyen normalmente en formato jar, este formato es un comprimido que almacena las clases, documentación y cualquier otro recurso necesario para la librería o para que funcione un programa (los jar pueden funcionar como ejecutable). En este punto se trata del uso de los jar, tratando posteriormente la generación de jar.

Sea la librería o más bien el motor de juegos Lightweight Java Game Library (LWJGL o Biblioteca Java Ligera para Juegos) que es posible descargar en: <https://www.lwjgl.org/>



Al descargar y descomprimir se observa que se tiene una gran cantidad de jar's:



Se crea un fichero .java que haga uso del motor (obtenido de <https://www.lwjgl.org/guide>)

```
import org.lwjgll.*;
import org.lwjgll.glfw.*;
import org.lwjgll.opengl.*;
import org.lwjgll.system.*;
import java.nio.*;
import static org.lwjgll.glfw.Callbacks.*;
import static org.lwjgll.glfw.GLFW.*;
import static org.lwjgll.opengl.GL11.*;
import static org.lwjgll.system.MemoryStack.*;
import static org.lwjgll.system.MemoryUtil.*;

public class principal {
    // The window handle
    private long window;
    public void run() {
        System.out.println("Hello LWJGL " + Version.getVersion() + "!");
        init();
        loop();
        // Free the window callbacks and destroy the window
        glfwFreeCallbacks(window);
        glfwDestroyWindow(window);
        // Terminate GLFW and free the error callback
        glfwTerminate();
        glfwSetErrorCallback(null).free();
    }
    private void init() {
        // Setup an error callback. The default implementation
        // will print the error message in System.err.
        GLFWErrorCallback.createPrint(System.err).set();
        // Initialize GLFW. Most GLFW functions will not work before doing
        this.
        if ( !glfwInit() )
            throw new IllegalStateException("Unable to initialize GLFW");
        // Configure GLFW
        glfwDefaultWindowHints(); // optional, the current window hints are
        already the default
        glfwWindowHint(GLFW_VISIBLE, GLFW_FALSE); // the window will stay
        hidden after creation
        glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE); // the window will be
        resizable
        // Create the window
        window = glfwCreateWindow(300, 300, "Hello World!", NULL, NULL);
        if ( window == NULL )
```

```
        throw new RuntimeException("Failed to create the GLFW
window");

    // Setup a key callback. It will be called every time a key is
    pressed, repeated or released.
    glfwSetKeyCallback(window, (window, key, scancode, action, mods) ->
{
    if ( key == GLFW_KEY_ESCAPE && action == GLFW_RELEASE )
        glfwSetWindowShouldClose(window, true); // We will
detect this in the rendering loop
});
    // Get the thread stack and push a new frame
    try ( MemoryStack stack = stackPush() ) {
        IntBuffer pWidth = stack.mallocInt(1); // int*
        IntBuffer pHeight = stack.mallocInt(1); // int*
        // Get the window size passed to glfwCreateWindow
        glfwGetWindowSize(window, pWidth, pHeight);
        // Get the resolution of the primary monitor
        GLFWVidMode vidmode =
glfwGetVideoMode(glfwGetPrimaryMonitor());
        // Center the window
        glfwSetWindowPos(
            window,
            (vidmode.width() - pWidth.get(0)) / 2,
            (vidmode.height() - pHeight.get(0)) / 2
        );
    } // the stack frame is popped automatically
    // Make the OpenGL context current
    glfwMakeContextCurrent(window);
    // Enable v-sync
    glfwSwapInterval(1);
    // Make the window visible
    glfwShowWindow(window);
}

private void loop() {
    // This line is critical for LWJGL's interoperation with GLFW's
    // OpenGL context, or any context that is managed externally.
    // LWJGL detects the context that is current in the current thread,
    // creates the GLCapabilities instance and makes the OpenGL
    // bindings available for use.
    GL.createCapabilities();
    // Set the clear color
    glClearColor(1.0f, 0.0f, 0.0f, 0.0f);
    // Run the rendering loop until the user has attempted to close
```



```
// the window or has pressed the ESCAPE key.
while ( !glfwWindowShouldClose(window) ) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // clear
the framebuffer
    glfwSwapBuffers(window); // swap the color buffers
    // Poll for window events. The key callback above will only be
    // invoked during this call.
    glfwPollEvents();
}
}

public static void main(String[] args) {
    new principal().run();
}
}
```

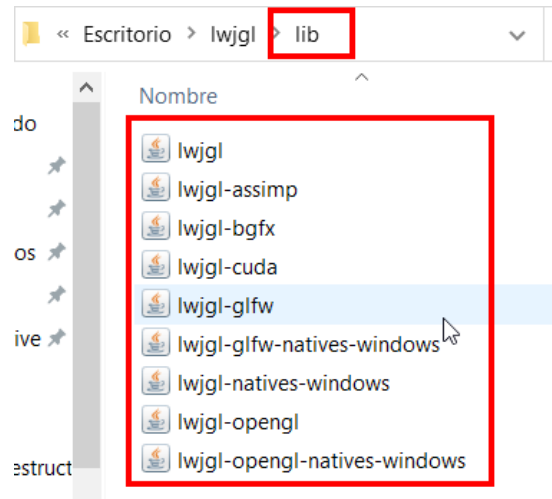
Observar en amarillo los paquetes usados en el código.

```
d----- 29/10/2021 12:54 lwjgl-ovr
d----- 29/10/2021 12:54 lwjgl-par
d----- 29/10/2021 12:54 lwjgl-remotery
d----- 29/10/2021 12:54 lwjgl-rpalloc
d----- 29/10/2021 12:54 lwjgl-shaderc
d----- 29/10/2021 12:54 lwjgl-sse
d----- 29/10/2021 12:54 lwjgl-stb
d----- 29/10/2021 12:54 lwjgl-tinyexr
d----- 29/10/2021 12:54 lwjgl-tinyfd
d----- 29/10/2021 12:54 lwjgl-tootle
d----- 29/10/2021 12:54 lwjgl-vma
d----- 29/10/2021 12:54 lwjgl-vulkan
d----- 29/10/2021 12:55 lwjgl-xxhash
d----- 29/10/2021 12:55 lwjgl-yoga
d----- 29/10/2021 12:55 lwjgl-zstd
-a----- 29/10/2021 12:54 20 build.txt
-a----- 29/10/2021 12:54 1518 LICENSE
-a----- 29/10/2021 13:11 3663 principal.java
```

Al compilar el código sin indicar las librerías se obtiene

```
PS C:\Users\Pedro\Desktop\lwjgl> javac .\principal.java
.\principal.java:9: error: package org.lwjgl.glfw does not exist
import static org.lwjgl.glfw.GLFW.*;
^
.\principal.java:10: error: package org.lwjgl.opengl does not exist
import static org.lwjgl.opengl.GL11.*;
^
.\principal.java:11: error: package org.lwjgl.system does not exist
import static org.lwjgl.system.MemoryStack.*;
^
.\principal.java:12: error: package org.lwjgl.system does not exist
import static org.lwjgl.system.MemoryUtil.*;
^
.\principal.java:1: error: package org.lwjgl does not exist
import org.lwjgl.*;
^
.\principal.java:2: error: package org.lwjgl.glfw does not exist
import org.lwjgl.glfw.*;
^
.\principal.java:3: error: package org.lwjgl.opengl does not exist
import org.lwjgl.opengl.*;
^
.\principal.java:4: error: package org.lwjgl.system does not exist
import org.lwjgl.system.*;
^
.\principal.java:40: error: cannot find symbol
    if ( !glfwInit() )
        ^
```

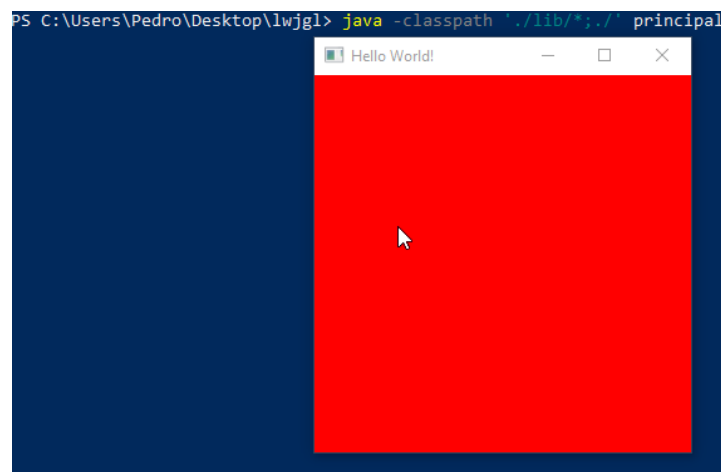
Se ha indicado el lugar en que se encuentran los ficheros jar usando el parámetro -classpath o -cp, por comodidad se han situado todas las necesarias en la carpeta lib.



```
javac .\principal.java -cp ./lib/*
```

Al ejecutar sucede lo mismo, es necesario indicar la localización de las librerías:

```
java -classpath './lib/*;./' principal
```



**Tanto para compilar como para ejecutar es necesario indicar la localización de las librerías y/o jar usando la opción -classpath o -cp tanto de javac como java.**

Este proceso (definir classpath, compilación, movimiento de ficheros...) es llevado a cabo por los IDE y por el software de gestión y construcción de proyectos como Ant, Maven o Gradle en Java, pero es necesario conocer el proceso interno para la generación.

## 2.2. Estructura interna de clase.

Una vez se es capaz de usar clases y objetos, importar librerías externa, compilar código en varios ficheros y configurar las opciones para ejecutar usando librerías externas se

profundiza en la estructura básica de una clase

Recordar que una clase permite modelar y abstraer elementos y conceptos necesarios para el software.

### 2.2.1 Constructores.

Suponer que se desea modelar con clases y objetos a personas, una persona se caracteriza por un nombre, una fecha de nacimiento, una altura, un peso, un nif..., un posible código para la clase que lo representa sería:

```
import java.util.Date;

public class Persona {
    String nombre;
    Date fecha_nacimiento;
    int altura;
    int peso;
    String nif;
}
```

Observar que la fecha de nacimiento es un **atributo cuyo tipo de dato es una clase** que gestiona fechas, por supuesto se ha realizado el import para poder utilizarla.

Al declarar e instanciar el objeto e intentar acceder a la fecha de nacimiento sucede esto:

```
public static void main(String args[]){
    Persona p;
    p= new Persona();
    System.out.println(p.fecha_nacimiento.getTime());
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.util.Date.getTime()" because "p.fecha_nacimiento" is null
```

Indica que p.fecha\_nacimiento es null (nulo), es decir no tiene espacio en memoria reservado.



Intentar explicar la razón del fallo anterior.

Para evitar estos fallos por no encontrarse el objeto **inicializado** con un estado correcto se definen unos métodos especiales llamados constructores que toda clase ha de poseer y que **permite inicializar el objeto al mismo tiempo que se instancia**.

**Toda clase posee un constructor por defecto sin parámetros (aunque no se defina, si existe internamente), este método se puede reemplazar, no devuelve ningún valor y ha de tener el mismo nombre.** En la clase anterior:

```
1. public class Persona {  
2.     String nombre;  
3.     Date fecha_nacimiento;  
4.     int altura;  
5.     int peso;  
6.     String nif;  
7.     public Persona() {  
8.         fecha_nacimiento= new Date();  
9.     }  
10.    public static void main(String args[]) {  
11.        Persona p;  
12.        p= new Persona();  
13.        System.out.println(p.fecha_nacimiento.getTime());  
14.    }  
15. }
```

En la línea 7 se define el constructor (se reemplaza el por defecto) y en la nueve se instancia el atributo fecha\_nacimiento.

En la línea 12 al hacer p= new Persona(), se declara p, se instancia y a continuación de forma automática se llama al constructor.

Es posible ejecutar cualquier código que sea necesario para que el objeto se encuentre en un estado correcto para su posterior uso, por ejemplo se pueden dar valores a altura y peso a -1 para indicar que no se han introducido.



**En caso de tener atributos que son tipos de datos compuestos (otras clases) se recomienda instancias e inicializar( si es necesario) dichos atributos para evitar errores de referencias nulas.**

### 2.2.1.1 Sobrecarga de constructores.

El polimorfismo es uno de los principios de la POO, si bien su uso se relaciona con otro de estos principios, la herencia, existe también el polimorfismo de métodos funciones. Este polimorfismo permite que existan diferentes métodos en clases y funciones en estructurado

con el mismo nombre pero con diferente tipo y/o número de parámetros.



**No puede existir métodos o funciones con el mismo número de parámetros y cuyos tipos sean iguales uno a uno, por ejemplo:**

```
1. metodo(int p1);  
2. void metodo(int p1,int p2);  
3. voidmetodo(int p1,float p2);  
4. void metodo(float p1);  
5. void metodo(int p2);  
6. void meotod(int p5,int p6);  
7. int metodo(int p3);
```

El método definido en el punto 5 es incorrecto (no compila y normalmente el IDE mostrará el error), ya que en la línea 1 se ha definido un método con el mismo número de parámetros y el mismo tipo (no sucediendo lo mismo con el método de la línea 4 ya que son de diferente tipo), en la línea 6 sucede lo mismo, ya que en la línea 2 se tiene definido un método o función con el mismo nombre y los mismos tipos, por último en la línea 7 sucede lo mismo destacando que el valor devuelto no influye en la sobrecarga. (C estándar no soporta sobrecarga de funciones, en cambio C++ si, y junto con Java también sobrecarga de métodos).

Por defecto aunque no se declare **toda clase posee al menos un constructor**, llamado í constructor por defecto, que no recibe parámetros ni realiza ninguna acción. En el punto anterior se ha visto que se puede **sobreescibir** dicho método para adaptarlo a las necesidades, pero también es posible definir nuevos constructores con diferentes parámetros, por ejemplo, en el caso anterior, se pueden dar valores a los atributos directamente:

Siguiendo el ejemplo de la clase persona se sobrecarga el constructor:

```
1. import java.util.Date;  
2. public class Persona {  
3.     String nombre;  
4.     Date fecha_nacimiento;  
5.     int altura;  
6.     int peso;  
7.     String nif;  
8.     //constructor por defecto  
9.     public Persona() {
```

```
10.         fecha_nacimiento = new Date();
11.     }
12.     //Sobrecarga 1
13.     public Persona(String nombre_persona) {
14.         fecha_nacimiento = new Date();
15.         nombre = nombre_persona;
16.     }
17.     //Sobrecarga 2
18.     public Persona(int peso_persona, String nif_persona){
19.         fecha_nacimiento =new Date();
20.         peso=peso_persona;
21.         nif=nif_persona;
22.     }
23.     //Sobrecarga 3
24.     public Persona(String nombre_persona, int altura_persona) {
25.         fecha_nacimiento = new Date();
26.         altura = altura_persona;
27.         nombre = nombre_persona;
28.     }
29.     public static void main(String args[]) {
30.         Persona p;
31.         Persona p2;
32.         p = new Persona();
33.         System.out.println(p.fecha_nacimiento.getTime());
34.         p2= new Persona(37,"3456788F");
35.         System.out.println(p2.peso+" "+p2.nif);
36.     }
37. }
```

Se han creado 4 constructores para la clase, el primero es el por defecto.



Los constructores 2 y 3 tienen el mismo número de parámetros, y no da fallo de compilación. ¿Cuál es la razón?

## 2.2.2 Encapsulación y visibilidad.

Uno de los principios de la programación orientada a objetos es la encapsulación y ocultación de información. De igual forma que cuando se enciende un ordenador los usuarios no conocen los mecanismos internos que hacen que se arranque, se inicie el sistema operativo...



En la POO es posible limitar el acceso a ciertas partes de la implementación con los llamados **modificadores de accesos** que se aplican a las clases, subclases (se verá en el tema de herencia), métodos y atributos. Estos modificadores son:

- public
- protected
- private
- default

Al modificar el acceso se está fijando si otras clases pueden usar esta, métodos o atributos de las mismas.

Para declarar una clase se tiene

```
modificador_acceso class nombreclase{  
    //métodos y atributos  
}
```

- En caso de **no indicarse modificador de acceso**, la clase es visible e instanciable dentro del paquete en el que se define, pero no en subclases (se verá en el tema de herencia).
- Si la clase se declara como **pública** esta es visible e instanciable por cualquier otra clase.
- Si la clase se declara como **protegida** es visible desde la misma clase, desde el paquete y desde las subclases.
- Si se declara como privada solo es visible por ella misma.

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

### 2.2.2.1 Default.

Sea la clase A, en el paquete modificadores, declarada por defecto, es decir es visible dentro de la clase y el paquete modificadores:

```
package modificadores;
```

```
/**
 *
 * @author Pedro
 */
class A {
    int x;
    public A() {
        x=5;
    }
}
```

Al intentar usarlo desde la clase B, que también se encuentra en el paquete modificadores:

```
package modificadores;

/**
 *
 * @author Pedro
 */
class B {
    public B() {
        A a= new A();
    }
    public static void main(String args[]){
        B b=new B();
    }
}
```

En cambio si se intenta acceder desde un paquete que no es modificadores:

```
package externo;

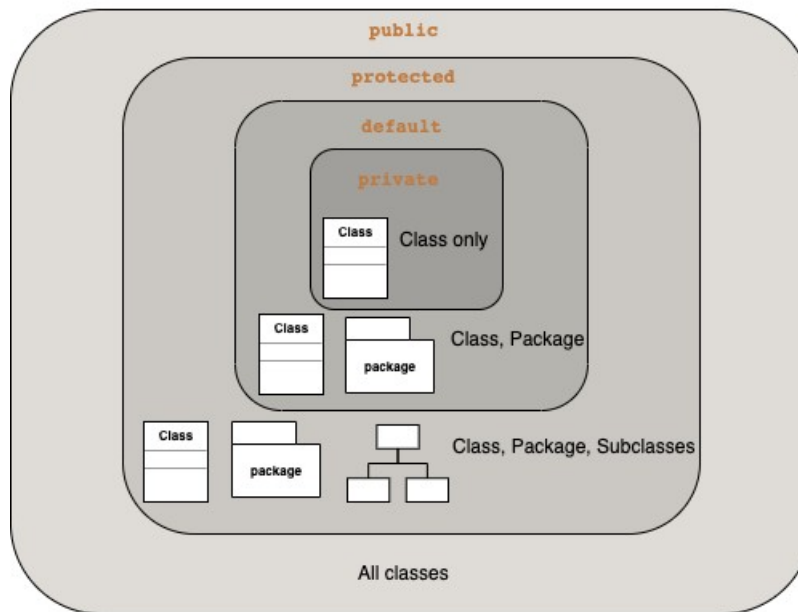
/**
 *
 * @author Pedro
 */
class C {

    public C() {
        A a = new A();
    }
    public static void main(String args[]) {
```

```
C c = new C();
}
}
```

Se obtiene un error de compilación.

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code
- Erroneous tree type: A
    at externo.C.<init>(C.java:14)
    at externo.C.main(C.java:18)
```



### 2.2.2.2 Private.

Siguiendo con el mismo ejemplo anterior, ahora se declara la clase A como privada, en principio esto no puede ser, ya que ha de existir al menos una clase accesible en cada fichero, para ver el comportamiento se introduce en el mismo fichero que B:

```
package modificadores;

/**
 *
 * @author Pedro
 */
class B {
    public B() {
        A a= new A();
    }

    public static void main(String args[]){
        B b=new B();
    }
}
```

```
}  
}  
private class A {  
    int x;  
    public A() {  
        x=5;  
    }  
}
```

Al compilar el error que muestra es el siguiente:

```
modificadores/B.java:[19,9] modifier private not allowed here
```

### 2.2.2.3 *Protected.*

En el caso de protected es posible acceder desde la clase, el paquete y las subclases. Las subclases se tratan en temas posteriores junto con la herencia, siendo el comportamiento sin subclases igual que en el caso de default.

### 2.2.2.4 *Public.*

En este caso la clase es accesible e instanciable desde cualquier punto, si se compara con el ejemplo de default con 3 clases A, B y C, pero en este caso A como public:

Clase A:

```
package modificadores;  
  
/**  
 *  
 * @author Pedro  
 */  
public class A {  
    int x;  
    public A() {  
        x=5;  
    }  
}
```

Clase C, observar como se crea en el constructor un objeto de clase A, esto es posible a pesar de encontrarse en diferentes paquetes ya que A es declarado como public:

package externo;

```
import modificadores.A;
```

```
/**
 *
 * @author Pedro
 */
class C {

    public C() {
        A a = new A();
    }

    public static void main(String args[]) {
        C c = new C();
    }
}
```

### 2.2.3 Definición de atributos.

Los atributos se definen en las clases y pueden ser de tipo primitivo como int, char... o de una clase, en este último caso es necesario inicializar dichos atributos antes de utilizarlo, ya que por defecto son nulos, esto se suele hacer en los constructores.

Los atributos también poseen modificadores de acceso al igual que las clases siendo su comportamiento idéntico pero añadiendo un nuevo nivel de concreción, pudiendo ser:

- public. Es accesible por todas las clases.
- Protected. Es accesible por las clases del paquete y las subclases.
- Private. Solo por la clase.
- Default. No es necesario indicarlo, por la clase y el paquete.

A partir del modificador de la clase y el del atributo se definirá quién tiene acceso a ese atributo, por ejemplo si se tiene una clase definida como protected y un atributo de tipo fecha como default, a la clase pueden acceder las clases del paquete y las subclases pero al atributo fecha solo la clase y las clases definidas en el paquete.



Si se define una clase como public y un atributo como protected ¿Quién tiene acceso a la clase? ¿Y al atributo?

Para definir un atributo en una clase se sigue la sintaxis:

```
[modificador_de_acceso] tipo_de_dato nombre;
```

Un ejemplo de clases para una carta y baraja de cartas:

```
package juego;

/**
 *
 * @author Pedro
 */
class Carta {
    protected enum ORLA {
        OROS,
        BASTOS,
        ESPADAS,
        COPAS
    }

    protected enum NUMERO{
        AS,
        UNO,
        DOS,
        TRES,
        CUATRO,
        CINCO,
        SEIS,
        SIETE,
        OCHO,
        NUEVE,
        DIEZ,
        SOTA,
        CABALLO,
        REY
    }

    private ORLA palo;
    private NUMERO numero;

    public Carta() {
    }

    public Carta(ORLA nuevo_palo, NUMERO nuevo_numero) {
        palo=nuevo_palo;
        numero=nuevo_numero;
    }

    public ORLA getPalo() {
        return palo;
    }
}
```



```
public void setPalo(ORLA palo) {  
    this.palo = palo;  
}  
public NUMERO getNumero() {  
    return numero;  
}  
public void setNumero(NUMERO numero) {  
    this.numero = numero;  
}  
}
```



¿Qué tipo de modificador de acceso tiene la clase carta?. ¿Se podría hacer new Carta desde una clase del mismo paquete? ¿Y desde una clase fuera del paquete?



¿Que tipo de modificadores tiene los dos atributos de la clase? ¿En ambos casos se puede acceder desde el mismo paquete en el que se define? ¿Y desde fuera del paquete?

## La clase Baraja

```
1. package juego;  
2.  
3. import juego.Carta.NUMERO;  
4. import juego.Carta.ORLA;  
5.  
6. /**  
7.  *  
8.  * @author Pedro  
9.  */  
10.    public class Baraja {  
11.  
12.        private Carta cartas[];  
13.        private int iteraciones_barajar = 5;  
14.        private static final int NUM_CARTAS=48;  
15.        public Baraja() {  
16.            //se crea el array  
17.            cartas = new Carta[Baraja.NUM_CARTAS];  
18.            int contador = 0;
```

```
19.          //se crean e inicializan cada una de las cartas
20.          for (ORLA o : ORLA.values()) {
21.              for (NUMERO n : Carta.NUMERO.values()) {
22.                  cartas[contador] = new Carta(o, n);
23.                  contador++;
24.              }
25.          }
26.      }
27.      public Carta getCarta(int index){
28.          if(index >0 && index <cartas.length)
29.              return cartas[index];
30.          else
31.              return null;
32.      }
33.      public void imprimir() {
34.          for (int i = 0; i < cartas.length; i++) {
35.              System.out.println("Carta orla:" +
36.                  cartas[i].getPalo().name() + ", numero:" + cartas[i].getNumero().name());
37.          }
38.      public void mezclar() {
39.          int aleatorio;
40.          Carta temporal;
41.          for (int i = 0; i < iteraciones_barajar; i++) {
42.              for (int j = 0; j < cartas.length; j++) {
43.                  aleatorio = (int) (Math.random() * cartas.length);
44.                  temporal = cartas[j];
45.                  cartas[j] = cartas[aleatorio];
46.                  cartas[aleatorio] = temporal;
47.              }
48.          }
49.      }
50.      public static void main(String args[]){
51.          Baraja b= new Baraja();
52.          b.mezclar();
53.          b.imprimir();
54.      }
55.  }
```

En la línea 3 y 4 se hace import de los enumerados ORLA y NUMERO, esto es posible ya

que Carta se ha definido con default, si se hubiera definido como private no hubiera sido posible, a pesar de tener protected en los enumerados.

En las líneas 12 y 13 se definen los atributos de la clase como private, esto hace que no sean accesibles ni desde fuera del paquete, ni desde dentro, ni subclases independientemente de como se declare la clase, solo es accesible desde el interior de la misma.

En la línea 14 se define un atributo estático (constante) pero a nivel de clase al tener la palabra reservada static.



**Observar la línea 15, en el constructor cómo se crea el array de objetos de tipo Carta y para cada índice se crea un objeto de la clase carta.**

### 2.2.3.1 *This.*

La palabra reservada this hace referencia al objeto en el que se está ejecutando el código y por tanto se dispone de todos los métodos y atributos de dicho objeto.

El uso de this ofrece una serie de ventajas:

- De forma rápida se ve que se están usando métodos o atributos del objeto actual.
- Parámetros del constructor u otros métodos se llamen igual que los atributos.
- Se puede usar como parámetro a métodos, ya sea de si mismo o de otros objetos.

En el constructora clase Carta los parámetros tienen un nombre diferente, para que se puedan llamar igual se usa this:

```
public Carta(ORLA palo,NUMERO numero){  
    this.palo=palo;  
    this.numero=numero;  
}
```

De igual forma se pueden llamar a los métodos del objeto de forma interna (se puede hacer también sin this, pero **se pierde legibilidad**), por ejemplo mezclar al crear la baraja:

```
public Baraja() {  
    //se crea el array  
    this.cartas = new Carta[Baraja.NUM_CARTAS];  
    int contador = 0;  
    //se crean e inicializan cada una de las cartas  
    for (ORLA o : ORLA.values()) {
```

```
for (NUMERO n : Carta.NUMERO.values()) {  
    this.cartas[contador] = new Carta(o, n);  
    contador++;  
}  
}  
this.mezclar();  
}
```

## 2.2.4 Métodos.

Los métodos definen el comportamiento y las acciones que puede realizar un objeto de una clase, estas acciones pueden cambiar el estado interno de los objetos (atributos), estos métodos poseen al igual que las clases y los atributos modificadores de acceso (public, protected, private y default) y se comportan igual que en los atributos.

Dentro de los métodos se pueden definir y utilizar variables de los tipos que se necesite, acceder. y modifica atributos del objeto y llamar a otros métodos del objeto, clase u objetos creados. La sintaxis básica de definición de un método es:

```
[modificador_acceso]  
[static]  
[final]  
tipo_de_dato_devuelto nombre ( tipo_de_dato p1, tipo_de_dato p2...){  
    BLOQUE CÓDIGO  
}
```

### 2.2.4.1 Definición de argumentos y parámetros en métodos.

El comportamiento es similar por no decir idéntico a las funciones, con la salvedad de que todo argumento se **pasa por valor en Java**, es decir una copia de parámetro.

```
class B {  
    public B() {  
    }  
    //EN JAVA SIEMPRE SE PASA POR VALOR  
    public void pasoparametro(int numero) {  
        numero++;  
        System.out.println("Dentro del método el valor al incrementar es " +  
numero);  
    }  
    public static void main(String args[]) {  
        int numero_externo = 6;
```

```
B b = new B();  
System.out.println("Antes de enviar el mensaje al método " +  
numero_externo);  
b.pasoparametro(numero_externo);  
System.out.println("A continuación de la llamada al método " +  
numero_externo);  
}  
}
```

La salida del anterior código es:

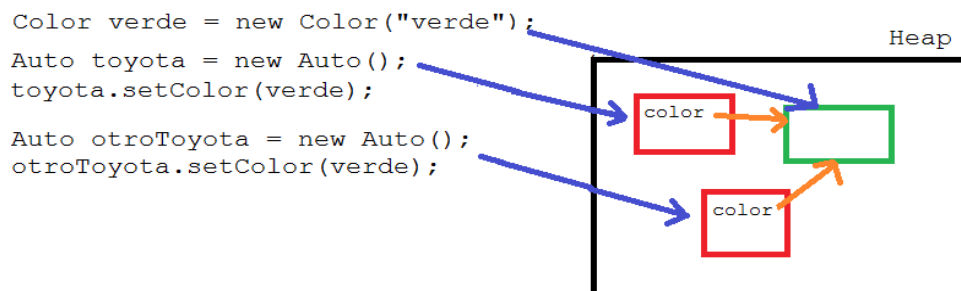
```
Antes de enviar el mensaje al método 6  
Dentro del método el valor al incrementar es 7  
A continuación de la llamada al método 6
```



**Recordar en JAVA los parámetros a los métodos SIEMPRE se pasan por valor nunca por referencia.**

Los objetos también se pasan por valor, pero a efectos prácticos se pasa por referencia, ya que **las modificaciones de los objetos dentro de los métodos son persistente**. La razón es la gestión que hace Java de los objetos y el concepto de referencia.

Una **referencia en Java es un identificador de la instancia de una clase**, es decir un objeto. Al hacer new se está creando una referencia al objeto en memoria. Es posible por tanto referenciar al mismo objeto con diferentes variables (conocido como aliasing).



**Cuando se pasa un objeto como parámetro en un método se está pasando una copia de la referencia al objeto, es decir sería por valor, pero al “apuntar” a la misma área de memoria los efectos prácticos son por referencia.**

En el siguiente ejemplo se encapsula un entero dentro de un objeto y se pasa como parámetro:

```
class B {
```

```
public B() {

}

public void pasoparametro(A referencia) {
    referencia.inc();
    System.out.println(referencia.hashCode());
    System.out.println("Dentro después de incrementar vale " +
referencia.getNumero());
}

public static void main(String args[]) {
    A a = new A(6);
    B b = new B();
    System.out.println(a.hashCode());
    System.out.println("Antes de enviar el mensaje al método " +
a.getNumero());
    b.pasoparametro(a);
    System.out.println("A continuación de la llamada al método " +
a.getNumero());
}
}

class A {
    private int numero;
    public A(int valor) {
        this.numero = valor;
    }
    public int getNumero() {
        return this.numero;
    }
    public void setNumero(int numero) {
        this.numero = numero;
    }
    public void inc() {
        this.numero++;
    }
}
```

La salida del programa anterior es:

1072591677

Antes de enviar el mensaje al método 6

1072591677

Dentro después de incrementar vale 7



## A continuación de la llamada al método 7

Se puede ver que es el mismo objeto (en memoria) usar el método hashCode() que devuelve una función resumen del objeto siendo el mismo.

### 2.2.5 Métodos y atributos estáticos.

En ocasiones existen ciertas operaciones y valores que son necesarios asociar a las clases y no a los objetos. A lo largo del curso se han usado en varias ocasiones, por ejemplo el método public static void main (String args[]), Math.PI o Math.pow.

Para ver los métodos y atributos estáticos simplemente poner el nombre de la clase, y aparecerá el listado de métodos y atributos estáticos de la clase (en la mayoría de los IDE's) o ir a la documentación o API.

Por ejemplo la clase Int posee casi todo su código como static:

The screenshot shows the Java IDE interface with the `Integer` class selected. The `CLASE` tab is active, displaying a list of static methods and attributes. A red box highlights the `Métodos y atributos` section, and another red box highlights the `Documentación` section.

**Métodos y atributos:**

- `MAX_VALUE` (int)
- `MIN_VALUE` (int)
- `SIZE` (int)
- `TYPE` (Class<Integer>)
- `bitCount(int i)` (int)
- `compare(int x, int y)` (int)
- `compareUnsigned(int x, int y)` (int)
- `decode(String nm)` (Integer)
- `divideUnsigned(int dividend, int divisor)` (int)
- `getInteger(String nm)` (Integer)
- `getInteger(String nm, Integer val)` (Integer)
- `getInteger(String nm, int val)` (Integer)
- `hashCode(int value)` (int)
- `highestOneBit(int i)` (int)
- `lowestOneBit(int i)` (int)
- `max(int a, int b)` (int)
- `min(int a, int b)` (int)

**Documentación:**

`java.lang.Integer`

public static final int BYTES = 4

The number of bytes used to represent an `int` value in two's complement binary form.

Since:

1.8

Para definir un atributo como estático en una clase simplemente indicar la palabra reservada **static** antes del tipo de dato, pudiéndose combinar con otras como la visibilidad.

Se ha de tener en cuenta que:

- Se ha de inicializar al declarar.

- Se puede modificar el valor.
- Si se define como **final** no es posible modificar el valor.

Un ejemplo:

```
public class Claseestatica {  
  
    public static int atributoestatico = 5;  
    public static final int atributofinalestatico = 45;  
  
    public static void main(String args[]) {  
        System.out.println("El valor de atributo estatico:" +  
Claseestatica.atributoestatico);  
        Claseestatica.atributoestatico = 6;  
        System.out.println("El valor de atributo estatico una vez modificado:"  
+ Claseestatica.atributoestatico);  
        System.out.println("El valor de atributo estatico final una vez  
modificado:"+Claseestatica.atributofinalestatico);  
        //no es posible modifcar el valor del atributo final  
        //Claseestatica.atributofinalestatico=55;  
    }  
}
```

En cuanto a los métodos es similar, pero se ha de tener en cuenta:

- Se recomienda usar para manipular atributos estáticos.
- Su funcionalidad suele ser auxiliar a la clase, o la misma clase es una clase auxiliar como Integer o Math.
- No tienen acceso a los elementos no estáticos de la clase, es decir a objetos.

Temas más avanzados son la inicialización de estructuras más complejas como atributos estáticos o las clases estáticas relacionadas con patrones como Singleton, para profundizar visitar:

<https://www.baeldung.com/java-static>

Un ejemplo de uso de métodos estáticos:

```
/**  
 *  
 * @author Pedro  
 */  
public class Coche {
```

```
//numero de objetos creado
private static int numerodeobjetos = 0;
//se obtiene en numero de coches creados
public static int getNumCoches() {
    return numerodeobjetos;
}
//servicio encargado de crear coches, no se llama al constructor
//patron muy usado en framework web
public static Coche service() {
    return new Coche();
}
private Coche() {
    Coche.numerodeobjetos++;
}

public static void main(String args[]) {
    Coche c= Coche.service();
    System.out.println("Se han creado "+Coche.getNumCoches());
}
}
```

## 2.2.6 Getters and setters.

La encapsulación y la ofuscación de información son principios de los lenguajes OO y los atributos forman parte de la estructura interna de la clase que han de ser ocultados a los usuarios de dicha clase. En caso de realizarse una modificación interna de la clase, por ejemplo un nuevo algoritmo de ordenación, si se protegen los atributos internos y por supuesto también los métodos que no sea necesario exponer en el exterior, **estos cambios internos no tendrán** consecuencias en el resto del código que haga uso del objeto.

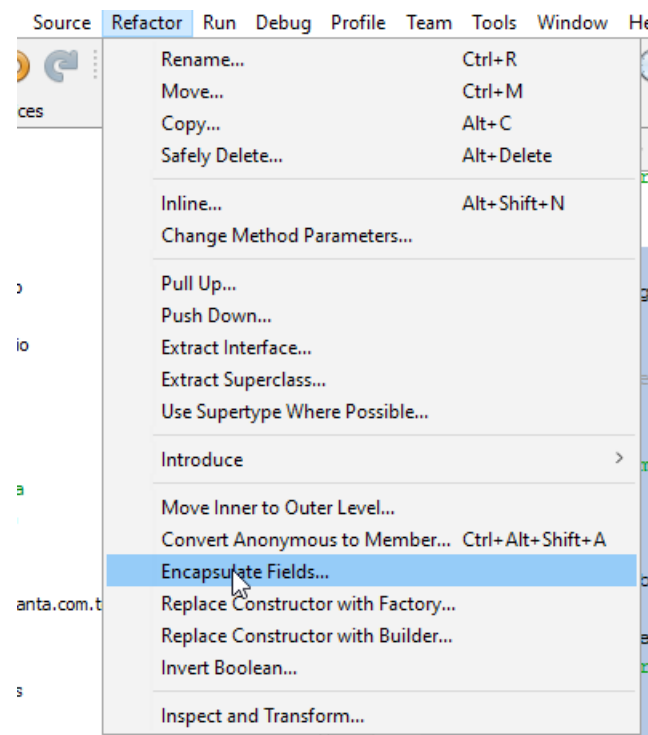


**Los atributos de una clase se han de declarar protected o private para ocultar la información y los detalles internos.**

Por convención cada atributo tiene un método que permite obtenerlo y un método que permite asignar un valor a dicho atributo. Por ejemplo sea la clase Persona con un atributo apellidos de tipo String y con acceso private, se definen dos métodos get y set para manipular dicho atributo.

```
class Persona{
    private String nombre;
    /**
     * @return the nombre
     */
    public String getNombre() {
        return nombre;
    }
    /**
     * @param nombre the nombre to set
     */
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Los IDE's suelen tener funcionalidades para la generación de código de forma automática, por ejemplo NetBeans en el menú Refactor posee una opción para genera de forma automática estos métodos:



En otros lenguajes como C# a la combinación de atributo, get y set **se le denomina propiedad**, definiéndose el atributo en minúsculas y la propiedad accesible desde el exterior con la primera letra en mayúsculas. Un ejemplo en C#:

```
class Person
```

```
{  
    private String name; // field  
  
    public String Name    // property  
    {  
        get { return name; }    // get method  
        set { name = value; }    // set method  
    }  
}
```

## 2.3. Objetos y gestión de memoria.

En Java la gestión de los objetos en memoria se delega en el recolector de basura, el desarrollador o desarrolladora solo se ha de encargar de crear el objeto en memoria y no de su eliminación cuando ya no es necesario.

En C/C++ la liberación de memoria reservada de forma dinámica(tiempo de ejecución) es tarea del desarrollados, y en caso de no realizarse de forma correcta puede llegar a desperdiciar memoria o incluso a llenarla, teniendo las clases en C++ un método adicional llamado destructor con el símbolo circunflejo ~ y el nombre de la clase. En Java se tiene el **método finalize**, y es llamado por el GC (recolector de basura) al liberar el objeto, usado si se tienen recursos del sistema usados por el objeto y se necesitan liberar.

El recolector de basura cada cierto tiempo analiza la memoria y si encuentra alguna región de la misma con objetos no referenciados por alguna variable marca dicha área de memoria como libre para poder volver a utilizarse. Para dejar de referenciar un objeto la variable a de apuntar a null

```
persona= null;
```

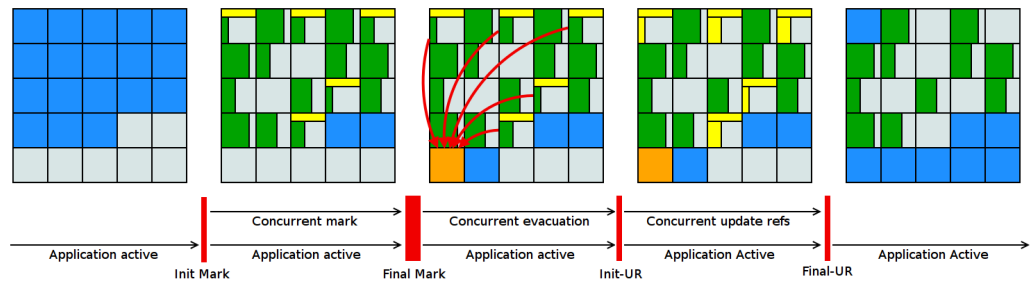
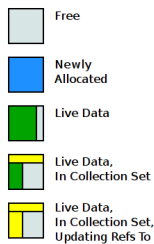
La próxima vez que el recolector evalúe la memoria pasara



¿Qué ventajas aporta el recolector de basura? ¿Y desventajas?

A lo largo de las diferentes versiones de Java se han desarrollado diversos recolectores de basura como el inicial G1, Epsilon GC o el más reciente **Shenandoah GC**.

Legend:



```
GC(3) Pause Init Mark 0.771ms
GC(3) Concurrent marking 76480M->77212M(102400M) 633.213ms
GC(3) Pause Final Mark 1.821ms
GC(3) Concurrent cleanup 77224M->66592M(102400M) 3.112ms
GC(3) Concurrent evacuation 66592M->75640M(102400M) 405.312ms
GC(3) Pause Init Update Refs 0.084ms
GC(3) Concurrent update references 75700M->76424M(102400M) 354.341ms
GC(3) Pause Final Update Refs 0.409ms
GC(3) Concurrent cleanup 76244M->56620M(102400M) 12.242ms
```

En la imagen anterior se puede ver como funciona el recolector Shenandoah.

Además es posible personalizar el comportamiento de cada uno de los diferentes recolectores de basura en cuanto a tiempo, tamaño. Hilos...

**Referido también a la memoria y los objetos es importante destacar que la operación == en las condiciones entre objetos hace referencia a las direcciones de memoria**, para comparar si dos objetos son iguales se ha de usar el **método equals** que todo objeto posee y sobrescribirlo (se tratará en la herencia). Un ejemplo con la clase persona:

```
public class Persona{
    private String nombre;
    public String getNombre() {
        return nombre;
    }
    @Override
    public boolean equals(Object p){
        return this.nombre.equals(((Persona)p).nombre);
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

```
public static void main (String args[]){  
    Persona p1= new Persona() ;  
    p1.setNombre ("Manolo") ;  
    Persona p2 = new Persona() ;  
    p2.setNombre ("Manolo") ;  
    System.out.println("Comparando con ==:"+(p1==p2)) ;  
    System.out.println("Comparando con equals:"+(p1.equals(p2))) ;  
  
    }  
}
```

La salida del programa es:

```
Comparando con ==:false  
Comparando con equals:true
```

Otros conceptos relacionados con los objetos y la memoria son clone y constructores con herencia tratados en próximos temas.

## 2.4. Empaquetado de clases. Organización de las clases en paquetes.

En puntos anteriores se ha tratado del uso de paquetes de tercero, pero en los proyectos propios con un tamaño medio o grande es necesario crear una estructura de las clases y código que facilite su desarrollo, mantenimiento y uso.

Los paquetes se definen con una estructura de árbol, es decir un paquete tiene un paquete superior o padre y puede tener varios paquetes inferiores o hijos, además cada paquete tiene clases en caso de ser hojas del árbol aunque también se pueden clases en paquetes intermedios.

En principio la estructura de los paquetes no ha de tener equivalencia en directorio y subdirectorios del sistema de archivo, aunque es recomendable que para cada nivel de paquete se defina también el directorio asociado.



**Lo primero que se tiene en el fichero es la palabra reservada package y la ruta del árbol de paquetes a la que pertenece el fichero.**

Un ejemplo sin guardar la equivalencia, todos los ficheros java en el mismo directorio pero con diferentes paquetes:

Fichero A.java, paquete paquetenive1:

```
package paquetenivel1;  
  
public class A {  
  
  
}
```

Fichero B.java, paquete paquetenive1.paquetenivel2:

```
package paquetenivel1.paquetenivel2;  
  
public class B {  
  
  
}
```

Y para usar ambas se define la clase Main con las importaciones correspondientes:

```
import paquetenivel1.A;  
import paquetenivel1.paquetenivel2.B;  
  
public class Main {  
    public static void main(String args[]) {  
        //se ha realizado el import de B se puede usar directamente  
        B b = new B();  
        //idem que el anterior.  
        A a = new A();  
    }  
}
```

Y para compilar:

```
javac Main.java A.java B.java
```

Al usar los IDE's la gestión de los paquetes se hace de forma casi automática.

Es importante, en especial al desarrollar librerías o proyectos en los que trabajan diferentes personas establecer estrategias para la estructura de los paquetes. Se busca una alta cohesión y un bajo acoplamiento estableciendo los siguientes principios.

Para una alta cohesión:

- Principio de equivalencia de reutilización-liberación: Cada paquete se ha de crear con clases **reutilizables y de la misma familia. No incluir clases no relacionadas con el propósito del paquete.**
- Principio de reutilización común: Las clases que tienen a utilizarse juntas han de pertenecer al mismo paquete.



- Principio del cierre común: El paquete no ha de tener más de una razón para cambiar, se han de empaquetar juntas si es posible las clases que puedan cambiar en el futuro.

Para un bajo acoplamiento.

- Principio de dependencias acíclicas. No pueden existir ciclos en las dependencias por ejemplo el paquete A necesitar el B, el B el C y a su vez el C necesitar A.
- Principio de dependencias estables. Los paquetes cambian de forma irremediable, este principio indica que se ha de evitar la dependencia hacia un paquete con alta probabilidad de cambio en uno con una probabilidad baja de cambio
- Principio de abstracciones estables. Un paquete estable ha de ser abstracto, para que su estabilidad no impida que se extienda y un paquete inestable (puede cambiar fácilmente) ha de ser concreto. La abstracción se trata en temas posteriores.

## 2.5. Ficheros jar.

Cuando en un proyecto se tienen una estructura de paquetes, clases u otros recursos como imágenes la gestión de todos ellos para su distribución puede ser problemática ya que en caso de perderse uno de ellos puede ser que el programa no funcione.

La solución es compactar todos los ficheros necesarios, en especial los paquetes del propio proyecto y las librerías externas (que son a su vez jar), en ficheros con extensión jar, aunque realmente son ficheros zips.

El comando jar permite crear estos ficheros, la sintaxis del comando es:

```
jar options jar-file [manifest-file] class-files...
```

Las principales opciones:

Opción	Descripción
c	Crear nuevo jar
u	Actualizar jar existente
x	Extraer ficheros
f	Nombre del fichero creado/modificado

v	Verbose. Muestra información del proceso
0	No comprimir el fichero, simplemente empaquetar
m	Fichero manifest, permite indicar opciones como la clase principal o el classpath
e	Indica la clase que contiene el método estático main y que se ejecutara, también se puede indicar con --main-class

Algunos ejemplos extraídos de la documentación oficial:

```
jar --create --file classes.jar Foo.class Bar.class
```

Crea un jar llamado classes.jar con las clases Foo.class y Bar.class

```
jar --create --file classes.jar --manifest mymanifest -C foo/
```

Crea un jar igual que el anterior, con el fichero mymanifest e incluye todos los ficheros que se encuentren en el directorio foo/

```
jar --create --file foo.jar --main-class com.foo.Main
```

Crea un fichero jar indicando que el punto de entrada es la clase com.foo.Main

Para ejecutar un jar simplemente, si se tiene configurado el punto de entrada.

```
java -jar fichero.jar
```

Un elemento importante de todo jar es el fichero manifest que contiene información variada como la clase principal o el classpath, la especificación del fichero para Java 8 se puede encontrar en:

<https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html>

Los IDE's actuales y las herramientas de gestión de proyectos como Ant, Maven o Gradle poseen guiones para la automatización de los ficheros jar, pero un buen profesional ha de conocer los fundamentos para poder modificar y crear jar en caso de ser necesario.

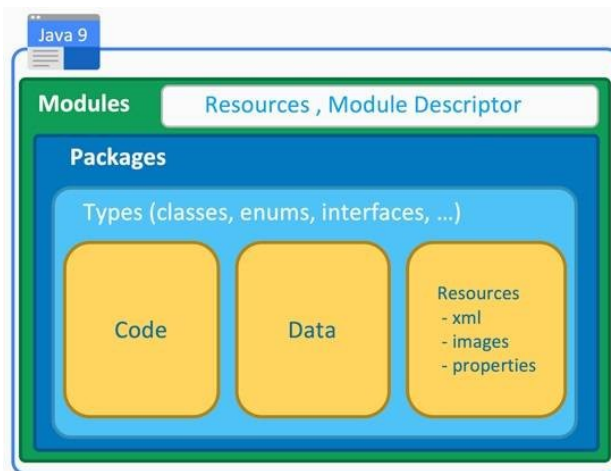
A continuación se realiza un ejemplo de creación de un pequeño juego usando la librería LitienGine

## 2.6. Módulos.

A partir de Java 9 se han introducido los módulos, estos solucionan algunos de los problemas de los ficheros jar como son:

- Limitar el tamaño del entorno de ejecución, por defecto se tiene disponible todas las librerías de J2SE, muchas de ellas no son necesarios. Esto es especialmente útil para las aplicaciones en IoT, a través del concepto de imagen.
- Comprueba previamente a iniciar la aplicación la existencia de las librerías necesarias.
- Los tipos y clases se cargan a mayor velocidad al no necesidad buscar en el classpath.
- Limita el acceso a ciertas clases aportando seguridad.

Un módulo es un jar que agrupa código, recursos y metadatos (describen dependencias con otros módulos y regula el acceso a dicho módulo).



El módulo contiene un fichero llamado module-info.java en el que se definen las características del módulo como:

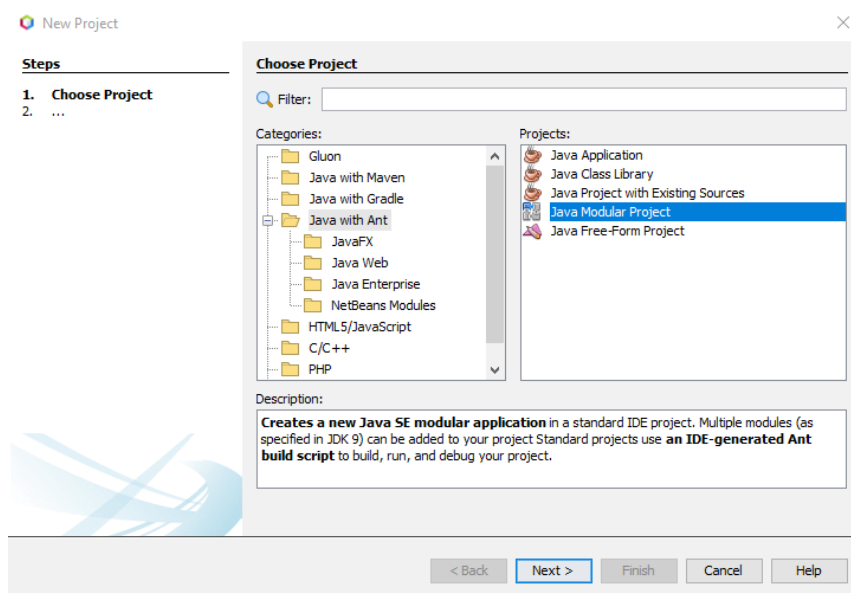
- Nombre del módulo.
- Paquetes expuestos.
- Dependencias con otros módulos.
- Servicios consumidos e implementados. (Los servicios se tratarán en otros temas y en módulos de segundo curso).
- Permisos de reflexión. La reflexión permite instanciar objetos de una clase, o llamar

a métodos a partir de una cadena.

Se establecen 4 tipos de módulos:

- De sistema. Incluidos en JSE y JDK.
- Aplicaciones: Empaquetan las aplicaciones, los módulos y todo lo necesario para tener una aplicación independiente, normalmente es el objetivo a la hora de crear los módulos. **No es necesario tener Java preinstalado. Se utiliza la herramienta Jlink** que permite crear una imagen de JSE con lo necesario para ejecutar una aplicación.
- Automáticos: Módulos no oficiales, tendrán acceso al resto de los módulos. Se crea a partir de ficheros JAR en versiones Java 8 o anterior. Estos módulos exportan todos los paquetes
- Sin nombre: Cuando se carga una clase o JAR en la ruta de las clases pero no en la del módulo. Utilizado para bibliotecas antiguas junto con el argumento -classpath

En NetBeans existen un tipo de proyecto con Ant que permite crear módulos:



Poner vídeo.

## 2.7. Documentación.

Las bibliotecas poseen decenas de clases, cada una de ellas con atributos, métodos, a su

vez estos métodos tiene un listado de parámetros con un tipo y un significado especial. Se hace por tanto imprescindible disponer de una documentación que explique las clases disponibles, los atributos, tarea de cada método, valores devueltos o que función tiene cada parámetro de los métodos entre otros.

Para ello se definen las interfaz de programación de aplicaciones o API. Normalmente esta API se ofrece en formato HTML, aunque se puede generar XML, RIF o RTF, y puede estar en una página web o incluirse en el jar.

Se puede definir documentación a nivel de paquete, de clase, de atributo y de método. Un comentario para Javadoc se inicia con **\*\*** (barra y dos asteriscos) y se finaliza con **\*\*** (dos asteriscos y barra).

Se definen a su vez una serie de “tags” que aportan significado a ciertas partes del código, como:

@author

@deprecated      Indica que ese atributo o método está próximo a retirarse del paquete y en próximas versiones puede que desaparezca.

@param            Se le da el nombre del parámetro un espacio y la descripción del mismo.

@return           Indica que devuelve el método.

@exception       El método puede lanzar una excepción, indicando el tipo, se trata en temas posteriores.

@see               Referencia a otra clase y método

@version          Versión del código

Un ejemplo de uso de documentar clases con Javadoc, en el que se solicita la creación de un sistema de lista de recursos multimedia (vídeos, url, imágenes y música), cada usuario posee varias colecciones (en este caso se usan vectores, en temas posteriores se tratarán estructuras más adecuadas), cada colección se compone de recursos, estos recursos son de los tipos anteriores (ahora se usa un enumerado para indicar el tipo, posteriormente se verán mejores formas de solucionarlo).

Clase Recurso:

```
import java.util.Date;

/**
 *
 * @author Pedro
 *
 */
enum TipoRecurso{
    VIDEO,
    MUSICA,
    URL,
    IMAGEN
}

/**
 * Gestion de recursos
 * @author Pedro
 * @see Coleccion
 */
public class Recurso {
    private String localizacion;
    private TipoRecurso recurso;
    private Date fecha_alta;
    /**
     * Constructor por defecto
     */
    public Recurso() {
    }
    /**
     * Constructor sobrecargado
     * @param recurso indica el tipo de recurso
     * @param localizacion cadena de texto que indica cómo llegar al recurso
     */
    public Recurso(TipoRecurso recurso,String localizacion){
        this.recurso=recurso;
        this.localizacion=localizacion;
    }
    /**
     * Constructor sobrecargado con la fecha
     * @param recurso indica el tipo de recurso
     * @param localizacion cadena de texto que indica cómo llegar al recurso
     * @param fecha fecha de alta del recurso
     */
}
```

```
*/  
  
public Recurso(TipoRecurso recurso,String localizacion,Date fecha){  
    this.recurso=recurso;  
    this.localizacion=localizacion;  
    this.fecha_alta=fecha;  
}  
/**  
 * @return the localizacion  
 */  
public String getLocalizacion() {  
    return localizacion;  
}  
  
/**  
 * @return the recurso  
 */  
public TipoRecurso getRecurso() {  
    return recurso;  
}  
  
/**  
 * @return the fecha_alta  
 */  
public Date getFecha_alta() {  
    return fecha_alta;  
}  
  
/**  
 * @param fecha_alta the fecha_alta to set  
 */  
public void setFecha_alta(Date fecha_alta) {  
    this.fecha_alta = fecha_alta;  
}  
  
/**  
 * @param localizacion the localizacion to set  
 */  
public void setLocalizacion(String localizacion) {  
    this.localizacion = localizacion;  
}
```

```
/**
 * @param recurso the recurso to set
 */
public void setRecurso(TipoRecurso recurso) {
    this.recurso = recurso;
}
}
```

Clase colección:

```
import java.util.Date;

/**
 *
 * @author Pedro
 * Gestion de recursos con una coleccion, permite añadir, borrar y obtener los
recursos.
 * @see Recurso
 * @see Usuario
 */
public class Coleccion {

    /**
     * Tamanyo por defecto de la coleccion
     */
    public static int tam_coleccion = 5;
    private String titulo;
    private Date fecha_alta;
    private Recurso recursos[];

    /**
     * constructor por defecto
     */
    public Coleccion() {
        this.recursos = new Recurso[tam_coleccion];
        //contiene la fecha actual
        this.fecha_alta = new Date();
    }

    /**
     * Constructor s obrecargado

```



```
*  
  
* @param titulo titulo de la coleccion  
* @param fecha_alta fecha de alta de la coleccion  
*/  
public Coleccion(String titulo, Date fecha_alta) {  
    this.recursos = new Recurso[tam_coleccion];  
    this.fecha_alta = fecha_alta;  
    this.titulo = titulo;  
}  
  
/**  
 * devuelve la posición del primer elemento del vector vacio (null)  
 *  
 * @return  
 */  
private int getIndexRecurso() {  
    int i = 0;  
    for (i = 0; i < this.recursos.length && this.recursos[i] != null; i++)  
{  
        }  
    return i;  
}  
  
/**  
 * Anyade un recurso a la coleccion  
 *  
 * @param r  
 * @return booleano que indica si se ha podido insertar o no  
 * @see Recurso  
 */  
public boolean addRecurso(Recurso r) {  
    int indice = this.getIndexRecurso();  
    if (indice < this.recursos.length) {  
        this.recursos[indice] = r;  
        return true;  
    }  
    return false;  
}  
  
/**  
 * devuelve
```

```
*  
  
* @param index  
* @return devuelve el recurso  
* @see Recurso  
*/  
  
public Recurso getRecurso(int index) {  
    if (index > 0 && index < this.recursos.length) {  
        return this.recursos[index];  
    } else {  
        return null;  
    }  
}  
/**  
 * Borra un recurso  
 * @param index  
 * @return devuelve si se pudo o no realizar la operacion  
 */  
public boolean removeRecurso(int index) {  
    int indice = this.getIndexRecurso();  
    if (indice < this.recursos.length) {  
        this.recursos[indice] = null;  
        return true;  
    }  
    return false;  
}  
  
/**  
 * @return the titulo  
 */  
public String getTitulo() {  
    return titulo;  
}  
  
/**  
 * @return the fecha_alta  
 */  
public Date getFecha_alta() {  
    return fecha_alta;  
}  
  
/**
```

```
* @return the recursos
*/
public Recurso[] getRecursos() {
    return recursos;
}

/**
 * @param titulo the titulo to set
 */
public void setTitulo(String titulo) {
    this.titulo = titulo;
}

/**
 * @param fecha_alta the fecha_alta to set
 */
public void setFecha_alta(Date fecha_alta) {
    this.fecha_alta = fecha_alta;
}

/**
 * @param recursos the recursos to set
 */
public void setRecursos(Recurso[] recursos) {
    this.recursos = recursos;
}
}
```

Clase usuario:

```
/**
 * Usuarios de la aplicacion
 * @author Pedro
 * @see Recurso
 * @see Coleccion
 */
public class Usuario {
    private static int tam_colecciones=5;

    private String usuario;
    private Coleccion colecciones[];
}
```

```
* Constructor por defecto
*/
public Usuario() {
    this.colecciones=new Coleccion[tam_colecciones];

}
/**
 * Constructor sobrecargado
 * @param nombre nombre del usuario
 */
public Usuario(String nombre){
    this.usuario=nombre;
    this.colecciones= new Coleccion[tam_colecciones];
}

/**
 * devuelve la posición del primer elemento del vector vacio (null)
 *
 * @return
 */
private int getIndexColeccion() {
    int i = 0;
    for (i = 0; i < this.colecciones.length && this.colecciones[i] != null;
i++) {
        }
    return i;
}

/**
 * Anyade un recurso a la coleccion
 *
 * @param c
 * @return booleano que indica si se ha podido insertar o no
 * @see Coleccion
 */
public boolean addColeccion(Coleccion c) {
    int indice = this.getIndexColeccion();
    if (indice < this.colecciones.length) {
        this.colecciones[indice] = c;
        return true;
    }
}
```

```
    }  
    return false;  
}  
  
/**  
 * devuelve  
 *  
 * @param index  
 * @return devuelve la coleccion  
 * @see Coleccion  
 */  
public Coleccion getRecurso(int index) {  
    if (index > 0 && index < this.colecciones.length) {  
        return this.colecciones[index];  
    } else {  
        return null;  
    }  
}  
  
/**  
 * Borra una coleccion  
 * @param index  
 * @return devuelve si se pudo o no realizar la operacion  
 */  
public boolean removeColeccion(int index) {  
    int indice = this.getIndexColeccion();  
    if (indice < this.colecciones.length) {  
        this.colecciones[indice] = null;  
        return true;  
    }  
    return false;  
}  
  
/**  
 * @return the usuario  
 */  
public String getUsuario() {  
    return usuario;  
}  
  
/**  
 * @return the colecciones
```

```

*/

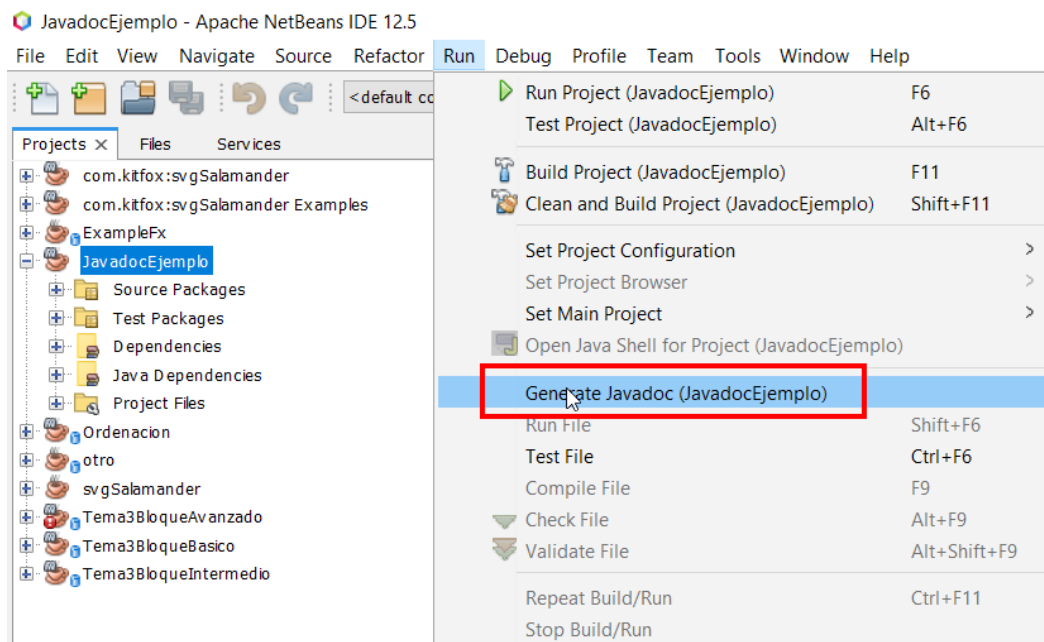
public Coleccion[] getColecciones() {
    return colecciones;
}

/**
 * @param usuario the usuario to set
 */
public void setUsuario(String usuario) {
    this.usuario = usuario;
}

/**
 * @param colecciones the colecciones to set
 */
public void setColecciones(Coleccion[] colecciones) {
    this.colecciones = colecciones;
}
}

```

Ahora se puede generar la documentación, por defecto en formato HTML. Se puede usar el IDE como NetBeans con Run→ Generate Javadoc.



O realizarlo desde la línea de comandos, con más opciones, la sintaxis es la siguiente:

```
javadoc PACKAGE|SOURCE_FILE OPTIONS @ARGFILES
```

Un ejemplo para el proyecto anterior:

```
C:\Users\Pedro\Desktop\JavadocEjemplo\src\main\java> javadoc -d ../../../../api
pedro.ieslaencanta.com.javadocejemplo.gestorrecurso
```

Que genera la documentación en formato HTML en C:\Users\Pedro\Desktop\JavadocEjemplo\api.

Package `pedro.ieslaencanta.com.javadocejemplo.gestorrecurso`

**Class Usuario**

`java.lang.Object`  
`pedro.ieslaencanta.com.javadocejemplo.gestorrecurso.Usuario`

`public class Usuario`  
`extends Object`

Usuarios de la aplicacion

See Also:  
`Recurso`, `Coleccion`

**Constructor Summary**

Constructor	Description
<code>Usuario()</code>	Constructor por defecto
<code>Usuario(String nombre)</code>	Constructor sobrecargado

**Method Summary**

Modifier and Type	Method	Description
<code>boolean</code>	<code>addColeccion(Coleccion c)</code>	Anyade un recurso a la coleccion
<code>Coleccion[]</code>	<code>getColecciones()</code>	
<code>Coleccion</code>	<code>getRecurso(int index)</code>	devuelve
<code>String</code>	<code>getUsuario()</code>	
<code>boolean</code>	<code>removeColeccion(int index)</code>	Borra una coleccion
<code>void</code>	<code>setColecciones(Coleccion[] colecciones)</code>	
<code>void</code>	<code>setUsuario(String usuario)</code>	

### 3. Actividades y ejercicios.

1. ¿Qué problemas tiene la programación estructurada?
2. Explicar con tus palabras los fundamentos de la POO.
3. ¿Qué es una clase? ¿Y un objeto? ¿Cuántos objetos de una clase se pueden tener?

4. Las clases unen los \_\_\_\_\_ y los \_\_\_\_\_ en \_\_\_\_\_ y \_\_\_\_\_, comunicándose entre ellos usando \_\_\_\_\_.
5. Crear una clase para representar el NIF de una persona.
6. ¿Cuántas clases puede haber en un fichero .java?
7. ¿Cómo incluir código externo en ficheros jar en un proyecto sin usar el IDE o gestores de proyecto?
8. ¿Qué es un fichero jar?
9. ¿Qué es un constructor? ¿Cuándo se llama?
10. ¿Qué puede suceder si al usar un atributo de una clase se obtiene una excepción de tipo java.lang.NullPointerException?
11. ¿Cuántos constructores se puede tener en una clase?
12. ¿Qué es el constructor por defecto? ¿Qué hace?
13. El siguiente código tiene al menos 2 fallos importantes, ¿Qué excepciones se causan al ejecutar? ¿A qué es debido?.

```
import java.util.Date;
public class Coche {
    Motor motor;
    String matricula;
    int puertas;
    int peso;
    public void arranca() {
        motor.arrancar();
    }
    public void parar() {
        motor.parar();
    }
    public void imprimir_fecha_revision() {
        System.out.println("La última revisión fue"+motor.fecha_revision.toString());
    }
    public static void main(String args[]) {
        Coche c;
        c= new Coche();
        c.arranca();
        c.parar();
    }
}
```



```
c.imprimir_fecha_revision();  
    }  
}  
  
enum ESTADOS_MOTOR{  
    ENCENDIDO,  
    APAGADO,  
    AVERIADO  
}  
  
class Motor{  
    int cilindrada;  
    int potencia;  
    Date fecha_revision;  
    ESTADOS_MOTOR estado;  
    public void arrancar() {  
        estado=ESTADOS_MOTOR.ENCENDIDO;  
    }  
    public void parar() {  
        estado=ESTADOS_MOTOR.APAGADO;  
    }  
}
```

14. Modificar el código anterior (razonando las modificaciones), para que funcione.
15. Se tiene una librería externa en ./lib para genera pdf llamada JavaPDF.jar. Indicar cómo compilar y ejecutar el programa que se encuentra en ./principal.java y hace uso de la librería.
16. ¿Qué son los modificadores de acceso y que función tienen? ¿A qué elementos se aplica?
17. Sea el siguiente código:
18. ¿Qué es this? ¿Para qué se utiliza this?
19. En Java los parámetros se pasan por valor, pero los cambios en los objetos dentro de métodos son permanentes. Explicar la razón.
20. Indicar la diferencia entre usar == y el método equals al comparar objetos.
21. ¿Qué es un atributo estático? ¿Para qué sirve?
22. Se tiene un juego de estrategia y se quiere contar el número de arqueros totales creado. ¿Cómo hacerlo?

23. ¿Es recomendable declarar los atributos como privados? De ser así ¿Cómo se accede y modifica dichos atributos?
24. ¿Qué es una propiedad? ¿Existe en Java?.
25. En C++ ¿Como se destruyen los objetos? ¿Es necesario desarrollar algo en la clase para poder destruirlos?
26. ¿Cómo se destruyen los objetos en Java?
27. ¿Cómo sabe el GC que puede liberar el espacio reservado para un objeto?
28. Definir qué es un paquete.
29. Explicar la forma de desarrollar paquetes con un algo grado de cohesión y bajo acoplamiento.
30. ¿Qué es un fichero JAR? ¿Qué contiene?
31. Se quiere distribuir una aplicación y asegurarse de que funcionará incluso sin Java instalado en los equipos. ¿Cómo hacerlo?
32. ¿Qué herramienta se utiliza la documentación en Java? ¿Qué son los tags?