# 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”};

* Serializable

Interfaz que no tiene ningún método y que sirve para especificar que un objeto instanciado puede convertirse en una cadena de bytes para su consiguiente almacenamiento o transmisión.

* Serial Versión UID

Número de versión que debe tener cada clase que implemente la interfaz Serializable.

Se utiliza durante la deserialización, para verificar que el emisor y el receptor de un objeto serializado hayan cargado las clases para ese objeto que son compatibles con respecto a la serialización. Si no es así, se producirá una InvalidClassException.

Si una clase serializable no declara de manera explícita este atributo, entonces se generará uno predeterminado en tiempo de ejecución.

Dicho atributo, debe ser estático, final y de tipo long. Además, puede llevar cualquier modificador de acceso.

* Transient

Un atributo se etiqueta como transient cuando no queremos que se incluya su serialización a la hora de almacenar o enviar dicho objeto.

De igual manera, disponemos de la anotación @Transient en JPA, que se utiliza para indicar que un campo no se debe almacenar dentro de la base de datos. Se usa para la persistencia.

* Volatile

Un atributo se marca como volatile para indicar que existe la posibilidad de que varios hilos quieran acceder y modificar su contenido a la vez.

Este atributo se guarda en memoria principal y no en la caché, para asegurarse de que los hilos leen siempre la última versión.

* 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.

* Creación de hilos

Tenemos dos maneras de crear hilos:

* Heredar la clase Thread (con extends en nuestra clase) y redefinir el método run().



Para instanciarlo, tenemos que proceder de la siguiente manera

Cuando llamemos al método start(), comenzará a ejecutarse el método run ();

* Crear una clase que implemente la interfaz Runnable, que nos obliga a definir el método run().



Para instanciarlo, tenemos que proceder de la siguiente manera:



En ambos casos escribimos un método run() que contendrá el código del hilo. Cuando se termine de ejecutar este método, también finalizará el hilo.

* Ciclo de vida de los hilos

Un hilo, pasará por varios estados durante su ciclo de vida.

* Nuevo hilo

Ocurre cuando se ha instanciado un hilo.

* Ejecutable

Se produce al llamar al método start().

* No ejecutable

Se produce por varios motivos:

* + Cuando se ejecuta el método sleep() para dormir el hilo.
  + Cuando se encuentre bloqueado por una llamada al método wait().
  + Cuando se encuentre bloqueado por una operación de E/S.
* Muerto

El código asignado al hilo ya se ha terminado de ejecutar.

Para saber si un hilo sigue vivo o no, podemos usar el método isAlive().



* Compartición de variables entre hilos

Los hilos comparten la misma zona de memoria.

Si no existe ninguna relación entre ellos, no habrá ningún conflicto, pero si no es así, tenemos que garantizar la consistencia de los datos que se comparten.

Necesitamos un mecanismo que nos permita sincronizar hilos, es decir, establecer unas reglas para acceder a los recursos compartidos.

Para marcar los métodos en los que se pueden producir problemas, se utiliza la palabra reservada synchronized.

* Executors

Librería que permite manejar la ejecución de tareas de manera concurrente.

Para ello, se crea un pool, que es una estructura que gestiona un conjunto de threads. Cuando van llegando tareas que se quieren realizar, el executor se encarga de pasarlos a ejecución, se les asigna a los threads, y cuando terminan, se vuelven a poner a la cola para realizar otras áreas si es necesario.

Nosotros no creamos los threads, solo indicamos las tareas a llevar a cabo y él se encarga de asignarlas.

Debemos tener en cuenta que Executor solo se puede usar cuando implementamos Runnable, con Thread no funciona.

* Future

Objeto que en algún momento contendrá el resultado de un método.

Se utiliza para evitar bloqueos y está ligado a la programación asíncrona.

A continuación, podemos ver los métodos que contiene la interfaz Future:



Los métodos get, sirven para acceder al resultado de nuestro objeto.

También disponemos de dos métodos de consulta para saber si el futuro se ha cancelado o se ha realizado.

Por último, tenemos un método para cancelarlo si ya no nos interesa el resultado que nos devuelve.

* 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 log, 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.

* + Patrón Builder

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 Factory Method

Proporciona una interfaz que nos permite crear objetos sin tener que especificar su clase exacta. Las subclases deciden qué clase instanciar.



La interfaz Product, define la estructura de los objetos que podemos crear con el factoryMethod().

ConcreteProduct representa una implementación concreta de la interfaz Product.

Factory, también denominada Creator, declara el factoryMethod() que nos devuelve un objeto genérico, es decir, de tipo Product.

ConcreteFactory se encarga de sobrescribir el factoryMethod() para devolver un objeto de algún tipo de ConcreteProduct.

* + Patrón Abstract Factory

Proporciona una interfaz que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



La interfaz Abstract Factory define un conjunto de métodos para la creación de productos abstractos.

Las ConcreteFactory sobrescriben los métodos de Abstract Factory para crear distintos objetos concretos.

AbstractProduct es una interfaz que define la estructura de los productos que se pueden crear.

Los ConcreteProduct implementan la interfaz AbstractProduct.

* Diferencias entre el patrón Factory Method y el Abstract Factory
* Factory Method se preocupa por crear un único producto.
* Abstract Factory lo que crea son familias de productos.
* Factory Method es un patrón de clase, por lo que se refiere a las relaciones entre las clases y sus subclases.
* Abstract Factory es un patrón de objeto, por lo que se refiere a las relaciones entre las instancias
* Cada método contenido en una Abstract Factory se trata de un Factory Method.
  + Patrón Singleton

Patrón que limita la creación de objetos pertenecientes a una clase, garantizando que solo exista una instancia de la misma para toda la aplicación y que el acceso a dicha instancia sea global.



Este patrón se implementa haciendo privado el constructor de la clase y teniendo dentro de ella un método estático que crea una instancia del objeto si este no existe todavía.

* Patrón estructural

Nos ayuda a definir la forma de composición de clases.

* + Patrón Facade (fachada)

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 Proxy

Proporciona un objeto intermediario entre el cliente y el objeto a utilizar, que permite configurar ciertas características sin tener que modificar la clase original.



Tenemos una interfaz que tiene toda la funcionalidad que queremos para nuestro objeto, y tanto el proxy como el objeto real tienen que definirla.

El sujeto real es la clase que se quiere proteger del usuario.

El proxy tendrá un atributo de la clase principal que el usuario podrá manipular como si fuese el objeto real.

* Patrón de comportamiento

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

* + Patrón Template Method

Patrón que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

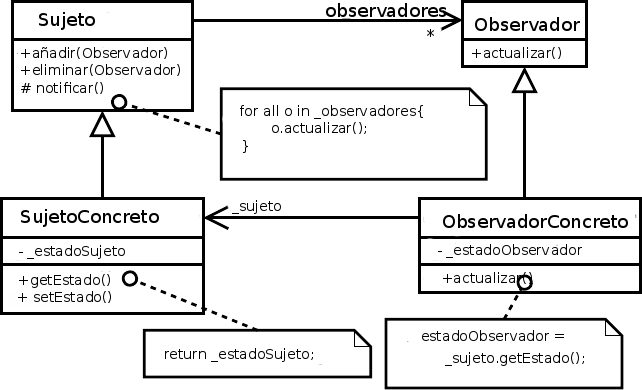


Tenemos una clase abstracta, en la que se declaran los métodos que actúan como pasos de un algoritmo, así como el método plantilla, que va a ser el esqueleto que invoca a cada uno de los métodos en un orden específico. Los pasos pueden ser abstractos o tener una implementación por defecto.

Las clases concretas pueden sobrescribir todos los pasos necesarios, pero nunca el método plantilla.

* + Patrón Observer

Patrón que permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



Tenemos una interfaz Sujeto que nos permite agregar y eliminar observadores.

El Sujeto Concreto, guarda el estado de interés y notifica a sus observadores cuando su estado cambia.

Tenemos la interfaz Observador, que define la interfaz de los objetos a los que se le tiene que notificar los cambios que se han producido en el sujeto que observan.

El Observador Concreto almacena el estado del sujeto que le resulta de interés.

* Patrón Delegate

Patrón de diseño que expresa un cierto comportamiento de cara al exterior pero que en realidad delega la responsabilidad de implementar dicho comportamiento a un objeto asociado, es decir, un objeto, en lugar de realizar una de sus tareas establecidas, deja esa tarea a un objeto auxiliar.

No aparece entre los definidos por GoF, pero se utiliza con frecuencia. Podría clasificarse dentro de los patrones estructurales.

Es similar al patrón proxy, aunque su propósito sea diferente.

* 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.

* Idempotencia

Propiedad para realizar una acción concreta varias veces y aun así obtener siempre el mismo resultado que se obtendría si se hiciera solo una vez.

Por ejemplo:

* Ley de Little John

Es conocida como la Teoría de colas o líneas de espera y se usa para calcular el rendimiento de los sistemas.

Su enunciado dice así: “*El número medio de usuarios N que hay en un sistema durante un tiempo determinado es igual a la velocidad media V a la que entran en el sistema multiplicada por el tiempo medio t que están dentro”*.

N = V x T

* Clases anidadas

Clases que se declaran dentro de otra.

No existen de manera independiente de su clase adjunta.

Se pueden diferenciar dos tipos de clases anidadas:



* Clases estáticas o clases anidadas estáticas

Clases internas a las que se puede acceder sin tener que instanciar un objeto de la clase externa que la contiene.

Funcionan igual que cualquier clase normal, ya que dentro pueden tener a su vez métodos tanto estáticos como no estáticos.

Desde estas clases se puede acceder a todos los miembros que sean estáticos de la clase externa pero al resto no.

Para acceder al contenido de la clase lo hacemos de la siguiente forma:



Indicamos el tipo de la clase que queremos instanciar, ponemos la palabra reservada new y accedemos al constructor.

Si después queremos acceder a algún método de ella, se hace de la manera habitual:



* Clases anidadas internas

Clases que se encuentran dentro de una clase externa y que tienen acceso a todos los miembros de la clase superior.

Necesitan que exista una instancia de la clase externa para poder acceder a su funcionalidad.

Para acceder al contenido de la clase tenemos dos formas diferentes, aunque el primer paso es instanciar un objeto de la clase superior:



Para la forma 1:



Para la forma 2:



Si después queremos acceder a algún método de ella, se hace de la manera habitual:



Dentro de este tipo de clases, nos encontramos las clases anónimas.

* + Clases anónimas

Clases que no tienen nombre y que nos va a permitir declarar una clase e instanciarla al mismo tiempo en la parte del código que nos haga falta.

Esto hace nuestro algoritmo más legible, ya que no tenemos que crear un nuevo archivo .java para implementarla.

Se escriben dentro del método de otra clase y solo desde ahí son visibles.

Son clases que no se puede reutilizar.

Se definen de la siguiente manera:

* Se escribe el operador new.
* Seguidamente el nombre de la interfaz a implementar o clase a extender.
* Paréntesis para colocar los argumentos del constructor.
* Y entre llaves el cuerpo de la clase, que implementa los métodos de la interfaz o redefine los de la clase.

La sintaxis para implementar una interfaz es:



La sintaxis para extender una clase quedaría así:



* Expresiones regulares en Java

Las expresiones regulares son secuencias de caracteres y símbolos que nos permiten encontrar coincidencias en cadenas de caracteres, reemplazar dichas coincidencias o validar que las cadenas cumplen con un patrón.

* Java Bean Validation

Especificación de Java basada en el uso de anotaciones que nos permiten verificar el correcto contenido de los componentes de un programa.

La implementación de referencia es Hibernate Validator aunque también existe la Apache BVal.

Para poder utilizarlo, tenemos que añadir las dependencias necesarias dentro de nuestro archivo pom.xml, por lo que tenemos que crear un proyecto Maven o Spring Boot.

Si usamos un proyecto de Spring Boot, tenemos que añadir la siguiente dependencia:



Si creáramos un proyecto de Maven, las dependencias a añadir son las siguientes:

* Para añadir el Hibernate Valitador



* Para agregar expresiones regulares:



El paquete javax.validation.constraints contiene los elementos que nos permiten comprobar dichas validaciones.

Vamos a hacer un ejemplo implementando una gestoría, en la que tendremos gestores y clientes con ciertas características.

El gestor irá revisando documentos que le pide al cliente, como son el nombre, DNI y la edad.

Estos datos, van a tener ciertas restricciones:

* El nombre tendrá una longitud máxima de 10 caracteres.



* El DNI vendrá dado por una expresión que comience por la letra X, seguido de 3 números y finalmente un carácter que puede ser A, B, C, D o E.



* La edad deberá ser mayor o igual que 18 y no superar los 50.



Una vez hecho esto, crearemos un método para validar las restricciones impuestas.



Nos creamos un validador y le pasamos el cliente que nos han facilitado para obtener las verificaciones que no se han cumplido.

Si no tenemos ninguna, el cliente es correcto y lo indicamos por pantalla.

Si no es así, decimo que hemos encontrado fallos y mostramos cuáles han sido.

Este método lo llamamos después de que el gestor haya recibido todos los datos de un cliente.



* @Valid

Anotación que nos permite verificar la validez de todo un objeto.

* @Validated

Anotación que nos permite verificar la validez de un conjunto de campos de un objeto.

* Testing

Realización de pruebas sobre un software con el fin de detectar errores de este.

* Test double

Objeto que utilizamos en sustitución de otro cuando queremos realizar una prueba.

Se pueden distinguir 5 categorías:



* Dummy

Objeto necesario para ejecutar un test pero que no interviene en la funcionalidad que estamos comprobando. Son objetos de relleno.

* Stub

Objetos parecidos a los dummy pero que proporcionan respuestas predefinidas a llamadas realizadas durante los test.

* Spy

Objetos que registran cierta información sobre las acciones que se realizan sobre ellos.

* Mock

Es un stub en el que sus métodos sí que implementan un comportamiento, pues se espera que reciban unos valores y, en función de ellos, se va a devolver una respuesta.

* Fake

Objeto completamente implementado equivalente al objeto real al que pretende simular, pero que falsea algo para que los test sean más fáciles de probar.

* Pirámide de testing o pirámide de Cohn

Representación piramidal de los pasos que hay seguir para comprobar la calidad de un software y su grado de comprobación.

Existen distintos tipos de pruebas y cada una está orientada a detectar y prevenir cierto tipo de errores de software.

En el siguiente dibujo, podemos ver las partes de las que consta:



* Pruebas unitarias

Con ellos lo que queremos es probar funcionalidades únicas. Cada test tiene un único propósito.

Son los tests más rápidos.

* Pruebas de integración

Sirven para asegurarse de que todos los componentes juntos funcionan adecuadamente.

* Pruebas funcionales

Con ellos, probaremos todo el sistema haciéndonos pasar por el usuario final.

Son la parte más lenta.

* JUnit

Herramienta para realizar pruebas unitarias en Java.

* Test parametrizados

Método de prueba que se realiza para diferentes parámetros.

Son útiles para no tener que repetir código en función de la entrada recibida, aunque Junit se encarga de crear internamente un test para cada uno de ellos. Si uno falla, no pasa nada porque el resto se pueden ejecutar.

En JUnit 4 se usa la siguiente sintaxis:

* @RunWith (Parameterized.class) -> para decirle con qué debe ejecutar la clase de prueba.
* @Parameters -> para anotar los datos de prueba.
* @Test -> para anotar el método de prueba.

En JUnit 5 se hace como aparece a continuación:

* @ParameterizedTest -> para declarar un test como parametrizado
* Para indicarle de dónde tiene que leer los datos, existen diversas anotaciones que se tienen que colocar justo debajo de la anterior. Algunas de ellas son:
  + @ValueSource -> usado para especificar un array de valores.



* + @EnumSource -> usado para especificar valores enumerados.



* + @MethodSource -> usado para especificar un método estático de la clase que proporciona un Stream de valores.



La clase que los contiene se declara así:



* + @CsvSource -> usado para especificar valores separados por comas, es decir, en formato CSV (comma separated value).



* + @CsvFileSource -> usado para especificar valores en formato CSV contenidos en un fichero



* Teorías

Métodos de prueba que permiten definir un número variable de valores para cada tipo de dato.

Aunque se ejecute para diferentes valores, solo se crea un test. Si uno de ellos falla, el test no continúa.

Se usa la siguiente sintaxis:

* @RunWith (Theories.class) -> para decirle con qué debe ejecutar la clase de prueba.
* @DataPoint / @DataPoints -> para anotar los datos de prueba (uno o varios).
* @Theory -> para anotar el método de prueba.
* Rule

Mecanismo que permite agregar funcionalidad adicional que se aplica a todas las pruebas de un test.

Para ello, se debe utilizar la anotación @Rule, y el campo anotado tiene que ser de una clase que implementa la interfaz TestRule.

Tiene un método llamado apply que hay que implementar.

Recibe un test y su descripción y, dentro, hay que llamar a la función evaluate() para que se pueda ejecutar.

Devolveremos un nuevo test con las condiciones que le hemos indicado dentro de nuestra regla.

En el ejemplo, lo que hemos hecho es mostrar la frase Hola mundo y la hora actual antes de ejecutar el test y, justo al terminar, escribimos el tiempo que ha tardado en ejecutarse la prueba.



Una vez creada nuestra rule, tenemos que poner la sentencia que aparece a continuación dentro de nuestra clase de test:



Se ejecutará antes de cada uno de los test que tengamos.

* AssertJ

Librería de Java que nos permite simplificar la escritura de las aserciones en pruebas de código.

Su objetivo principal es mejorar la legibilidad de los test, permitiéndonos encadenar múltiples aserciones.

Para poder usarla, tenemos que añadir la siguiente dependencia dentro del pom.xml de nuestro proyecto:



Además, debemos importarla de manera estática dentro de la clase que va a contener las pruebas:



Todas las aserciones empiezan por el método base llamado



Aunque él por si solo no va a fallar nunca, tenemos que indicarle a continuación lo que queremos comprobar, por ejemplo:



Para facilitarnos su uso dentro de eclipse, seguimos los siguientes pasos:

Window -> Preferences -> Java -> Editor -> Content Assist -> Favorites -> New Type -> escribir org.assertj.core.api.Assertions y confirmar.

* Mockito

Marco de trabajo que nos permite la creación de objetos simulados (test dobles) con el propósito de realizar pruebas unitarias en Java.

Para poder utilizarlo, tenemos que añadir la siguiente dependencia dentro del pom:



* @Mock

Anotación de Mockito que sirve para simular métodos o campos. Se trata de un objeto falso, simulado.

Su comportamiento predeterminado es no hacer nada.

Esta anotación también existe en Spring Boot, pero se llama @MockBean.

* @Spy

Anotación de Mockito que sirve para simular métodos o campos. Se trata de un objeto real, en el que estamos espiando información del mismo.

Su comportamiento por defecto es llamar al método real.

Esta anotación también existe en Spring Boot, pero se llama @SpyBean.

* Programación funcional

Lenguaje de programación declarativa (le decimos al programa lo que tiene que hacer, no cómo debe llevarlo a cabo) que se basa en el uso de funciones matemáticas.

* Expresiones Lambda

Son funciones anónimas que se escriben en el mismo lugar en el que se usan.

Como cualquier función, reciben 0 o más argumentos y devuelven uno o ningún valor de retorno.

Desde ellas se puede acceder a las variables locales del método en el que se usan, pero sola las podrá utilizar para lectura, en ningún caso será capaz de modificarlas.

La sintaxis es muy sencilla. Tendremos una primera parte donde aparecen los parámetros que va a utilizar y una segunda en la que tendremos el cuerpo de la función, que especifica las acciones a realizar. Irá entre llaves si consta de más de una expresión. Para separar ambas partes se utiliza el operador lambda, también llamado operador flecha (->).



* Interfaz funcional

Tipo de interfaz que tiene un único método abstracto.

Estas interfaces pueden tener además otros métodos predeterminados (default) o estáticos, pero no puede haber ninguno más que sea abstracto.

Se pueden utilizar como destino de una asignación de una expresión lambda, como referencia de un método o como referencia a un constructor.

Están definidas dentro del paquete java.util.function.

Lo mejor es anotarlas con @FuncionalInterface, ya que si no cumplimos las reglas, el propio IDE nos va a avisar del error.

Algunos tipos de interfaces funcionales son:

* Consumer

Interfaz funcional que acepta un único argumento de entrada y no devuelve ningún resultado.



Se construyen de la siguiente manera:



* Function

Interfaz funcional que recibe uno o más parámetros y produce una salida.



Se construyen como sigue:



* Producer (Supplier)

Interfaz funcional que no recibe ningún argumento pero produce un resultado.



Se construyen como se puede ver a continuación:



* Predicate

Especialización de function, ya que recibe un parámetro y produce como salida un valor booleano.



Se construyen de la siguiente manera:



* Java Streams

Librería de Java que nos permite procesar colecciones de objetos.

Su uso es muy sencillo, ya que solo tenemos que llamar al método stream(). A partir de él, se puede llamar al resto de funciones disponibles.

* Métodos referenciados (reference methods)

Tipo especial de expresiones lambda que nos proporciona una notación abreviada para llamar a métodos que ya tienen un nombre.

Para definir la referencia al método, utilizamos el operador ::.

La sintaxis para los distintos tipos es la que sigue:



* Optional

Clase que encapsula un objeto para poder hacer ciertas operaciones sobre él. Este objeto puede o no contener un valor nulo.

Para poder usarlo, tenemos que incluir en nuestra clase su import correspondiente:



Una instancia de la clase nos proporciona inicialmente tres métodos:

* Nos permite almacenar una instancia vacía:



* Nos permite almacenar el objeto especificado si no es nulo. Si no se guarda una instancia vacía:



* Nos permite almacenar el objeto no nulo especificado:



Para saber si esa instancia contiene un valor o no, se usa el método que nos devuelve un valor booleano:



Para poder recuperarlo entonces se aplica el método:



Si no sabemos si tiene o no contenido y queremos que nos devuelva siempre algo, podemos usar el siguiente método, donde le tenemos que indicar el valor por defecto:



* Reflection

Librería de Java que nos permite manipular clases, interfaces y sus miembros en tiempo de ejecución para trabajar con ellos, sin conocer los nombres de los mismos en tiempo de compilación.

Nos aporta una gran versatilidad, pero también afecta notablemente al rendimiento del programa.

* Spring

Plataforma de código abierto que permite crear aplicaciones de alto rendimiento, livianas y reutilizables en Java, Groovy o Kotlin.

Su finalidad es estandarizar, agilizar, manejar y resolver problemas que puedan ir ocurriendo durante la programación.

Spring está formado por un conjunto de módulos que nos ofrecen diversas funcionalidades.

Estos módulos son opcionales, por lo que podemos utilizar solo los que necesitemos.



Vamos a explicar en qué consiste cada uno:

* Core Container

Proporciona los submódulos básicos que se encargan de la creación y gestión de los Beans (clases de nuestro programa cuya función es almacenar información relevante), la lectura de los ficheros de configuración, el manejo de las dependencias, el uso del contexto para almacenar los Beans en memoria y la utilización del Spring Expression Language para manipular los objetos en tiempo de ejecución.

* Infraestructura

Está compuesto por los submódulos donde se manejan las transacciones, loggings o seguridad de las aplicaciones.

* Data Access/Integration

Los submódulos de esta categoría están diseñados para ofrecer a las aplicaciones los instrumentos para que interactúen con otras aplicaciones, así como controlar el acceso a los datos.

* Web

Aquí dentro se encuentran los submódulos específicos para el desarrollo y soporte de las aplicaciones web.

* Test

El módulo de pruebas nos permite comprobar la correcta funcionalidad de los componentes de la aplicación.

* Spring Boot

Herramienta que nos permite construir aplicaciones de Spring de manera rápida, evitando tener que realizar manualmente la configuración inicial.

Dispone de un servidor embebido que permite el despliegue de las aplicaciones.

* Spring Data JPA (Java Persistence API)

Módulo de Spring que nos facilita el acceso y uso de los datos de una aplicación siguiendo la especificación que proporciona JPA, que es el estándar que define cómo se debe persistir los datos de Java.

Para poder usarlo, tenemos que añadir la siguiente dependencia si trabajamos con Spring:



O la que se ve a continuación si usamos Spring Boot:

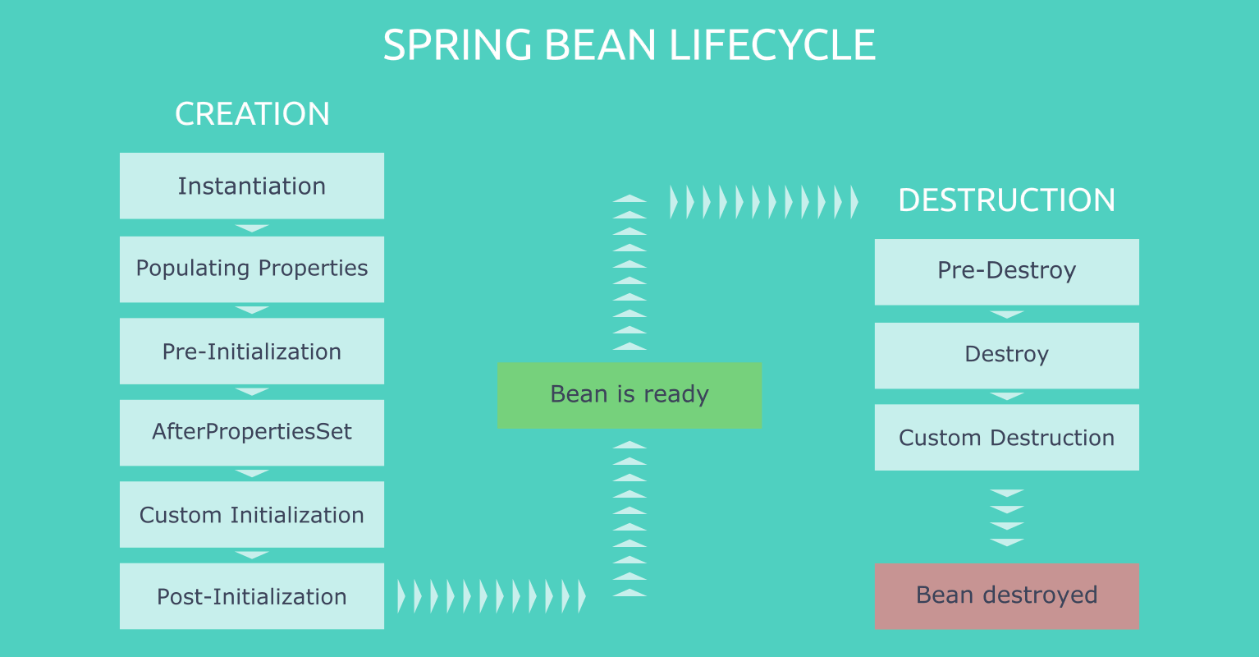


Además, tenemos que añadir la dependencia referente a la base de datos que queremos utilizar.

* Bean

Clase de nuestro programa cuya función es almacenar información relevante.

* Ciclo de vida de los Beans



Fases de creación de Beans:

* Instanciación: Spring crea una instancia de los objetos Beans de la misma maneara que nosotros crearíamos una instancia de un objeto Java
* Rellenar propiedades: busca los Beans que implementan interfaces Aware y establece las propiedades relevantes. Estas interfaces, permiten ver el comportamiento interno de Spring
* Pre-inicialización: se ejecutan los métodos postProcessBeforeInitialization() de BeanPostProcessor.
* AfterPropertiesSet: Spring ejecuta los métodos afterPropertiesSet() de los Beans que implementan Initializing Bean.
* Inicialización personalizada: Spring activa los initMethod de nuestras anotaciones @Bean.
* Post-inicialización: se ejecutan los métodos postProcessAfterInitialization() de BeanPostProcessor.

Fases de destrucción de Beans:

* Pre-destrución: Spring activa los métodos anotados con @PreDestroy.
* Destrucción: Spring ejecuta los métodos destroy() de la implementación DisposableBean.
* Destrucción personalizada: se activan los destroyMethod de nuestras anotaciones @Bean.
* Cargar Beans de manera perezosa (@Lazy)

El Bean se carga solo cuando cualquier otro método o clase necesita una instancia de dicho Bean.

Los Beans, por defecto, se crean al inicio del arranque de la aplicación.

* Modelo Vista Controlador (MVC)

Patrón de arquitectura del software que separa el código en función de sus responsabilidades, manteniendo diversas capas que se encargan de una tarea concreta, como son los datos de una aplicación, la interfaz de usuario y la lógica de control.

Consta de tres componentes distintos: el modelo, la vista y el controlador.

* Modelo

Contiene una representación de los datos que maneja el sistema, su lógica de negocio y los mecanismos de persistencia de la información.

* Vista

Está compuesta por la información que se le envía al cliente y los mecanismos de interacción con éste.

* Controlador

Actúa como intermediario entre el modelo y la vista gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

A continuación, podemos ver los pasos que conforman su ciclo de vida:



Este patrón se basa en las ideas de reutilización de código y separación de conceptos, que son clave para el desarrollo de aplicaciones y su posterior mantenimiento.

* Modelo Vista Vista Modelo (MVVM)

Patrón de arquitectura del software que se caracteriza por tratar de desacoplar lo máximo posible la interfaz de usuario de la lógica de la aplicación.

No se controlan manualmente los cambios que ocurren en la vista o en los datos, si no que estos se actualizan directamente cuando se produce un cambio en ellos.

Existen tres componentes que nos ayudan a lograrlo:

* Modelo

Contiene una representación de los datos que maneja el sistema, su lógica de negocio y los mecanismos de persistencia de la información.

* Vista

Está compuesta por la información que se le envía al cliente y los mecanismos de interacción con éste. Estos datos son activos y a que reaccionan a cambios en el modelo.

* Vista-Modelo

Es la capa intermedia entre el modelo y la vista, y se encarga de procesar las peticiones que tenga la vista hacia el modelo.



Mantener una separación entre la interfaz de usuario y la lógica de la aplicación ayuda a abordar los problemas de desarrollo y hace a la aplicación más fácil de probar, mantener y desarrollar. También mejora la reutilización de código.

* Patrón Pizarra

Patrón de arquitectura del software utilizado en sistemas basados en el conocimiento, que constan de múltiples agentes.

Está formado por tres componentes principales:

* Agentes

Componentes que están especializados en resolver una tarea concreta.

La función de los agentes es cooperar para alcanzar una meta común. Examinan la pizarra, realizan su tarea y escriben las conclusiones en la propia pizarra. Así, otro agente podrá trabajar con dichos resultados.

* Pizarra

Se trata de la estructura de datos central, que provee una interfaz que permite a los agentes leer esa información y añadir nueva.

* Elemento de control

Se encarga de monitorizar los cambios que se producen en la pizarra y decide qué acciones se deben tomar.

Este papel lo puede hacer la propia pizarra.



El estado inicial de la pizarra es una descripción inicial del problema que hay que resolver y el estado final será su solución.

La computación termina cuando se alcanza alguna condición deseada entre los resultados escritos en la pizarra.

* JSON (JavaScript Object Notation)

Formato de texto usado para el intercambio de datos entre sistemas.

Resulta sencillo de leer y escribir para los programadores y simple de interpretar y generar para las máquinas.

Estos archivos contienen solo texto y utilizan la extensión .json.

Un objeto JSON está formado por dos elementos:

* Clave

Son cadenas de caracteres encerrados entre comillas.

El primer carácter debe ser una letra, un guion bajo (\_) o el signo del dólar($).

* Valor

Son los tipos de datos que JSON puede representar. Hablamos de strings, números, booleans, valores nulos, arrays y objetos.

* + String

Son cadenas de cero o más caracteres y se escriben entre comillas.



* + Número

Se admiten números negativos y con parte fraccional separada por un punto.



* + Boolean



* + Valor nulo

Se utiliza para mostrar que no hay información.



* + Array

Un array se representa utilizando corchetes [] para englobar su contenido, y cada dato se separa mediante el uso de la coma ,.



* + Objeto

Un objeto, termina y comienza con llaves {} y puede tener varios pares clave valor dentro separados por comas.

Además, cada clave se separa del valor que se le asigna usando dos puntos (:).



No se deben incluir comentarios dentro de los objetos JSON.

* JSONPath

Lenguaje de consulta para JSON.

Se basa en la utilización de expresiones o selectores que nos permiten obtener información de un JSON.

Para poder utilizarlo en nuestros proyectos, tenemos que añadir en el pom la siguiente dependencia:



Vamos a ver algunos de los elementos más importantes de su sintaxis:

* $ -> símbolo con el que empiezan las expresiones de JSONPath, aunque también es cierto que se puede omitir. Hace referencia al elemento raíz.
* Para denotar un elemento hijo, se utiliza la notación . o [‘ ‘] con el nombre del elemento entre comillas simples.



* Si queremos acceder a un elemento de ese objeto, utilizaremos el mismo operador de forma anidada.



* De la misma manera, podemos usar .. seguido del nombre de una propiedad para que se haga una búsqueda recursiva hasta encontrar el elemento que hemos indicado.



* También podemos usar los corchetes para indicar dentro los elementos que queremos obtener que se encuentran dentro de un array.

Si solo queremos uno:



Si queremos que nos devuelva una lista de los elementos indicados:



Si queremos los elementos de un array dentro de un rango:



El elemento de la posición end no lo recoge. Si además no ponemos el elemento end, nos devuelve todos los datos hasta el final.



* @ -> se utiliza en expresiones de filtro para referirse al nodo actual.



* \* -> es el operador comodín y se puede usar para representar un número o un nombre.



Cabe destacar que JSONPath distingue entre el uso de mayúsculas y minúsculas.

* Servicio REST (Representational State Transfer)

Conjunto de restricciones por las que se puede diseñar una comunicación entre cliente y servidor utilizando el protocolo HTTP.

Nos sirve para obtener y generar datos y operaciones que nos son devueltos en formato JSON o XML.

* Niveles de REST



* Nivel 0 (Swamp of POX): Uso de HTTP para realizar interacciones remotas. Se intercambian mensajes XML entre cliente y servidor.
* Nivel 1 (Recursos): Son los datos centrales sobre los que actúa la aplicación.
* Nivel 2 (Verbos HTTP): Acceso a los datos gracias al uso de los métodos HTTP.
* Nivel 3 (Controles hipermedia-HATEOAS): Los recursos pueden estar relacionados entre ellos mediante el uso de links.
* Métodos HTTP (REST)

Definen la acción que se realizará sobre un determinado recurso.

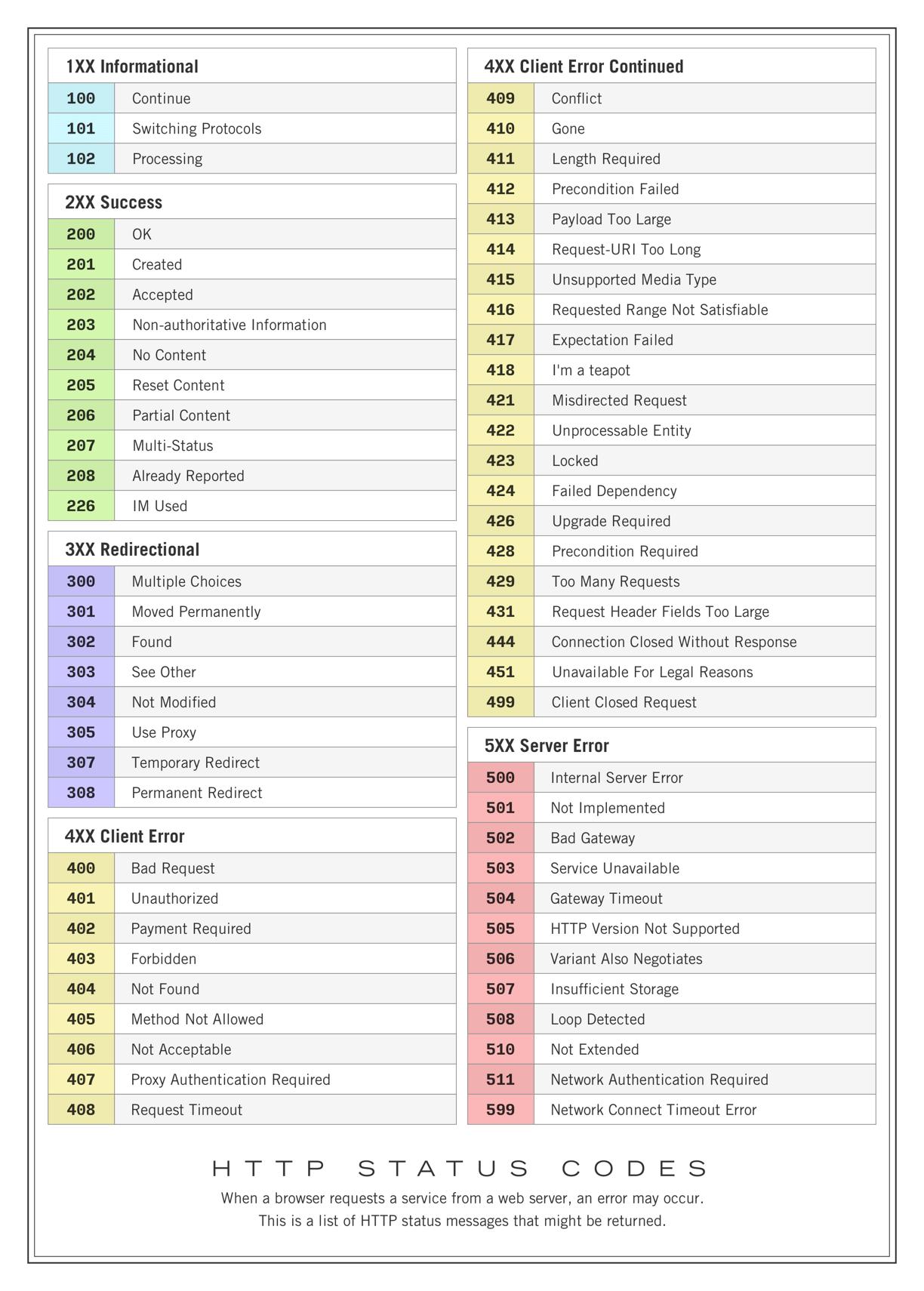
* Get: utilizado para obtener información del servidor.
* Post: usado para la creación de un nuevo registro.
* Put: sirve para actualizar por completo un registro ya existente.
* Patch: utilizada para actualizar solo un fragmento del registro.
* Delete: usado para eliminar un registro ya existente.
* Head: sirve para obtener la cabecera de respuesta que devuelve el servidor al hacer una petición sobre este.
* Options: utilizado para saber qué otros métodos HTTP están disponibles en el servidor.
* HTTP Status Codes

Mensajes del servidor que te permiten saber si se ha realizado satisfactoriamente una determinada petición.

El primer dígito del código de estado especifica uno de los 5 tipos de respuestas definidas que existen:

* 1XX -> Respuestas informativas
* 2XX -> Peticiones correctas
* 3XX -> Redirecciones
* 4XX -> Errores del cliente
* 5XX -> Errores del servidor

A continuación, se pueden ver los códigos de cada una de las clases:



* Recurso REST

Entidad que representa un concepto de nuestra aplicación.

Se almacenan en un servidor y pueden ser accesibles de manera pública bajo petición del cliente.

* Clave primaria compuesta Spring Data JPA

A veces, para identificar inequívocamente un objeto, necesitamos hacer uso de la clave primaria compuesta.

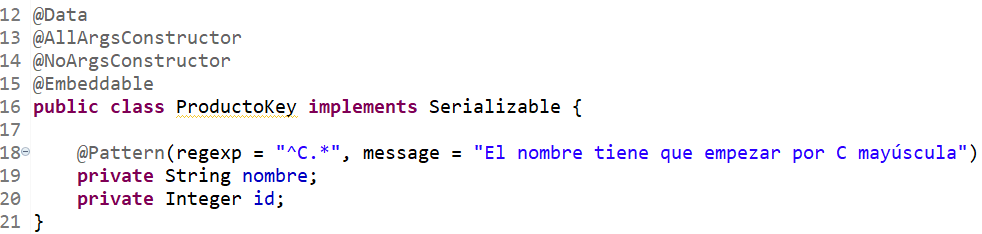
Se trata de una combinación de dos o más columnas que representan información relevante del objeto.

Hay dos maneras distintas de generarla:

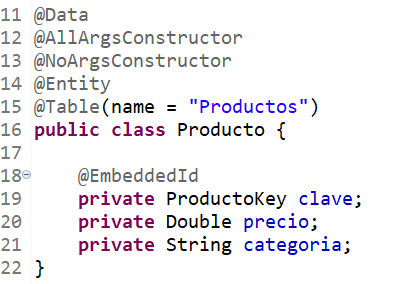
* @EmbeddedId

Creamos una clase que va a contener las variables que conforman la clave.

La clase tiene que implementar la interfaz Serializable y tenemos que anotarla con @Embeddable.



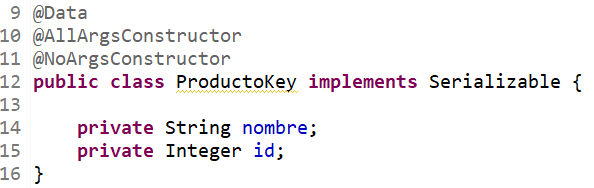
Dentro de la clase que contiene nuestro objeto, añadimos una variable del tipo recién creado y lo anotamos como @EmbeddedId. Además, prescindimos de las variables por solitario, ya que se engloban dentro de la nueva.



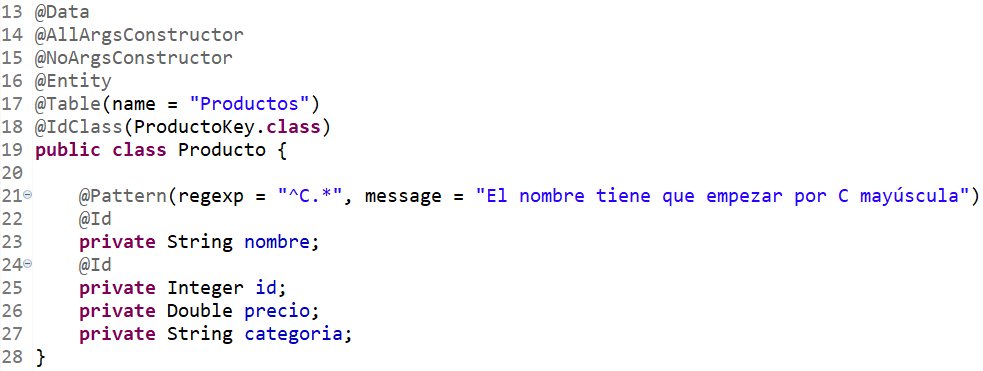
* @IdClass

Creamos una clase que va a contener las variables que conforman la clave.

La clase tiene que implementar la interfaz Serializable.



Si ahora nos vamos a la clase que define nuestro objeto, tenemos que añadirle la anotación @IdClass (nombreDeLaClase.class) para saber a qué clase hace referencia. Además, debemos anotar las variables que forman parte de la clave compuesta con @Id.



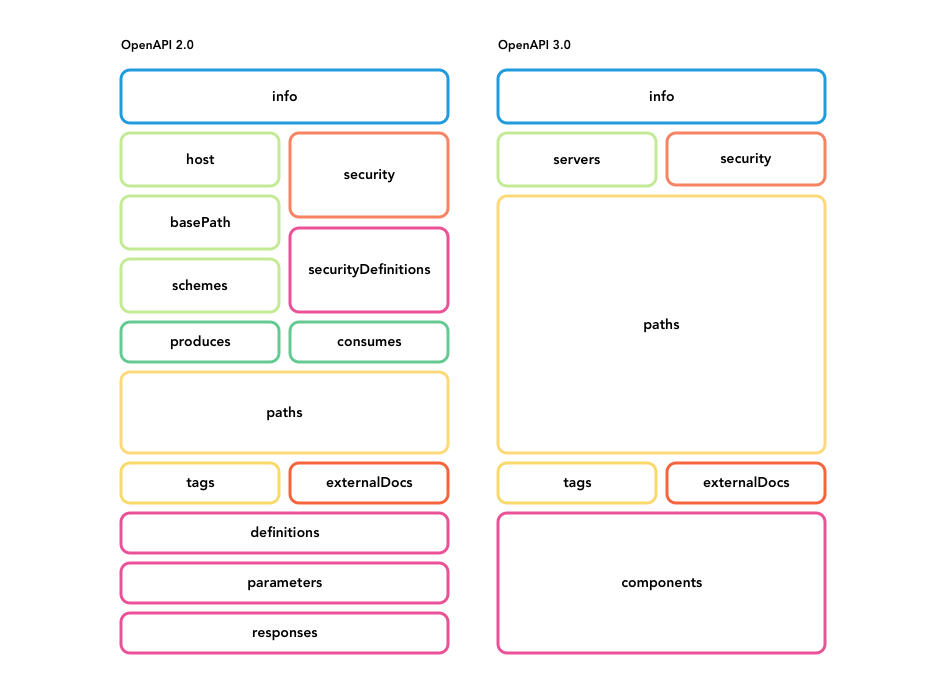
* OpenAPI 3.0

Estándar que proporciona un marco común para la descripción de interfaces de programación o APIs.

Puede usarse para describir, desarrollar, probar y documentar las API compatibles con servicios REST.

La definición de la API se debe plasmar en un documento en formato YAML o JSON.

Tendrá la siguiente estructura:



* Cómo crear clases con OpenAPI

Una vez tenemos nuestro archivo YALM, lo que tenemos que hacer es generar las clases que se recogen en él.

Para ello, creamos un proyecto de Spring Boot con las dependencias necesarias.

Añadimos el plugin que va a ejecutar nuestro proyecto:



Nos posicionamos en el directorio sobre el pom -> Run As -> Maven clean -> Maven install.

Una vez hecho esto, solo nos queda esperar a que se generen las clases.

Aparecerán dentro de la carpeta target:

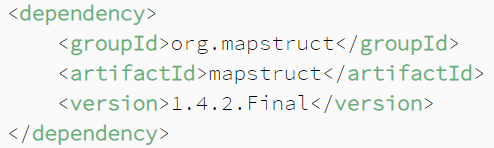


* MapStruct

Generador de código que simplifica la implementación de mapeos entre tipos de Java Beans.

Funciona en tiempo de compilación.

Para poder utilizarlo, debemos añadir la siguiente dependencia:



Además, debemos agregar la sección annotationProcessorPaths a la configuración del complemento de maven-compiler-plugin.

Se utiliza para generar la implementación del mapeador durante la compilación.



Si estamos utilizando lombok para generar las clases que están implicadas en el mapeo, tenemos que añadir nuevas dependencias dentro de annotationProcessorPaths.



Una vez tenemos configurado nuestro proyecto, vamos a crear el mapeador.

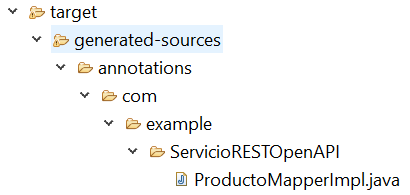
Generamos una interfaz que anotaremos con @Mapper.



Dentro, escribiremos los métodos necesarios para cubrir los mapeos que necesitamos.



Al ejecutar nuestra aplicación, se creará la implementación de nuestro mapeador.



* CSS (Cascading Style Sheet - Hojas de Estilo en Cascada)

Lenguaje de hojas de estilo creado para controlar el aspecto y la presentación de los documentos escritos con un lenguaje de marcado (HTML).

Es la mejor forma de separar los contenidos de su representación visual.

Hay tres formas distintas de añadir el estilo a nuestras páginas HTML:

* CSS Externo -> Etiqueta <link>

El código se escribe en un archivo .css aparte. Es el método más habitual.

* CSS Interno -> Etiqueta <style>

El código se escribe dentro de la etiqueta <style> en el propio documento .html.

* Estilos en línea -> Atributo style=” “

El código se escribe dentro de las etiquetas de HTML.

Las reglas se definen de la siguiente manera:



* Programación Orientada a Aspectos (AOP)

Técnica de programación que permite extraer código transversal y aplicarlo en aquellos puntos de la aplicación donde sea necesario sin estar mezclado con el código al que se aplica.

Posibilita la separación de las responsabilidades.

Los aspectos solo se aplican se desde otra clase se llama a los métodos a los que afectan los aspectos.

Dichos métodos pueden ser tanto public como protected.

Si nosotros llamamos a un método dentro de otro método de la misma clase, me da igual si es public, protected o private, que el aspecto no se va a aplicar a dicho método que se ha llamado internamente.

* Programación Reactiva

Paradigma de programación enfocado en el trabajo con flujos de datos finitos o infinitos de manera que no coinciden en el tiempo y sin producirse bloqueos.

En resumen, se trata de unas determinadas directrices que nos indican cómo debemos programar nuestro sistema para trabajar con datos que no sabemos cuándo se van a generar.

Está relacionada con el patrón de diseño Observer, ya que cuando hay un cambio de estado en un objeto, los otros objetos son notificados y actualizados de manera correcta.