# Fundamentos de la Programación Orientada a Objetos

* Programación orientada a objetos

Paradigma de programación basado en el concepto de clases y objetos y las relaciones que se pueden establecer entre ellos.

* Clase

Plantilla que va a definir de manera genérica la tipología de los objetos.

* Objeto

Entidad concreta provista de un conjunto de propiedades y comportamientos que se crea a partir de una clase.

* Instancia

Particularización de una determinada clase.

* Encapsulación

Mecanismo que nos permite controlar el acceso a los componentes de una clase.

* Abstracción

Proceso mediante el cual representamos inequívocamente las características de un objeto real.

* Herencia

Relación jerárquica que permite la reutilización de código definiendo de una clase otras nuevas que supongan una especialización o una generalización.

* Polimorfismo

Capacidad que tienen los objetos de una clase de ofrecer una respuesta en función de sus características.

* SOLID

Acrónimo que representa los cinco principios básicos de la programación orientada a objetos.

* S: principio de responsabilidad única

Cada clase solo debería ocuparse de una cosa.

* Declaración e inicialización de un array
* tipo\_dato nombre\_array [ ]= new tipo\_dato[tamaño];

Ejemplo: String b[] = new String[9];

b[0] = “Madrid”;

b[1] = “Segovia”;

b[2] = “Toledo”;

* tipo\_dato nombre\_array [ ]= {elemento1, elemento2, elemento3, … };

Ejemplo: String b[ ] = {“Madrid”, “Segovia”, “Toledo};

* Proceso

Instancia de un programa en ejecución.

En el momento en el que un proceso se lanza, el sistema operativo proporciona un espacio de direcciones de memoria en los que el proceso puede ejecutarse.

Un proceso se puede dividir en tareas, también llamadas hilos.

* Hilos (threads)

Trozo de código de nuestro programa que puede ser ejecutado al mismo tiempo que otro.

Para ello, lo que tenemos que hacer es heredar la clase Thread (con extends en nuestra clase) y redefinir el método run(), ya que Thread implementa la interface Runnable.



* Diferencia entre hilos y procesos

La principal diferencia entre un proceso y un hilo es que los procesos tienen su propia zona de memoria mientras que los hilos de un proceso comparten la misma zona de memoria.

Por ello, un proceso es más costoso de lanzar mientras que un hilo es más ligero.

Los hilos no son independientes entre sí.

* Stack (Pila)

Sección de memoria que se utiliza para almacenar variables locales, variables de referencia, parámetros y valores de retorno, es decir, resultados parciales. Además, se utiliza para llevar el control de la invocación y el retorno de los métodos.

* Heap (Montículo)

Zona de memoria que almacena objetos y sus variables de instancia. Se trata de un espacio de memoria dinámica que se crea al inicio de la máquina virtual y además es único.

* Garbage Collector (Recolector de Basura)

Administrador de la memoria dinámica. Cuando un objeto no es referenciado, el garbage collector se encarga de liberar esa memoria.

* Anotación

Información adicional que se añade a nuestro código. Se conoce como metadatos.

Se identifican con un @ delante del nombre de la anotación.

Ejemplo: @Anotacion

Las siguientes anotaciones vienen ya definidas en Java y son de propósito general:

* @Override

Aplicamos esta anotación a un método para indicar que con él sobrescribimos el comportamiento de ese método heredado de otra clase o interfaz, anulando el de la superclase.

* @Deprecated

Anotación que nos permite indicar que un método, clase o atributo está obsoleto y no se debería usar. Si se usan, el compilador producirá una advertencia.

* @SuppressWarnings

Evita que en nuestro entorno de desarrollo se muestren una o más advertencias sobre nuestro código. Las advertencias que queremos suprimir se especifican por su nombre y en forma de String. Tenemos dos categorías, deprecation o unchecked.

* @SafeVarargs

Anotación que vamos a usar para garantizar que no se harán operaciones inseguras con parámetros variables. Se puede aplicar tanto a métodos como a constructores.

* @FunctionalInterface

Anotación que se utiliza en la declaración de una interfaz para indicar que es funcional, es decir, que solo contiene un método abstracto. Se trata de una anotación informativa, ya que cualquier interfaz con un método abstracto es una interfaz funcional.

* @Retention

Especifica el método de almacenamiento de las anotaciones. Se pueden diferenciar 3:

* RetentionPolicy.SOURCE

Las anotaciones solo se mantienen en el nivel de fuente y el compilador las ignora.

* RetentionPolicy.CLASS

Las anotaciones son recogidas por el compilador, pero la Máquina Virtual de Java (JVM) las ignora.

* RetentionPolicy.RUNTIME

Las anotaciones son recogidas por la JVM, por lo que el entorno de ejecución puede utilizarlas.

* @Documented

Se utiliza para indicar que se pueden recoger los elementos de las anotaciones dentro del javadoc.

* @Target

Especifica los tipos de elementos de Java a los que se puede aplicar una anotación. Tiene un argumento en el que se indica el o los ElementType:

* ElementType.ANNOTATION\_TYPE

Se aplica a tipos de anotaciones.

* ElementType.CONSTRUCTOR

Se aplica a los constructores.

* ElementType.FIELD

Se aplica a los atributos

* ElementType.LOCAL\_VARIABLE

Se aplica a las variables locales.

* ElementType.METHOD

Se aplica a los métodos.

* ElementType.PACKAGE

Se aplica a declaraciones de paquetes

* ElementType.PARAMETER

Se aplica a la declaración de parámetros

* ElementType.TYPE

Se aplica a la clase, interfaz o enumeración.

Si @Traget no está especificado, la notación se puede utilizar en cualquier declaración.

* @Inherited

Anotación que hace que la clase hija herede las anotaciones de la clase padre en el caso de que la clase hija no tenga ninguna anotación.

Acabamos de ver algunas anotaciones predefinidas, pero también podemos crear anotaciones personalizadas.

Para crear nuestras propias a notaciones tenemos que seguir los siguientes pasos:

* Inicialmente, creamos un proyecto (File -> New Project -> Java Proyect).
* Añadimos una anotación al proyecto (New -> Annotation) y podemos ver a continuación el contenido del archivo que se genera:



Una anotación se define haciendo uso de la palabra reservada @interface.

También podemos crearla añadiendo una clase y cambiando la palabra class por @interface.

* Ahora, tenemos que añadir los parámetros que queremos que contenga.

En este ejemplo, nuestra anotación se escribirá como:

@MiAnotacion( parametro1 = “valor2”)

Pero vamos a tener dentro 3 parámetros:

* Parametro1: obligatorio de tipo String.
* Parametro2: opcional de tipo array cuyo valor por defecto es “canada, españa”.
* Parametro3: opcional de tipo entero cuyo valor por defecto es 10.

Tras añadir estos datos, nuestra anotación queda así:



Para indicar si un parámetro es o no opcional nos vamos a valer de la palabra reservada default seguida del valor por defecto.

Si el parámetro es obligatorio, no hay que añadir nada más:



* Queremos que nuestra anotación sirva tanto para clases como para métodos. Por ello, vamos a servirnos de la anotación @Target.

Nuestra anotación quedaría así:



* Lombok

Librería de Java basada en las anotaciones que permite reducir el tiempo de creación de código.

Algunas de las anotaciones que utiliza junto con su uso, aparecen a continuación:

* @Data

Anotación que agrupa las características de @ToString, @EqualsAndHashCode, @Getter/@Setter y @RequiredArgsConstructor.

* @Builder

Anotación que permite generar automáticamente el código requerido para que una clase sea instanciable.

Se basa en el uso del patrón de diseño Builder.

* @NoArgsConstructor

Anotación que genera un constructor sin parámetros.

* @RequiredArgsConstructor

Anotación que genera un constructor con un parámetro para cada atributo que requiere un manejo especial, como los atributos finales o los que no pueden ser nulos.

* @AllArgsConstructor

Anotación que genera un constructor con un parámetro para cada atributo.

* @Getter/@Setter

Anotación que permite generar automáticamente los métodos getter y setter de un atributo.

Estas anotaciones se pueden usar a nivel de clase para generar estos métodos para todos los atributos.

* @Log

Anotación que genera un campo de registro de tipo Logger y que nos permite crear mensajes para el seguimiento de una aplicación.

Para configurarlo, podemos hacerlo mediante el uso de código, mantener la configuración por defecto o editando el archivo application.properties que aparece en nuestro proyecto dentro de la carpeta src/java/resources.

* + Appender de Log

Cada una de las salidas por las que puede mostrarse un mensaje de log.

Existen varios appenders disponibles y configurados, aunque también podemos crear y configurar nuestros propios appenders.

Usando Spring Boot, la salida está configurada para que salga por consola. Si queremos que la envía también a un archivo, usamos la sentencia

Logging.file.name = nombreDelFichero

* + Niveles de Log

Filtros que nos permiten configurar dónde queremos ver los mensajes de log.

Dependen del log que estemos utilizando, pero en general existen los siguientes niveles de prioridad ordenados de menor a mayor detalle:

* Off

Nivel de mínimo detalle. Deshabilita todos los logs.

* Fatal

Se usa para notificar los mensajes críticos del sistema. Lo habitual es que después de lanzar ese mensaje, el programa se aborta.

* Error

Se utiliza para mostrar los mensajes de error de la aplicación. Afectan al programa, pero no lo terminan.

* Warn

Se usa para notificar mensajes de alerta sobre los que se quiere tener constancia pero no afectan al funcionamiento de la aplicación.

* Info

Se utiliza para mostrar mensajes que muestran información del programa durante su ejecución.

* Debug

Se usa para escribir mensajes de depuración mientras se está desarrollando la aplicación.

* Trace

Se utiliza para mostrar mensajes con un mayor nivel de detalle que debug

* All

Es el máximo nivel de detalle y habilita todos los logs. Suele ser lo mismo que usar Trace.

Los de mayor detalle, muestran también los mensajes que se recogen en los niveles más bajos.

* + MDC (Mapped Diagnostic Context) de Log

Mecanismo que permite almacenar pares de clave-valor y que se utiliza para hacer que la información relevante de un sistema esté disponible en sus mensajes de registro.

Vamos a ver un ejemplo muy sencillo sobre cómo podemos utilizarlo.

* Creamos un proyecto con un main y una clase (o hacemos usando Maven o Spring Boot)
* Dentro de la clase, vamos a crear dos variables, que serán las que posteriormente se guardarán en el MDC de log.



* Les damos valor dentro del constructor.



* Creamos un método que se encargará de añadir estas variables al MDC.

Para ello, necesitamos importar la librería que nos permite usarlo.



Utilizamos los métodos que nos proporciona, para guardar los datos.

Para ello, le damos un nombre al id, y seguidamente ponemos la variable que queremos guardar.



Después, llamaremos al método que va a poner esta información en la salida.



* Creamos el método que se va a encargar de hacer el log.

Primero, para tener un log, vamos a anotar la clase con @Slf4j y la importamos



Queremos que se escriban los daros guardados en el MDC en el nivel de info. Hemos hecho dos ejemplos para ver las diferencias.



* Para terminar, tenemos que irnos al archivo application.properties para indicar la o las salidas y el formato de la misma.



En primer lugar, ponemos que queremos que nos muestre el nivel info.

A continuación, que nos lo escriba también dentro de un archivo de nombre Fichero.log.

Para terminar, podemos ver el formato. Necesitamos usar %X para que nos cambie el id indicado por el valor al que representa. %5p hace referencia al nivel del lo, para que nos lo escriba también.

* Creamos un objeto dentro de nuestro main, y llamamos al método que hemos creado.



* Lo ejecutamos y nos fijamos en la salida



Podemos ver que nos muestra perfectamente las variables guardadas en el MDC.

Hemos hecho dos ejemplos con mensajes diferentes que se observan a continuación:



* + Diferencias entre @Log y @Slf4j
* @Log se importa de la librería lombok.extern.java.Log.

@Slf4j se importa de la librería lombok.extern.slf4j.Slf4j.

* Los niveles del @Log son: info, warning y severe.

Los niveles de @Slf4j son: trace, debug, info, warn y error.

Warning = warn y severe = error

* La creación de la variable log usando @Log se hace internamente con la siguiente sentencia:

private static final java.util.logging.Logger.log = java.util.logging.Logger.getLogger (LogExample.class.getName());

La creación de la variable log usando @Slf4j se hace internamente con la siguiente sentencia:

private static final org.slf4j.Logger.log = org.slf4j.LoggerFactory.getLogger (LogExample.class);

* + Logback

Librería de registro de eventos de logging desarrollada en Java.

Se ha creado con el objetivo de ser la sucesora de Log4j, ya que se rediseña su código y se añaden varias mejoras.

Ha sido diseñada para ser una implementación de SLF4J.

Por defecto, envía los mensajes a la consola y el nivel es debug.

Es necesario añadir el archivo .jar a nuestro proyecto para que detecte las anotaciones utilizadas.

* Patrón de diseño

Solución reutilizable para resolver un problema común en el desarrollo de software.

Contamos con diversos patrones de diseño que se pueden clasificar en los siguientes grupos según su propósito:

* Patrón creacional

Facilita la creación de nuevos objetos.

Entre ellos, cabe destacar el patrón Builder, que nos permite construir objetos complejos paso a paso. Separa la creación de un objeto de su estructura, de manera que el mismo proceso de construcción nos puede servir para crear representaciones distintas.

Vamos a ver y explicar el siguiente ejemplo:



En primer lugar, tenemos nuestra clase abstracta Builder.

En ella, se declaran los pasos de construcción del producto que todos los tipos de constructores tienen en común.

Está compuesta por una clase director, que define el orden en el que se invocarán dichos pasos.

Además, tenemos en este caso dos constructores concretos que heredan de Builder, y que ofrecen distintas implementaciones de los pasos de construcción. Son las encargadas de crear los objetos concretos.

Los productos son los objetos resultantes.

* Patrón estructural

Nos ayuda a definir la forma de composición de clases.

Uno de ellos es el patrón Facade (fachada), que proporciona una interfaz unificada para un conjunto de clases complejas haciendo que la comunicación con las mismas sea más sencilla.



Como podemos ver, el cliente solo se comunica con el sistema a través de Facade.

Facade oculta el comportamiento y la complejidad de los subsistemas y es la que se encarga de todo.

* Patrón de comportamiento

Nos ayuda a definir la forma en la que los objetos interaccionan entre ellos.

* Static

Una clase, método o atributo se declara como static si puede ser accedido o invocado sin la necesidad de tener que instanciar un objeto de la clase.

* Bloque static (o de inicialización)

Código que se ejecuta en el momento en el que la clase que lo contiene se carga por primera vez.

* Métodos estáticos genéricos

Métodos que admiten cualquier tipo de dato y que no pertenecen a ninguna clase en concreto.

Cuando usamos la palabra static para definir un método, éste se convierte en un método de clase, por lo que puede invocarse sin haber creado previamente una instancia. Pero si no tenemos una instancia, no sabemos cuál es el tipo al que hace referencia el genérico, por lo que tenemos que decirle explícitamente al compilador qué tipo debe esperar el método.

* Interfaces fluidas

Son interfaces o clases que cuando invocamos a un método concreto, nos devuelve el mismo objeto modificado, de tal forma que podemos llamar a otro método del objeto y encadenar más operaciones, es decir, se basan en el encadenamiento de métodos.

Esos métodos no pueden devolver void.

* Inmutabilidad de los objetos

Un objeto inmutable es aquel que no se puede modificar una vez que se ha creado.

Sus atributos se han definido como final o utilizan una copia defensiva para protegerse frente a cambios desde el código cliente.

La forma de “modificar” un objeto inmutable es crear una copia modificando los valores necesarios.

* Mutabilidad de los objetos

Un objeto es mutable si puede ser modificado tras su creación.

* Excepción

Error que ocurre en tiempo de ejecución.

El manejo de excepciones se gestiona a través de las palabras clave try, catch, throw, throws y finally.

Existe una jerarquía para las excepciones que es la que se puede ver en la siguiente imagen:



Exception hereda de Throwable que a su vez hereda de Object.

Las excepciones se pueden dividir en dos categorías:

* Las excepciones que han sido verificadas o IOException.

Errores que no puede evitar el programador, generalmente relacionadas con la entrada/salida del programa.

* Las excepciones que no se han comprobado o RuntimeException.

Se deben a un error de programación que se localiza durante la ejecución y que el compilador no es capaz de solucionar.

Java tiene ya definidas algunas excepciones pero también podemos crear las nuestras.

Para ello, vamos a seguir los siguientes pasos:

* Creamos una clase que va a contener nuestra excepción y que en este caso a extender la clase Exception, pero podríamos utilizar cualquier subclase de un tipo de excepción ya existente, la que mejor nos vaya.



* Le añadimos los atributos que necesitamos y un constructor que utilice el de la clase padre.



* Para enviar un mensaje, tenemos que reescribir el método getMessage() de la clase padre. También podríamos editar el mensaje indicándolo en el super() del constructor.



* Cuando ya lo tenemos, vamos a crear nuestra clase principal para ver que funciona correctamente.



* Como se puede ver, en este ejemplo lanzaremos una excepción dependiendo del número que se indique, la capturaremos y mostraremos el error según el mensaje que hemos creado.



* Spring Initializr

Herramienta que permite crear una estructura básica de un proyecto Spring Boot. Se genera un ZIP que podemos importar a nuestro IDE para empezar a trabajar sobre él.

* Bajo acoplamiento

Idea de tener las clases lo menos ligadas entre sí que se pueda, de tal forma que en el caso de producirse una modificación en alguna de ellas, se tenga la menor repercusión posible en el resto de clases, potenciando la reutilización, y disminuyendo la dependencia entre clases.

* Alta cohesión

Medida en la que un componente se dedica a realiza solo la tarea para la que fue creado, delegando las tareas complementarias a otros componentes.

* Ley de Demeter

Principio básico de la programación orientada a objetos.

Su lema es “habla solo con tus amigos”, es decir, cada clase debe tener un limitado conocimiento sobre otras clases y únicamente debe conocer aquellas clases relacionadas estrechamente con la clase actual, las inmediatamente superiores o inferiores.

* Memory Analyzer Eclipse (MAT)

Herramienta que permite analizar la memoria Heap de un programa de Java.

Con él se pueden encontrar fugas de memoria y revisar programas que tengan un alto consumo de espacio.

Para utilizarlo, necesitamos una instantánea de la memoria cuando se produce un OutOfMemoryError, que se almacena en un archivo de extensión .hprof.

Para conseguirlo, debemos crear una configuración en tiempo de ejecución.

Nos posicionamos en el main del programa a ejecutar, Run As -> Run Configurations.

En la pestaña Arguments, tenemos que escribir -XX:+HeapDumpOnOutOfMemoryError en los argumentos de VM.



Además, podemos indicar dónde queremos que se genere el archivo usando la directiva

-XX:HeapDumpPath=/ruta

Una vez generado el archivo, si queremos analizarlo abrimos la perspectiva de Memory Analyzer, vamos al menú, pulsamos File -> Open Heap Dump y buscamos la imagen generada.