BACK

GUÍA COMPLETA

autentia

GUIA PARA DIRECTIVOS Y TÉCNICOS

V.3

Back

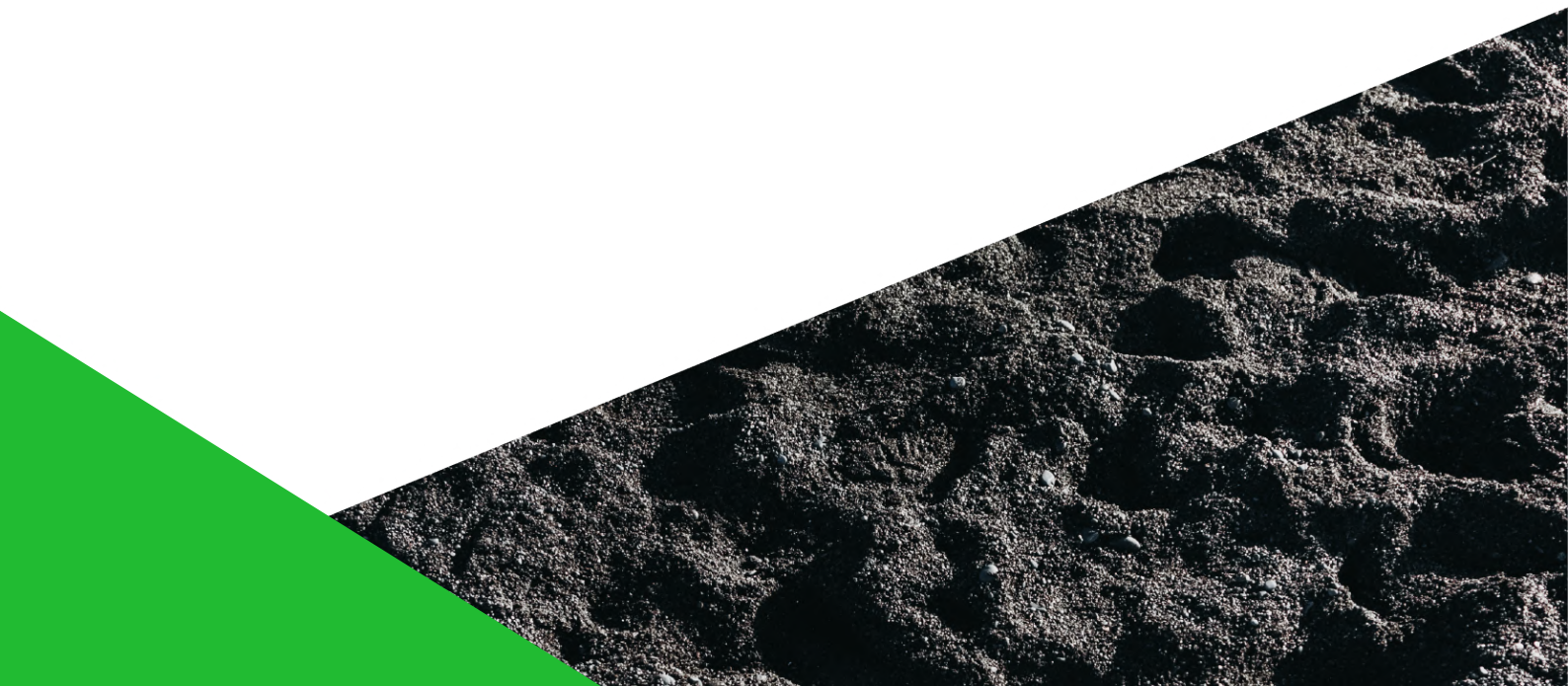
Guía completa

Este documento forma parte de las guías de onboarding de Autentia. Si te apasiona el desarrollo de software de calidad ayúdanos a difundirlas y anímate a unirte al equipo. Este es un documento vivo y puedes encontrar la última versión, así como el resto de guías completas, en nuestra web.

https://www.autentia.com/libros/

Si alguna vez, estando con amigos o familiares preguntan a qué me dedico exactamente, suelo contestar que mi trabajo consiste en colaborar en la construcción de la parte de las aplicaciones que hace que salgas en los periódicos únicamente cuando lo haces mal. **Eso es el Backend, eso que no se ve pero que todo el mundo da por supuesto**: se da por supuesto que los datos se intercambian de manera segura, que no se pierden o corrompen y que son vistos o modificados únicamente por los que tienen el permiso para hacerlo; se da por supuesto que el sistema debe responder con celeridad y que siempre está disponible; se da por supuesto que la información se intercambia entre distintos sistemas con fiabilidad y registrando en todo momento qué, quién y cuándo se accedió a esa información, etc. En definitiva, **se da por supuesto que funciona.**

“El Backend es la parte de las aplicaciones que el usuario percibe únicamente cuando NO funciona”



Quizás por su posición de elemento básico o por su naturaleza invisible al usuario, se ha extendido la percepción de ser un trabajo poco creativo, falto de imaginación o mucho peor: que para esto vale cualquiera, frase que no es tan rara de escuchar. Y esta estrecha y limitada visión produce sistemas frágiles y poco fiables, aplicaciones inconsistentes e inseguras abandonadas en manos de desarrolladores con poca experiencia y cuya única motivación es sobrevivir un día más, manteniendo una aplicación de la que hace tiempo perdieron el control, dirigidos por profesionales obsoletos porque hace tiempo que abandonaron las trincheras y son ya incapaces de ayudarles. Y sin embargo, éste es un campo que **sí necesita creatividad,** una que nace del análisis de los datos y de un conocimiento profundo del amplísimo ecosistema en el que se desenvuelven las aplicaciones, capacidad que surge del trabajo, de la mejora y el aprendizaje contínuo, de la experiencia y del compromiso, del trabajo en equipo y de la **enseñanza e inspiración** que algunos profesionales puedan ejercer sobre aquellos que comienzan en este apasionante mundo del Backend.

Haz un buen estudio de mercado, encuentra patrocinadores, diseña tu producto acorde a tus usuarios objetivo. Haz una espectacular interfaz de usuario que permita conquistar el mundo en un solo click. Construye una aplicación Web que sea fidelísima al diseño entregado y que además, sea responsive, adaptable y accesible. Despliega en la nube, con una arquitectura autoescalable y todos los extras que te ofrece tu plataforma favorita. Pero ten por seguro, que si la parte sobre la que se fundamenta todo el sistema no funciona, **nada de lo que has hecho servirá.**

Back

Guía completa

Índice

    - for-each
    - while
    - do/while
- Operadores
    - Operadores aritméticos
    - Operadores de asignación
    - Operadores de comparación
    - Operadores lógicos
    - Operadores bit a bit
    - Otros operadores
    - Prioridad entre operadores
- Clases, interfaces y anotaciones
    - Clases
    - Herencia y clases abstractas
    - Interfaces
    - Anotaciones
- Control de excepciones
    - try-with-resources
    - RuntimeException
- APIs básicas del lenguaje
    - Object
    - Arrays
    - Clases envoltorio
    - String
    - Fechas
    - Formateado de texto
- Concurrencia
    - Estados de un Hilo
    - Prioridades en los Hilos
    - Sincronización de hilos
    - Pools de hilos
    - ThreadLocal
    - Recomendaciones sobre concurrencia

- Generics
- Colecciones
    - Concurrencia y colecciones
- Lambdas
    - Sintaxis
    - Interfaces funcionales
    - Dónde pueden usarse las lambdas
    - Referencias a métodos
    - Interfaces funcionales estándar más importantes
- Data processing Streams
- IO
    - Serializable
- Optional

Parte 2 - Herramientas y técnicas

- Introducción a Git
    - Instalación inicial
    - Estructura interna de un repositorio
    - Ciclo de vida de un fichero
    - Comandos básicos
    - Herramientas comunes
    - Ramas
    - Problemas comunes y soluciones
        - Conflictos al mergear con otra rama
        - Cambiar el mensaje de un commit
        - Añadir cambios a un commit
        - Deshacer commits locales
        - Deshacer commits ya pusheados

- Introducción a la gestión de la configuración
    - Maven
        - Estructura de directorios
        - Ciclos de vida
        - Goals
        - Dependencias y Repositorios
        - Arquetipos
    - Gradle
- Introducción al testing
    - TDD y las pruebas como técnica de diseño
    - JUnit
        - Cambios entre JUnit4 y JUnit5
    - Hamcrest
    - AssertJ
    - Cobertura de código y JaCoCo
    - Dobles de Test
    - Recomendaciones
        - FIRST
        - Arrange - Act - Assert
- Entorno de ejecución
    - Depuración
        - Breakpoints
        - Observar variables
    - Gestion de logs

Parte 3 - El mundo de los microservicios

- Introducción a Spring
    - Spring IoC e Inyección de Dependencias
    - Spring Beans

        - Tipos de bean
        - Ciclo de vida
    - Tipos de configuración
        - XML
        - Anotaciones
        - Java
    - Resolución de conflictos entre beans
    - Spring Data
        - Introducción
        - Conceptos
        - ¿Qué es JDBC?
        - Spring JDBC
        - Spring Data JDBC
        - Gestión de la transaccionalidad
        - Connection Pooling
        - ¿Qué ofrece Spring respecto a Connection Pooling?
    - SpringMVC
        - Introducción
        - Modelo, vista, controlador
        - Flujo de ejecución
    - Rest
        - Niveles de cumplimiento de los principios REST
        - API REST con SpringMVC
        - Nuevas anotaciones
        - Clase ResponseEntity<T>
        - Solucionar problema CORS en nuestra API
- Introducción al desarrollo de microservicios
    - Qué son
    - Patrones de los microservicios
        - Service Discovery
        - Circuit Breaker
            - Hystrix
        - Bulkhead pattern

        - Externalized Configuration
        - API Gateway
        - Distributed Tracing y Central Log Analysis
        - Control loop
        - Centralized Monitoring
    - Spring Boot
        - Introducción
        - Convención frente a configuración
        - Starters
        - Fat Jar file
        - @SpringBootApplication
        - Microservicios con Spring Boot
            - Spring Cloud
            - Spring Cloud Netflix
- Micronaut
    - Introducción
    - GraalVM
    - Diferencias con Spring
    - Micronaut CLI
    - Inyección de dependencias en tiempo de compilación
    - Reactive IO
    - Cloud Native Features

Parte 4 - Kotlin

- Introducción a Kotlin
    - Un poco de historia
    - ¿Por qué usar Kotlin?
- Instalación de Kotlin
    - Instalación de Kotlin localmente
    - Ejecución de Kotlin en local
        - Kotlinc
        - Script
        - REPL
    - Ejecución de Kotlin en remoto
- Características de Kotlin
    - Conciso
        - No necesario punto y coma
        - Inferencia de tipos
    - Tipos en Kotlin
        - Numbers
        - Characters
        - Booleans
        - Arrays
    - Conversión explícita entre tipos
    - Diferencia entre val y var
    - String templates
- Programación orientada a objetos en Kotlin
    - Clases
    - Constructores
    - Propiedades
    - Accesores
    - Herencia

    - Interfaces
    - Niveles de acceso
    - Extensiones
    - Data Classes
    - Sealed Classes
    - Genéricos
        - Comodines para tipos genéricos
    - Clases anidadas
    - Clases Enumeradas
    - Objetos anónimos
    - Type Aliases
- Control de flujo
    - If / else
    - For
    - While / Do-While
    - When
- Nulos en Kotlin
    - Tipos nullables
    - Usando Safe Calls
    - Operador Elvis
    - Operador Not Null Assertion
    - Any y Nothing
- Funciones
    - Block body y el tipo de retorno opcionales
    - El tipo Unit
    - Vararg y spreads
    - Destructuring
    - Lambdas
    - Scope Functions

Parte 1

Introducción al Backend y Java

Tipos de aplicaciones

No todas las aplicaciones tienen las mismas características. En función del entorno en el que se ejecutan, podemos distinguir dos grandes grupos: aplicaciones de escritorio y aplicaciones web. A continuación, se ofrece una comparativa con sus principales ventajas y desventajas:

	Escritorio	Web
Ventajas	• Acceso completo a recursos. • Pueden funcionar sin conexión.	• No hace falta instalarlas. • Todos tienen la misma versión. • Válidas para cualquier S.O.
Desventajas	• Despliegue más complicado. • Específicas para un S.O. • Conflictos entre versiones.	• Requieren conexión. • Compatibilidad con distintos navegadores.

Aplicaciones de escritorio

Son las aplicaciones más tradicionales que podemos instalar en nuestro equipo. Las aplicaciones móviles también pertenecen a este grupo, aunque su planteamiento dista de aquellas que se desarrollaban en décadas pasadas.

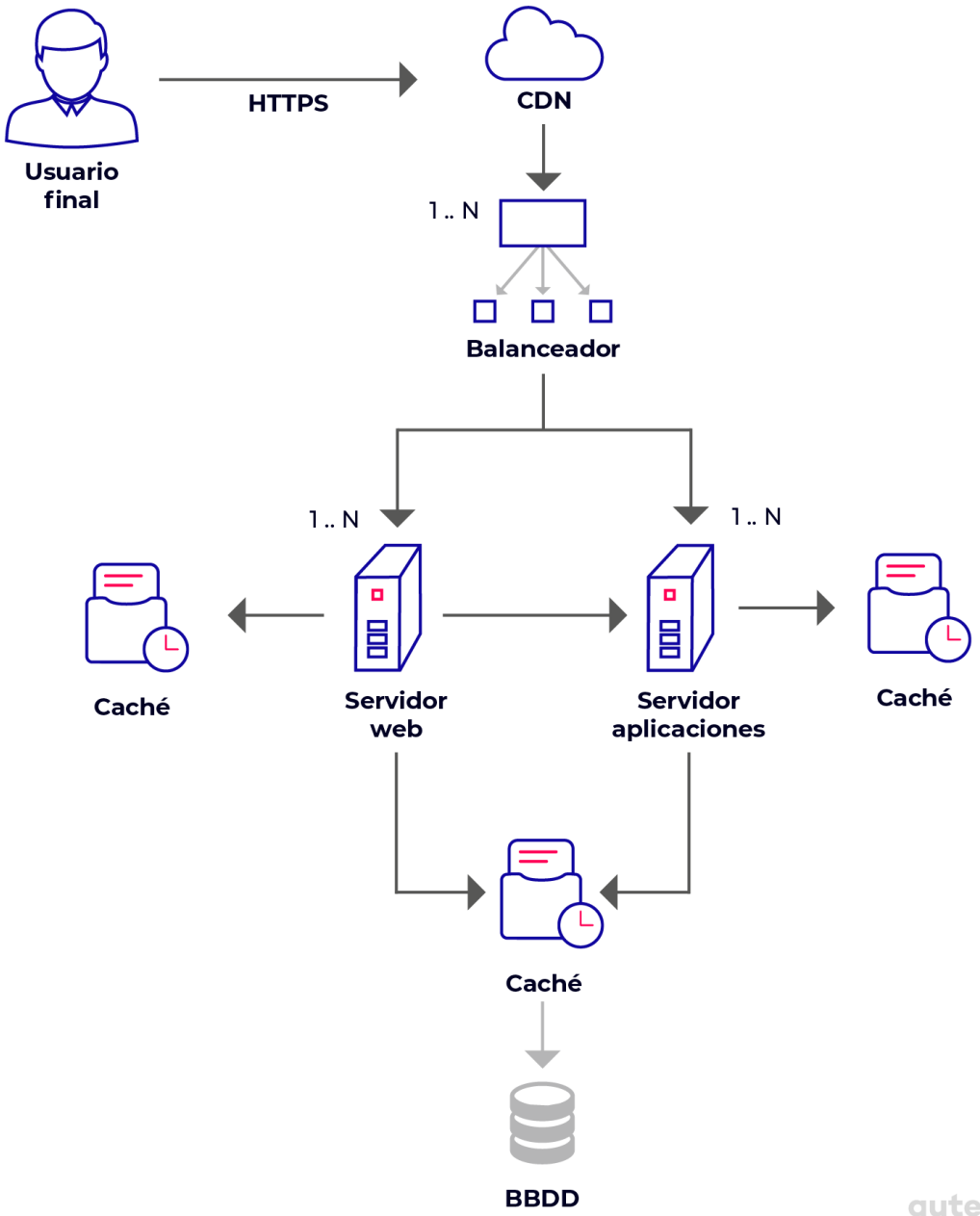
Este tipo de aplicaciones se caracteriza por tener, en mayor o menor grado, una buena parte del procesamiento de los datos en el propio dispositivo. Por tanto, tienen también un lado back instalado. La interfaz puede generarse de muchas formas. Pueden usar componentes del sistema operativo, interfaces web, renderizar componentes propios, etc.

Actualmente, muchas de estas aplicaciones también consumen datos desde servicios remotos. Algunas de ellas podrían llegar a considerarse una mera interfaz nativa para estos servicios.

Aplicaciones Web

Son la piedra angular de internet. Se encargan de gestionar las peticiones de millones de clientes a lo largo de todo el mundo. Mantienen la coherencia y la seguridad de los datos. Intercambian información con otras aplicaciones para ofrecernos servicios de interés.

En un inicio, se basaban en una arquitectura cliente-servidor que contenía todo lo necesario para funcionar, incluida una interfaz web para acceder y manipular la información. Esta visión ha evolucionado hacia aplicaciones que se distribuyen a lo largo de varios servidores o instancias en la nube, y se enfocan a actuar como back para atender peticiones de múltiples clientes a través de servicios SOAP o REST. Incluso, en muchas ocasiones, encontramos que esta relación se establece entre varias aplicaciones web, donde una no puede funcionar sin acceso a las otras.


Usuario final
HTTPS
CDN
1 .. N
Balanceador
1 .. N
1 .. N
Caché
Servidor web
Servidor aplicaciones
Caché
Caché
BBDD
autentia

Lenguajes de programación

Un lenguaje de programación no es más que un conjunto de reglas gramaticales que son usadas para decirle a un ordenador cómo llevar a cabo una determinada tarea.

Los lenguajes de programación pueden ser categorizados en base a distintos criterios. Por ejemplo, podemos clasificarlos entre **lenguajes de alto o bajo nivel.** Un lenguaje será de más alto nivel cuanto mayores sean las abstracciones que nos permitan trabajar con él de una forma más similar a la que puede pensar un humano y no una máquina.

Otra forma de categorizarlos es por **paradigma**. Existen varios paradigmas

de programación y los lenguajes pueden adoptar uno o varios de estos paradigmas. A veces, está en la mano del programador escribir el código usando un paradigma u otro dentro del mismo lenguaje o incluso combinando varios paradigmas. Por ejemplo, a partir de **Java 8** podemos escribir programas usando mayormente la **programación orientada a objetos** pero aprovechando algunas de las ventajas de la **programación funcional.**

Algunos paradigmas son:

- Programación imperativa.
- Programación declarativa.
- Programación lógica.
- Programación funcional.
- Programación estructurada.
- Programación orientada a objetos.
- Programación reactiva.

También podemos diferenciar los lenguajes entre **lenguajes compilados e interpretados**. Un **lenguaje compilado** será un lenguaje que a través de un compilador es convertido a código máquina que el procesador es capaz de ejecutar directamente. Mientras que un **lenguaje interpretado** será traducido por un intérprete al momento de ejecutarse y este intérprete ejecutará el código máquina correspondiente

Existen casos un poco más especiales como el de Java, que aunque es compilado, no es compilado a código máquina si no a **bytecode**, un lenguaje intermedio que solo la **JVM** (Java Virtual Machine) es capaz de interpretar, siendo necesario disponer de una para poder ejecutar el programa.

Lenguaje interpretado vs. compilado

autentia

¿En qué consiste?

Los lenguajes de programación **compilados** son aquellos que requieren de un paso previo, en el que un compilador traduce todas las instrucciones del programa a código máquina. Un lenguaje **interpretado** realiza dicha traducción en tiempo de ejecución instrucción a instrucción.

CONSIDERACIONES

A día de hoy es raro encontrar lenguajes que se puedan considerar puramente interpretados. Lenguajes como Java, JavaScript, Python, etc., que históricamente fueron considerados como interpretados, hoy en día en realidad se podría decir que son un híbrido, ya que la mayoría realizan un paso previo en el cual traducen el código a **bytecode**, el cual es agnóstico a la plataforma en la cual se ejecuta el código y se suelen apoyar en máquinas virtuales que interpretan dicho **bytecode** y lo traducen a código máquina (por ejemplo la JVM para Java o el motor V8 para Javascript). De esta forma consiguen mayor eficiencia.

Por otra parte, en el caso de lenguajes de scripting, como bash y otros tipos de *shell* es más fácil considerarlos lenguajes interpretados.

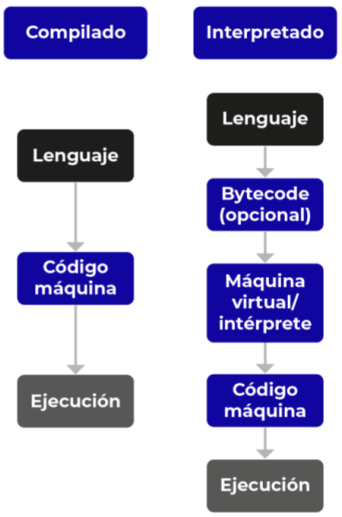
EJEMPLOS

Lenguajes compilados:

- Swift.
- C.
- C++.
- C#.
- Objective-C.
- Kotlin/Native.

Lenguajes interpretados:

- Bash, sh, zsh...
- Java.
- JavaScript.
- Python.
- Perl.
- Kotlin (JVM).



Java Virtual Machine - JVM

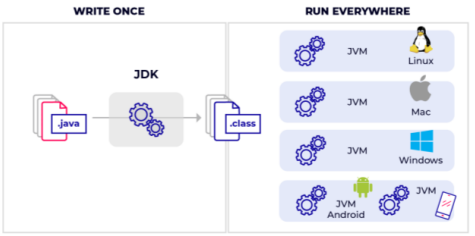
autentia

¿Qué es?

Máquina virtual que **permite ejecutar código desarrollado en Java en cualquier sistema operativo**. JVM interpreta y compila a código nativo (de la plataforma concreta de ejecución) las instrucciones expresadas en un código binario especial (**bytecode**), el cual es generado por el compilador del JDK (Java Development Kit).

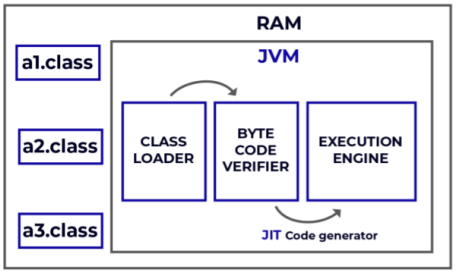
OBJETIVO

La idea cuando se desarrolló Java era **poder ejecutar el código en cualquier plataforma (Write Once Run Anywhere)**. Esto significa que un desarrollador puede escribir código Java en un sistema específico y sabe que se va a poder ejecutar en cualquier otro sin ninguna configuración adicional. La JVM es la que permite diversificar su uso y cada plataforma tiene su propia JVM, que se adapta a cada arquitectura/SO.



¿CÓMO FUNCIONA?

1. Un fichero *example.java* se compila (gracias al compilador del JDK) y se genera un fichero *example.class* que contiene el bytecode capaz de ser interpretado por la JVM.
2. Durante la ejecución del código, ***Class Loader*** se encarga de llevar los ficheros .class a la JVM que reside en la RAM del ordenador y ***ByteCode Verifier*** se encarga de verificar que el código en bytecode procede de una compilación válida.
3. El **compilador Just in Time** (esto se hace en tiempo de ejecución) compila el bytecode a código nativo de la máquina y se ejecuta directamente.



Paradigmas

Programación orientada a objetos (POO)

Este paradigma representa entidades del mundo real o internas del sistema mediante objetos, que son estructuras que tienen datos, normalmente llamados **propiedades o atributos**, y a la vez comportamientos (funciones), normalmente llamados **métodos**.

En la mayoría de los lenguajes orientados a objetos, los objetos son creados a partir de **clases**. Llamaremos instancia de una clase a un objeto creado a partir de la misma. Las clases definen qué atributos y métodos tendrán sus objetos.

Cada lenguaje puede tener su propia forma de implementar este paradigma y las ideas aquí expresadas son ideas generales que no tienen porqué aplicarse idénticamente en todos los lenguajes que soportan este paradigma.

Algunas de las principales características de la POO son:

Herencia

La herencia es uno de los recursos principales para reutilizar código en POO, aunque no siempre el más recomendado. Consiste en la posibilidad de **heredar** desde una clase, métodos y propiedades de otra. Por lo general, definimos una clase como una subclase de otra, esto significa que todos los objetos de la subclase son también objetos de la clase padre. Por ejemplo, una clase Trabajador podría heredar de una clase Persona y diríamos, por lo tanto, que un Trabajador **es** una Persona.

Abstracción

La herencia a veces se nos queda corta. Cuando queremos que todos los hijos tengan cierto comportamiento, cierta funcionalidad pero no queremos dar una implementación de la misma, entonces usamos Abstracción. La

abstracción nos permite obligar a que nuestros hijos o sus sucesivos hijos, se vean obligados a implementar cierta funcionalidad. Es posible, incluso, utilizar esa funcionalidad desde otras funciones de una clase abstracta. Esto es así porque el lenguaje se asegura de que esa funcionalidad va a estar implementada cuando se use. No se deja instanciar objetos de clases que tengan alguna funcionalidad abstracta. Se tiene que haber implementado para poder instanciar un objeto de esa clase.

Polimorfismo

La idea es que cualquier referencia de una subclase puede ser utilizada donde la superclase (clase de la que se hereda) pueda ser usada. De esta forma, el comportamiento de la subclase en concreto será ejecutado.

Es decir, volviendo al ejemplo de un Trabajador que hereda de Persona, podremos utilizar un objeto de la clase Trabajador en cualquier otro sitio donde una Persona pueda ser utilizada. Ya que un trabajador es también una persona.

Encapsulación

Consiste en agrupar los elementos que corresponden a una misma entidad en el mismo nivel de abstracción. Estos luego se protegen con distintos mecanismos. Estos mecanismos de protección pueden depender del lenguaje.

Principio de ocultación

Se ocultan las propiedades del objeto de forma que estos solo puedan ser accedidos a través de sus métodos.

Alta cohesión y bajo acoplamiento

Uno de los objetivos de la POO es conseguir una alta cohesión y un bajo

acoplamiento.

Una alta cohesión consiste en que una clase o módulo tenga un propósito claro y los conceptos que son parecidos o iguales se mantengan juntos.

Un bajo acoplamiento se refiere a que las clases o módulos tienen que depender y conocer el funcionamiento lo menos posible de otros módulos o clases del software.

Programación funcional

Este paradigma de programación sigue un estilo de desarrollo declarativo y está basado en el uso encadenado de funciones. Aunque hoy en día, el desarrollo sigue estando más enfocado a la metodología imperativa, cada vez más lenguajes como Java, C#, Python, Kotlin, Php, etc., están incorporando funciones y librerías para el desarrollo funcional. Un ejemplo muy común son las expresiones Lambda.

Algunas características de este paradigma son:

- Funciones de orden superior: una función puede recibir una o más funciones por parámetro y a su vez, podría retornar otra. Además, las funciones pueden ser asignadas a una variable.
- “Qué” en vez de “Cómo”: su enfoque principal es "qué resolver", en contraste con el estilo imperativo donde el enfoque es "cómo resolver".
- No soporta estructuras de control: aplica la recursividad para resolver problemas que en lenguajes imperativos se resolverían con bucles o condicionales.
- Funciones puras: el valor retornado por una función será el mismo siempre que los parámetros de entrada sean iguales. Esto significa que durante el proceso no va a haber efectos secundarios que muten el estado de otras funciones. Esto ayuda a reducir los bugs en los programas y facilita su testeo y depuración.

Programación reactiva

No debemos confundir programación reactiva con sistemas reactivos. Los sistemas reactivos están definidos en el Manifiesto Reactivo.

Al igual que el paradigma anterior, la programación reactiva está basada en el desarrollo declarativo. Se enfoca en flujos (streams) de datos asíncronos y en un modelo basado en eventos, permitiendo la propagación de los cambios de forma automática, donde la información se envía al consumidor a medida que está disponible. Esto permite realizar tareas en paralelo no bloqueantes que ofrecen una mejor experiencia al usuario.

Los lenguajes que soportan la programación reactiva suelen tener su propia librería con una serie de funciones para crear, transformar, combinar o filtrar los streams. Un stream es una secuencia de eventos (pudiendo ser de cualquier tipo) ordenados en el tiempo que puede devolver tres tipos de resultados: un valor, un error o una señal de completado. Estos resultados

generados se emiten de forma asíncrona a través de una función que es ejecutada por el suscriptor u observador. Lo mencionado es, básicamente, el patrón Observer, ya que tenemos un sujeto (el stream) que está siendo observado por las funciones mencionadas (observadores o suscriptores).

Comportamiento - Observer autentia

Reaccionando a cambios de estado

El patrón observador (*observer* en inglés) describe una solución que se asemeja al manejo de eventos. Principalmente es utilizado para permitir que ciertos objetos puedan reaccionar a los cambios que suceden en un momento dado en otros objetos.

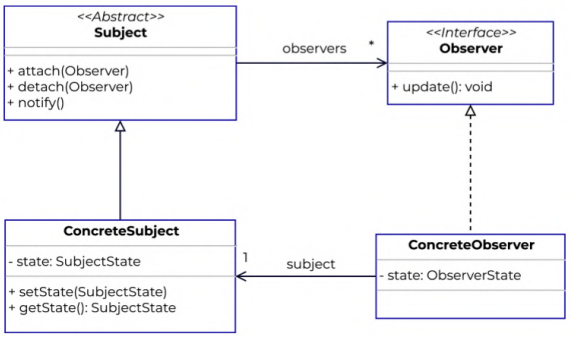
CONCEPTO

Dado el diagrama, tenemos una clase abstracta, **Subject**, que implementa una serie de métodos para enlazar observadores a la clase. Cuando el estado del subject cambie, notificará a sus observadores, invocando el método *update* en cada una de ellas.

La interfaz **Observer** expone un solo método, cuya invocación dependerá del Subject. La lógica de este método es una reacción al cambio sucedido en Subject.

Las clases **ConcreteSubject** y **ConcreteObserver** denotan una manera de implementar el patrón. Se observa que el ConcreteObserver tiene una referencia al ConcreteSubject, lo cual le permite obtener su estado. De esta manera, cuando el Subject notifique a sus Observers, esta implementación en particular, tendrá una referencia directa al Subject para poder reaccionar en base a los cambios sucedidos.

Esta no es la única forma de diseñar el patrón Observer, pero sí es la clásica descrita por el *GoF* (Gang of Four).

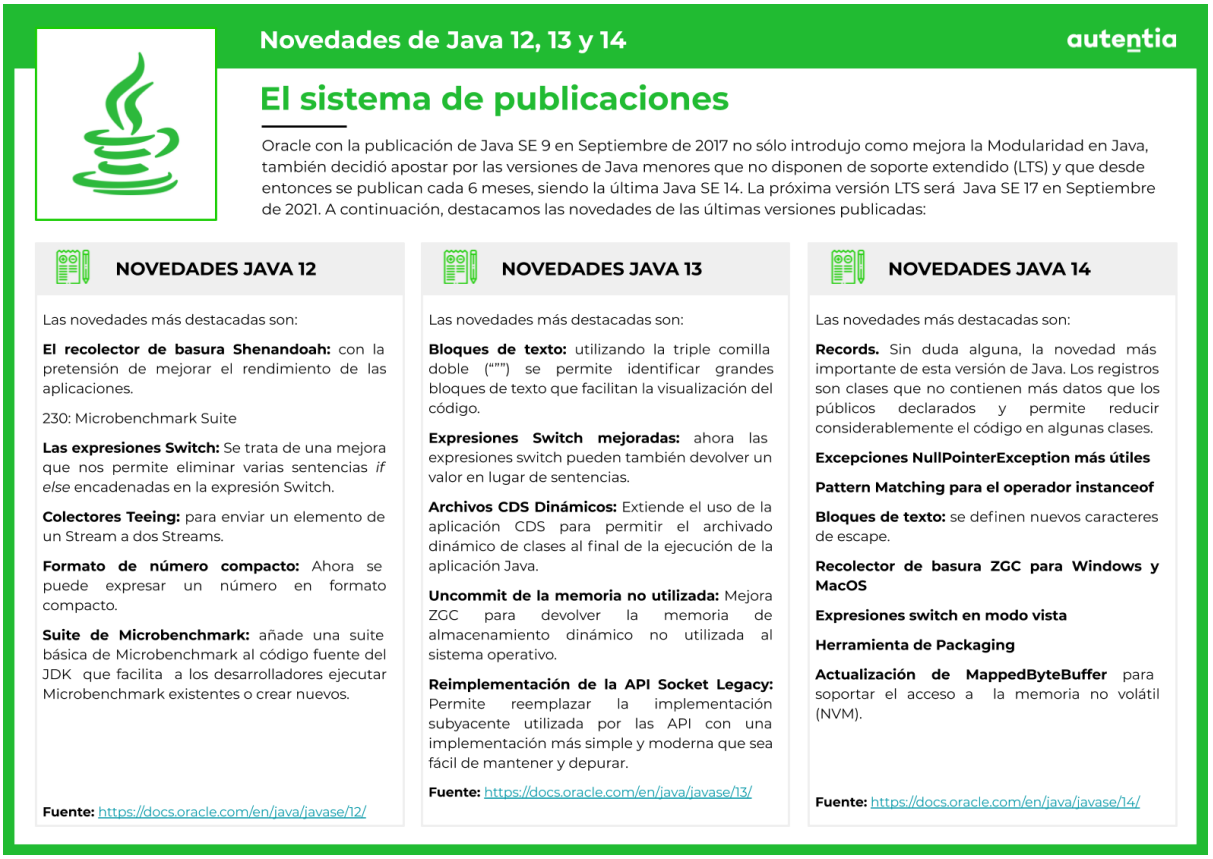


Java

A mediados de la década de los 90, **Sun Microsystems** definió Java como “un lenguaje de programación ‘sencillo’ **orientado a objetos**, distribuido, con una arquitectura neutra y portable, seguro y concurrente”.

La evolución de Java se lleva a cabo por el **Java community Process (JCP)** a través de **Java Specification Request (JSR)**. Su desarrollo ha pasado por varias manos, empezando en **Sun Microsystems** que fue comprada por **Oracle** en 2009. Sin embargo, también se han realizado implementaciones **open source** de la plataforma. Hoy en día tienen más relevancia que nunca.

Pero Java no es sólo el lenguaje, sino que engloba también las plataformas que permiten su uso: **SE** (standard Edition, la más habitual), ME (Micro Edition), pensada para móviles, Java Embedded (IoT), EE (Enterprise Edition), enfocada a servidores y aplicaciones web, Java TV, Java Card...



Java tiene dos componentes principales:

- Java Runtime Environment (**JRE**): es el entorno de ejecución de Java. Incluye la máquina virtual (**JVM**), las librerías básicas del lenguaje y otras herramientas relacionadas como Java Access Bridge (JAB), Remote Method Invocation (RMI), herramientas de monitorización...
- Java Development Kit (**JDK**): además del JRE, incluye el compilador, el debugger, el empaquetador JAR, herramientas para generar documentación...

Classpath

El classpath indica a Java dónde debe buscar las clases de usuario; esto es, aquellas que no pertenecen al JRE y que son necesarias para poder compilar o ejecutar la aplicación. Por defecto, el classpath se limita al directorio actual. Podemos modificar el classpath de dos maneras distintas:

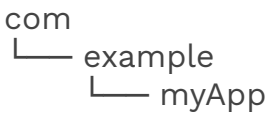
- Mediante la opción -cp en la línea de comandos. Es el método preferido ya que especifica un classpath diferente para cada aplicación, sin que afecte al resto.
- Declarándolo como una variable de entorno.

Se pueden declarar cuantas ubicaciones sean necesarias en el classpath.

Paquetes

En Java, el código se organiza en **paquetes**. Cada paquete forma un **namespace** propio, de forma que se evitan los conflictos de nombres entre elementos de distintos paquetes.

Los paquetes se corresponden con estructuras de árbol de directorios. Por convención, se utiliza un dominio del que tengamos la propiedad como prefijo de los paquetes, aunque a la inversa. Por ejemplo, si estamos desarrollando MyApp y tenemos en propiedad el dominio www.example.com, podríamos nombrar nuestro paquete como com.example.myApp y se correspondería con la siguiente estructura de directorios:



Dentro del código fuente de nuestra clase, también deberemos indicar el paquete al que pertenece. Si no coincide con la estructura de directorios, el compilador lanzará errores, pues no encontrará las clases que necesita. Esto se hace al principio del fichero con la siguiente sentencia:

```
package com.example.myApp;
```

Si queremos referenciar una clase dentro de un paquete, debemos escribir todo el nombre completo. No obstante, Java proporciona un método de

importación que permite abreviar esta nomenclatura en nuestro código, siempre que no haya conflicto entre dos nombres de diferentes paquetes.

```
import java.util.List; // Podremos referenciarlo como List
```

Los componentes del paquete java.lang siempre están cargados y no necesitan ser importados para usarse.

Compilar

Una vez hemos escrito nuestro código, el siguiente paso es compilarlo. Para ello, necesitaremos el JDK. Hay que tener en cuenta que nuestro código debe adaptarse a la versión del JDK que tengamos. Revisa las características y especificaciones que incluye cada versión de Java.

Podemos encontrar el compilador javac dentro del directorio bin de nuestra instalación. Ejecutaremos el comando de la siguiente forma:

```
$ javac MyApp.java
```

Este comando generará uno o varios ficheros .class a partir de nuestro archivo fuente .java. Estos son los archivos que puede ejecutar la máquina virtual de Java.

Ejecutar

Para ejecutar una aplicación, usaremos java, que lo podemos encontrar en el directorio bin del JRE o JDK. Para ello, debemos hacer referencia a una clase que contenga un método estático main, el cual es siempre el punto de entrada de las aplicaciones en Java. Además, si forma parte de un paquete, deberemos escribir la ruta completa desde la base del árbol.

```
$ java com.example.myApp.MyApp
```

Como se aprecia, no es necesario incluir la extensión .class. Podemos declarar algunas opciones adicionales, como el classpath en caso de necesitarlo o ampliar la memoria disponible para la máquina virtual, si encontramos que la aplicación es pesada y no funciona o tiene un rendimiento bajo.

Empaquetado de aplicaciones y librerías

Java permite empaquetar las aplicaciones y librerías en archivos comprimidos. De esta forma, es más sencillo poder reutilizar el código a través de distintas aplicaciones o desplegar nuevas versiones de la aplicación. Estos archivos pueden ser:

- JAR: librerías o aplicaciones de escritorio.
- WAR: aplicaciones web.

El comando para crear un archivo JAR es el siguiente:

```
$ jar cf jar-file files-to-package
```

La opción c indica que se desea crear el archivo y la opción f especifica el nombre del archivo. Este comando genera un comprimido .jar que contiene todas las clases que indiquemos, incluyendo directorios de forma recursiva. Además, genera un archivo de manifiesto.

Si el archivo de manifiesto especifica el header Main-Class, podremos ejecutar la aplicación desde el archivo JAR de la siguiente forma:

```
$ java -jar jar-file
```

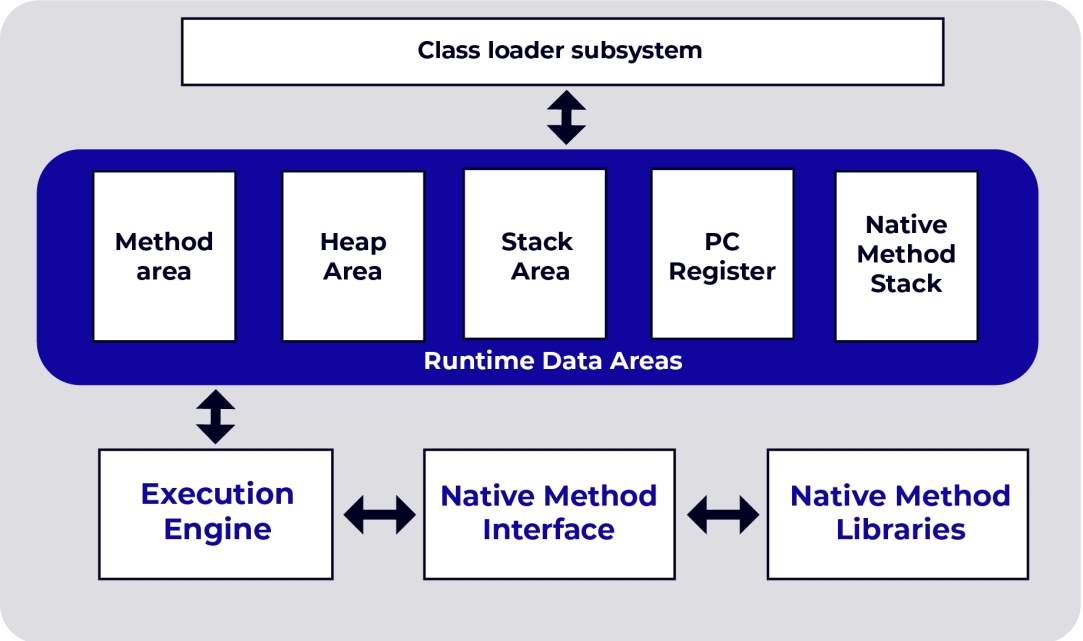
Los archivos JAR también pueden ser agregados al classpath, de forma que las aplicaciones puedan obtener sus dependencias al explorar dentro de su contenido. Es la principal forma de distribución de librerías. Normalmente, cuando descargamos una aplicación Java, esta trae sus propios JAR además de las dependencias.

Java Virtual Machine (JVM)

La **Máquina Virtual de Java**, en inglés Java Virtual Machine (JVM), es un componente dentro de JRE (Java Runtime Environment) necesario para la ejecución del código desarrollado en Java, es decir, es la máquina virtual la que permite ejecutar código Java en cualquier sistema operativo o arquitectura. De aquí que se conozca Java como un lenguaje multiplataforma.

JVM **interpreta y ejecuta** instrucciones expresadas en un código máquina especial (**bytecode**), el cual es generado por el compilador de Java (también ocurre con los generados por los compiladores de lenguajes como Kotlin y Scala). Dicho de otra forma, es un proceso escrito en C o C++ que se encarga de interpretar el bytecode generado por el compilador y hacerlo funcionar sobre la infraestructura de ejecución. Como hay una versión de la JVM para cada entorno que sí conoce los detalles de ejecución de cada sistema, puede utilizar el código máquina equivalente para cada una de las instrucciones bytecode.

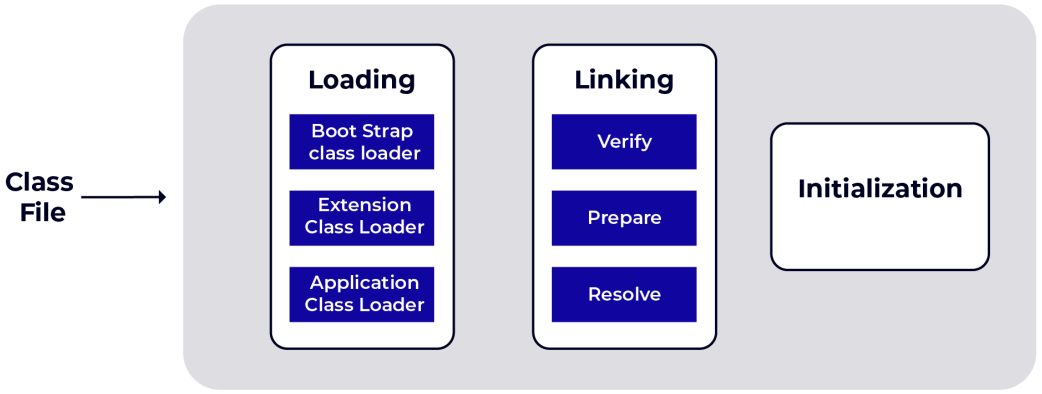
JVM se divide en 3 subsistemas que vamos a explicar a continuación:



autentia

Class Loader Subsystem

Cuando una clase Java necesita ser ejecutada, existe un componente llamado **Java Class Loader Subsystem** que se encarga de **cargar, vincular e inicializar** de forma dinámica y en tiempo de ejecución las distintas clases en la JVM. Se dice que el proceso es dinámico porque la carga de los ficheros se hace gradualmente, según se necesiten.



Existen tres tipos de **Loaders** y cada uno tiene una ruta predefinida desde donde cargar las clases:

- **Bootstrap/Primordial ClassLoader:** es el padre de los loaders y su función es cargar las clases principales desde jre/lib/rt.jar, fichero que contiene las clases esenciales del lenguaje.
- **Extension ClassLoader:** delega la carga de clases a su padre (bootstrap) y, en caso fallido, las carga él mismo desde los directorios de extensión de JRE (jre/lib/ext)
- **System/Application ClassLoader:** es responsable de cargar clases específicas desde la variable de entorno *CLASSPATH* o desde la opción por línea de comandos *-cp*.

Linking es el proceso de añadir los bytecodes cargados de una clase en el Java Runtime System para que pueda ser usado por la JVM. Existen 3 pasos en el proceso de Linking, aunque el último es opcional.

- **Verify: Bytecode Verifier** comprueba que el bytecode generado es correcto. En caso de no serlo, se devuelve un error.
- **Prepare:** una vez se ha verificado, se procede a asignar memoria a las variables de las clases y se inicializan con valores por defecto (tabla inferior) dependiendo de su tipo. Importante saber que las variables

de clase **no se inicializan con sus valores iniciales correctos hasta la fase de Initialization.**

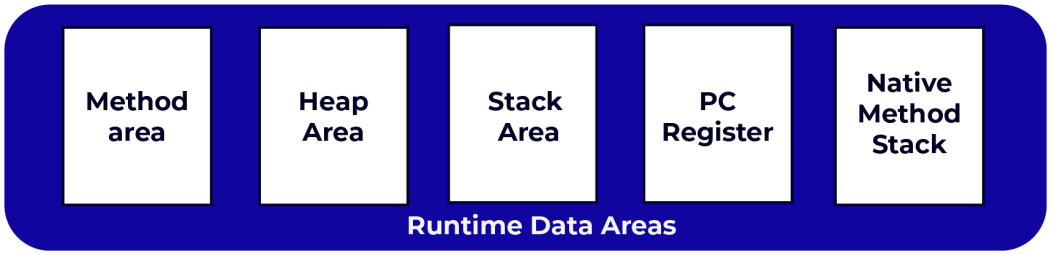
Tipo	Valor inicial
int	0
long	0L
short	(short) 0
char	“\u0000”
byte	(byte) 0
boolean	false
reference	null
float	0.0f
double	0.0d

- **Resolve:** JVM localiza las clases, interfaces, campos y métodos referenciados en una tabla llamada *constant pool (CP)* y determina los valores concretos a partir de su referencia simbólica. Cuando se compila una clase Java, todas las referencias a variables y métodos se almacenan en el CP como referencia simbólica. Una referencia simbólica, de forma muy breve, es un string que puede usarse para devolver el objeto actual. El CP es un área de memoria con valores únicos que se almacenan para reducir la redundancia. Para el siguiente ejemplo `System.err.println("Autentia");` `System.out.println("Autentia");` en el CP solo habría un objeto String “Autentia” .

El último paso en el proceso del ClassLoader es **Initialization,** que se encarga de que las variables de clase se inicialicen correctamente con los valores que el desarrollador especificó en el código.

Runtime Data Areas

JVM define varias áreas de datos que se utilizan durante la ejecución de un programa y que se podrían dividir en dos grupos. Algunas de estas áreas se crean al inicializarse la JVM y se destruyen una vez la JVM finaliza (compartidas por todos los hilos). Otras se inicializan cuando el hilo se crea y se destruyen cuando el hilo se ha completado (una por hilo).

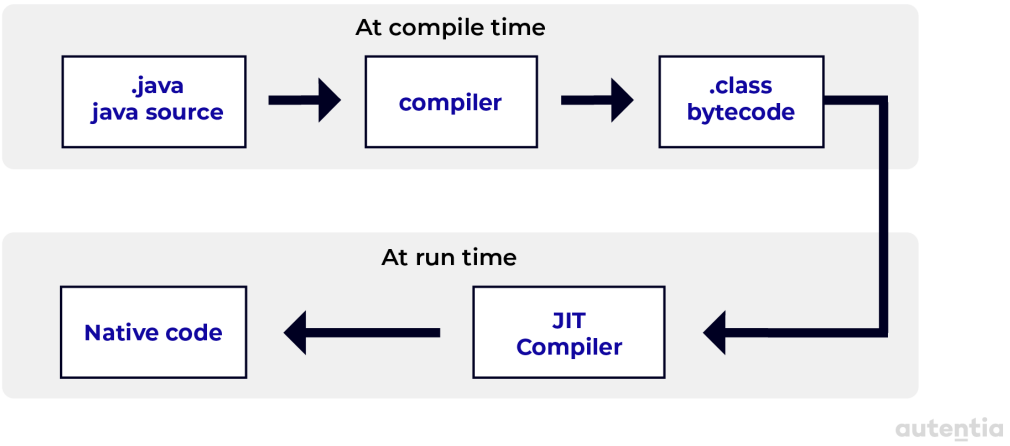


- **Method Area:** es parte de *Heap Area*. Contiene el esqueleto de la clase (métodos, constantes, variables, atributos, constructor, etc.).
- **Heap Area:** fragmento de memoria donde se almacenan los objetos creados (todo lo que se inicialice con el operador *new*). Si el objeto se borra, el *Garbage Collector* se encarga de liberar su espacio. Solo hay un Heap Area por JVM, por lo que es un recurso compartido (igual que Method Area).
- **Stack Area:** fragmento de memoria donde se almacenan las variables locales, parámetros, resultados intermedios y otros datos. Cada hilo tiene una private JVM stack, creada al mismo tiempo que el hilo.
- **PC Register:** contiene la dirección actual de la instrucción que se está ejecutando (una por hilo).
- **Native Method Stack:** igual que Stack, pero para métodos nativos, normalmente escritos en C o C++.

Execution Engine

El bytecode que es asignado a las áreas de datos en la JVM es ejecutado por el Execution Engine, ya que este puede comunicarse con distintas áreas de memoria de la JVM. El *Execution Engine* tiene los siguientes componentes.

- **Interpreter:** es el encargado de ir leyendo el bytecode y ejecutar el código nativo correspondiente. Esto afecta considerablemente al rendimiento de la aplicación.
- **JIT Compiler:** interactúa en tiempo de ejecución con la JVM para compilar el bytecode a código nativo y optimizarlo. Esto permite mejorar el rendimiento del programa. Esto se hace a través del **HotSpot compiler.**



- **Garbage Collector:** libera zonas de memoria que han dejado de ser referenciadas por un objeto.

Para poder ejecutar código Java, necesitamos una VM como la que acabamos de ver. Si nos vamos al mundo de JavaScript, necesitamos el motor V8 que usa Google Chrome. Estaría bien poder tener una sola VM para distintos lenguajes y aquí es donde entra **GraalVM**. Es una extensión de la JVM tradicional que **permite ejecutar cualquier lenguaje en una única VM** (JavaScript, R, Ruby, Python, WebAssembly, C, C++, etc.). El objetivo principal es mejorar el rendimiento de la JVM tradicional para llegar a tener

el de los lenguajes nativos, un desarrollo políglota, así como reducir la velocidad de inicio a través de la compilación **Ahead of Time (AOT)**. Esto permite compilar algunas clases antes de arrancar la aplicación (en tiempo de compilación).

Control de flujo

Son las sentencias que permiten controlar el flujo y orden de la ejecución de un programa.

if/else

Los bloques if/else nos permiten ejecutar solo ciertas partes del código en función de las condiciones que pasemos.

```
if (expresión booleana 1) {
  // Código a ejecutar si la expresión 1 es verdadera
} else if (expresión booleana 2) {
  // Código a ejecutar si la expresión 1 es falsa y la 2 es
verdadera
} else {
  // Código a ejecutar si ninguna expresión es verdadera
}
```

switch

Switch evalúa la expresión entre paréntesis y ejecuta las sentencias que siguen al caso que coincide con el nuestro. Switch seguirá ejecutando todas las sentencias que siguen, aunque sean parte de otro caso, hasta que se encuentre un break. Funciona con los primitivos byte, short, char, int y sus wrappers. También funciona con enums y la clase String.

```
switch (expresión) {
  case "ABC":
    // Código a ejecutar si la expresión es "ABC"
  case "DEF":
    // Código a ejecutar si la expresión es "ABC" o "DEF"
    break;
  case "GHI":
    // Código a ejecutar si la expresión es "GHI"
    break;
  default:
    // Código a ejecutar si la expresión no es ninguna de las
anteriores
    break;
}
```

for

El bucle for repite una serie de sentencias mientras se cumpla una condición. En la primera expresión antes del punto y coma podemos definir y asignar una variable, en la segunda establecemos la condición que tiene

que cumplir el bucle para continuar y al final, tenemos el incremento.

```
for (int i = 0; i < 10; i++) {
  // Sentencias
}
```

for-each

Funciona con arrays y clases que implementen Iterable. Permite iterar sobre todos los elementos de una colección de manera sencilla

```
for (Clase elemento : colección) {
  // Sentencias
  // No puede modificarse la colección mientras se recorre, ya que resultará en un ConcurrentModificationException
}
```

while

El bucle while ejecutará una serie de sentencias mientras una expresión se cumpla.

```
while (expresión) {
  // Sentencias
}
```

do/while

El código se ejecutará al menos una vez y se seguirá ejecutando mientras se cumpla la condición.

```
do {
  // Sentencias
} while(expresión);
```

Operadores

Java nos proporciona multitud de operadores para manipular variables. Podemos clasificarlos como operadores unarios, binarios o ternarios en función de si actúan sobre uno, dos o tres elementos, respectivamente. También podemos clasificarlos en función del tipo de datos sobre los que actúan.

Operadores aritméticos

Nos permiten hacer operaciones matemáticas con tipos numéricos (int, long, double y float). Tenemos las operaciones matemáticas usuales *+, -, *, / y %*. También tenemos los operadores unarios *++* y *--*, que incrementan y decrementan en uno el valor de una variable, respectivamente. Tenemos que tener en cuenta que al hacer operaciones aritméticas entre datos de tipo int, el resultado siempre va a ser de tipo int.

```
5 + 7;   // 12
5 - 7;   // -2
6 * 7;   // 42
9 / 2;   // 4
9.0 / 2; // 4.5
9 % 2;   // 1

int num = 5;
double otroNum = 11.5;

num++;      // num ahora vale 6
otroNum--;  // otroNum ahora vale 10.5
```

Operadores de asignación

Nos permiten asignar valores a variables. El operador más usado es *=*, que asigna un valor concreto a una variable. También son bastante usados los operadores *+=* y *-=* que nos permiten incrementar o decrementar,

respectivamente, una variable el valor que especifiquemos.

```
String hola = "Hola!!";  // La variable hola ahora vale "Hola!!"
int num = 7;  //La variable num ahora vale 7

num += 2; // Es equivalente a escribir num = num + 2
num -= 5; // Es equivalente a escribir num = num - 5
num *= 7; // Es equivalente a escribir num = num * 7
num /= 9; // Es equivalente a escribir num = num / 9
```

Operadores de comparación

Nos permiten comparar dos valores. Tenemos los siguientes operadores:

- ***==:*** devuelve true si dos valores son iguales.
- ***!=:*** devuelve true si dos valores son diferentes.
- ***<:*** devuelve true si el primer valor es estrictamente menor que el segundo.
- ***<=:*** devuelve true si el primer valor es menor o igual que el segundo.
- ***>:*** devuelve true si el primer valor es estrictamente mayor que el segundo.
- ***>=:*** devuelve true si el primer valor es mayor o igual que el segundo.

```
“Hola” == “Adios”; // false

1 != 2;   // true
1 < 2;    // true
1 <= 5;   // true
1 > 1;    // false
1 >= 1;   // true
```

Podemos usar los operadores de desigualdades (<, <=, >, >=) con variables no numéricas, pero no se recomienda este uso ya que puede dar lugar a muchas confusiones. Para comparar objetos no debemos usar el operador ==, sino *.equals(),* ya que == comprueba si ambos son el mismo objeto y no

su valor.

```
'a' < 'b'; // true
'a' < 'B'; // false ya que no se usa su posición en el abecedario
sino su valor ASCII

String str1 = new String("Hola mundo");
String str2 = new String("Hola mundo");

str1 == str2;       // false a pesar de que ambos valen Hola mundo
str1.equals(str2)); // true
```

Operadores lógicos

Nos permiten realizar operaciones con valores booleanos. Tenemos los siguientes operadores:

- ***&&***: and lógico, devuelve true si ambas expresiones son true.
- ***||***: or lógico, devuelve true si una de las expresiones es true.
- ***!***: *not lógico,* devuelve el contrario del valor de la expresión.

```
int x = 5;

x < 6 && x = 8 // false,   true && false = false
x < 6 || x = 8 // true,    true || false = true
! (x < 6)      // false,   !true = false
```

Operadores bit a bit

Realizan operaciones bit a bit. No se recomienda usarlas pues su resultado es poco intuitivo.

```
int x = 5; // 5 = 0101
int y = 3; // 0011

x & y;     // 1,     0101 & 0011 = 0001
```

```
x | y;     // 7,     0101 | 0011 = 0111
x << 2;   // 20,    0101 << 2 = 010100
```

Otros operadores

El operador *+* también se puede usar para concatenar cadenas de texto.

```
"Hola" + "mundo" // "Holamundo"
```

El operador ternario *?:* tiene la estructura `condicion ? valorSiTrue : valorSiFalse`. El operador evalúa la condición pasada como primer primer argumento. Si la condición es cierta se devuelve el primer valor y si es falsa, se devuelve el segundo. Se suele usar para sustituir bloques if-else.

```
// El operador ?: tiene la siguiente estructura:
// condición ? expresión si condición es true : expresión si no

String str =  x > 5 ? "x es mayor que 5" : "x es menor o igual que 5";

// Es equivalente a:
String str = "";

if (x > 5) {
  str = "x es mayor que 5";
} else {
  str = "x es menor o igual que 5";
}
```

Prioridad entre operadores

Si en una expresión tenemos más de un operador se evaluarán siempre siguiendo el siguiente orden:

- *++* y *--*
- *!*
- **, /* y *%*
- *+, -*

- *<, <=, > y >=*
- *== y !=*
- *&&*
- *||*
- *?:*
- *=, +=, -=, *= y /=*

```
int x = 5 + 7 * 6;

// Primero se evalúa * y tenemos:
int x = 5 + 42;
// Ahora se evalúa + y tenemos:
int x = 47;

// Ahora se evalúa = y tenemos que x vale 47
```

Un caso específico en el que es importante tener en cuenta la prioridad de operadores es al usar *++*.

- Si hacemos *++var,* primero se incrementa el valor de *var* y luego el resto de la expresión.
- Si hacemos *var++*, primero se evalúa la expresión y luego se incrementa el valor de *var*.

Se puede ver bien la diferencia en el siguiente ejemplo:

```
int num1 = 5;
int num2 = 5;

int var1 = num1++; // var1 = 5,   Primero se asigna valor a var1 y
luego se incrementa el valor de num1

int var2 = ++num2; // var2 = 6,   Primero se incrementa el valor
de num2 y luego se asigna valor a var2
```

Clases, interfaces y anotaciones

Java utiliza dos elementos principales para implementar la orientación a objetos, clases e interfaces, además de un sistema de anotaciones de metadatos para facilitar la introducción de algunos comportamientos y funcionalidades.

Clases

Una clase define el comportamiento y el estado que pueden tener los objetos que son instanciados a partir de ella. El estado se define mediante atributos y el comportamiento mediante métodos. Ambos elementos son tipados. Los primeros marcan el tipo de dato que pueden almacenar y los segundos el que devuelven.

Además, estos elementos se acompañan de un modificador de visibilidad. Éste indica qué objetos pueden o no pueden acceder a estos atributos o métodos. La visibilidad puede ser:

- public: cualquier clase puede acceder.
- protected: solo clases descendientes de la clase o del mismo paquete pueden acceder.
- default: solo clases del mismo paquete pueden acceder.
- private: solo se puede acceder desde la propia clase.

Estos modificadores también se aplican a las propias clases. Cada clase pública debe estar en un fichero .java con el mismo nombre. Los atributos se marcan como private, siguiendo el principio de ocultación. Para acceder a ellos, se utilizan los métodos conocidos como getter/setter, o incluso otros, en función del acceso que queramos darles.

Los objetos se crean a través de un constructor. Éste es un tipo de método especial que se invoca mediante la palabra reservada new. No tiene nombre y no retorna ningún valor. Se encarga de recibir parámetros, en su caso, para inicializar el estado del objeto.

Cabe destacar que podemos utilizar la palabra reservada this para referirnos a un atributo o método del objeto. Esto puede ser útil para distinguirlo de parámetros o variables locales.

Vamos a ver un ejemplo con todo lo visto hasta el momento:

```
public class SpeedCalc {

  private double time, distance;

  // El constructor
  public SpeedCalc(double time, double distance) {
    this.time = time;
    this.distance = distance;
  }

  // Getter y setter
  public double getTime() {
    return time; // Como no hay conflicto, se puede omitir this.
  }

  public void setTime(double time) {
    this.time = time;
  }

  // ...

  // Un método cualquiera.
  public double getSpeed() {
    return distance / time;
  }

}
```

Para utilizar esta clase, haríamos lo siguiente:

```
SpeedCalc calc = new SpeedCalc(3.0, 60.0);
System.out.println(calc.getSpeed()); // Output: 20
calc.setTime(4.0);
System.out.println(calc.getSpeed()); // Output: 15
```

El modificador static indica que un atributo está vinculado a la clase en sí y no a sus instancias. Esto quiere decir que podemos acceder a ellos sin necesidad de instanciar objetos de dicha clase. La invocación se realiza utilizando la propia clase directamente. Un ejemplo es el método main, que sirve como punto de entrada para cualquier aplicación en Java:

```
public class App {
  public static void main(String[] args) {
    // ...
  }
}
```

El modificador final indica que una variable no puede ser modificada. Es la manera de conseguir que se comporten como constantes en Java. Cuando se aplica en métodos es para indicar que no pueden ser extendidos por sus clases descendientes.

Un uso típico es crear una constante global que pueda ser accedida desde cualquier parte de la aplicación. En estos casos, los atributos se marcan como públicos y estáticos, y se les suele dar un nombre en mayúsculas por convención, separando las palabras con guiones bajos:

```
public class Globals {
  public static final String APP_VERSION = "1.1.4";
}

// En cualquier parte de la aplicación.
System.out.println(Globals.APP_VERSION); // Output: 1.1.4
```

Herencia y clases abstractas

Podemos establecer una relación de herencia entre dos clases mediante la palabra reservada extends. Esto hará que la clase hija herede todos los métodos y atributos de la clase padre. Hay que tener en cuenta que sólo

podrá acceder a ellos si no están declarados como private.

Una clase hija puede definir cuantos atributos y métodos adicionales quiera, pero también puede modificar el comportamiento de los métodos de su padre. Para ello, se utiliza la anotación @Override, que indica que el método pretende reimplementar un método de su padre. Esto asegura que el compilador nos avisará en caso de que no lo estemos haciendo. Podemos acceder a la implementación de los métodos de la clase padre mediante la referencia super. Para constructores, no es necesario añadir ningún nombre de método.

Veamos estos conceptos con un ejemplo:

```
public class Publication {
  private String title;
  private int pages;

  public Publication(String title, int pages) {
    this.title = title;
    this.pages = pages;
  }

  // Omitidos getters y setters.

  public void read() {
    System.out.println("Leyendo la publicación... " + title);
  }

}

public class Magazine extends Publication {

  private int number;

  public Magazine(String title, int pages, int number) {
    super(title, pages);
    this.number = number;
  }
```

```
  // Getter y setter para number omitidos.

  @Override
  public void read() {
    super.read();
    System.out.println("Es el número " + number + " y tiene " +
getPages() + " páginas.");
  }

}
```

Existe también la posibilidad de dejar un método de una clase sin implementar. Para ello, se define la firma del método y se le añade el modificador abstract. Las clases con este tipo de métodos se denominan abstractas y también deben llevar este modificador. Una clase abstracta no puede instanciarse directamente. Sólo se pueden instanciar clases hijas que sí implementen el comportamiento.

```
public abstract class Animal {
  public abstract String getSound();
}

public class Cat extends Animal {
  public String getSound() {
    return "Miau";
  }
}

// En cualquier otra parte.
Animal animal = new Cat();
System.out.println(animal.getSound()); // Output: Miau
```

En este ejemplo, hemos visto algo interesante. Una variable de tipo Animal a la que le asignamos un valor de tipo Cat. Esto es posible gracias al concepto de abstracción. Al ser una clase hija, podemos considerar que Cat es un Animal.

Además, también incluimos el concepto de polimorfismo. Como todos los

animales tienen el método getSound(), podemos llamarlo sin preocuparnos de qué animal concreto es. El resultado dependerá de la clase hija concreta que hayamos instanciado. Podríamos tener una clase Dog que devolviera “guau” e intercambiarlas dinámicamente.

Interfaces

Las interfaces son un paso más en el proceso de abstracción. A diferencia de las clases, no implementan métodos ni atributos. Sólo declaran la firma de los métodos. Existe una excepción, los métodos marcados con la palabra reservada default. Estos proveen una implementación por defecto. Las interfaces son implementadas por clases y cada clase puede implementar un número indefinido de ellas.

La forma más simple de verlo es que con las interfaces definimos qué hay que hacer, pero no cómo hacerlo. Esto permite que clases muy diferentes entre sí puedan compartir comportamientos. Por ejemplo, un ave puede volar, pero un avión también. La diferencia es cómo lo hacen.

```
public interface Flying {
  void fly();
}

public class Bird implements Flying {
  public void fly() {
    System.out.println("Batir de alas");
  }
}

public class Plane implements Flying {
  public void fly() {
    System.out.println("Arrancar motores");
  }
}

// En cualquier otra parte de la aplicación.
Flying flying1 = new Bird();
```

```
flying1.fly(); // Output: Batir de alas
Flying flying2 = new Plane();
flying2.fly(); // Output: Arrancar motores
```

Como se puede apreciar, mediante el uso de interfaces se puede alcanzar un grado mayor de abstracción y polimorfismo, ya que podemos definir comportamientos iguales para objetos que no tienen nada que ver, con implementaciones muy distintas de los mismos.

Es recomendable definir atributos, parámetros y tipos de retorno de los métodos como interfaces siempre que podamos. Esto hará que nuestra aplicación sea más tolerante al cambio, pues no nos atamos a una implementación concreta.

Anotaciones

Las anotaciones son una forma de añadir metadatos a los elementos de nuestras aplicaciones. Estos metadatos pueden ser luego utilizados por el compilador, librerías o frameworks para tratar de una forma determinada esas piezas de nuestro código. No afectan de ninguna manera al código en sí.

Su introducción tiene que ver con la extensibilidad y mantenibilidad del código. El funcionamiento de las librerías y frameworks hasta entonces se basaba en la implementación o extensión de ciertas interfaces y clases, la firma de clases y métodos, y archivos XML poco legibles. Después de su aparición con Java 5, todo esto se simplificó con la posibilidad de anotar cada elemento en el propio código.

Las anotaciones van precedidas del símbolo @. Ya hemos utilizado la anotación @Override para sobreescribir el comportamiento de un método. Afectan al elemento que las sigue y pueden apilarse varias sobre un mismo elemento. Pueden aceptar o no parámetros, en cuyo caso, éstos se especifican entre paréntesis.

Las anotaciones y su uso, dependen fundamentalmente del entorno para el que desarrollemos nuestras aplicaciones. Pueden definirse anotaciones propias, pero eso es un aspecto que va más allá del objetivo de esta guía.

Java - Anotaciones

autentia

¿Qué son?

Las anotaciones son una característica de Java que nos permite asociar un elemento de nuestro código a una serie de metadatos. Estos metadatos son utilizados por el compilador o durante la ejecución del programa, donde otros frameworks y librerías se pueden aprovechar para generar código o realizar otras operaciones.

¿CÓMO DECLARAR UNA ANOTACIÓN?

Una anotación se declara con **@** seguido por su nombre. **Debe preceder el elemento que queremos anotar**. Por ejemplo, un caso sencillo es la anotación **@Override**, indicando que el elemento siguiente sobreescribirá un elemento de la superclase donde está definida.

Algunas anotaciones pueden tener elementos asignables. En este caso se definen entre paréntesis después del nombre de la anotación:

@Author(nombre="José", apellido="Palacios")

Otras características:

- Se pueden declarar varias anotaciones para un mismo elemento.
- Normalmente, una anotación ocupará su propia línea, pero es meramente, una convención de estilo.

```
@Override
@Author(nombre="Juan", apellido="Palacios")
private String title;
```

EJEMPLO

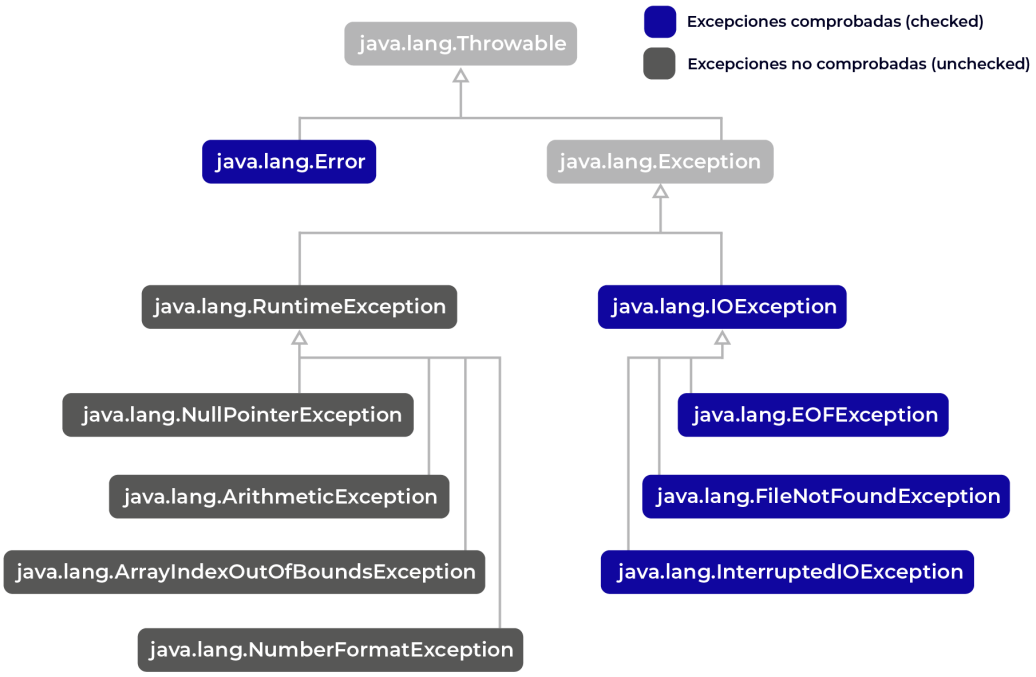
Lombok es una librería que ofrece anotaciones para generar código. Una de sus anotaciones, @Getter, se asocia a una variable dentro de una clase:

```
public class LombokExample {

    @Getter
    private int someField;

}
```

Al compilarla, se añadiría un método a la clase compilada:

```
public class LombokExample {

    private int someField;

    public int getSomeField() {
        return this.someField;
    }

}
```

Control de excepciones



Hay ocasiones en las que una determinada operación puede salir mal. Parámetros inválidos, recursos no disponibles, estados inconsistentes, etc. Nuestras aplicaciones deben ser robustas y tolerar estos fallos. No solo deben seguir funcionando, sino que debe asegurarse que el estado global queda consistente.

Para gestionar estas situaciones, Java nos proporciona el bloque de control try/catch. El programa trata de ejecutar las instrucciones incluidas en el bloque try, pero si se produce una excepción, pasa inmediatamente a un bloque catch que acepte ese tipo de excepción.

Las excepciones, como casi todo en Java, son objetos. Todas ellas descienden de la clase Exception. Pueden almacenar, además de la traza de llamadas que la provocó, un mensaje y alguna otra excepción asociada que provocó la actual.

Podemos crear nuestras propias excepciones extendiendo la clase Exception. Para lanzar una excepción, utilizamos la palabra reservada ***throw***. Si no la tratamos inmediatamente con un bloque try/catch, debemos indicar en la firma del método que puede lanzar ese tipo de excepción con la palabra reservada **throws**.

Veamos un ejemplo:

```
public class MyException extends Exception {

  public MyException(String message) {
    super(message);
  }

}

public class ExceptionThrower {

  public static void throwException() throws MyException {
    throw new MyException("¡Excepción!");
  }

  public static void main(String[] args) {
    try {
      throwException();
    } catch (MyException e) {
       System.err.println(e.getMessage());
       e.printStackTrace();
    } catch (Exception e) { //Si es cualquier otra excepción
       e.printStackTrace();
    }
  }
}
```

También podemos añadir un bloque **finally** después de los bloques catch. Este bloque se ejecutará siempre, vaya bien la ejecución del try o no. Se utiliza normalmente, para cerrar cualquier recurso que se haya abierto.

try-with-resources

Desde Java 7 existe la fórmula try-with-resources que permite vincular el cerrado de recursos a la conclusión del try, de modo que no se nos olvide hacerlo manualmente.

```
// Con finally
String line = null;
BufferedReader br = new BufferedReader(new FileReader("myfile"));
try {
  line = br.readLine();
} catch (Exception e) {
  e.printStackTrace();
} finally {
  if (br != null) br.close();
}

// Con try-with-resources
String line = null;
try (BufferedReader br = new BufferedReader(new
FileReader("myfile"))) {
  line = br.readLine();
} catch (Exception e) {
  e.printStackTrace();
}
```

Como se puede observar, definimos los recursos que deben ser cerrados automáticamente después del *try* y entre paréntesis. Podemos incluir varios recursos separándolos por punto y coma. Al escribirse de esta forma se llamará al método close del BufferedReader al acabar la ejecución del bloque, se produzcan errores o no.

Todos los recursos que se utilicen dentro de un *try-with-resources* deben implementar la interfaz *AutoCloseable*, la cual tiene un único método close que define cómo se debe cerrar el recurso.

Antes de Java 9, los recursos necesitaban inicializarse en el bloque try, pero

a partir de Java 9, pueden ser inicializados antes e incluirlos en el bloque después, siempre que las variables sean final o efectivamente final. La sintaxis es la siguiente:

```
public static void main(String args[]) throws IOException {
  final FileWriter fw = new FileWriter("C:\\file.txt");
  try (fw) {
    fw.write("Welcome to Autentia Onboarding");
  }
}
```

RuntimeException

Si llevas programando un tiempo en Java, te habrás dado cuenta de que, en ocasiones, tu código ha generado excepciones que el compilador no te ha obligado a envolver en un bloque try/catch o en un método con throws. Esto puede ocurrir cuando invocamos un método sobre una referencia a objeto null, cuando accedemos a un índice de un array que excede sus dimensiones, etc.

Todas estas excepciones extienden **RuntimeException**. Se trata de excepciones por problemas que se producen en tiempo de ejecución. Estas excepciones no se controlan ya que tienen un carácter imprevisible, provocado habitualmente por errores de programación. Ten cuidado cuando crees tus propias RuntimeException ya que el compilador no actuará como recordatorio de que deben ser controladas.

APIs básicas del lenguaje

En este apartado vamos a ver algunas APIs básicas que Java nos ofrece

para tratar ciertas situaciones recurrentes.

Object

Object es la clase de la que heredan todas las clases en Java en última instancia. Declara algunos métodos útiles que pueden ser invocados desde cualquier clase:

- getClass() devuelve la clase a la que pertenece el objeto.
- equals() comprueba si dos objetos son iguales. Sobreescribirlo permite que cada clase defina su propio concepto de igualdad.
- hashCode() obtiene un código hash para un objeto. Este código se calcula a partir de algunos datos del objeto. Dos objetos iguales deben tener el mismo hashCode, pero dos objetos con el mismo hashCode no tienen por qué ser iguales. Se utiliza generalmente para colecciones clave-valor que utilizan el hashCode de la clave para ubicar el objeto. Como puede haber varios objetos con el mismo hashCode, también es necesario comprobar la igualdad.
- clone() permite obtener una copia del objeto.

Además, contamos con la clase Objects, que incorpora otros métodos útiles. Algunos de estos métodos se superponen con los de la clase Object, pero permiten lidiar de una forma más cómoda con valores null.

Arrays

Los arrays son la forma más básica de agrupar valores y objetos. Tienen una longitud fija y pueden ser de una o varias dimensiones. Cuando trabajamos con arrays multidimensionales, simplemente anidamos unos arrays dentro de otros.

Los arrays en Java son tipados. Esto quiere decir que sólo pueden almacenar un tipo de valor u objeto. Para declararlos, se añade [] al tipo o nombre de la variable. Para crearlo, podemos utilizar la palabra reservada

new con el tamaño del array o utilizar un array de literales por defecto.

```
String[] array1 = new String[5]; // Array vacío de String con 5 posiciones.
int[] primos = {2, 3, 5, 7, 11} // Array de int con 5 posiciones.
```

Para acceder a una posición, utilizamos la sintaxis variable[posicion]. Por ejemplo:

```
System.out.println(primos[0]); // 2
array1 [0] = "Hola mundo";
```

Además, disponemos de la clase Arrays, que contiene métodos estáticos útiles para manipular arrays. Podemos ordenarlos, hacer búsquedas, etc..

Clases envoltorio

El paquete java.lang trae consigo algunas clases envoltorio que nos ofrecen métodos para trabajar con tipos primitivos. Su nombre es el mismo que el de estos tipos, pero con la inicial en mayúscula, como normalmente ocurre con las clases; excepto en el caso de int, cuya clase correspondiente es Integer.

Algunas de las funcionalidades que nos ofrecen estas clases son:

- Conversión del tipo a String y viceversa.
- Conversión de unos tipos numéricos a otros.
- Valores máximos y mínimos para los tipos numéricos.
- Operaciones de bit. Por ejemplo:

```
Integer a = 5;
Integer b = 4;
Integer r = a + b;
String str = r.toString();
Long l = Long.parseLong(str);
```

Además, podemos utilizar los objetos de estas clases envoltorio como si fueran tipos primitivos y viceversa. Esto se conoce como boxing/unboxing. Es necesario tener en cuenta este comportamiento en términos de rendimiento, ya que el compilador crea una nueva variable del tipo envoltorio cuando realiza el boxing por nosotros. Es especialmente importante cuando se utiliza en bucle.

```
Integer a = 5;
// Integer a = Integer.valueOf(5);
Integer b = 4;
// Integer b = Integer.valueOf(4);

Integer r = a + b;
// Integer r = Integer.valueOf(a.intValue() + b.intValue());
```

String

En Java, String es una clase, no un tipo primitivo. Aun así, podemos crear nuevas instancias de forma sencilla con literales entre comillas dobles. Los objetos de esta clase son inmutables. Todas las operaciones que modifican la cadena original devuelven un nuevo objeto sin alterar el primero.

Para concatenar dos cadenas, podemos usar el método concat() o el operador +. También podemos concatenar un tipo primitivo o un objeto, en cuyo caso se utilizará de forma transparente el método toString() que implemente cada clase.

Hay que destacar que cuando concatenamos cadenas, estamos reservando memoria para la nueva cadena como consecuencia de ser inmutables. Si hacemos esto repetidamente, como en un bucle, puede repercutir en el rendimiento de la aplicación. Para evitarlo, podemos recurrir a la clase StringBuilder, que nos ofrece una implementación mutable para este propósito. Si necesitamos que sea thread safe, usaremos entonces

StringBuffer.

```
String s1 = "Hola" + " mundo" + "!";
String s2 = new StringBuilder("Hola").append("
mundo").append("!").toString();
```

Java utiliza una codificación Unicode de 16 bits para representar los caracteres. Esto no es suficiente para representar todos los posibles caracteres que se pueden encontrar en el estándar con un solo code point. Para solventar el problema sin romper la compatibilidad con aplicaciones ya en uso, Java introdujo el concepto de caracteres suplementarios, que se codifican mediante dos code points, en lugar de uno.

Fechas

La forma clásica de trabajar con fechas en Java es con las clases Date y Calendar. La primera representa un punto cronológico en el tiempo, expresado en milisegundos. La segunda nos ofrece una interfaz más rica con la que poder trabajar con fechas.

Un ejemplo de uso sería el siguiente:

```
Calendar calendar = Calendar.getInstance(); // Instancia con el
tiempo local
calendar.set(Calendar.YEAR, 1990);
calendar.set(Calendar.MONTH, 3);
calendar.set(Calendar.DATE, 10);
calendar.set(Calendar.HOUR_OF_DAY, 15);
calendar.set(Calendar.MINUTE, 32);

Date date = calendar.getTime(); // Convertimos de Calendar a Date.
```

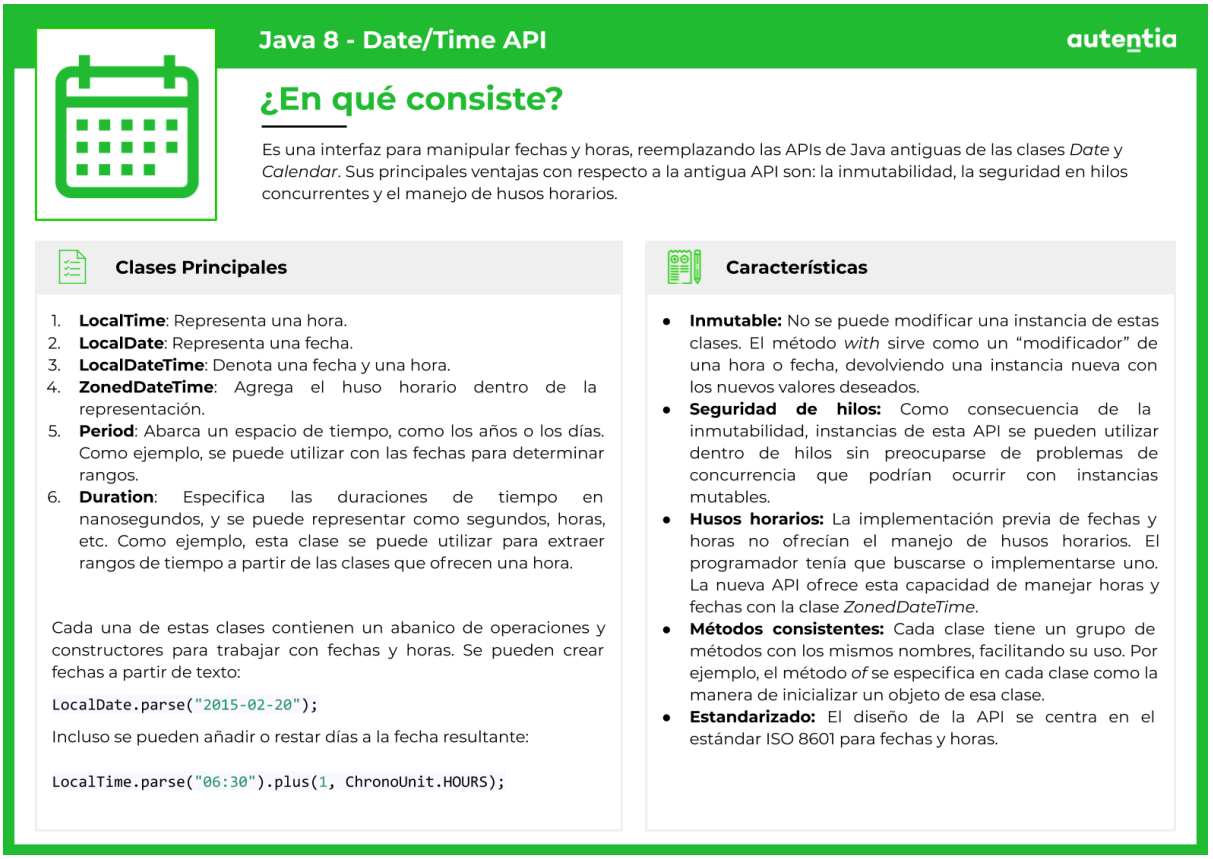
Sin embargo, esta API tiene algunos problemas. No tiene un diseño demasiado bueno, puesto que no pueden utilizarse fechas y horas por separado, y no es thread safe, lo que puede ocasionar problemas de

concurrencia. Por eso, Java 8 introdujo una nueva API que ofrecía fechas inmutables, adaptadas a diferentes calendarios y con un diseño mejorado que nos ofrece métodos factoría. Podemos encontrar esta API en el paquete java.time.

Las clases base son LocalTime, LocalDate y LocalDateTime.

```
LocalDateTime timepoint = LocalDateTime.now(); // Fecha y hora
actual
LocalDate date = LocalDate.of(2020, Month.JULY, 27); // Obtenemos
la fecha indicada
LocalTime.of(17, 30); // Obtenemos la hora indicada
LocalTime.parse("17:30:00"); // Otra forma para la hora

// Usamos la convención estándar para get.
Month month = timepoint.getMonth();
int day = timepoint.getDayOfMonth();

// Son inmutables, así que cambiar el valor retorna un objeto
LocalDateTime happyTwenties = timepoint.withYear(1920)
.withMonth(Month.January)
.withDayOfMonth(1)
.plusWeeks(3);
```

El paquete también ofrece otras clases adicionales, como Period o Duration, que sirven para representar lapsos de tiempo, algo que no estaba soportado por Date y Calendar. Además, existen otras herramientas y conceptos más avanzados que puedes investigar si estás interesado.



Formateado de texto

Si tratamos de sacar por pantalla el valor de números decimales o fechas, podemos encontrarnos con que el resultado no es demasiado legible ni atractivo. Java nos ofrece algunas clases para trabajar con el formato del texto en el paquete java.text. Veamos un ejemplo a continuación:

```
Date date = new Date(); // Actual
SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy");
String date = df.format(date);
```

Hay que tener cuidado, pues estos formateadores pueden no ser thread safe y pueden ser fuente de error en entornos productivos.

Concurrencia

La concurrencia es la **capacidad de ejecutar varias partes de un programa en paralelo,** aunque no necesariamente tienen que estar ejecutándose en el mismo instante. Una aplicación Java se ejecuta por defecto en un proceso, que puede trabajar con varios hilos para lograr un comportamiento asincrónico. Pero, ¿qué es un proceso? Un proceso corresponde con un programa a nivel de sistema operativo. Normalmente, suele tener un espacio aislado de ejecución con un conjunto de memoria reservada exclusivamente para él. Además, comparte recursos con otros procesos como el disco, la pantalla, la red, etc., y todo esto lo gestiona el propio S.O. Dentro de los procesos podemos encontrar hilos (threads). Un hilo corresponde con una unidad de ejecución más pequeña, compartiendo el proceso de ejecución y los datos del mismo.

Un concepto importante a conocer es el de **Condición de Carrera** *y surge cuando dos procesos 'compiten' por llegar antes a un recurso específico*. Cuando un proceso es dependiente de una serie de eventos que no siguen un patrón al ejecutarse y trabajan sobre un recurso compartido, se podría producir un error si los eventos no llegan en el orden esperado. Pero si se realiza una buena sincronización de los eventos que estén en condición de carrera, no se producirán problemas de inconsistencia en el futuro.

La JVM permite que una aplicación tenga múltiples hilos ejecutándose simultáneamente. Existen dos formas de crear Hilos en java:

1. **Extendiendo de la Clase *Thread*** (que realmente implementa la interfaz Runnable) y sobreescribiendo el método *run()*. Después de instanciar nuestro hilo, podemos ejecutarlo con el método *start()*.

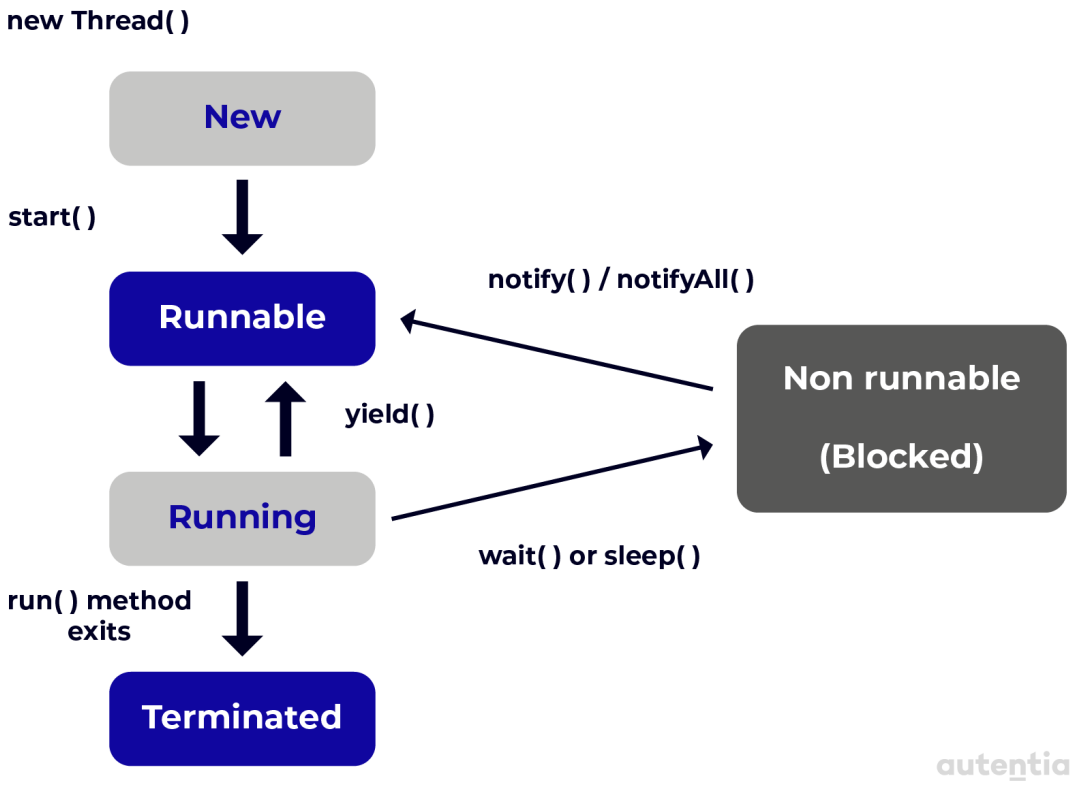
```
public class MyThread extends Thread {
  @Override
  public void run() {
     ...
  }
}

...
Thread thread = new MyThread();
thread.start();
...
```

2. **Implementado la Interfaz *Runnable*** y pasando por parámetro dicha instancia a la clase *Thread*. Si nuestra clase no extiende de la clase Thread, nuestra instancia de clase no se tratará como thread. Por este motivo, se crea explícitamente una instancia de la clase Thread en el siguiente ejemplo.

```
public class MyRunnable implements Runnable {
  @Override
  public void run() {
     ...
  }
}

...
Thread thread = new Thread(new MyRunnable());
thread.start();
...
```

Estados de un Hilo

El **thread scheduler** es un componente de la JVM que decide qué hilo debe ser seleccionado para ejecutarse y por lo tanto, se consideraría que está en un estado de *Running* (no es lo mismo que *Runnable*).



- **New**: cuando se crea una nueva instancia de la clase de Thread pero sin llegar a invocar el método *start()*.
- **Runnable:** tras invocar el método *start()*, un thread es considerado *Runnable* aunque podría pasar o no a estado Running si es el seleccionado por el thread scheduler.
- **Running**: indica el hilo actual que se está ejecutando.
- **Non runnable (blocked):** tras invocar el método *sleep()* o *wait()*. También se puede pasar a este estado en caso de que haya algún bloqueo por una operación de entrada/salida o cuando se está esperando a que otro hilo termine a través del método join().
- **Terminated/dead**: cuando el método *run()* finaliza.

A parte de los métodos vistos, podemos encontrar otros para manipular el estado de un hilo como *yield()* que permite pausar el hilo actual en ejecución, dando la posibilidad de que otros hilos puedan ejecutarse o

getState() que permite conocer el estado actual de un hilo, además de muchos otros que podemos encontrar en la documentación.

Prioridades en los Hilos

Cada hilo tiene una prioridad que se indica con un rango de números entre el 1 y el 10. Al crear un hilo nuevo, podemos usar el método *setPriority(int)* que recibe por parámetro un entero con la prioridad deseada. También podemos usar valores por defecto MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY (los valores asignados son 1, 5, 10).

```
...
Thread thread = new MyThread();
thread.setPriority(Thread.MIN_PRIORITY);
thread.start();
...
```

Sincronización de hilos

Cuando dos hilos acceden a un mismo recurso, debemos gestionar su acceso para evitar colisiones entre ellos (a esto se le conoce como **Thread safety).** Una sección de código en la que se actualizan datos comunes a varios hilos se le conoce como *sección crítica.* Cuando se identifica una sección crítica, se ha de proveer un mecanismo para conseguir que el acceso sea exclusivo de un solo hilo. A esto se le conoce como *exclusión mutua*. En java, las secciones críticas se marcan con ***synchronized***.

A un objeto de una clase con métodos synchronized se le conoce como ***monitor.*** Cuando un hilo accede al interior de un método *synchronized* se dice que el hilo ha adquirido el monitor. En ese momento, ningún hilo podrá acceder a ninguno de los métodos *synchronized* hasta que el hilo libere el monitor. A través del método *wait()*, podemos indicar a un hilo que espere para ocupar el monitor y con *notify()/notifyAll()*, indicamos al hilo que ya

puede acceder al recurso.

En el siguiente ejemplo se observa cómo se pone en pausa el hilo en caso de que el índice sea menor a 0. Cuando el valor del índice avance y se asigne un valor a esa posición del array, lo notificaremos mediante el método *notifyAll()* y desbloquearemos el hilo correspondiente.

```
public class Stack {

  private int index = -1;
  private int [] store = new int [100];

  public synchronized void push(int newValue) {
    index++;
    store[index]=newValue;
    notifyAll(); //Será necesario gestionar excepciones.
  }

  public synchronized int pop() {
    if(index<0) {
      wait();  //Será necesario gestionar excepciones.
    } else {
      index--;
    }
    return store[index];
  }

}
```

Existen también métodos de sincronización específicos para ciertas situaciones. Java nos proporciona algunas clases que los implementan:

- Semaphore: ofrece un número limitado de permisos que los hilos deben adquirir al entrar en la región crítica y liberarlos al salir. Cuando no hay permisos disponibles, esperan.
- CountDownLatch: obliga a esperar a que un número de eventos tengan lugar antes de permitir el paso de los hilos.

- CyclicBarrier: permite establecer un punto al que todos los hilos deben llegar antes de proseguir la ejecución.
- Phaser: otro tipo de barrera más flexible que las anteriores.
- Exchanger: permite crear un punto de sincronización en el que un par de hilos pueden intercambiar elementos.
- Tipos atómicos: son clases especiales para tipos básicos que realizan ciertas operaciones de forma atómica, de modo que un hilo no puede modificar su valor mientras otro está realizando una operación.

Pools de hilos

Un pool de hilos no es más que una reserva de hilos que están instanciados, a la espera de ejecutar alguna rutina. Esto nos permite, por una parte, agilizar el lanzamiento de tareas concurrentes y, por otra, limitar el número máximo de hilos activos, de forma que no se dispare el consumo de recursos de la aplicación.

Estos hilos son totalmente agnósticos. Se limitan a ejecutar la tarea para la que son requeridos. Una vez terminada, se liberan y vuelven a estar disponibles en el pool. Si en un momento dado no hay hilos disponibles en el pool, la tarea se encola a la espera de recibir uno.

Los pools implementan la interfaz ExecutorService y pueden configurarse fácilmente gracias a los métodos factoría que proporciona la clase Executors. A continuación se muestra un ejemplo:

```
ExecutorService pool = Executors.newFixedThreadPool(10);
/* Creamos un pool con 10 hilos.
* 10 de las tareas son atendidas enseguida.
Las otras 10 se atenderán según se liberen los hilos del pool. */
for (int i = 0; i<20; i++) {
  pool.execute(new MyRunnableClass());
}
```

```
pool.shutdown();
```

Es importante apagar el pool con el método shutdown cuando ya no lo necesitemos. El pool no aceptará más tareas pero terminará con normalidad aquellas que están en ejecución o en cola. ExecutorService ofrece los métodos execute, que permite ejecutar una instancia de la interfaz Runnable; y submit, que permite ejecutar una instancia de la interfaz Callable y devuelve un Future. Este segundo método puede utilizarse cuando esperamos obtener un resultado de la ejecución. Por ejemplo:

```
public class App {
  public static void main(String[] args) throws Exception {
    int n = 1000;
    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new CallableSum(n));
    pool.shutdown();
    System.out.println("El sumatorio de " + n + " es " + result.get());
  }

  public static class CallableSum implements Callable<Integer> {

    private int sumTo;

    CallableSum(int sumTo) {
      this.sumTo = sumTo;
    }

    public Integer call() {
      int sum = 0;
      for (int i = 0; i <= sumTo; i++) {
        sum += i;
      }
      return Integer.valueOf(sum);
    }
  }
}
```

ThreadPoolExecutor puede rechazar la ejecución de una tarea nueva. Esto puede ocurrir porque el pool está apagado (mediante el método shutdown)

o porque el tamaño de la cola y de hilos máximos se ha excedido. En este caso, el resultado dependerá de la política que utilice el pool. Podemos seleccionarla mediante el método setRejectedExecutionHandler. Las políticas que tenemos a disposición son:

- ThreadPoolExecutor.AbortPolicy: la tarea se aborta y se lanza una excepción.Es la política por defecto.
- ThreadPoolExecutor.CallerRunsPolicy: el hilo que ha realizado la llamada se encarga de la ejecución.
- ThreadPoolExecutor.DiscardPolicy: la tarea se descarta.
- ThreadPoolExecutor.DiscardOldestPolicy: la tarea en la cabeza de la cola se descarta y se intenta volver a ejecutar la nueva tarea.

ThreadLocal

Esta clase nos permite crear variables confinadas al ámbito de memoria de cada hilo. Normalmente, se suele utilizar como atributo estático de una clase, desde donde podremos recuperar una instancia diferente dependiendo del hilo que la consulte. Podemos almacenar en ella datos importantes que necesitemos en varios puntos de la ejecución del hilo, sin necesidad de pasarlos como parámetros continuamente. ID de usuario, de producto o similares son candidatos típicos.

Se puede inicializar ThreadLocal con el método withInitial, que acepta una interfaz funcional. Un ejemplo de uso sería el siguiente:

```
public class MyGlobals {
  private static final AtomicInteger nextId = new
AtomicInteger(0);
  public static final ThreadLocal<Integer> threadId =
ThreadLocal.withInitial(() -> nextId.getAndIncrement());
}

// ...
```

```
// El ID será distinto en función del hilo.
MyGlobals.threadId.get();
// Y sólo se modificará en el hilo en curso.
MyGlobals.threadId.set(77);
```

También tenemos la clase InheritableThreadLocal. Esta permite que las variables locales de un hilo se compartan con sus hilos hijos. Esto puede ser útil si necesitamos realizar alguna tarea de forma asíncrona que requiera alguno de los datos que hemos almacenado en el hilo padre.

El funcionamiento es simple. Aquellas variables de tipo InheritableThreadLocal que tengan un valor para el hilo padre son inicializadas con el mismo valor para el hilo hijo. Esto no impide que el hijo pueda modificar su valor en cualquier momento.

```
public class MyGlobals {
  public static final ThreadLocal<Integer> userId =
    new InheritableThreadLocal<Integer>();
}
// Seleccionamos el valor desde el padre.
MyGlobals.userId.set(20);
// ...
// Consultamos el valor desde un hilo hijo.
MyGlobals.userId.get(); // 20
```

Todas las variables declaradas a un hilo permanecen mientras el hilo esté vivo y se mantenga una referencia al ThreadLocal. Cuando el hilo termina, todas las copias locales desaparecen.

Sin embargo, puede ser interesante limpiar el valor de una variable después de concluir la tarea para la que es necesaria. Esto es debido a que los hilos pueden ser reutilizados, como en los pools. Para ello, podemos invocar el método remove() de ThreadLocal.

Recomendaciones sobre concurrencia

Si vas a utilizar concurrencia en tu aplicación, es mejor que te tomes tu tiempo para entender cómo funciona cada pieza exactamente. Un mal uso de la concurrencia puede desembocar en datos corruptos y uso excesivo de recursos. Aquí van unos consejos:

- Utiliza las colecciones concurrentes. Son mucho más eficientes y te ahorrarás problemas.
- Elimina los datos de ThreadLocal con el método remove() al finalizar la tarea.
- Apaga los pools de hilos con el método shutdown cuando ya no los necesites.
- Utiliza ThreadPoolExecutor.CallerRunsPolicy como política de rechazo para tus pools. Así te asegurarás de que no quedan tareas sin hacer.
- En caso de que estés desarrollando una aplicación web, la necesidad de trabajar con concurrencia debe estar muy bien justificada. Ten en cuenta que los servidores ya utilizan sus propios pools de hilos para conexiones http, de base de datos, etc. Además, el acceso concurrente a los datos suele manejarse mediante transacciones. Una aplicación no tiene por qué estar alojada en un solo servidor, de forma que todo esto se complica un poco más. Normalmente, la tecnología que utilices te proveerá de herramientas específicas mejores que la gestión manual de la concurrencia.

Por último, animar a quien esté interesado en ahondar en la materia. La concurrencia es un campo vasto de conocimiento, con multitud de enfoques y detalles. Aquí sólo hemos expuesto lo básico, pero Java trae consigo muchas más herramientas que pueden ayudarte a conseguir exactamente lo que necesitas.

Generics

El término “Generic” viene a ser como un **tipo parametrizado**, es un tipo de dato especial del lenguaje que permite centrarnos en el algoritmo sin importar el tipo de dato específico que finalmente se utilice en él. Muchos algoritmos son los mismos, independientemente del tipo de dato que maneje. Por ejemplo, un algoritmo de ordenación, como puede ser “la burbuja”, es el mismo, independientemente de si estamos ordenando tipos como: String, Integer, Object, etc. La mayoría de los lenguajes de programación los integran y muchas de las implementaciones que nos ofrecen los usan. Mapas, listas, conjuntos o colas son algunas de las implementaciones que usan genéricos.

Se llaman parametrizados porque el tipo de dato con el que opera la funcionalidad se pasa como parámetro. Pueden usarse en clases, interfaces y métodos, denominándose clases, interfaces o métodos genéricos respectivamente. En Java, la declaración de un tipo genérico se hace entre símbolos <>, pudiendo definir uno o más parámetros, por ejemplo: <T>, <K, V>. Existe una convención a la hora de nombrarlos:

- E – Element (usado bastante por Java Collections Framework).
- K – Key (usado en mapas).
- N – Number (para números).
- T – Type (representa un tipo, es decir, una clase).
- V – Value (representa el valor, también se usa en mapas).
- S, U, V etc. – usado para representar otros tipos.

A continuación, se muestran dos ejemplos de declaración, el primero define una clase genérica y el segundo, un método genérico.

```
//Definición de una clase genérica que usa dos genéricos
// GenericClass.java file
public class GenericClass<T, K> {
```

```
  private T g1;
  private K g2;

  public GenericClass(T g1, K g2) {
    this.g1 = g1;
    this.g2 = g2;
  }

  public T getGenericOne() {
    return g1;
  }

  public K getGenericTwo() {
    return g2;
  }

}
// Main.java file
public class Main{
  public static void main(String[] args) throws Exception {
    GenericClass<Integer, String> clazz =
      new GenericClass<>(1, "generic");

    Integer param1 = clazz.getGenericOne();
    String param2 = clazz.getGenericTwo();

    System.out.println(String.format("Param1 %d - Param2 %s",
param1, param2));
  }
}
```

```
// WhiteBoard.java file
public class WhiteBoard {

  //Definición de un método genérico
  public <T> void draw(T figure ) {
    ...
  }
}
```

```
// Main.java file
public static void main(String[] args) throws Exception {
  WhiteBoard board = new WhiteBoard();

  Figure circle = new Circle(1.5);

  board.draw(circle);
}
```

Al trabajar con genéricos hay que tener en cuenta ciertas consideraciones: el parámetro tipo no puede ser un tipo primitivo, ya que los genéricos sólo trabajan con tipos de referencia. Tampoco se pueden usar en la implementación del genérico los métodos de la clase/interfaz del tipo que se defina en la instanciación, a menos que se indique explícitamente. Los tipos parametrizados pueden definir límites o especializaciones que permiten trabajar con determinados tipos en la definición del genérico:

```
public static <T extends Comparable<T>> int compare(T t1, T t2){
  return t1.compareTo(t2);
}
```

En el ejemplo anterior, estamos indicando que el tipo T debe ser un subtipo de Comparable, esto permite que dentro del método se puedan usar los métodos definidos en la interfaz Comparable.

Los tipos parametrizados también permiten el uso de comodín “?” para definir que un tipo parametrizado es desconocido. Los comodines se pueden usar como tipo de un parámetro, atributo o variable local y algunas veces como tipo de salida. En las declaraciones donde se utilizan tipos parametrizados con comodín, se pueden usar las palabras clave “super” o “extends”. El uso de uno u otro difiere en función de si la implementación que usa el genérico lo consume o produce. Debemos seguir la regla mnemotécnica “PECS” (Producers Extends, Consumers Super), es decir,

“super” se utiliza para cuando se consume el tipo parametrizado y “extends” cuando se produce.

```
List<? extends Vehicle> garage = new ArrayList<>();
garage.add(new Vehicle());  // error de compilación
garage.add(new Car());      // error de compilación
garage.add(new Bus());      // error de compilación

Vehicle vehicle = garageB.get(1);
```

Cuando se usa “? extends”, el compilador de Java sabe que esta lista podría contener cualquier subtipo de clase Vehicle pero no sabe qué tipo, podemos tener Bike, Car, Bus, etc. El compilador no permitirá al desarrollador insertar cualquier tipo de elemento en la lista, preservando la seguridad de tipos. En cambio, cuando se recupera un elemento de la lista, se garantiza que cualquier elemento recuperado es una subclase de Vehicle. De ahí que se diga que el uso de “extends” como comodín es usado para productores.

```
List<? super Car> garage = new ArrayList<>();
garage.add(new BMW());
garage.add(new Alto());
garage.add(new Vehicle());    // error de compilación

Object object = garage.get(0);    // No retorna un Car.
```

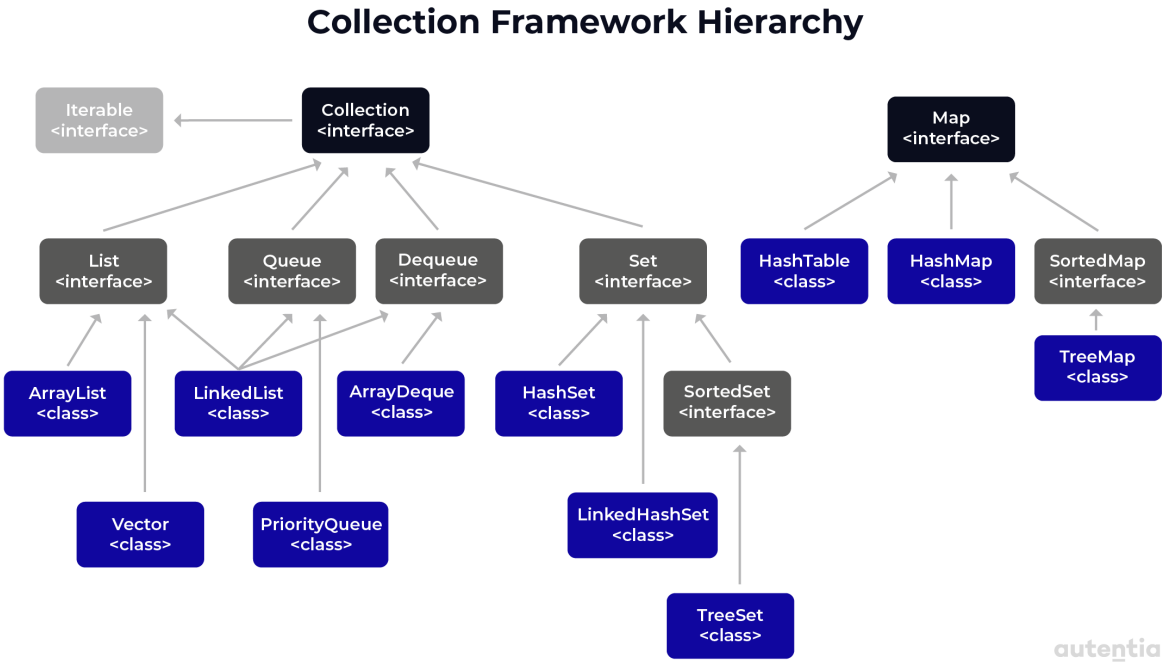
En cambio cuando se usa “? super” nos encontramos con el caso contrario, el compilador de Java sabe que los elementos almacenados son los tipos superiores de la clase Car, pero no sabe qué supertipo está almacenando en la lista ya que podría ser Vehicle u Object, de ahí que, cuando intentamos recuperar un valor de la lista, este retorne un Object. Sin embargo, cualquier clase hija de Car podrá ser insertada en la lista pero no sus clases padre, como puede ser Vehicle. Por tanto, el uso de “super” con

comodín se circunscribe a consumidores.

El uso de comodines con genéricos es recomendable cuando estemos desarrollando frameworks o librerías que son usadas por terceros y donde queremos indicar explícitamente el uso del genérico dentro de la implementación.

Colecciones

Una colección es un objeto que agrupa múltiples elementos bajo una sola entidad. A diferencia de los arrays, las colecciones no tienen un tamaño fijo y se crean y manipulan exactamente igual que cualquier otro objeto. En Java, se conoce como **Collection Framework Hierarchy** a la arquitectura que representa y manipula las colecciones**.** Se observa en la imagen inferior como se emplea la interfaz genérica Collection y la interfaz Map para este propósito. Podemos almacenar cualquier tipo de objeto y usar una serie de métodos comunes como: añadir, eliminar, obtener el tamaño de la colección, etc. Partiendo de la interfaz genérica, extienden otra serie de subinterfaces que aportan funcionalidades más concretas sobre la interfaz anterior y se adaptan a distintas necesidades.



Algunas de las interfaces e implementaciones más comunes son:

- **List:** admite elementos repetidos y mantiene un orden inicial.
    - **ArrayList:** array redimensionable que aumenta su tamaño según crece la colección de elementos.
    - **LinkedList:** se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

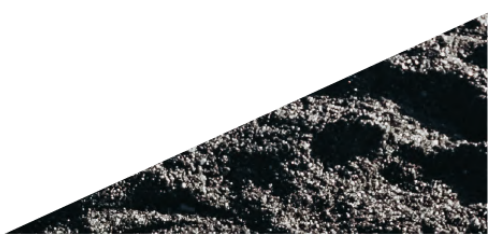
¿Uso ArrayList o LinkedList? Si necesitamos manipular los datos constantemente (insertar o eliminar), LinkedList nos ofrece un mejor rendimiento (O(1) vs. O(n)). En caso de necesitar hacer más operaciones de búsqueda (*get()*) y no tanto de inserción o eliminación, ArrayList ofrece un mejor rendimiento (O(1) vs. O(n)).

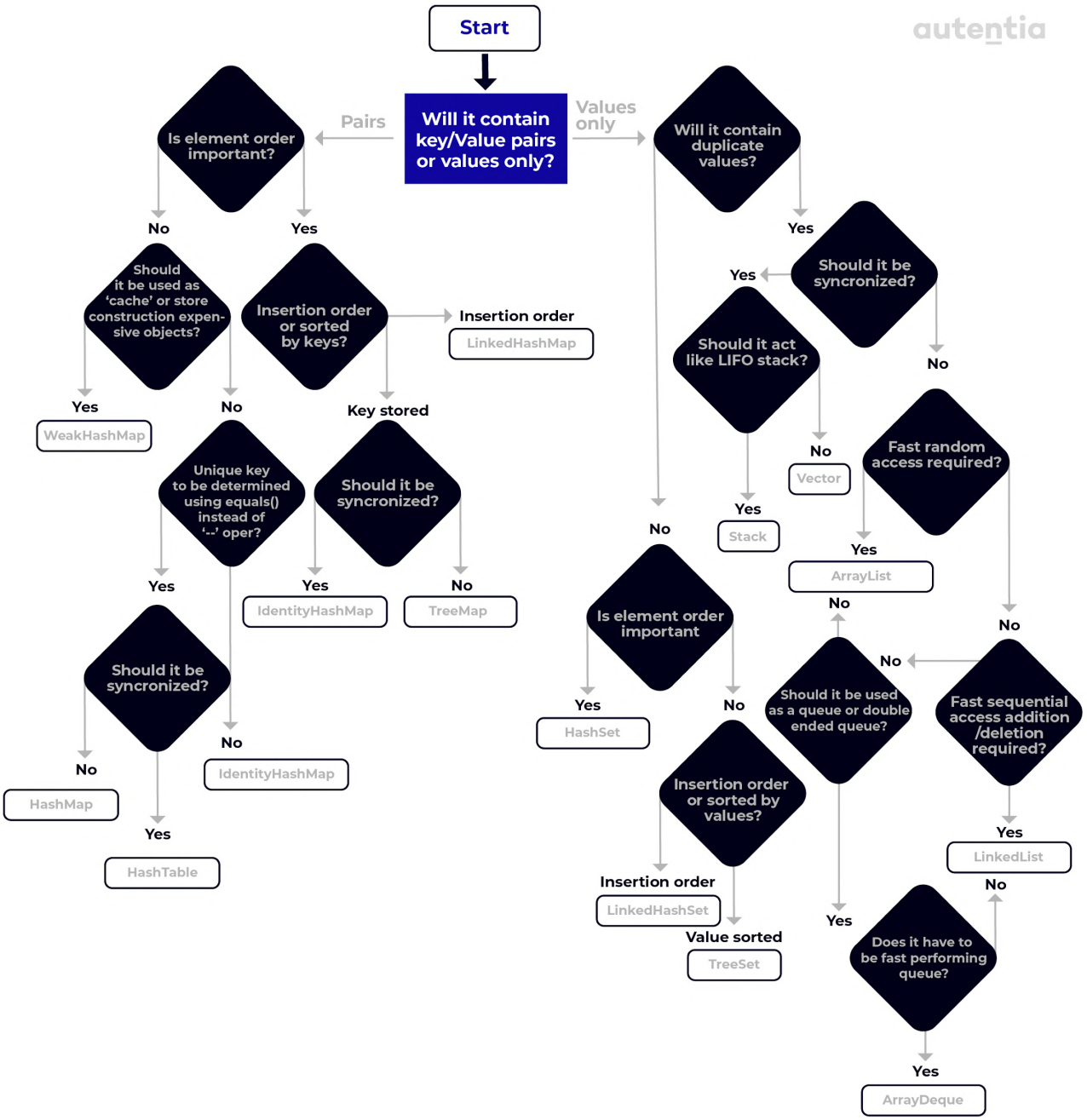
- **Set:** colección que no admite elementos repetidos. Es importante destacar que, para comprobar si los elementos están duplicados o no, es necesario tener implementados de forma correcta los métodos *equals()* y *hashCode()*.
    - **HashSet:** almacena los elementos en una tabla hash y no

garantiza ningún orden a la hora de realizar iteraciones. Es la implementación más usada debido a su rendimiento y a que, generalmente, no nos importa el orden que ocupen los elementos.

- **TreeSet:** almacena los elementos ordenándolos en función del criterio establecido por lo que es más lento que HashSet. Los elementos almacenados deben implementar la interfaz Comparable. Esto produce un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- **LinkedHashSet:** igual que Hashset, pero esta vez almacena los elementos en función del orden de inserción. Es un poco más costosa que HashSet.

- **Map:** conjunto de pares clave/valor, sin repetición de claves.
  - **HashMap:** almacena las claves en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones.
  - **TreeMap:** almacena las claves ordenándolas en función del criterio establecido, por lo que es más lento que HashMap. Las claves almacenadas deben implementar la interfaz Comparable. Esto produce un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
  - **LinkedHashMap:** Igual que Hashmap pero almacena las claves en función del orden de inserción. Es un poco más costosa que HashMap.





La interfaz Iterable ofrece un método con el que podemos obtener un objeto Iterator para una colección. Este objeto permite iterar por la colección, accediendo sucesivamente a cada uno de sus elementos. En el caso de las listas, existe la interfaz ListIterator que nos permite iterar también hacia atrás. Por ejemplo:

```
List<Integer> myMarks = new ArrayList();
myMarks.add(7);
```

```
myMarks.add(8);
myMarks.add(9);
Iterator it = myMarks.iterator();
Integer n1 = it.next(); // n1 = 7;
while (it.hasNext()) {
  System.out.println(it.next()); // Output: 8, 9.
}
```

Como vemos en el ejemplo, el iterador recuerda la posición en la que se quedó por última vez. No hay ningún impedimento para mantener referencias a varios iteradores diferentes sobre una misma colección. Como vimos anteriormente, la estructura for each se basa en el uso de iteradores.

Concurrencia y colecciones

Las colecciones de Java son mutables. Esto hace que trabajar con ellas cuando varios hilos tienen acceso pueda producir problemas. Una manera de lidiar con esto es envolverlas de forma que todos sus métodos sean synchronized. La clase Collections nos ofrece métodos para llevar esto a cabo según el tipo de colección.

Además, si se quiere utilizar un iterador sobre la colección, se debe sincronizar su uso sobre la colección devuelta. De lo contrario, se obtendrían resultados impredecibles. Por ejemplo:

```
List list = Collections.synchronizedList(new ArrayList());

synchronized(list) {
  Iterator it = list.iterator();
  while (it.hasNext()) {
    // Hacer algo con it.next()
  }
}
```

Sin embargo, esta aproximación supone un problema de rendimiento. Sólo un hilo podrá acceder a la vez a la colección. Para subsanar este

inconveniente, Java ofrece interfaces y clases específicas para colecciones concurrentes que puedes encontrar en el paquete java.util.concurrent. Estas estructuras de datos son mucho más eficientes ya que han sido pensadas para esta casuística y procuran crear los menores bloqueos posibles.

Lambdas

Las lambdas fueron introducidas a partir de Java 8. No son más que funciones anónimas que nos permiten programar en Java con un estilo más funcional y, en ocasiones, declarativo.

Sintaxis

La sintaxis de una lambda es la siguiente:

```
( tipo1 param1, tipoN paramN) -> { cuerpo de la lambda }
```

El operador flecha `->` es característico de las lambda y separa los parámetros del cuerpo de la función.

No es necesario incluir el tipo ya que este puede ser inferido. El paréntesis de los parámetros puede omitirse cuando sólo existe un parámetro y no incluimos el tipo. Si no hay parámetros los paréntesis son necesarios.

```
(param1, param2) -> { cuerpo }
```

```
param1 -> { cuerpo }
```

```
() -> { cuerpo }
```

En el caso del cuerpo, si solo tenemos una sentencia, podremos omitir las llaves y el return, por ejemplo:

```
numero -> String.valueOf(numero)
```

Si tenemos más de una, las llaves serán necesarias:

```
numero -> {
  String cadena = String.valueOf(numero);
  return cadena;
}
```

Interfaces funcionales

En Java, se considera interfaz funcional a toda interfaz que contenga un único método abstracto. Es decir, interfaces que tienen métodos estáticos o por defecto (default) seguirán siendo funcionales si solo tienen un único método abstracto.

Ejemplo:

```
@FunctionalInterface
public interface SalaryToPrettyStringMapper {

  default List<String> map(List<Salary> list) {
    return list.stream()
      .map(this::map)
      .collect(Collectors.toList());
  }

  String map(Salary salary);
}
```

La anotación `@FunctionalInterface` denota que es una interfaz funcional, pero es opcional y, aunque no estuviese, la interfaz seguiría siendo funcional.

En cualquier caso, es recomendado añadirla si queremos que la interfaz sea funcional, ya que en caso de que alguien añada más métodos a la interfaz, el compilador lanzará un error si tiene la anotación.

Dónde pueden usarse las lambdas

Las lambdas pueden usarse en cualquier parte que acepte una **interfaz funcional**. La lambda tendrá que corresponder con la **firma del método abstracto** de la interfaz funcional.

Pueden asignarse a variables tipadas con la interfaz funcional que representan:

```
Predicate<Integer> isOdd = n -> n % 2 != 0;
isOdd.test(2); // false
```

Pueden ser parte del return de un método:

```
private Predicate<Integer> isOddPredicate() {
  return n -> n % 2 != 0;
}
```

Y, finalmente y lo más habitual, en las llamadas a métodos que acepten interfaces funcionales:

```
IntStream.range(0, 2)
  .mapToObj(entero -> String.format("entero = %s", entero))
  .forEach(cadena -> System.out.println(cadena));
// Salida:
// entero = 0
// entero = 1
```

Referencias a métodos

Cuando un método cualquiera coincida con la firma de una interfaz

funcional, podremos usar una referencia al método en vez de la sintaxis habitual de las lambdas.

Utilizando el ejemplo del apartado anterior, podemos modificar la lambda del forEach, ya que System.out.println coincide exactamente con la firma del método que espera.

```
IntStream.range(0, 2)
  .mapToObj(entero -> String.format("entero = %s", entero))
  .forEach(System.out::println); // <- Referencia a método
```

Para usar referencias a métodos, ponemos `::` justo antes del método, en vez de un punto, e ignoramos los paréntesis. Así pues, estas podrían ser referencias válidas a métodos:

```
System.out::println
this::miMetodo
super::metodoDeSuper
unObjeto::suMetodo
```

Interfaces funcionales estándar más importantes

Con la llegada de Java 8 y las lambdas, también se incluyeron varias interfaces funcionales en el API estándar de Java. Del mismo modo, interfaces que existían previamente y que contenían un único método abstracto, fueron marcadas oficialmente como interfaces funcionales.

Las nuevas inclusiones pueden encontrarse en el paquete java.util.function, y se pueden encontrar fácilmente en la documentación de Java 8.

Estas son algunas de las más importantes:

- Function.
- Supplier.
- Consumer.

- Predicate.

Mientras que algunas de las interfaces antiguas que a partir de Java 8 son funcionales son:

- Runnable.
- Callable.
- Comparator.

Data processing Streams

Prácticamente todas las aplicaciones tienen que trabajar con colecciones. Buscar elementos con un determinado valor, ordenarlas, transformar sus datos, etc. Típicamente, esto se ha hecho con bucles, iterando una y otra vez sobre ellos, repitiendo el mismo código. Además, para hacer el trabajo de forma eficiente, pretendemos utilizar varios núcleos de nuestra CPU. Eso

es difícil y una fuente de errores.

Los data processing streams vienen a solucionar este problema desde Java 8. Presentan las siguientes características:

- Operaciones fácilmente paralelizables.
- Estilo declarativo de operaciones.
- Concatenación de operaciones en un pipeline.

Para utilizar los streams sobre una colección, basta con invocar al método stream() o parallelStream(), en función de si queremos paralelizar las operaciones o no.

Un stream no almacena los valores, sino que se limita a computarlos. Obtiene los datos de una colección y genera un resultado tras el procesado de las operaciones intermedias del pipeline mediante una operación terminal. Es importante tener en cuenta que las operaciones intermedias devuelven un stream, mientras que las operaciones terminales no. Las operaciones intermedias no se ejecutan hasta que se realiza una operación terminal.

Por ejemplo, podemos realizar lo siguiente:

```
List<Other> l2 = l1.stream()
  .filter(elem -> elem.getAge() < 65)
  .sorted() // Ordena según la implementación de Comparable
  .map(elem -> new Other(elem.getName,() elem.getAge()))
  .collect(toList());
```

En función de su objetivo, podemos dividir las operaciones por grupos:

- Filtrado.
- Búsqueda.
- Mapeado.
- Matching.
- Reducción.

- Iteración.

Puedes encontrar un listado completo de las operaciones soportadas por los streams en la interfaz java.util.stream.Stream.

Además de crear un stream para una colección, se pueden construir streams para valores, un array, un fichero o incluso una función. Para valores, se utiliza el método estático Stream.of, mientras que para arrays se utiliza el método Arrays.stream.

```
int[] array = {1, 2, 3, 4, 5};
int sum = Arrays.stream(array).sum();
```

Para convertir un archivo en un stream de líneas, podemos utilizar Files.lines() como en el siguiente ejemplo:

```
long numberOfLines = Files.lines(
    Paths.get("yourFile.txt"),
    Charset.defaultCharset()
  ).count();
```

El hecho de que los streams computen elementos, hace que podamos crear streams infinitos a partir de funciones mediante Stream.generate y Stream.iterate. Por ejemplo, puede ser interesante para obtener un valor constante o número aleatorio.

Por último, aclarar que el método collect() es una operación terminal que acepta un parámetro de tipo Collector. Podemos importar métodos factoría para estos desde la clase Collectors. En función del tipo que utilicemos, la colección resultante será diferente.

I/O

Java realiza la entrada-salida, en inglés Input-Output (I/O), de datos a través de canales, mejor conocidos como **Streams**. Normalmente, el flujo para trabajar con streams siempre es el mismo:

1. Se abre el canal.
2. Se realiza una operación.
3. Se cierra el canal.

Los streams pertenecen a la paquetería de **java.io** y podemos encontrar dos tipos, por bytes y por caracteres.

- **Byte Stream:** gestionan el I/O de datos en formato binario (imágenes, sonidos, etc.). Podemos encontrar dos superclases abstractas que son **InputStream** y **OutputStream.** Algunos ejemplos más comunes que heredan de las clases nombradas son:
    - **FileInputStream:** permite leer un fichero.

Lo primero es cargar el fichero. Para poder leerlo, necesitamos usar el método *read()*. Cuando ya no haya más datos por leer, el método devuelve un -1 indicandolo. El último paso es cerrar el canal.

```
public static void main(String args[]) {
  FileInputStream fis = new FileInputStream("C:\\file.txt");
  try (fis) {
    int i = 0;
    while ((i = fis.read()) != -1) {
      System.out.print((char) i);
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

- **FileOutputStream:** permite escribir en un fichero.

Asociamos el fichero a un stream y obtenemos sus bytes. Para poder escribir, necesitamos el método *write()*. El último paso es cerrar el canal.

```
public static void main(String args[]) {
  FileOutputStream fos = new FileOutputStream("C:\\file.txt");
  try (fos) {
    String message = "Welcome to Autentia Onboarding";
    byte messageInBytes[] = message.getBytes();
    fos.write(messageInBytes);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Si queremos mejorar el rendimiento, ya sea de lectura o escritura, podemos utilizar las clases **BufferedInputStream** y **BufferedOutputStream,** respectivamente**.**

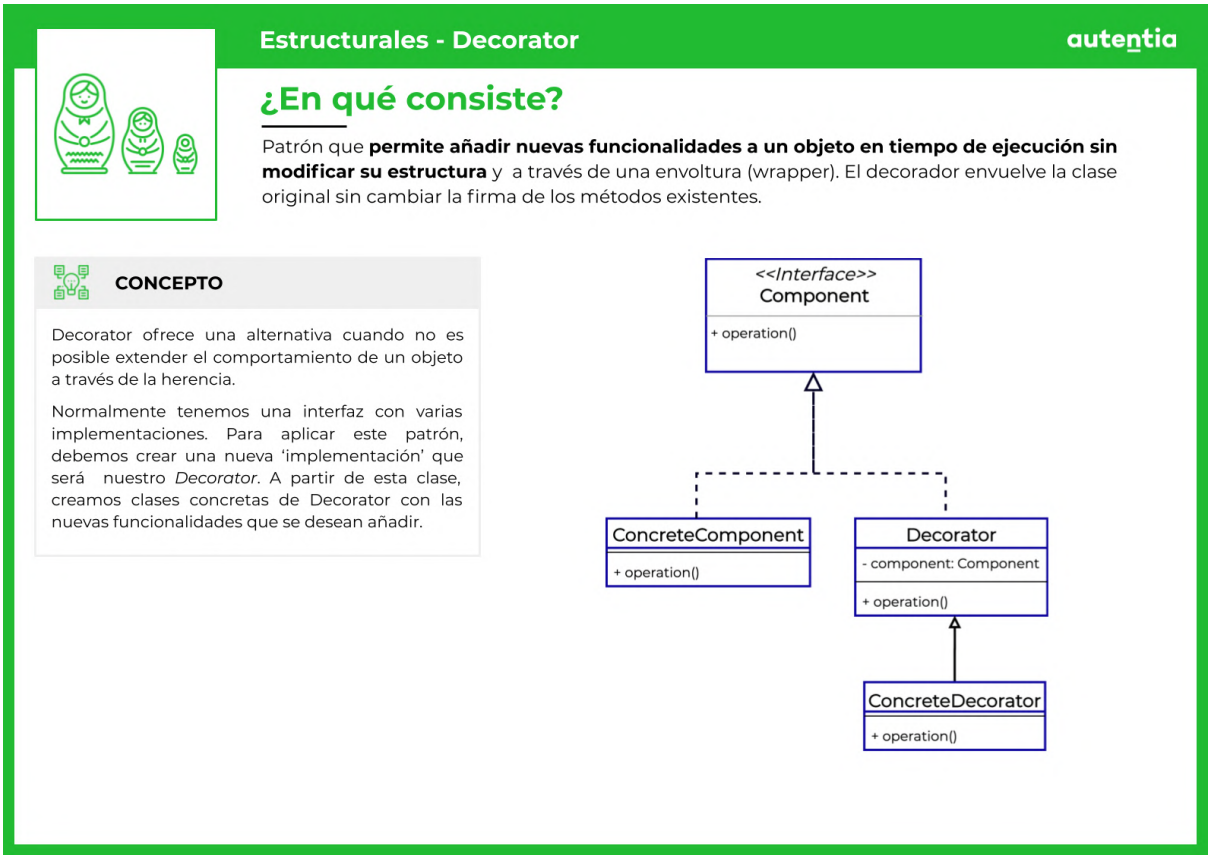
En el siguiente ejemplo, se observa como el BufferedOutputStream recibe por parámetro FileOutputStream y una vez se ha escrito en el buffer (a través de *write()*), se hace un flush para asegurarnos y forzar a que el buffer escriba todos los datos. No debemos olvidarnos de cerrar ambos canales.

```
public static void main(String args[]) throws Exception {
  FileOutputStream fos = new FileOutputStream("C:\\file.txt");
  BufferedOutputStream bos = new BufferedOutputStream(fos);
  try (fos; bos) {
    String message = "Welcome to Autentia Onboarding";
    byte messageInBytes[] = message.getBytes();
    bos.write(messageInBytes);
    bos.flush();
  }
}
```

Para la lectura, el ejemplo es muy parecido.

```
public static void main(String args[]) {
  FileInputStream fis = new FileInputStream("C:\\file.txt");
  BufferedInputStream bis = new BufferedInputStream(fis);
  try (fis; bis) {
    int i;
    while ((i = bis.read()) != -1) {
      System.out.print((char) i);
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Podemos encontrar otras clases como **PrintStream,** que permite escribir datos en otros streams, o **DataInputStream** que permite leer datos primitivos u objetos más complejos. Es un decorador sobre el InputStream que ofrece más funcionalidades que la clase básica de FileInputStream.



- **Character Stream:** gestionan el I/O de datos en formato texto (ficheros en texto plano, entradas por teclado, etc.). Podemos encontrar dos superclases abstractas que son **Reader** y **Writer**. Algunos ejemplos más comunes que heredan de las clases nombradas son:
    - **FileReader**: permite leer un fichero. Es igual que el FileInputStream visto anteriormente.

```
public static void main(String args[]) throws Exception {
  FileReader fr = new FileReader("C:\\file.txt");
  try (fr) {
    int i;
    while ((i = fr.read()) != -1)
      System.out.print((char) i)
  } catch (Exception e) {
    System.out.println(e);
  }
}
```

- **FileWriter:** permite escribir en un fichero. Igual que FileOutputStream, pero en este caso no necesitamos convertir la string en un array de bytes, ya que hay un método *write()* que recibe una string por parámetro.

```
public static void main(String args[]) {
  FileWriter fw = new FileWriter("C:\\file.txt");
  try (fw) {
    fw.write("Welcome to Autentia Onboarding");
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Al igual que vimos antes, también tenemos las clases **BufferedReader** y **BufferedWriter**. Otras clases interesantes pueden ser **InputStreamReader** y **OutputStreamReader** que actúan de puente entre un stream de bytes y un stream de caracteres.

Serializable

Cuando queremos que un objeto pueda ser enviado a través de algún canal, para persistirlo en un archivo o en una base de datos o para enviarlo a través de una conexión, debemos hacer que la clase implemente Serializable. Esta interfaz es sólo de marcado y no define ningún método.

Si la clase define algún atributo, como un objeto en lugar de un tipo primitivo, la clase de ese objeto también deberá ser Serializable.

Todas las clases serializables deberían definir un campo versión. Éste es útil cuando se modifica la clase y se producen conflictos con otras aplicaciones que utilizan versiones antiguas de la misma librería de clases. Su aplicación sería la siguiente:

```
public class MyClass implements Serializable {
  private static final long serialVersionUID = 31;
  // ...
}
```

Optional

Entre las muchas características que Java 8 incorporó al lenguaje, está Optional: una clase genérica que permite aplicar el patrón Option, nacido en los lenguajes funcionales e incorporado en esta versión de Java debido a la inclusión de las lambdas. Este patrón permite indicar explícitamente que un método puede o no devolver el valor deseado, obligando al desarrollador a controlar la posible ausencia de valor de forma explícita.

La clase Optional no dispone de un constructor público, delegando cualquier construcción a sus métodos de factoría estáticos.

```
public static <T> Optional<T> empty()
public static <T> Optional<T> ofNullable(T value)
public static <T> Optional<T> of(T value)
```

El primero nos permite retornar un objeto Optional vacío, es decir, sin valor. El segundo retorna un objeto con valor, pero si el parámetro es nulo, retorna uno vacío, y el último nos retorna un objeto con valor, y si se pasa un valor nulo, lanzará una excepción NullPointerException.

El objeto nos proporciona un conjunto de métodos básicos para trabajar con él:

```
public boolean isPresent()
public T get()

public Optional<T> filter(Function f)
```

```
public <U> Optional<U> map(Function f)
public <U> Optional<U> flatMap(Function f)

public T orElse(T other)
public T orElseGet(Function f)
public <X extends Throwable> T orElseThrow(Function f)
```

El método “isPresent” nos indica si el objeto tiene o no valor, es como si estuviéramos realizando la comprobación "variable == null", y el método “get” retorna el valor almacenado, devolviendo una excepción en caso de no existir.

El siguiente grupo es bastante útil para trabajar con el valor sin la necesidad de comprobar continuamente su presencia. Podremos ejecutar las operaciones “filter”, “map” y “flatMap” que habitualmente se usan cuando trabajamos con Streams.

Y el último bloque permite resolver la posible nulidad ejecutando una determinada acción. Por ejemplo, el método “orElse” nos retorna el valor o, si es nulo, el valor que pasamos por parámetro; “orElseGet” exactamente lo mismo, pero esta vez retornará el valor devuelto por la ejecución de la función y “orElseThrow” retorna el valor o, si no existe, lanzará una excepción que retorne la ejecución de la función pasada.

Destacar que no debemos utilizar este patrón como solución al problema de errores motivados por NullPointerException. A simple vista, uno puede interpretarlo así y, de hecho, en muchos desarrollos se ha usado este planteamiento incurriendo en un antipatrón. Por ejemplo, imaginad un desarrollo de una aplicación donde queremos obtener el número de alumnos que pueden ser escolarizados en un municipio e intentamos resolver el problema de la nulidad usando Optional. El código usando programación imperativa queda como se muestra abajo. Es muy difícil de seguir y puede incurrir en errores si alguien tuviera que modificarlo.

```
private static Optional<Integer> getStudentsOfCity(String name) {

  Optional<City> cityOptional = getCity(name);
  if (cityOptional.isPresent()) {
    City city = cityOptional.get();
    Optional<List<HighSchool>> highSchoolsOptional =
getHighSchools(city.getName());
    integer students = 0;
    if (highSchoolsOptional.isPresent()) {
      List<HighSchool> highSchools = highSchoolsOptional.get();
      for (HighSchool highSchool : highSchools) {
        students += track.getNumberOfStudents();
      }
      return Optional.of(students);
    } else {
      return Optional.empty();
    }
  } else {
    return Optional.empty();
  }
}
```

Algunas recomendaciones respecto al uso de los Optional para evitar varios de los antipatrones que se han usado en el ejemplo anterior serían:

- **Retornar un valor por defecto o que represente la nulidad**: muchos de los casos donde puede ser retornado un Optional, puede ser resuelto usando un valor por defecto o usar el patrón NullObject. Por ejemplo, la clase HighSchool en vez de tener:

```
public class HighSchool {
  private String name;
  private Optional<Integer> numberOfStudents;

  public Optional<Integer> getNumberOfStudents() {
    return numberOfStudents.
  }
```

```
}
```

Podríamos haber retornado un valor por defecto.

```
public class HighSchool {
  private String name;
  private Integer numberOfStudents;

  public Integer getNumberOfStudents() {
    return numberOfStudents == null ? 0 : numberOfStudents.
  }
}
```

- **No usar en atributos de un objeto**: la clase Optional no implementa serializable. Esto puede provocar errores si se usa en un atributo de una clase.

```
public class Student {
  private String name;
  private Optional<Address> address;
  ...
}
```

En estos casos se puede resolver de esta forma:

```
public class Student {
  private String name;
  private Address address;

  public Optional<Address> getAddress() {
    return Optional.ofNullable(address);
  }
}
```

- **No usar en colecciones**: este uso es un mal olor. Suele ser mejor

retornar una lista vacía.

- **No usar como parámetro de un método**: Opcional es una clase basada en valores, por lo tanto, no tiene ningún constructor público, es creada utilizando sus métodos estáticos de factoría. Su uso en parámetros supone código adicional que dificulta su legibilidad, siendo mejor no usarlo como tal.

Teniendo en cuenta estas recomendaciones, el código inicial quedaría mucho más legible:

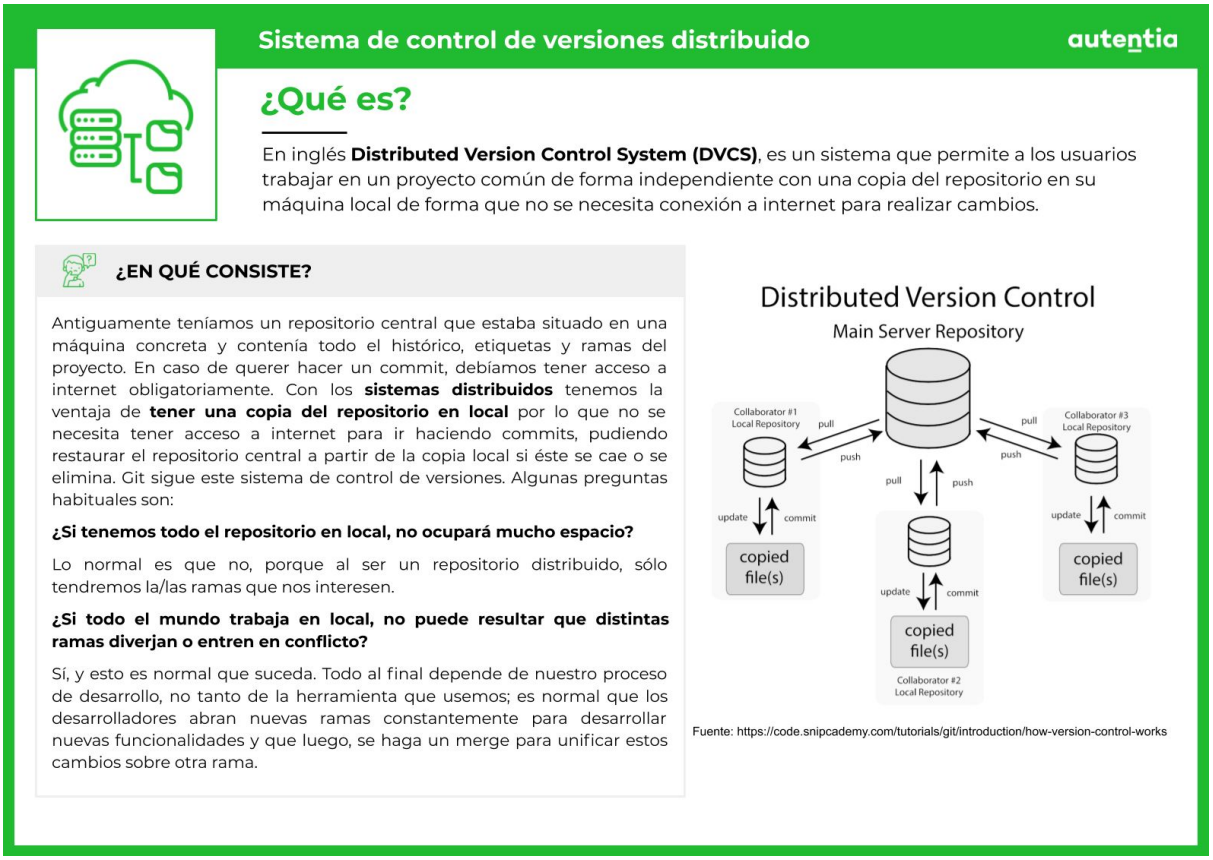
```
private static int getStudentsOfCity(String name) {
  int students = 0;
  City city = getCity(name);
  List<HighSchool> highSchools = getHighSchools(city.getName());

  for (HighSchool highSchool : highSchools) {
    students += track.getNumberOfStudents();
  }

  return students;
}
```

Parte 2

Herramientas y técnicas

Introducción a Git

GIT es un Sistema de control de versiones distribuido (DVCS) y Open Source que permite a los usuarios trabajar en un proyecto común y de forma independiente. Esto se hace a través de una copia del repositorio en la máquina local, de manera que no se necesita conexión a internet para realizar cambios. Cuando se necesite compartir los nuevos cambios con el equipo, se publican en el repositorio remoto.



¿Qué ventajas nos aporta?

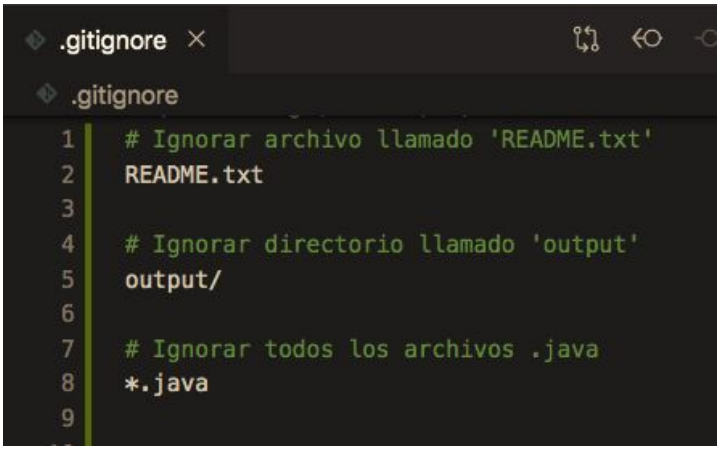
- Velocidad en el desarrollo, hace el trabajo en equipo más efectivo.
- Visualización de la evolución del proyecto.
- Sirve como Backup en caso de errores o problemas.

Algunas de las plataformas más conocidas para el control de versiones son Github, Gitlab o Bitbucket

Instalación inicial

Git proporciona diferentes instaladores para varias plataformas: Linux, Mac, Windows. En su web puedes descargarlos. Para comprobar que Git se ha instalado correctamente, podemos usar el comando *git --version*.

Vamos a crear nuestro primer repositorio local. Para ello, tenemos que crear una carpeta con el comando *"mkdir myRepository"* y acceder a ella *con "cd myRepository"*. Una vez dentro, ejecutamos el comando *"git init"*. Veremos cómo se crea una carpeta *.git* que contiene toda la información necesaria para el control de versiones del proyecto (commits, dirección de repositorio remoto, logs, etc.). También debemos crear un fichero llamado *.gitignore* que indicará todo aquello que no queramos subir al repositorio remoto. En él podemos definir archivos con cierta extensión o incluso, un directorio entero.



Estructura interna de un repositorio

Todos los datos de los repositorios son almacenados dentro de la carpeta '.git'. Un repositorio contiene un conjunto de objetos de commit y un conjunto de referencias head.

- Commit: contiene un conjunto de ficheros que refleja el estado del proyecto en ese punto, referencias a sus padres y un hashing que lo identifica unívocamente.
- Head: son referencias a un commit específico. Un repositorio puede tener varios heads, pero sólo un HEAD (en mayúscula) que identifica el head actual. Por defecto, hay un head en cada repositorio llamado 'master'.

Además, los repositorios locales cuentan con un directorio de trabajo. Este directorio está fuera del repositorio (la carpeta '.git'). En él, se encuentran los ficheros sobre los que se está trabajando. Pueden tener cambios sobre HEAD o no.

Este directorio de trabajo cuenta con dos áreas especiales que podemos utilizar para gestionar cómo se van a manejar los cambios en los ficheros:

- Área de staging: es el área donde se sitúan los ficheros con cambios que van a ser incluidos en el próximo commit. La inserción de los ficheros modificados en este área es manual.
- Área de stashing: en este área se sitúan ficheros con cambios que todavía no están listos para ser incluidos en un commit. Permite almacenarlos temporalmente mientras se resuelven otras tareas.

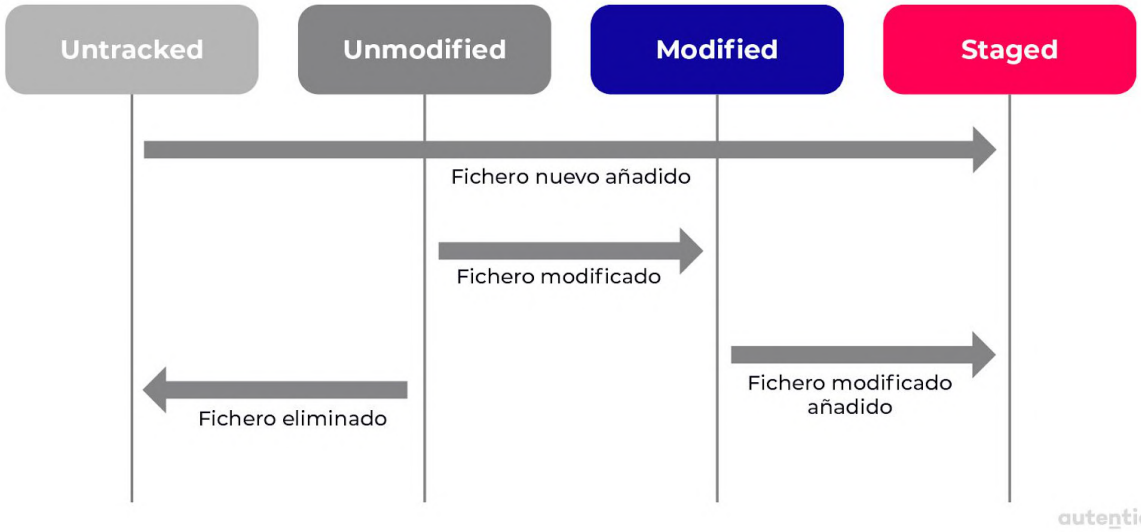
Ciclo de vida de un fichero

Cada uno de los ficheros que estén en el directorio de trabajo de git pueden encontrarse en varios estados:

- **Untracked**: son los archivos que no han sido añadidos al área de staged y que pueden ser consolidados una vez han pasado al estado Staged haciendo commit. Se puede cambiar el estado de estos archivos utilizando el comando "git add".
- **Unmodified**: una vez se ha realizado un commit, se podría decir que los archivos se quedan en un estado de Unmodified que puede cambiar a Modified en caso de que cambie algo.
- **Modified**: archivos ya existentes en Staged pero que han sido editados.
- **Staged**: son los archivos que se han añadido para hacer un commit. Para llegar a este estado, se ha tenido que ejecutar "git add". Si queremos pasar

algún archivo al área de Untracked, debemos ejecutar “git rm --cached [nombre_archivo]”.

Por ejemplo, cuando creamos un nuevo archivo llamado myFile.txt, el estado en el que se encuentra sería Untracked. Hacemos “git add myFile.txt” y pasaría al estado Staged. Ahora modificamos myFile.txt y añadimos una linea que diga “aprendiendo git”, el archivo sigue estando en el estado Staged pero los nuevos cambios que hemos añadido están en el estado Modified por lo que debemos hacer otra vez un “git add myFile.txt” para actualizarlo. Posteriormente, hacemos nuestro primer commit con el comando ‘git commit -m “my first commit”’. Una vez hecho esto todos los archivos comiteados pasan al estado Unmodified y el proceso comenzaría de nuevo.



Comandos básicos

Aunque GIT cuenta con un gran número de comandos y opciones, aquí se explican aquellos que se usan con más frecuencia:

- **init**: inicializa un repositorio nuevo. La opción --bare crea un repositorio sin árbol de trabajo, ideal para compartir el trabajo con otros usuarios. A menudo, estos repositorios se nombran con el sufijo ‘.git’.
- **clone**: crea un repositorio a partir de un repositorio remoto. El repositorio

remoto se añade como ‘origin’ al registro de repositorios remotos.

- **status**: comprueba el estado del repositorio y sus archivos.
- **log**: muestra el histórico del repositorio. Podemos usar la opción --oneline para que sea más fácil de leer.
- **add**: añade uno o varios ficheros con cambios al área de staging. Es un previo paso para consolidar los cambios.
- **commit**: consolida los cambios del área de staging y crea un nuevo commit. Requiere un mensaje de commit que se puede introducir con el parámetro -m o en el editor predeterminado, si no se especifica este parámetro.
- checkout: permite modificar el área de trabajo entre versiones. Puede aplicarse sobre commits o ramas.
- **branch**: crea una rama nueva a partir de un commit.
- **merge**: combina los cambios que se han producido en dos ramas desde el punto en que se bifurcaron. Se ejecuta con el área de trabajo en el head de la rama en la que queremos hacer merge de los cambios y se especifica el nombre de la rama a fusionar como parámetro. Si hay conflictos, es necesario resolverlos antes de finalizar el merge.
- **stash**: almacena los cambios en el área de stash y los elimina del área de trabajo. Es útil cuando tienes que cambiar de tarea pero los cambios no están listos para ser consolidados. Puedes tener más de un stash a la vez. Se puede utilizar de las siguientes maneras:
    - git stash: crea un stash nuevo.
    - git stash save “mensaje”: crea un stash con mensaje para poder distinguirlo mejor.
    - git stash list: muestra la lista de stash activos.
    - git stash pop: aplica el último stash creado al área de trabajo.
    - git stash pop stashId: aplica el stash al área de trabajo.
    - git stash apply: aplica el stash al área de trabajo pero no lo elimina

del área de stash.

- **remote**: permite administrar los repositorios remotos. Podemos usarlo de las siguientes formas:
    - git remote: devuelve el nombre de los repositorios configurados.
    - git remote -v: devuelve el nombre y las URL de los repositorios configurados.
    - git remote add nombre URL: añade un repositorio remoto al registro.
    - git remote rename old new: modifica el nombre de un repositorio en el registro.
    - git remote remove nombre: elimina un repositorio remoto del registro.
- **git fetch**: actualiza los cambios que se han producido en un repositorio remoto pero no actualiza el área de trabajo.
- **git pull**: actualiza los cambios que se han producido en un repositorio remoto y actualiza el área de trabajo. Puede causar conflictos. Es equivalente a hacer un git fetch + git merge.
- **git push**: permite publicar los cambios locales en un repositorio remoto. Se debe especificar el nombre del repositorio y la rama a publicar.

Existen más comandos y opciones que se pueden utilizar, pero con esto podrás desenvolverte en la mayoría de las ocasiones. Consulta la documentación de Git para más detalles.

Herramientas comunes

Además de la línea de comandos, podemos utilizar Git a través de aplicaciones que nos proporcionan interfaces gráficas para diferentes propósitos. Un ejemplo clásico son los sistemas de control de versiones integrados en los diferentes IDEs, como Eclipse, Visual Studio Code o IntelliJ. Estos nos permiten utilizar los comandos más habituales de una forma cómoda.

Además de éstos, existen otras herramientas utilizadas con frecuencia, tales como:

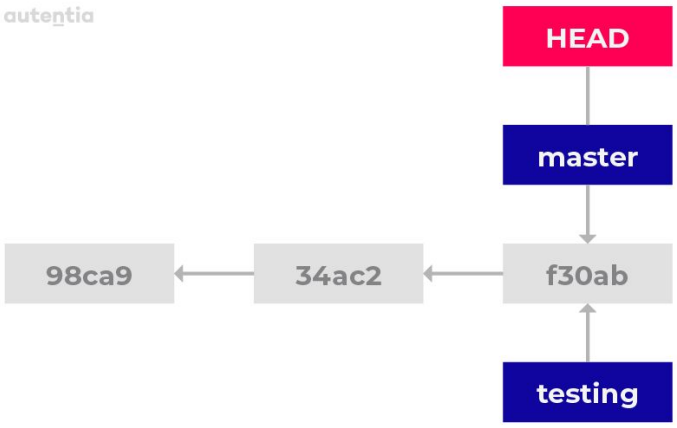
- gitk: es un visor del histórico de commits. Se instala junto a Git.
- git-gui: facilita la preparación y elaboración de commits. Se instala junto a git.
- GitKraken: es una interfaz gráfica para Linux, Mac y Windows con funcionalidades muy útiles como poder deshacer y rehacer acciones con un solo click. Su interfaz gráfica es muy intuitiva y nos permite hacer merge simplemente arrastrando con el ratón de una rama a otra.
- SourceTree: se trata de una interfaz gráfica para Mac y Windows que permite controlar nuestros repositorios locales y remotos, y realizar las operaciones de gestión de versiones de los proyectos.

Existen muchas más herramientas. Puedes encontrar un listado más completo en la web de Git.

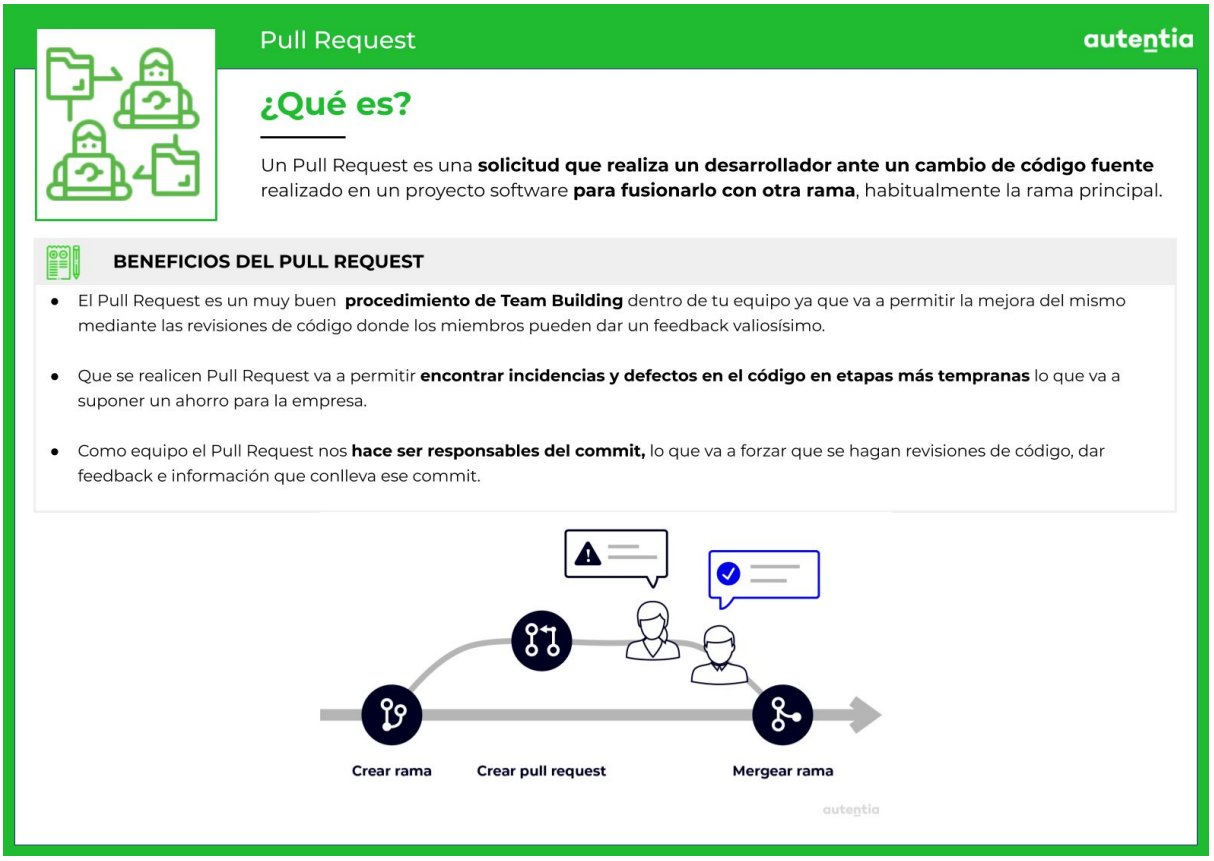
Ramas

Una rama Git es simplemente un puntero a uno de los commits. Cada vez que se confirmen los cambios del stage, se crea un nuevo commit y la rama apuntará a éste. El nuevo commit guarda un puntero al commit precedente. La rama por defecto de Git es la rama **master.**

Al crear una nueva rama, a través del comando “git branch”, se creará un nuevo puntero con el nombre. Por ejemplo, podemos crear una nueva rama “testing” con el comando “git branch testing”:



Para que el HEAD apunte a este nuevo puntero habrá que ejecutar “git checkout testing”. Existe una forma de hacer estos dos pasos en uno solo usando la opción -b. El comando completo sería “git checkout -b testing”



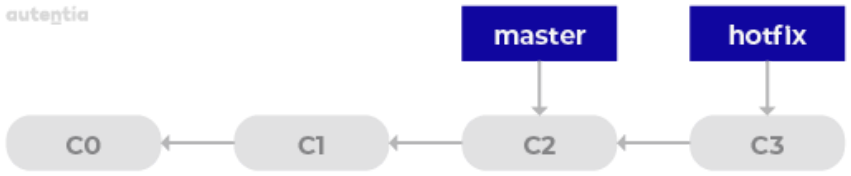
Problemas comunes y soluciones

La mayor parte de los problemas que podemos tener a la hora de trabajar con Git podemos resolverlos con los comandos básicos indicados anteriormente.

Conflictos al mergear con otra rama

Un conflicto sucede cuando Git descubre que las ramas que se van a mergear tienen cambios en la misma parte de uno o varios ficheros. Git deshace el merge y es el desarrollador/a el que debe resolver los conflictos. El desarrollador/a puede decidir quedarse con los cambios de la rama destino, de la rama con la que se va a mergear o hacer cambios manualmente. Una vez se resuelven los conflictos, se continúa con el merge.

Puede ocurrir que a la hora de mergear una rama con otra, los cambios sean simplemente lineales: por ejemplo, tenemos una rama “hotfix” que queremos mergear en nuestra rama “master”. Esta rama hotfix difiere de master en el commit C3:



Al mergear hotfix en master, Git realiza un **fast forward merge**, actualizando el puntero de la rama master a C3:

En este caso, los últimos cambios se incorporan sin necesidad de resolver conflictos y pueden subirse directamente al repositorio remoto.

Cambiar el mensaje de un commit

Puede pasar que al escribir el mensaje asociado al commit tengamos alguna errata como por ejemplo un error de ortografía, un mensaje poco descriptivo, etc. Para cambiar el mensaje asociado a un commit:

```
git commit --amend -m "Nuevo mensaje"
```

Añadir cambios a un commit

Se nos ha olvidado cambiar algo en alguno de los ficheros (p. ej. un comentario) asociados al último commit y no queremos añadir un nuevo commit para algo tan trivial:

```
git add fichero_modificado
git commit --amend

// En un solo paso
git commit -a --amend
```

Deshacer commits locales

A veces, nos damos cuenta de que alguno de los cambios asociados a los últimos commits, que se encuentran a nivel local, tienen algún error. Si queremos eliminar los últimos “n” commits del histórico pero no descartar los cambios ejecutamos:

```
git reset HEAD-n
```

Si queremos descartar los cambios:

```
git reset --hard HEAD-n
```

Deshacer commits ya pusheados

Se pueden deshacer commits a partir de los comandos:

```
git revert a345eq // Deshacer el commit a partir de su hash id
git revert HEAD^  // Deshacer penúltimo commit (^ indica al que apunta)
```

```
git revert HEAD~4..HEAD~2 // Deshacer un rango de commits
```

Gestión de ramas I

autentia

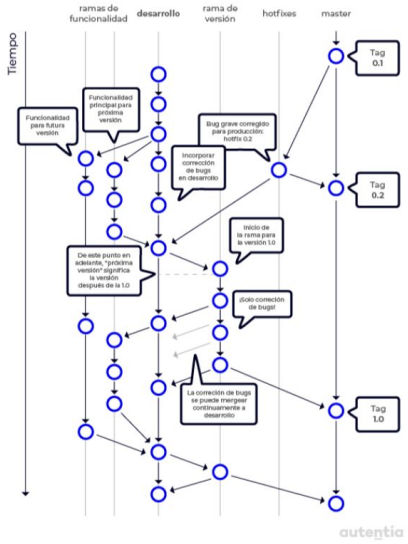
¿Cómo gestionarlas?

A la hora de trabajar con un sistema de control de versiones y en un equipo multidisciplinar, es necesario definir el flujo de trabajo que va a seguirse. GitFlow es una de las estrategias para la sincronización de ramas más comunes, fusionando los cambios introducidos por el equipo a través de Merge/Pull requests.

GIT FLOW

Modelo que establece una política de ramas bastante adaptable a cualquier entorno aunque puede llegar a ser compleja. Tiene una rama de desarrollo principal que suele llamarse **develop** y a partir de ella se crean subramas llamadas **features** para desarrollar nuevas funcionalidades. La rama **release** es una rama previa a producción que si finalmente todo está bien, se hace merge a **master**.

¿Qué pasa si encontramos un bug en producción? las ramas **hotfix** nacen a partir de master para solucionar problemas de urgencia y posteriormente se hace el merge a master y develop. **Los únicos commits que podemos encontrar en master son los merge con la rama release y hotfix**.



PULL REQUEST Y MERGE REQUEST

Previamente a hacer un merge, podemos realizar lo que se conoce como pull request en Github o merge request en Gitlab. Es un mecanismo que permite controlar los cambios de código introducidos en ciertas ramas, mediante un sencillo proceso de aprobación. Los motivos de rechazo pueden ser muy variados, desde conflictos en el código hasta desechar completamente una funcionalidad.

Gestión de ramas II

autentia

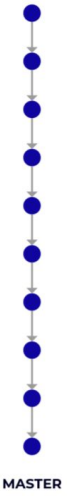
¿Cómo gestionarlas?

A la hora de trabajar con un sistema de control de versiones y en un equipo multidisciplinar, es necesario definir el flujo de trabajo que va a seguirse. Trunk based development es la estrategia utilizada cuando se opta por una entrega continua, integrando pequeñas funcionalidades de manera frecuente.

TRUNK BASED DEVELOPMENT

Modelo de desarrollo muy simple pero también usado de forma diferente según el tamaño del equipo. La idea general es que **todos los desarrolladores trabajen en una sola rama**, normalmente suele ser la rama maestra (master), también conocida como *trunk*. Se suele usar este modelo cuando se necesita iterar rápido y el equipo de desarrollo está en continua comunicación.

Tlene la ventaja de que se **integran los cambios de forma muy rápida** ya que obliga al equipo a realizar pequeños commits e ir haciendo push con mucha frecuencia. Además, cada persona verifica que todo los tests, incluidos los de integración, pasan sin ningún problema. Otra ventaja es que el tamaño de los merge suele ser muy pequeño (se evita el *merge hell* de cientos de ficheros) y por consiguiente se reducen los conflictos. Es un metodología opuesta a Git Flow.



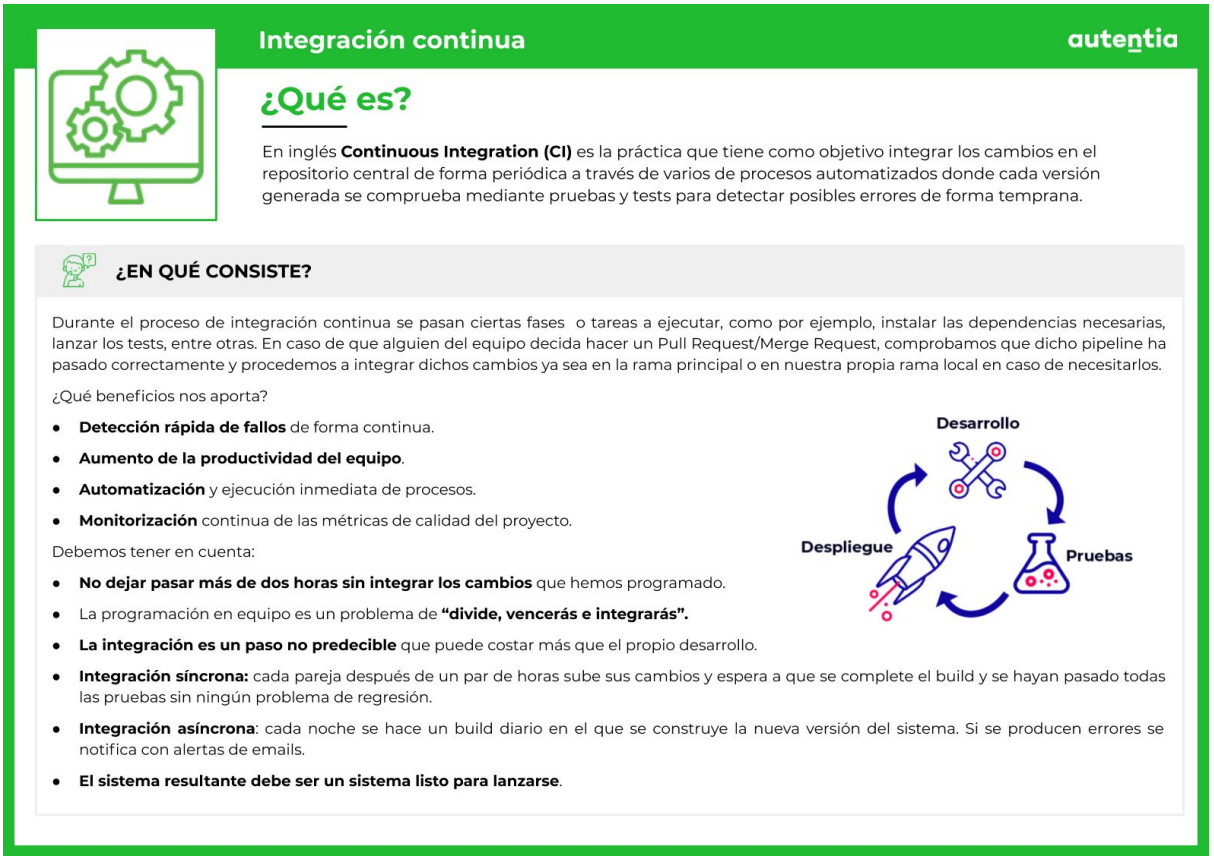
MASTER

OTROS MODELOS

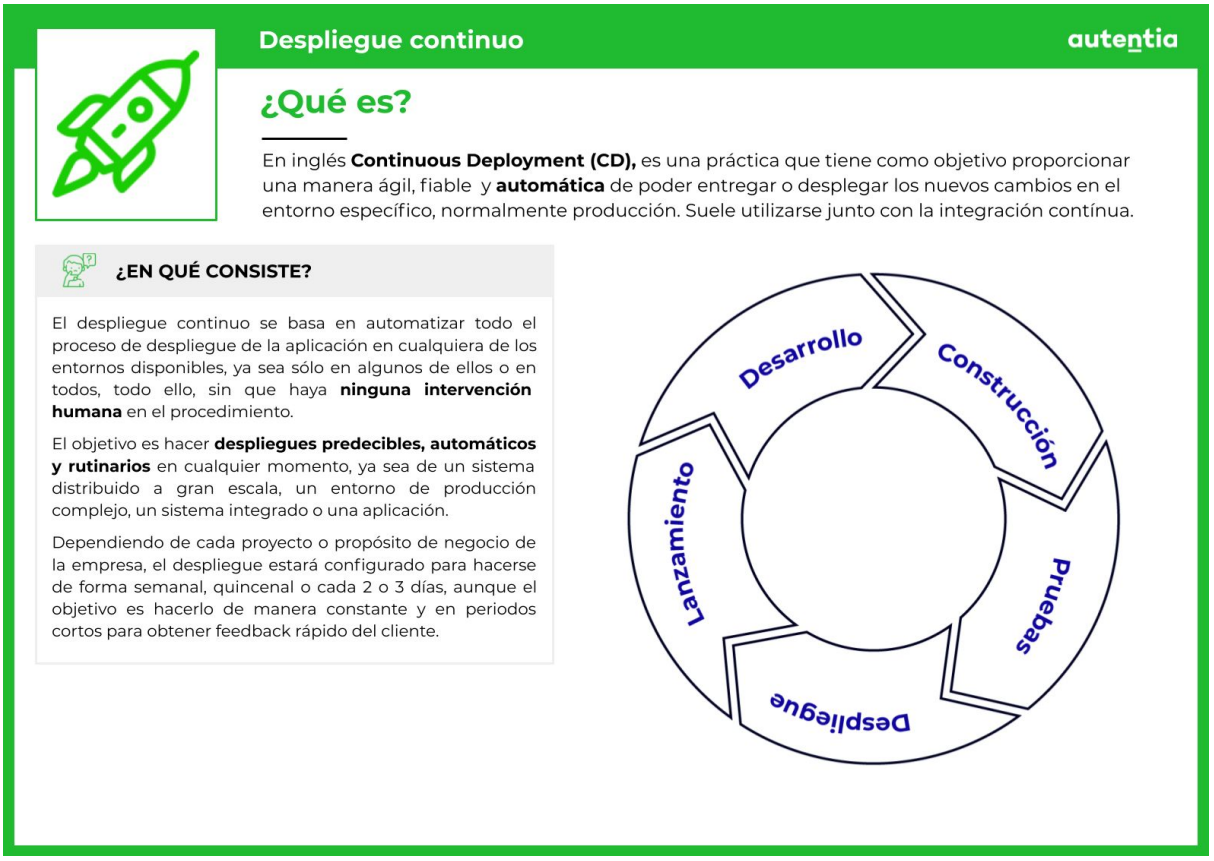
A parte de los dos modelos vistos en estas dos fichas, existen otros modelos de gestión también conocidos como **Github Flow** que solo tiene dos ramas master y las features y se van haciendo pull request para hacer el merge a master, **Gitlab Flow, Release Flow** por Microsoft, entre otros.

Introducción a la gestión de la configuración

Dentro de un proyecto de software existen una serie de tareas de gestión comunes: compilación, empaquetado, versionado, perfilado, control de dependencias, generación de documentación, automatización de pruebas, publicación de releases o cumplir unos mínimos criterios de calidad. Todas estas tareas son lo que comúnmente llamamos la gestión de la configuración de nuestro proyecto software.



Las herramientas para gestionar los proyectos software han evolucionado con el paso del tiempo. En un primer momento, la gestión se hacía a través de GNU Make y Apache Ant, la primera herramienta que centralizaba la configuración de proyectos Java en un fichero XML y que era totalmente independiente de la estructura. Sin embargo, debido a las limitaciones de Ant para ciertas tareas como la gestión de dependencias, nació Apache Maven y un poco más tarde Gradle.



Maven

Antiguamente, si queríamos compilar y generar ejecutables de un proyecto, se debía realizar un previo análisis sobre la estructura, las dependencias del proyecto, librerías, qué ficheros se querían compilar, etc. y luego, el propio equipo ejecutar las acciones deseadas. El principal cambio con Ant/Make es que con Maven de manera **declarativa** se define cómo es el proyecto de una manera estándar y es la propia herramienta la que ejecuta las acciones definidas. De esta

manera, se permite también la rápida inclusión de personal ajeno al proyecto.

Es a través del POM (Project Object Model) definido mediante el fichero pom.xml, donde se declaran las particularidades de nuestro proyecto. Contiene toda la información del mismo: de qué librerías depende, qué versión de JVM va a utilizar para compilar, qué informes hay que generar, etc. De este modo, nosotros indicamos las dependencias que queremos usar y él, automáticamente, realiza las tareas necesarias para obtenerlas del repositorio.

La configuración por defecto en el pom.xml del código fuente es 1.6. Esto significa que si la aplicación utiliza cualquier novedad de java 1.8 o superior el código fuente no compilará. No importa si en las variables de entorno del equipo se utiliza java 1.8, si el Maven compiler plugin no está definido, Maven utilizará java 1.6 para compilarlo. Para configurar esto debemos modificar lo siguiente:

```
<properties>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.source>1.8</maven.compiler.source>
</properties>
```

O definiendo el plugin:

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-compiler-plugin</artifactId>
     <version>3.8.0</version>
     <configuration>
          <source>1.8</source>
          <target>1.8</target>
     </configuration>
</plugin>
```

Maven se basa en una serie de patrones (fases del ciclo de vida de construcción de un proyecto) y estándares (estructura de directorios) que veremos a continuación.

Estructura de directorios

Maven establece una estructura común de directorios para todos los proyectos.

src/main/java	Código fuente con nuestras clases Java.
src/main/resources	Recursos de la aplicación (imágenes, ficheros de configuración como el logback.xml, application.properties, etc.).
src/main/webapp	Ficheros de la aplicación web (html, css, js).
src/main/sql	Scripts de BBDD.
src/test/java	Código fuente con las clases de tests.
src/test/resources	Igual que el de src/main pero para los tests.
target	Directorio donde Maven sitúa los desplegables (jar, war, ear, etc.). Con mvn clean eliminados el código previamente compilado. Para volver a generarlo se debe ejecutar mvn install.
pom.xml	Fichero descriptivo del proyecto.

Ciclos de vida

Las fases más comunes son (el resto de fases se pueden encontrar en la documentación oficial):

- **validate**: valida si la información del proyecto es correcta.
- **compile**: compila el código.
- **test**: ejecuta los test unitarios.
- **package**: empaqueta el proyecto generando un jar/war/ear. Recordemos que los jar pueden ser ejecutados desde la línea de comandos si se ha generado como un jar ejecutable. Por el contrario, se necesita un servidor para ejecutar un war.
- **verify**: comprueba la validez y calidad del paquete.
- **install**: instala el paquete en el repositorio local. Este paso crea el directorio /target. Podemos ejecutar esta fase obviando los tests con el comando `mvn clean install -Dmaven.test.skip=true`

- **deploy**: se hace en un entorno de integración o release y se encarga de copiar el proyecto en el repositorio remoto.

Debemos tener en cuenta que cuando ejecutamos una fase, por ejemplo package, implícitamente también se ejecutarán en orden, las fases previas a ésta, es decir, validate, compile y test.

Existe una fase especial, **clean**, que solo se ejecuta si se indica explícitamente y por lo tanto, está fuera del ciclo de vida de Maven. Esta fase limpia todas las clases compiladas del proyecto.

Goals

Cada fase vista anteriormente tiene una serie de goals por defecto que se encargan de realizar una tarea. Cada vez que ejecutamos una fase, se ejecutan sus goals por defecto, aunque también podemos ejecutar únicamente un goal sin tener que ejecutar su fase.

Estos son algunos de los goals asociados a algunas fases:

clean	clean:clean
compile	compiler:compile
test	surefire:test
package	jar:jar par:par rar:rar war:war ejb:ejb ejb3:ejb3
install	install:install
deploy	deploy:deploy

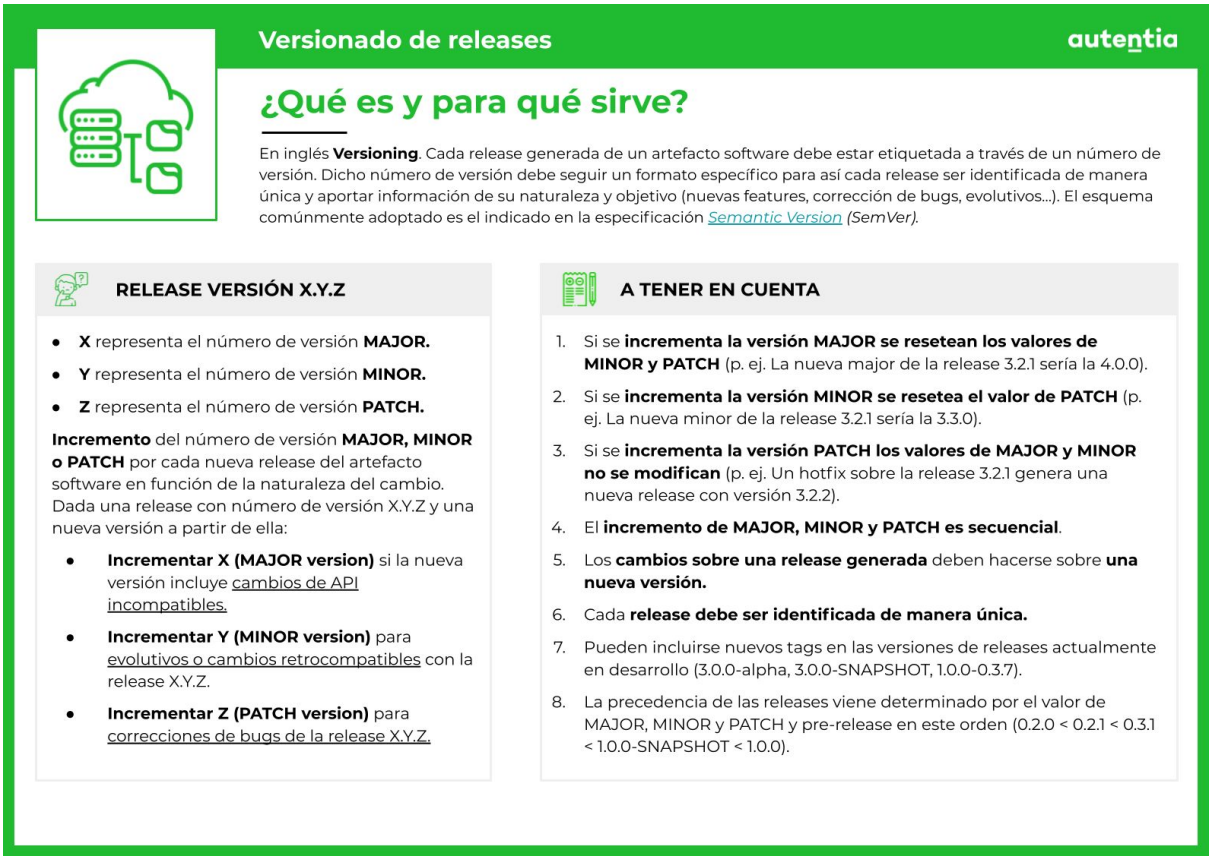
Si tenemos curiosidad por saber los goals ejecutados en un fase, podemos usar el comando `mvn help:describe -Dcmd=NOMBRE_FASE`

Dependencias y Repositorios

La gestión de dependencias con Maven es muy sencilla. A través de maven

repository podemos buscar aquellas que sean necesarias para nuestro proyecto y añadirlas dentro del apartado <dependencies> de nuestro fichero pom.xml. En el siguiente ejemplo tenemos la dependencia de JUnit 5:

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.7.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```



Un repositorio es un lugar donde se almacenan todas estas dependencias de uso cotidiano que pueden ser accedidas por el resto de proyectos. Maven se encarga de buscar las dependencias primero en el repositorio local y, si no las encuentra, las buscará en los repositorios remotos que le hayamos indicado en el pom.xml (por defecto https://repo.maven.apache.org/maven2).

Los repositorios se pueden definir en el pom.xml o el <USER_HOME>/.m2/settings.xml. El repositorio local por defecto se encuentra en el directorio <USER_HOME>/.m2/repository.

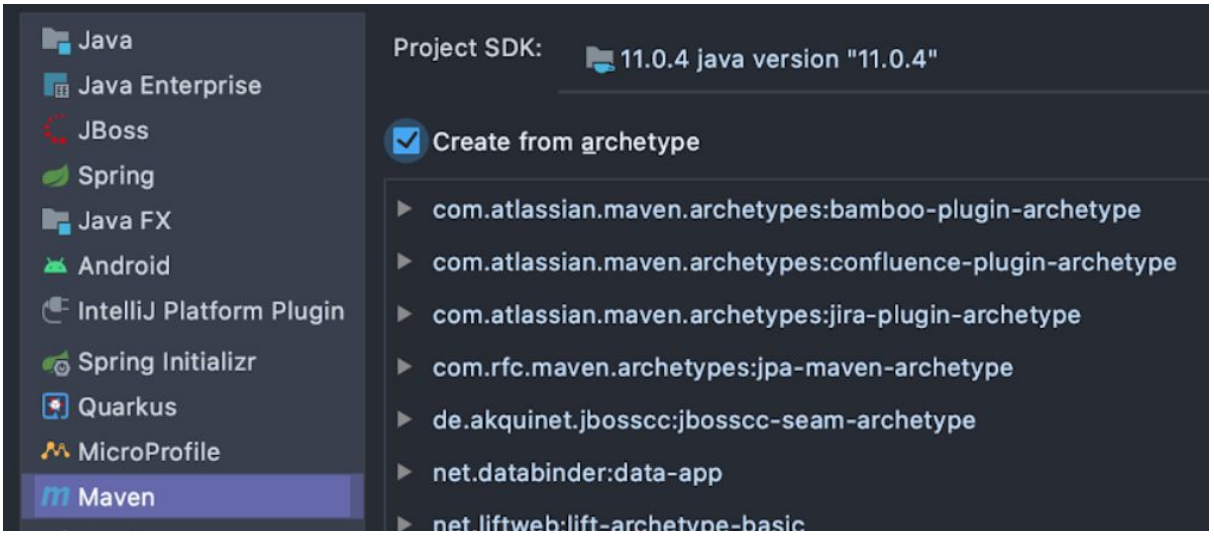
En muchos proyectos el repositorio de Maven se queda corto. Por ejemplo, las dependencias de Oracle como: j2ee, JTA o Activation no se encuentran. Esto incrementa la necesidad de crear un repositorio compartido en la organización que almacene estas librerías de terceros y las propias de la organización. Además, presenta la ventaja de que todos los miembros de la organización tienen actualizado su repositorio con las últimas versiones y reduce el ancho de banda ya que los desarrolladores se conectan contra un repositorio de la organización.

Arquetipos

Los arquetipos son las plantillas que podemos utilizar para generar nuestros proyectos con Maven. Estas plantillas evitan el llamado “miedo al folio en blanco” ya que nos generan una estructura de directorios y código de ejemplo, acorde con la naturaleza del proyecto que queramos realizar.

Con el comando `mvn archetype:generate` podemos ver una lista con todos los arquetipos que tiene por defecto cada tipo de proyecto y proporcionar la información del groupId, el artifactId y la versión (por defecto 1.0-SNAPSHOT). Si queremos filtrar los archetype por el paquete usamos la opción -Dfilter con el formato [groupId:]artifactId

Si estamos utilizando algún IDE tipo IntelliJ, al crear un nuevo proyecto podemos seleccionar si queremos crearlo a partir de un arquetipo.



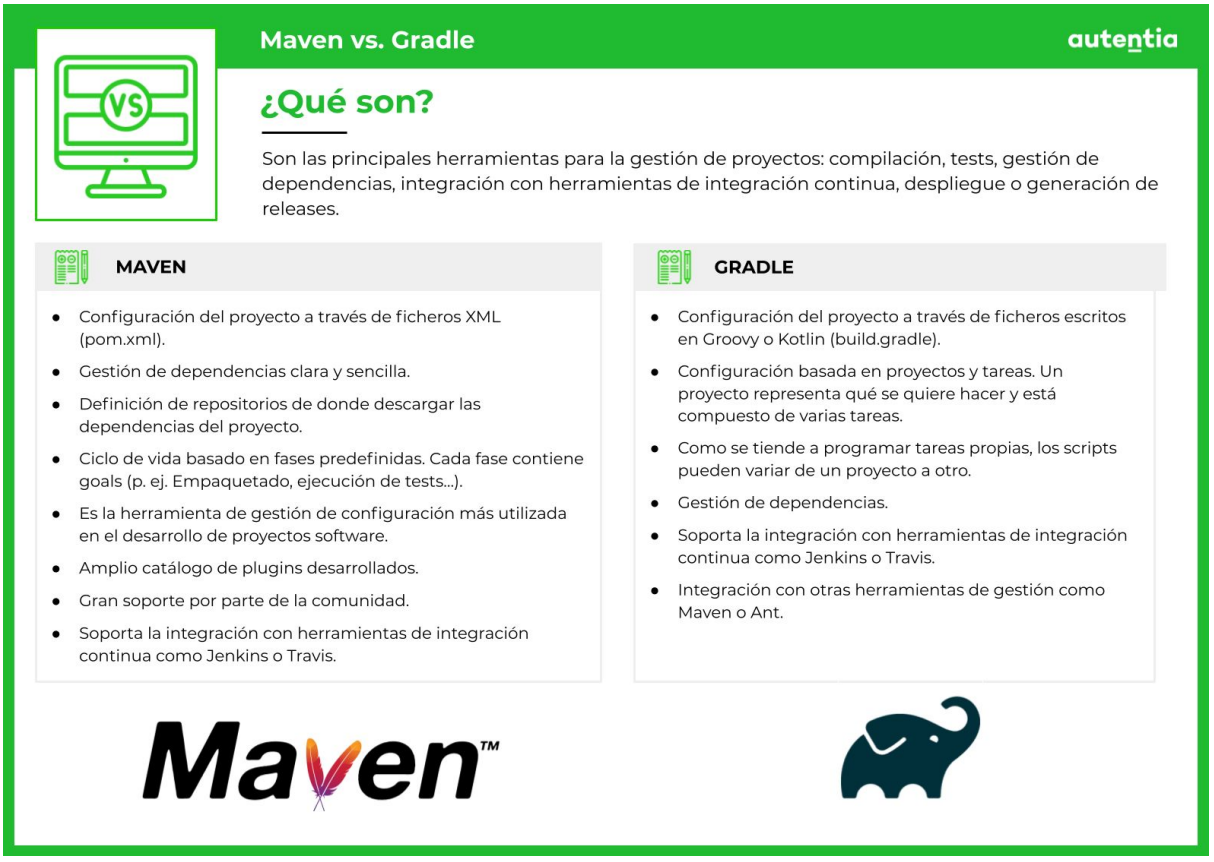
Gradle

Gradle es una herramienta similar a Maven con la que podemos gestionar las dependencias y las distintas fases de nuestro proyecto con las siguientes diferencias:

- La configuración se especifica, en vez de en XML, en un lenguaje basado en Groovy o Kotlin.
- Ficheros de configuración muy explicativos y que ocupan muy pocas líneas. Por ejemplo, un fichero de configuración para una aplicación en la que hay que verificar el código con pmd, checkstyle, findBugs e incluir la dependencia de JUnit quedaría en unas pocas líneas:

```
apply plugin:'java'
apply plugin:'checkstyle'
apply plugin:'findbugs'
apply plugin:'pmd'
version ='1.0'
repositories {
    mavenCentral()
}
dependencies {
    testCompile group:'junit', name:'junit', version:'4.11'
}
```

- Las distintas fases que se pueden definir en la configuración son etiquetadas como “task”.
- Gestión de dependencias.
- Construcción rápida, ya que evita la ejecución de aquellas tareas que tengan como resultado la misma salida.



Introducción al testing

Las pruebas, en ingeniería de software, son los procesos que permiten verificar y revelar la calidad del producto. Con la prueba, se lleva a cabo la ejecución de un programa que, mediante técnicas experimentales, trata de evitar errores que se producirían en tiempo de ejecución y comprueba la funcionalidad.

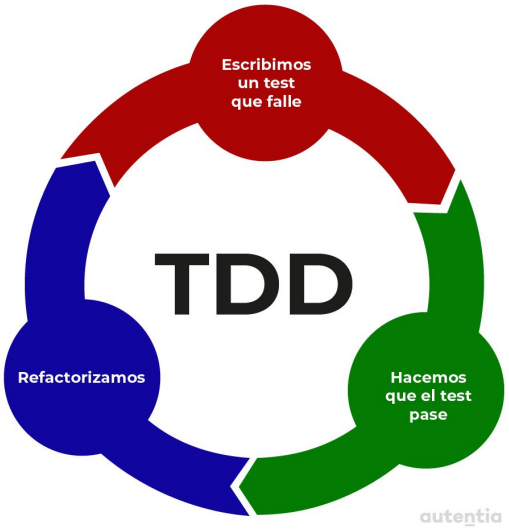
Hay muchos tipos distintos de pruebas: unitarias, de integración, funcionales, de aceptación, de regresión, etc.

El hecho de tener pruebas sobre nuestro código nos asegura que lo que funciona hoy, seguirá funcionando mañana; sobre todo, si la ejecución de las mismas está automatizada e integrada dentro del ecosistema de desarrollo, con el soporte de un servidor de integración continua.

Sin una buena batería de tests, cualquier modificación en el código puede ser el origen de un nuevo bug; con los tests se pierde el miedo al cambio y cuanta mayor cobertura, menos miedo tendremos. Eso sí, debemos tener cuidado y probar los casos estrictamente necesarios. Muchas veces, por tener un porcentaje de cobertura del 100%, se testean casos que son innecesarios. Lo normal es tener un test por cada regla de negocio.

TDD y las pruebas como técnica de diseño

Los tests deben ser un medio para realizar el diseño de la funcionalidad de negocio de nuestra aplicación. Aplicando TDD (Diseño Dirigido por Tests o Test Driven Development) podemos afrontar el problema desde una perspectiva más general y posteriormente podemos dividirlo en componentes más pequeños con el siguiente ciclo de vida:



RED: primero comenzamos escribiendo el código del test, que no compilará puesto que aún no hemos escrito nuestras clases, y no pasará porque no tiene lógica de negocio.

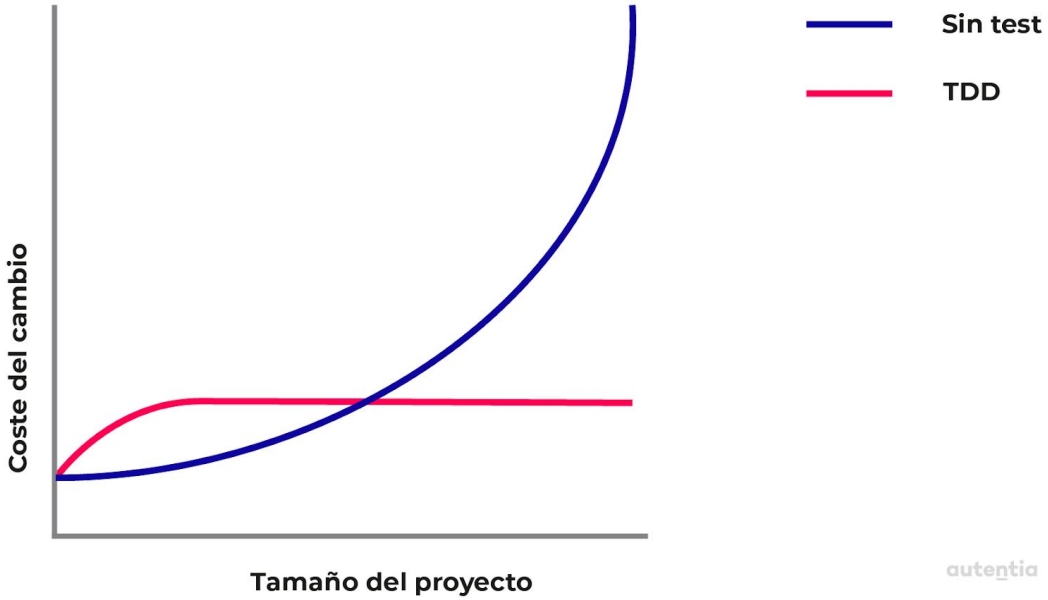
GREEN: después escribimos el código de nuestras clases de negocio para que el test compile y pase con el mínimo código posible.

REFACTOR: por último, eliminamos redundancia y mejoramos la implementación, con técnicas de refactoring y principios SOLID, una vez que disponemos del test que comprueba que todo sigue funcionando correctamente.

Siguiendo la técnica del **RED - GREEN - REFACTOR** nos aseguramos que no escribimos una línea de código que no esté probada mediante un test y, con ello, no escribimos una línea de código innecesaria. TDD nos sitúa en el punto de vista de quién tiene que usar la funcionalidad que estamos implementando. Esto resulta en no construir más código del necesario para cubrir la funcionalidad, sin complicar innecesariamente la aplicación.

Siguiendo con la misma filosofía, sólo deberíamos generar tests que cubren la funcionalidad de historias de usuario o casos de uso. Los tests nos ayudan a documentar el código que se va escribiendo. ¿Cómo? Cada test generado es una regla de negocio que estamos probando y nos obliga a pensar en un buen naming que describa exactamente lo que estamos validando, lo que incide directamente en un mejor diseño.

Si seguimos un diseño guiado por tests, el tiempo de desarrollo al principio puede ser más alto, hasta que se comience a dominar la técnica, sin embargo, en un corto plazo de tiempo obtendremos un beneficio tal que tanto el cliente, como los desarrolladores agradecerán. Podremos modificar o añadir nueva funcionalidad, con la certeza de que no vamos a 'romper' nada, y si hemos hecho algún cambio crítico, los tests estarán ahí para avisarnos. A partir de ese momento, nos preguntaremos asombrados por qué no lo hacíamos antes así.



JUnit

JUnit es la librería opensource más usada para el desarrollo de test unitarios en aplicaciones Java.

Para que un método de una clase se convierta en un test, basta con añadir a su firma la anotación @Test:

```
@Test
public void test_name() {
    ...
}
```

Un aspecto fundamental de las pruebas es verificar que el código fuente probado realmente hace lo que debe. Para hacer este seguimiento, JUnit proporciona la

clase Assert con la que a través de una serie de métodos podemos delegar ciertas comprobaciones: que un objeto no sea nulo, que sea nulo, que dos objetos deban ser iguales... Cuando alguna de estas comprobaciones no pasa, lanzará un AssertionError y ese test fallará.
A continuación se exponen algunos métodos importantes y se reccomienda hacer uso de los mismos con importaciones estáticas para simplificar el código del test y que sea más legible:

- **Assert.assertNotNull**: recibe un objeto y comprueba que no sea nulo.
- **Assert.assertNull**: recibe un objeto y comprueba que sea nulo.
- **Assert.assertEquals**: recibe dos objetos y comprueba que sean iguales. Importante sobreescribir e implementar la lógica de los métodos equals y hashCode en los objetos que queremos comprobar.
- **Assert.assertNotSame**: recibe dos objetos y comprueba que no sean el mismo.
- **Assert.failNotEquals**: este método está pensado para forzar el fallo si los objetos que le pasamos no son iguales.
- **Assert.fail()**: provoca explícitamente un fallo dentro del test. Sirve para forzar errores que en condiciones normales no deberían existir. Aunque si lo que se quiere probar es que se lanza una excepción, en JUnit5 se puede usar Assertions.assertThrows.

Cambios entre JUnit4 y JUnit5

Algunos cambios importantes que podemos destacar entre una versión y otra son los siguientes:

JUnit4	JUnit5	Descripción
@Ignored	@Disabled	Deshabilitamos la ejecución del test.
@Before y @After	@BeforeEach y @AfterEach	Permite ejecutar código antes y después de cada test.
@BeforeClass y @AfterClass	@BeforeAll y @AfterAll	Permite ejecutar código antes y después de que se ejecuten todos los tests.

Para lanzar excepción en JUnit 4 se usaba expected:

```
@Test(expected = NullPointerException.class)
public void this_test_will_throw_a_null_pointer_exception() {
    //..
}
```

Y si queríamos obtener el mensaje de error de la excepción, había que crearse una regla con la anotación @Rule:

```
@Rule
public ExpectedException expectedEx = ExpectedException.none();

@Test
public void this_test_will_get_the_null_pointer_exception_message() {
    expectedEx.expect(NullPointerException.class);
    expectedEx.expectMessage("my exception message");
}
```

En JUnit5 esto ha cambiado y ahora nos proporcionan un Assertions.assertThrows:

```
@Test
public void this_test_will_throw_a_null_pointer_exception() {
   Exception exception = assertThrows(NullPointerException.class, () ->{
        //...
    });

   String message = exception.getMessage();

   assertTrue(message.contains("my exception message"));
}
```

Hamcrest

Es una librería que nos permite añadir expresividad en los asserts de los test y así hacerlos más legibles.

Uso tradicional:

```
 ...
    assertEquals(expected, actual);
    assertNotEquals(expected, actual)
 ...
```

Con Hamcrest:

```
...
   // Los 3 primeros ejemplos son equivalentes
   assertThat(a, equalTo(b));
   assertThat(a, is(equalTo(b)));
```

```
   assertThat(a, is(b));

   assertThat(actual, is(not(equalTo(expected))));
   assertThat(a, nullValue());
...
```

Estos son ejemplos muy sencillos pero ya se puede ver la potencia de los matchers de Hamcrest. Estos matchers son los que nos permiten expresar lo que queremos comprobar en el assert y sus principales ventajas son:

- Gran cantidad de matchers predefinidos.
- Mejora la legibilidad de los tests.
- Podemos hacer nuestros propios matchers reutilizables.
- Son perfectamente compatibles tanto con JUnit como con Mockito.

AssertJ

Es una alternativa a Hamcrest que permite también escribir tests con un lenguaje más expresivo. Su sintaxis podría decirse que es prácticamente igual.

Algunos ejemplos son:

```
assertThat(user.getName()).isEqualTo("Autentia");
assertThat(user).isNotEqualTo("any");
assertThat(actual).isNull();
```

Cobertura de código y JaCoCo

La cobertura de código es una medida (porcentual) en las pruebas de software que mide el grado en que el código fuente de un programa ha sido testeado. Una alta cobertura implica un código robusto y de confianza, pero no debemos confundirnos con esto, ya que la cobertura debe ser un indicador y no un objetivo.

JaCoCo es un plugin de Maven que se utiliza para realizar reportes basados en las métricas de cobertura de código. Para añadir JaCoCo a un proyecto, podemos hacerlo añadiendo el plugin correspondiente de Maven en nuestro fichero

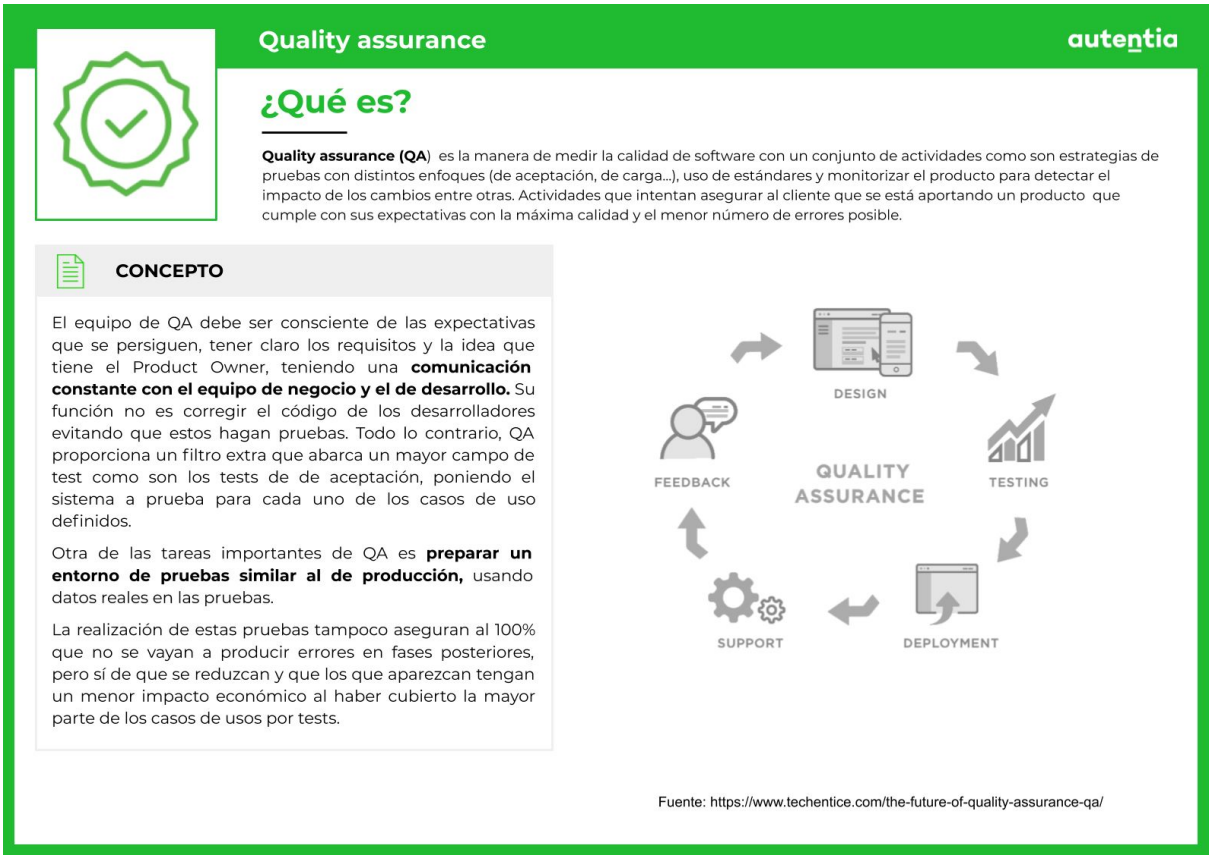
pom.xml:

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>X.X.X</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Para ver el reporte generado, podemos ejecutar el goal de Maven `mvn jacoco:report` que nos creará un html con el resultado. ¿Dónde? Dentro de la carpeta /target/site/jacoco/index.html.

Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
145 of 262	44%	25 of 32	21%	20	26	8	26	4	10	0	2

Estos reportes pueden ser utilizados para ser cargados y visualizados en otras herramientas de integración continua como Sonar o Travis.



Dobles de Test

A veces, estamos testeando un componente que tiene dependencias con apis de terceros o incluso se conecta a una base de datos para recuperar cierta información. Pero, lo que realmente queremos testear es el comportamiento del componente. Imaginemos que siempre que existen dependencias, realizamos una conexión con la base de datos, en un proyecto pequeño el rendimiento de los tests podría ser inapreciable, pero en un proyecto grande con cientos, incluso miles de tests, la duración de todos ellos podría ser inmensa. Aquí es cuando entran en juego los dobles de tests para simular dicho comportamiento que nos permita centrarnos y testar solo lo que realmente necesitamos. Permiten “engañar” al código para que se crea que colabora correctamente con otras clases, es como si fueran los dobles de las películas para las escenas peligrosas.

Existen los siguientes tipos de dobles ordenados de menor a mayor complejidad:

dummy: se usa cuando no nos importa cómo se colabora con este objeto. Por ejemplo, cuando sabemos que no se va a usar en absoluto. Lo necesitamos porque

nos interesa su interfaz pero no su implementación. La implementación de los métodos de estos dobles no hacen nada y devuelven null. Normalmente, se usa para rellenar una lista de parámetros.

```
class DummyRepositoryClass implements RepositoryClass {
     @Override
     public String getHelloWorld() {
         throw new RuntimeException("Not expected to be called");
     }
}

class ServiceTest{
    @Test
    public void example_dummy_test() {
        DummyRepositoryClass dummy = new DummyRepositoryClass();
        ServiceClass myService = new ServiceClass(dummy);
    }
}
```

Se debe tener en cuenta que el uso de un framework de mocks también es una alternativa al ejemplo anterior y suele ser más común. Si usamos, por ejemplo, mockito, se haría de la siguiente forma:

```
DummyRepositoryClass dummy = mock(DummyRepositoryClass.class);
```

stub: es como un dummy pero que devuelve valores fijos distintos de null. Por ejemplo, un método de autenticación devolvería siempre true y así podríamos usar este doble para probar todos los escenarios donde la autenticación ha sido correcta, sin necesidad de hacer la llamada real. En el ejemplo anterior, el método `getHelloWorld()`, podría devolver siempre la misma string, esto se consideraría un stub. Como vimos antes, esto también se podría hacer con un mock, donde podremos especificar el valor que queremos que devuelva siempre.

spy: es como un stub pero que espía a quien lo llama. Esto permite luego, comprobar el número de veces que se ha llamado al método, el número de argumentos que se le pasan, etc. Estos dobles son peligrosos porque acoplan el test con la implementación concreta, lo que provocará que si se cambia la

implementación, aunque no cambie el comportamiento, el test fallará. Son tests frágiles, por lo que debemos evitarlos.

Mockito nos ofrece el método verify(), que comprueba que se llama al método e incluso el número de veces que ha debido ser invocado. Por ejemplo:

```
private final AuthenticationService spyAs =
        mock(AuthenticationService.class);
    ...
    when(spyAs.isAuthenticated()).thenReturn(true);
    ...
    verify(spyAs).isAuthenticated();
```

En el código anterior se comprueba que efectivamente se ha llamado al método isAuthenticated() una sola vez (el valor por defecto). Si quisiéramos comprobar que se ha llamado tres veces:

```
verify(spyAs, times(3)).isAuthenticated();
```

mock: es como un spy que sabe lo que está probando exactamente. Así, al propio mock, en la sección de aserciones, se le preguntará si ha ido bien o mal el test. El mock sabe el comportamiento de cómo se debe llamar al doble, cuántas veces se le ha llamado, con qué parámetros, etc. Es una de las formas más conocidas y usadas hoy en día por los desarrolladores ya que ofrece múltiples opciones para probar nuestro código.

Vamos a ver con Mockito dos ejemplos sobre cómo especificar el resultado que queremos que nos devuelva nuestro mock. Imaginemos que tenemos un servicio que devuelve una lista de productos de una tienda. Esa lista es del tipo **List<Product>.** La primera forma que veremos a continuación es ***type safe***, esto quiere decir que tiene en cuenta el tipo devuelto y por tanto, nos saldría un error en tiempo de compilación indicandonos que se espera una lista de productos y se está devolviendo una string:

```
when(productService.getProducts()).thenReturn("This should be a list of
products, not a string") //Shows an error
```

La segunda forma no es **type safe**, esto quiere decir que no tiene en cuenta el tipo devuelto y por tanto, no nos saldría ningún error en tiempo de compilación pero sí

al ejecutar el test, provocando su fallo:

```
doReturn("This should be a list of products, not a
string").when(productService).getProducts(); //Does not show any error
```

fake: es un tipo totalmente distinto a los anteriores. Un fake implementa los métodos con lógica de negocio, es como un simulador que puede ser muy sencillo o extremadamente complicado. Por ejemplo, si usamos una base de datos en memoria para simular una base de datos real, esta base de datos en memoria se considera un fake.

De forma coloquial, también es muy común denominar a todos estos dobles como “mocks”.

Recomendaciones

El principal objetivo de los tests es comprobar que todas las partes implicadas de una aplicación queden libres de errores de forma unitaria e integrada para prevenir problemas en sucesivas fases del ciclo de vida del proyecto.

FIRST

Si bien los propios tests deben perseguir también un buen diseño, para evitar que la propia infraestructura de tests se convierta en un problema, debería cumplir con el principio FIRST:

Fast: los tests deben ser de rápida ejecución, por eso debemos poner especial énfasis en implementar tests unitarios y, solo test de integración en aquellos casos en los que realmente necesitemos el contexto de un sistema externo para ser ejecutados. Si nombramos correctamente los tests de integración, podemos definir una fase concreta para la ejecución de los mismos dentro del ciclo de vida de Maven, pudiendo ahorrar la ejecución de tal fase en una build normal y recopilar estadísticas de cobertura independientes distinguiendo entre tests unitarios y de integración.

Independent: para facilitarnos la tarea de detección de errores es muy importante

que los tests sean independientes los unos de los otros. Para lograrlo debemos evitar que las salidas de unos tests sean utilizadas como entradas de otros y no debería importar el orden en el cual se vayan a ejecutar los tests, ya que cada ejecución debe ser independiente de la otra. Si tenemos una batería de tests de integración contra base de datos, debemos mantener la transaccionalidad en las operaciones, de modo que el entorno siempre quede consistente tras su ejecución.

Repeatable: deben soportar su ejecución más de una vez sin cambiar el resultado ni el estado del sistema independientemente de su entorno o contexto.

Self-validating: deben ser autoevaluables, es decir, que el propio test identifique si el test ha funcionado correctamente o no. Esta autoevaluación se realiza mediante aserciones (asserts).

Timely: deben escribirse en el momento oportuno, es decir antes del código de producción, y el motivo es muy simple: es más fácil hacer tests para un código que todavía no está escrito que para uno que ya ha sido creado, del mismo modo que es más fácil hacer crecer recto un árbol que todavía no ha brotado con una guía, que enderezar uno que tiene varios metros de altura.

Arrange - Act - Assert

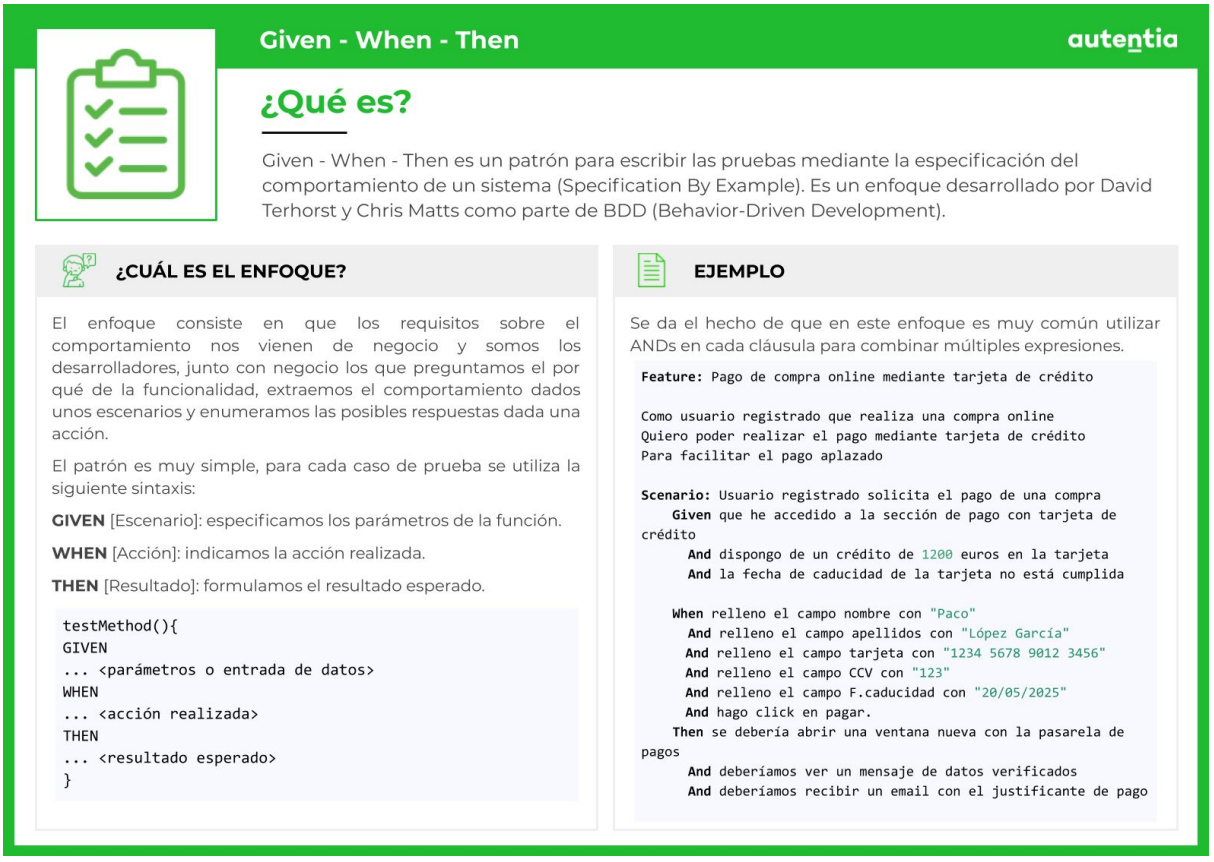
Todo test debería tener tres secciones claramente diferenciadas:

- **Arrange** o preparación: implica una serie de tareas de inicialización de las clases de servicio o preparación de los datos previo a la invocación a la lógica de negocio. Esto es lo que se denomina la fixture.
- **Act** o actuar: consiste en invocar a la lógica de negocio con los datos previamente preparados. La invocación a un método o función específica.
- **Assert** o afirmar: se comprueba que lo que se invoca coincide con el resultado esperado. Un test sin aserciones no es un test autoevaluable.

```
@Test
public void should_check_product_is_added_to_cart() {
   //Given
   Cart cart = new Cart();
   cart.addProduct(new Product("Autentia book"));
```

```
    //When
    String result = cart.getProductByName("Autentia book");

    //Then
    assertThat(result, is("Autentia book"));
}
```

Estas tres secciones coinciden con el también conocido **Given**, **When**, **Then** de BDD.



Entorno de ejecución

Depuración

En muchas ocasiones nuestro software no se comporta como esperamos o produce un error no controlado. Depurar nuestro código nos ayuda a detectar dónde está el fallo y así poder corregirlo.

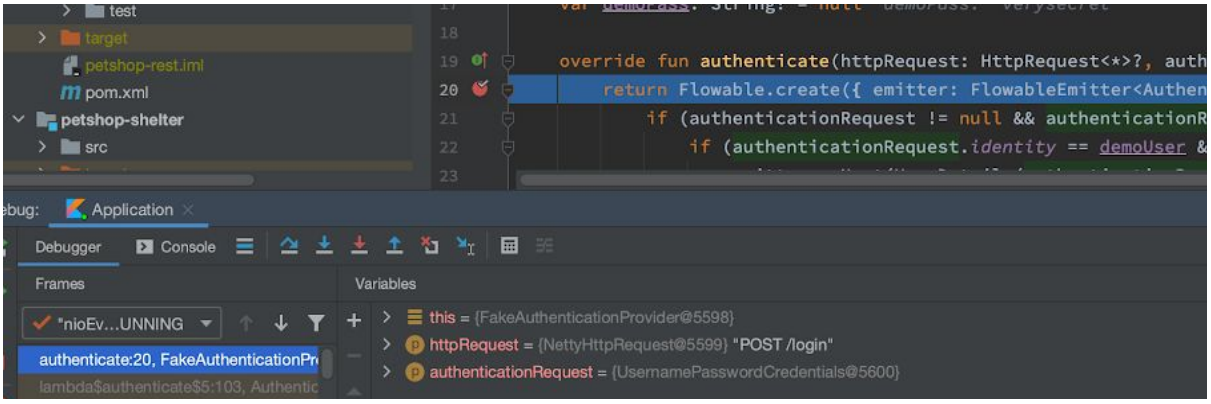
La depuración, conforme vamos añadiendo tests, debería ir desapareciendo, quedando acotada a situaciones anómalas que no puedan ser reproducidas por la batería de tests.

Breakpoints

Los puntos de ruptura, también llamados breakpoints, ayudan al desarrollador a parar la ejecución en un punto de código de manera que podamos inspeccionar el estado de la aplicación, continuar con la ejecución en la siguiente línea, en un nivel más (dentro del método que se va a invocar) o cancelar la ejecución actual.

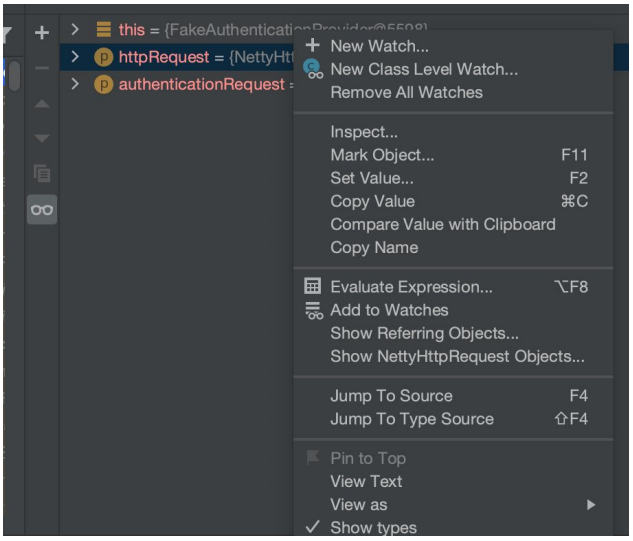
Los IDE ofrecen la posibilidad de añadir puntos de ruptura de manera sencilla e incluso condicionales, de manera que la ejecución se pare si se cumple una expresión.

Para que los breakpoints se disparen, la aplicación debe compilarse y levantarse en modo debug.



Observar variables

Una vez que un punto de ruptura se ha disparado, podemos observar el valor de las variables e incluso cambiar su valor en caliente. Esto es posible hacerlo porque Java es compatible con la JPDA (Java Platform Debugger Architecture), que es la que permite cambiar código en ejecución.



Gestión de logs

La gestión de logs es una parte fundamental en el desarrollo de nuestro software ya que proporciona información sobre posibles errores u otros datos que podrían ser de interés para resolver algún problema, ofreciendo una depuración rápida y un mantenimiento sencillo. Es una práctica común intercalar instrucciones de código que van informando del estado de la ejecución de las aplicaciones,

generando así un log. Entonces, ¿en qué consiste hacer logging o sacar trazas de una aplicación? En obtener un listado de mensajes que genera un sistema durante su ejecución. Ya sean operaciones que realizan los usuarios o lo que hacen los diferentes componentes de la aplicación.

Existen dos tipos de logs:

- **Logs de ejecución:** informan sobre distintos problemas en el código, aunque también pueden ser muy útiles para los administradores de sistemas.
- **Logs operacionales:** dan información sobre el funcionamiento de la aplicación. Es decir, informan de eventos a nivel semántico o de negocio.

En Java existe una librería llamada Log4Java (log4j) que nos permite gestionar estas tareas de una forma simple. Para mostrar mensajes de log en una clase, se debe crear un objeto de tipo Logger. También se debe tener en cuenta que no todos los mensajes de log de una traza tienen la misma importancia y estos se clasifican en niveles de criticidad:

OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL

Cuantos más datos de traza, más fácil será encontrar los problemas. ¿Por qué no mostrar el nivel máximo siempre? Lo primero, un exceso de información puede llegar a ser contraproducente. Segundo, el rendimiento de la aplicación se puede ver afectado. Por último, decir que el tamaño para almacenar los logs generados puede llegar a ser muy grande, algo a tener en cuenta a la hora de mantener nuestros entornos de producción.

El siguiente código de ejemplo consulta un array en memoria en una posición equivocada para forzar una excepción en tiempo de ejecución:

```
import org.apache.log4j.*;

public class LoggerExample {
    private static Logger final LOGGER =
Logger.getLogger(LoggerExample.class);

    public static void main(String[] args) throws SecurityException,
IOException {
        LOGGER.info("info message");
        int[] int_vector = {1,2,3,4};
```

```
        int i = 6;
        try {
            int_vector[i];
        } catch (ArrayIndexOutOfBoundsException e) {
            LOGGER.error("Error!", e);
        }
    }
}
```

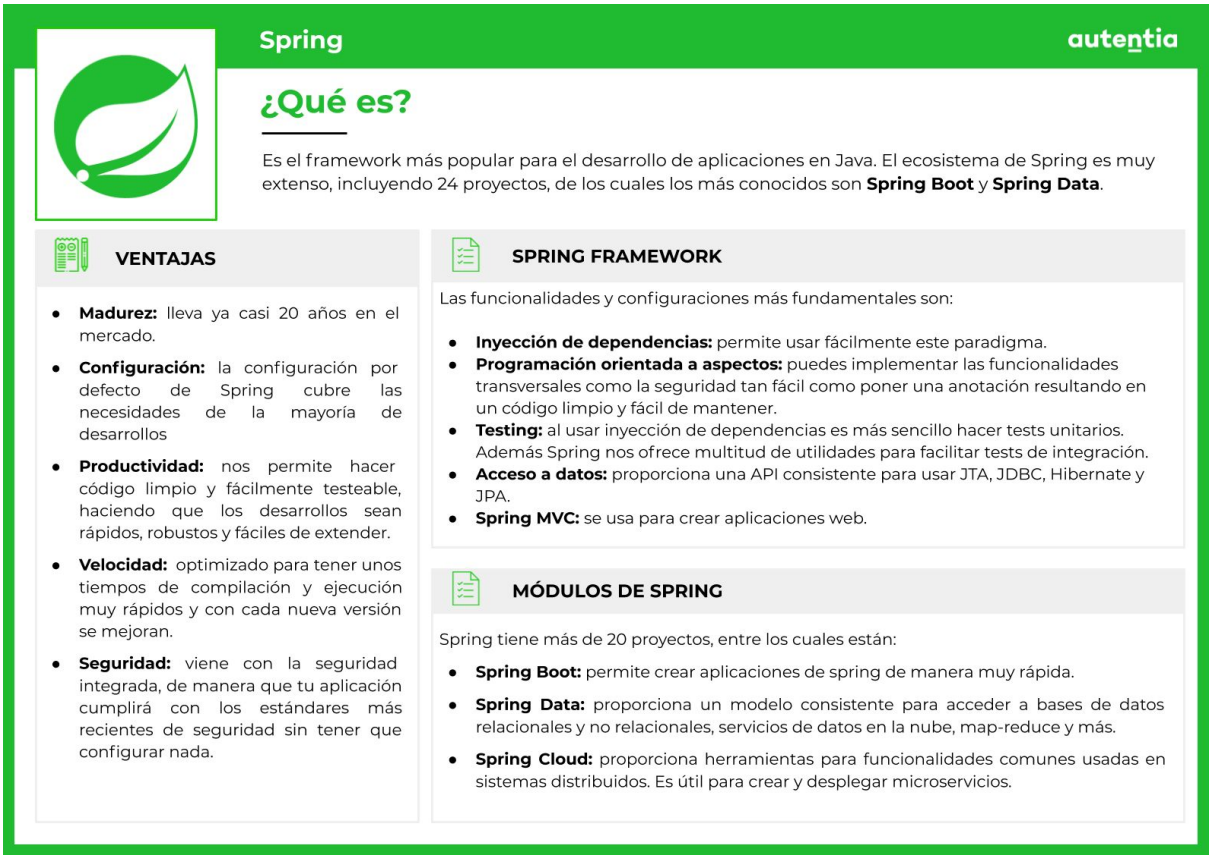
Es importante nombrar a SLF4J (Simple Logger Façade For Java), que es una fachada de un conjunto de librerías de logging como: java.util.logging, logback o log4j, permitiendo trabajar con sus diferentes implementaciones a través de una abstracción. Para resumir, SLF4J no reemplaza a log4j ni a otro framework de logs si no que funcionan juntos.

Parte 3

El mundo de los microservicios

Introducción a Spring

Spring Framework es una solución que nace con el objetivo de unificar y facilitar la construcción de aplicaciones en la plataforma Java, incluyendo soporte para Kotlin o Groovy. Es un framework modular, lo que nos permite utilizar sólo aquellos módulos que realmente necesitamos. Su característica más importante y sobre lo que se fundamenta, es el contenedor de **Inversion of Control (IoC)** del que hablaremos más adelante**.** El framework soporta gestión de transacciones declarativas, acceso remoto mediante RMI o servicios Web y varias opciones para persistir sus datos. Ofrece un framework MVC (Model View Controller) completo y le permite integrar AOP (Aspect Oriented Programming) de forma transparente en su software. Spring permite a los desarrolladores centrarse principalmente en la **lógica de negocio** a la hora de construir aplicaciones mientras que él asume el peso de las piezas de infraestructura.



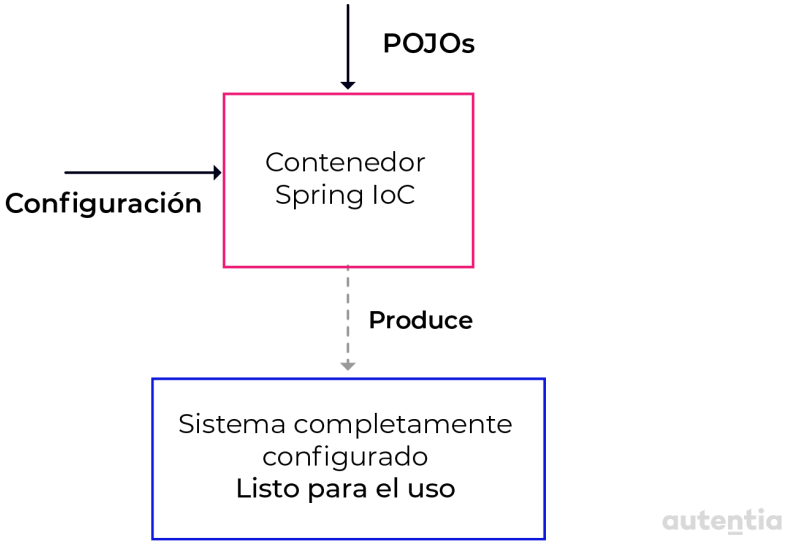
Spring IoC e Inyección de Dependencias

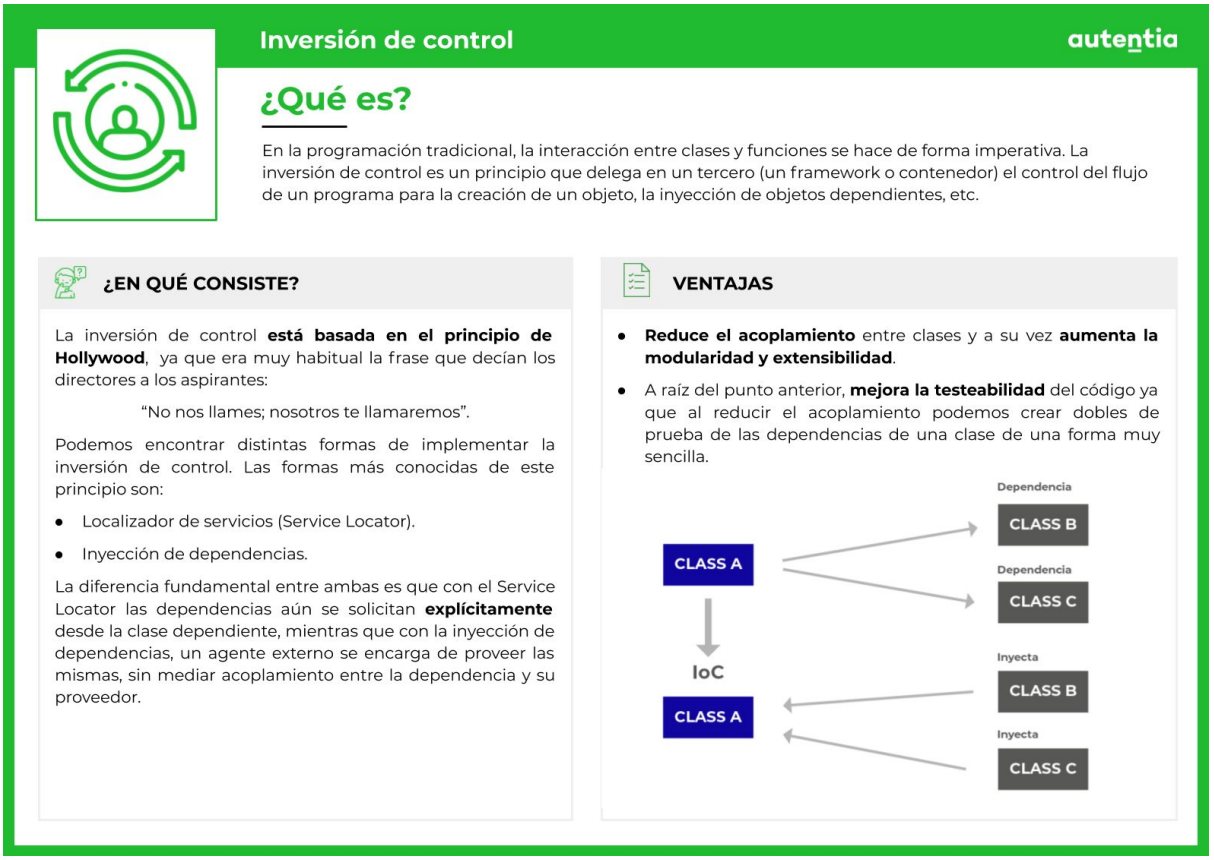
Como se puede consultar en nuestra guía de principios y patrones de diseño, la inversión de control se utiliza en el diseño orientado a objetos para **delegar en un tercero diferentes tipos de flujos de control** para lograr un bajo acoplamiento. **Uno de los patrones que implementan la inversión de control es la Inyección de Dependencias**, de modo que la creación y establecimiento de las dependencias de una clase es controlada por un framework o contenedor.

En el caso de Spring, es el **contenedor** (Spring IoC container) el que **asume la responsabilidad de la gestión del ciclo de vida de las dependencias**, conocidas como **Beans**, y de su **inyección** en aquellas clases que las necesiten. Dicha inyección se realiza a través del constructor o seteo de la propiedad en las clases dependientes y se representa a través de la

interfaz ApplicationContext.

El siguiente diagrama muestra cómo funciona Spring. A partir de las clases de la aplicación y una configuración declarada en XML, Java o anotaciones, se crea e inicializa la instancia del ApplicationContext que gestionará el ciclo de vida los beans definidos en nuestra aplicación.





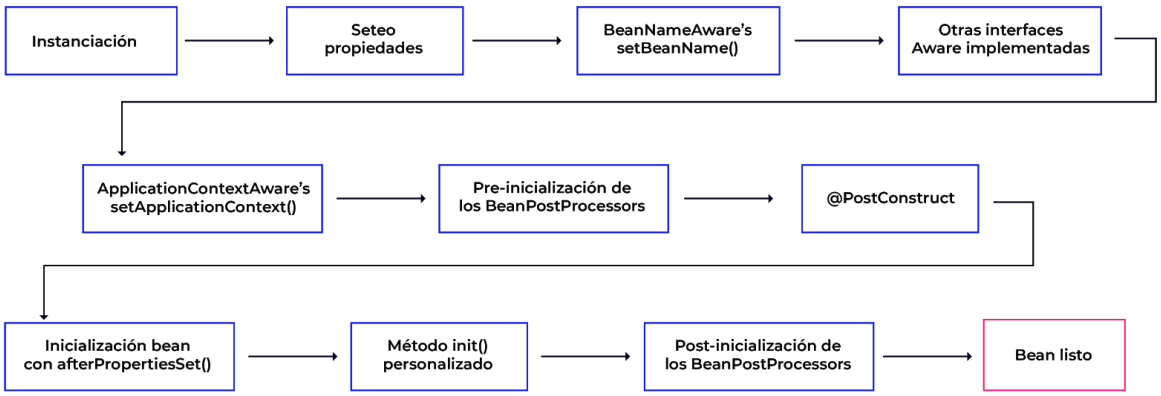
Spring Beans

Spring denomina beans a los objetos cuyo ciclo de vida es gestionado por el contenedor de dependencias.

Ciclo de vida

Es a la hora de levantar nuestra aplicación cuando Spring crea el contexto y los beans se inicializan. Los beans tienen un ciclo de vida para su creación definido en las siguientes fases:

- **Instanciación:** se crea la instancia de la clase, llamando a su constructor y se asigna valor a las propiedades igual que se haría de manera programática.
- **Configuración** del bean a través de las instancias de la interfaz Aware. Se configuran valores como el nombre del bean, la instancia de ApplicationContext que maneja el bean, etc. Se pueden crear implementaciones propias de Aware para cambiar el comportamiento del framework.
- Ejecución de los métodos anotados con **@PostConstruct**.
- **Inicialización del bean** a través de la ejecución del método afterPropertiesSet() de la interfaz InitializingBean y método init() implementado.
- El bean está listo y etiquetado para su uso en el contenedor de dependencias.



La destrucción de un bean tiene también un ciclo de vida definido:

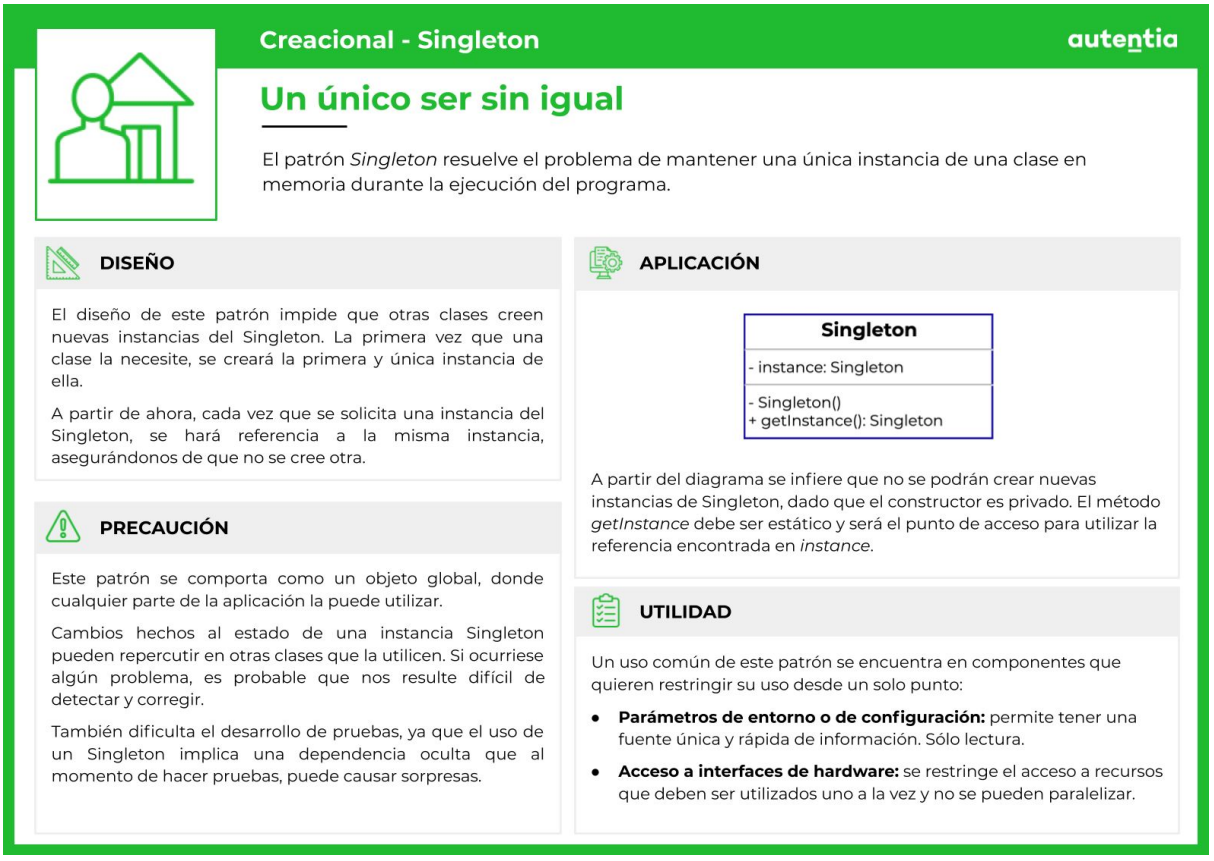
- Ejecución de los métodos anotados como **@PreDestroy**.
- Ejecución método **destroy()** de la interfaz DisposableBean.

- Ejecución del método **destroy()** implementado.

Tipos de bean

Existen cinco ámbitos diferentes que clasifican los beans:

- **Singleton:** solo una instancia del bean existe en el contenedor. Es el comportamiento por defecto. Todos aquellos beans que referencian a beans de tipo singleton apuntan a esta única instancia.



- **Prototype:** pueden existir múltiples instancias de un bean definido. El contenedor de Spring crea tantas instancias como veces sea definido desde otros beans como dependencia.
- **Request:** el contenedor de Spring creará una nueva instancia del bean por cada nueva petición HTTP.
- **Session:** el contenedor creará una nueva instancia del bean durante el ciclo de vida de una sesión HTTP.
- **Global session:** similar a session pero orientado a portlets. Se creará una nueva instancia de bean durante el ciclo de vida de una *global session*.
- **Personalizado:** se pueden definir ámbitos personalizados para los beans a través del método *registerScope* de la interfaz *configurableBeanFactory*.

Tipos de configuración

Nuestras clases pueden definirse como beans a partir de configuración, a través de fichero XML, con anotaciones o también de manera programática.

XML

Antes de Spring 3.0, la definición de beans se hacía a través de XML, normalmente el fichero de configuración tenía el nombre de *applicationContext.xml*. Aunque todavía se puede utilizar la configuración con XML, ha entrado en desuso a favor de la definición de beans con anotaciones y configuraciones programáticas. Un ejemplo de applicationContext es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="teacher" class="com.autentia.Teacher"></bean>
    <bean id="course" class="com.autentia.Course">
        <property name="teacher" ref="teacher"></property>
    </bean>
</beans>
```

Anotaciones

Las clases que queramos definir como beans podemos marcarlas a través de las anotaciones que nos proporciona el paquete *org.springframework.stereotype*. En este apartado vamos a repasar las anotaciones más comunes que podemos utilizar para etiquetar nuestras

clases como beans:

- **@ComponentScan**: esta anotación acompaña a la anotación @Configuration. Indica los paquetes que deben escanearse para identificar las clases etiquetadas como beans. Si no se especifica el atributo basePackage, por defecto se escanea el paquete donde se encuentra y sus subpaquetes. En el siguiente ejemplo, se indica que deben escanearse todas las clases del paquete com.autentia.courses y sus subpaquetes:

```
@Configuration
@ComponentScan(basePackage = com.autentia.courses)
class AppConfig {}
```

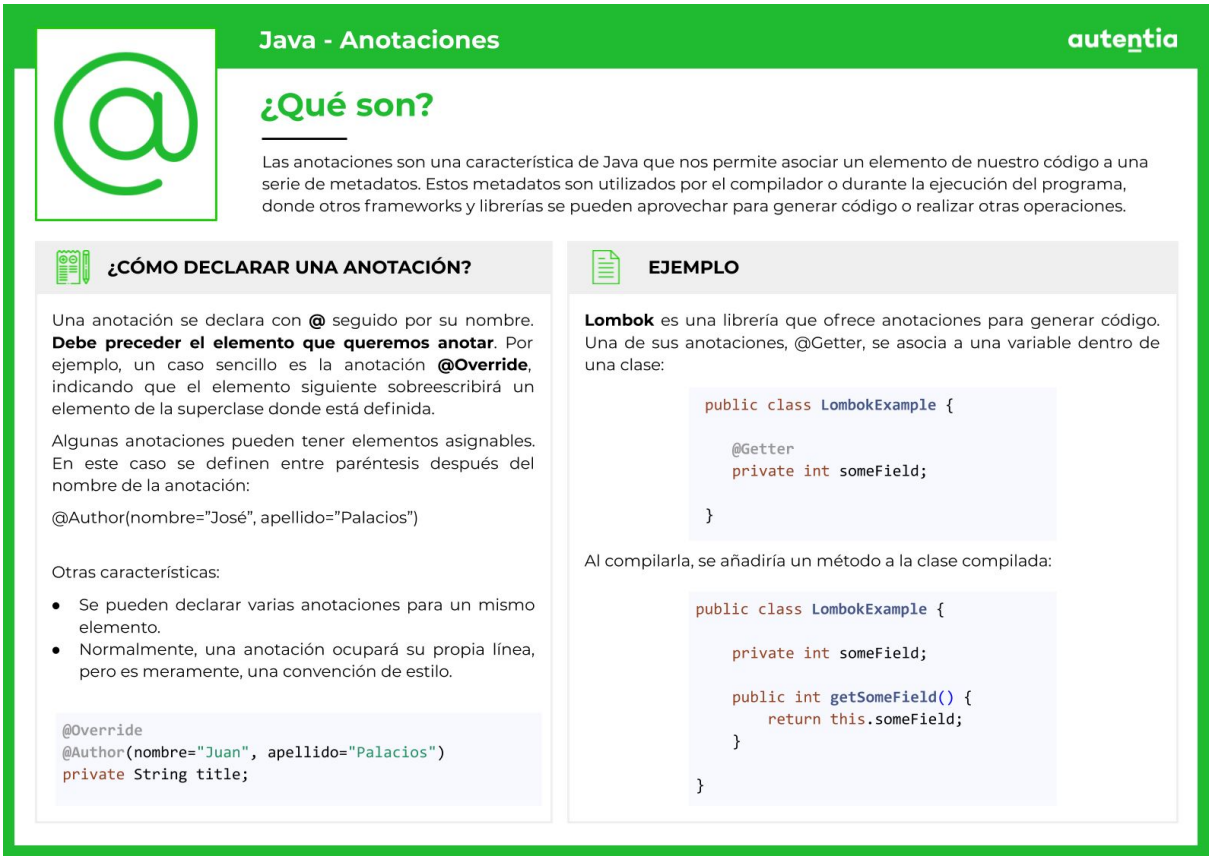
- **@Component:** a nivel de clase, es la anotación general para indicar que una clase es un bean.
- **@Controller, @Service, @Repository:** a efectos de identificación de bean es lo mismo que @Component. Estas anotaciones permiten clasificar nuestros beans según el contexto de la clase: clases de controlador, de capa de servicio o capa DAO/repositorio.
- @**Autowired:** es la anotación que, a nivel de constructor o propiedad de la clase, se utiliza para indicar que Spring debe ser el encargado de inyectar esos beans como dependencia. ¿Cuál es la diferencia entre inyectar las dependencias por propiedad o constructor? En términos de inyección de dependencias se hace igualmente, pero al hacer la inyección por constructor podemos hacer nuestras clases inmutables sin detrimento de su testabilidad al poder sustituir las dependencias por dobles de prueba.

Java

La definición de beans también puede hacerse de manera programática. Para ello, es necesario crear una clase y etiquetarla con la anotación

@Configuration. Dicha clase contiene métodos que crean las instancias de clase a tratar como beans. Dichos métodos deben ser etiquetados con **@Bean:**

```
@Configuration
class AppConfiguration {

    @Bean
    SlackNotifier slackNotifier() {
        return new SlackNotifier();
    }
}
```



Resolución de conflictos entre beans

Cuando tenemos múltiples implementaciones de una interfaz, al utilizar la anotación @Autowired, Spring no sabrá que bean resolver y un error

parecido a este saldrá por consola:

```
Field “campo anotado con @Autowired” in ... required a single bean,
but 2 were found
```

Por ejemplo: tenemos una interfaz ‘CarService’ y dos implementaciones ‘ElectricCarService’ y ‘HybridCarService’.

```
public interface CarService { ... }

@Service
public class ElectricCarService implements CarService { ... }

@Service
public class HybridCarService implements CarService { ... }
```

```
@Controller
public class CarController {

      @Autowired
      private CarService carService;

      ...
}
```

Spring no sabrá qué implementación inyectar en el campo ‘carService’ y nos informará de ello con un error.

Para evitar el error tenemos dos soluciones:

1. **Anotar el bean** que queremos con **@Primary** para indicar que cuando no se especifica el bean a inyectar, este se debe usar por defecto. Por ejemplo, si queremos que por defecto se inyecte la implementación ‘ElectricCarService’:

```
@Service
@Primary
```

```
public class ElectricCarService implements CarService { ... }
```

2. **Anotar el campo** sobre el que usamos la anotación @Autowired con la anotación **@Qualifier** y pasar entre paréntesis el nombre de la implementación que queremos que spring inyecte. Por ejemplo, si queremos la implementación “HybridCarService”:

```
    @Autowired
    @Qualifier("hybridCarService")
    private CarService carService;
```

Spring Data

Introducción

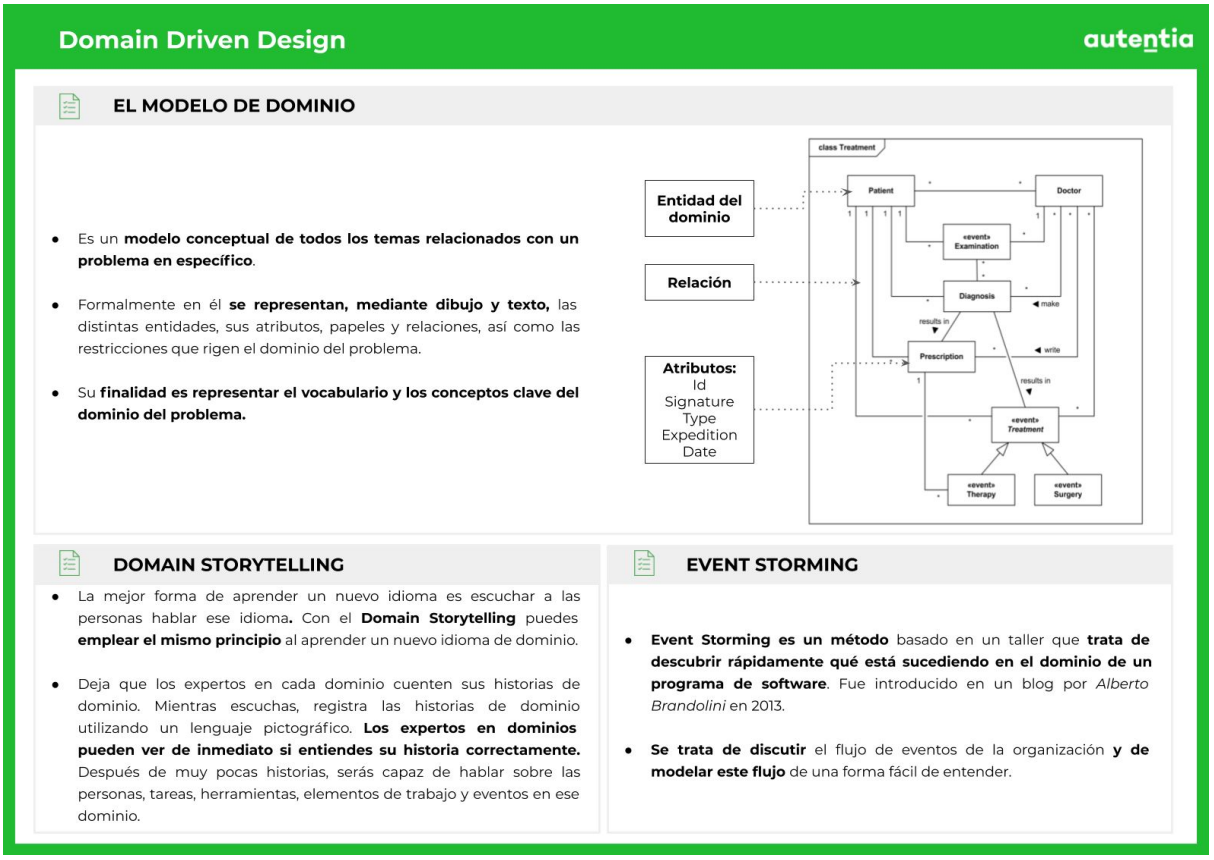
Spring Data es un módulo de Spring cuyo propósito es **unificar y facilitar el acceso a tecnologías de acceso a datos**, tanto a bases de datos relacionales como a NoSQL así como a servicios basados en la nube, etc. Aporta la mayor parte del código que tendríamos que implementar para trabajar con esas tecnologías y está dividido a su vez en varios módulos específicos de la tecnología que se va a utilizar. Algunos de ellos son:

- Spring Data JDBC.
- Spring Data JPA.
- Spring Data Rest.
- Spring Data Redis.
- Spring Data for Apache Cassandra.
- Spring Data Elasticsearch.

La lista completa con los módulos mantenidos por Spring Boot y la comunidad se puede ver aquí.

Conceptos

Spring Data se inspira en los conceptos "Aggregate", "Aggregate Root" y "Repository" de Domain Driven Design.



1. **Aggregate**: es un conjunto de entidades de dominio (Order, OrderLine, Post y Comment) que están estrechamente relacionadas entre sí y forman un todo, un único conjunto. Por ejemplo, un post puede tener varios comentarios. Un comentario no puede existir sin tener asociado un post.
2. **Aggregate Root**: cada agregado que se define tiene una entidad raíz de la que colgarán el resto de entidades. Si seguimos con la analogía de los posts, un usuario puede publicar varios posts y cada post tener varios comentarios. El usuario es la entidad raíz o raíz del agregado. Es importante destacar que una entidad raíz no tiene otra entidad de la que dependa, si no que es la raíz de todas las demás entidades.
3. **Repository**: un repositorio trabaja con el Aggregate Root para obtener los agregados. En nuestra analogía no tendría sentido recuperar los posts sin saber a qué usuario pertenecen los post publicados. Si una entidad que forma parte de un agregado se tiene que persistir en la

base de datos se hará a través del Aggregate Root usando el repositorio de esa entidad raíz para realizar la operación.

¿Qué es JDBC?

JDBC (Java Database Connectivity) es una API de Java para simplificar la conexión a una base de datos, ofreciendo la misma interfaz para diferentes bases de datos. JDBC funciona a bajo nivel y por tanto el desarrollador se tiene que encargar de la mayoría de operaciones, de abrir y cerrar la conexión, gestionar las excepciones, ejecutar las consultas, etc. Lo que lleva a un código repetitivo y propenso a errores. Por ejemplo:

1. Creamos la tabla en la base de datos.

```
CREATE TABLE User(
   Id   INT NOT NULL AUTO_INCREMENT,
   first_name VARCHAR(50) NOT NULL,
   last_name  VARCHAR(50) NOT NULL,
   PRIMARY KEY (ID)
);
```

2. Recuperamos los usuarios de la base de datos usando JDBC.

```
public List<User> findAll() throws SQLException {
  try {
    Class.forName("com.mysql.jdbc.Driver");
    Connection con = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/db_name","db_username","db_password")
;
    Statement stmt = con.createStatement();
    ResultSet result = stmt.executeQuery("select * from users");

    List<User> users = new List()
    while (result.next()) {
      Long id = result.getLong("id");
      String firstName = result.getString("first_name");
      String lastName = result.getString("last_name");
```

```
      // Construimos el objeto User
      User user = new User(id, firstName, lastName);
      users.add(user);
    }

    con.close();
    return users
  } catch(SQLException e) {
      System.out.println(e);
    }
  }
```

En el código de arriba hemos definido dentro de un bloque try-catch, la conexión con la base de datos, hemos creado y ejecutado la consulta SQL y hemos iterado sobre el resultado de esa consulta para obtener todos los usuarios. Finalmente, si todo ha ido bien, cerramos la conexión a la base de datos y devolvemos los usuarios, en caso contrario JDBC lanzará la excepción “SQLException” que imprimimos por consola.

Spring JDBC

Spring JDBC es una abstracción sobre JDBC para ocuparnos únicamente de:

1. Definir los parámetros de conexión.
2. Especificar la operación (consulta, actualización, etc.).
3. Realizar alguna tarea con esos resultados.

Spring se encargará del resto: abrir y cerrar la conexión a la base de datos, gestionar las excepciones, iterar sobre los resultados, etc.

Para realizar la operación hay tres cosas que son importantes.

1. Añadir la librería con el driver de JDBC para la base de datos hacia la que vamos a enviar las consultas SQL.

2. Configurar la conexión a esa base de datos en el archivo de configuración de Spring. Por defecto “application.properties”

```
## MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/db_name
spring.datasource.username=db_username
spring.datasource.password=db_password
```

3. Crear el POJO de la tabla a la que vamos a mapear el resultado de la consulta SQL.

```
public class User {
    private Long id;
    private String firstName;
    private String lastName;
    // constructors, getters, setters
}
```

4. Crear el repositorio con la lógica para realizar las operaciones en la base de datos. Spring Boot creará y configurará la clase ‘JdbcTemplate’ con la conexión que se ha especificado en el paso anterior y el driver JDBC de la librería instalada.

```
@Repository
public class JdbcUserRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public List<User> findAll() {
        return jdbcTemplate.query(
                "select * from users",
                (rs, rowNum) ->
                        new User(
                                rs.getLong("id"),
                                rs.getString("first_name"),
                                rs.getString("last_name")
```

```
                        )
        );
    }
}
```

Spring Data JDBC

Spring Data JDBC es una abstracción de Spring Data sobre Spring JDBC. Aplicando los mismos conceptos de “Repository”, “Aggregate”, “Root Aggregate” de Spring Data, éste implementará la mayor parte del código necesario. Por ejemplo:

```
public class User {

    @Id
    private Long id;
    private String firstName;
    private String lastName;

    // constructors, getters, setters
}
```

Spring Data JDBC utiliza, por defecto, una estrategia que asigna POJOs a tablas de la base de datos y atributos a nombres de columnas. Para ello, los nombres “CamelCase” se mapean a “snake_case”.

Por ejemplo, “firstName” se mapea a “first_name”.

```
@Repository
public interface UserRepository extends CrudRepository<User, Long>
{}
```

La línea de código de arriba es todo lo que necesitamos para crear las consultas CRUD de la tabla User.

Gestión de la transaccionalidad

Una transacción es un bloque de código que accede y posiblemente modifica el contenido de una base de datos. Si todo ha ido bien, los datos se persisten en la base de datos haciendo un commit, pero si algo ha ido mal y recibimos una excepción, se hará un rollback automáticamente de los cambios y volvemos al estado que teníamos antes de ejecutar el código. Para gestionar las excepciones, Spring proporciona la anotación **@Transactional** que podemos aplicar a un método o a la clase para indicar que todos sus métodos son transaccionales.

Para habilitar la gestión de transacciones con Spring debemos usar la anotación **@EnableTransactionManagement**, en Spring Boot la gestión de transacciones es configurada automáticamente. Se puede hacer a nivel general, por ejemplo:

```
@Configuration
@EnableTransactionManagement
class AppConfig {}
```

La anotación **@Transactional** soporta algunos parámetros para configurar la transacción, los más comunes son:

1. **Propagation**: define cómo se relacionan las transacciones entre sí, las opciones más comunes son:
    - **PROPAGATION_REQUIRED**: es la opción por defecto. Si un primer método abre la transacción, el segundo método la aprovecha, por lo que ambos están dentro de la misma transacción.
    - **REQUIRES_NEW**: el código siempre se ejecuta abriendo una transacción nueva, suspendiendo la transacción actual, si existe. Permite solo hacer rollback del método que ha fallado.
2. **ReadOnly**: es un boolean que marca la transacción de solo lectura,

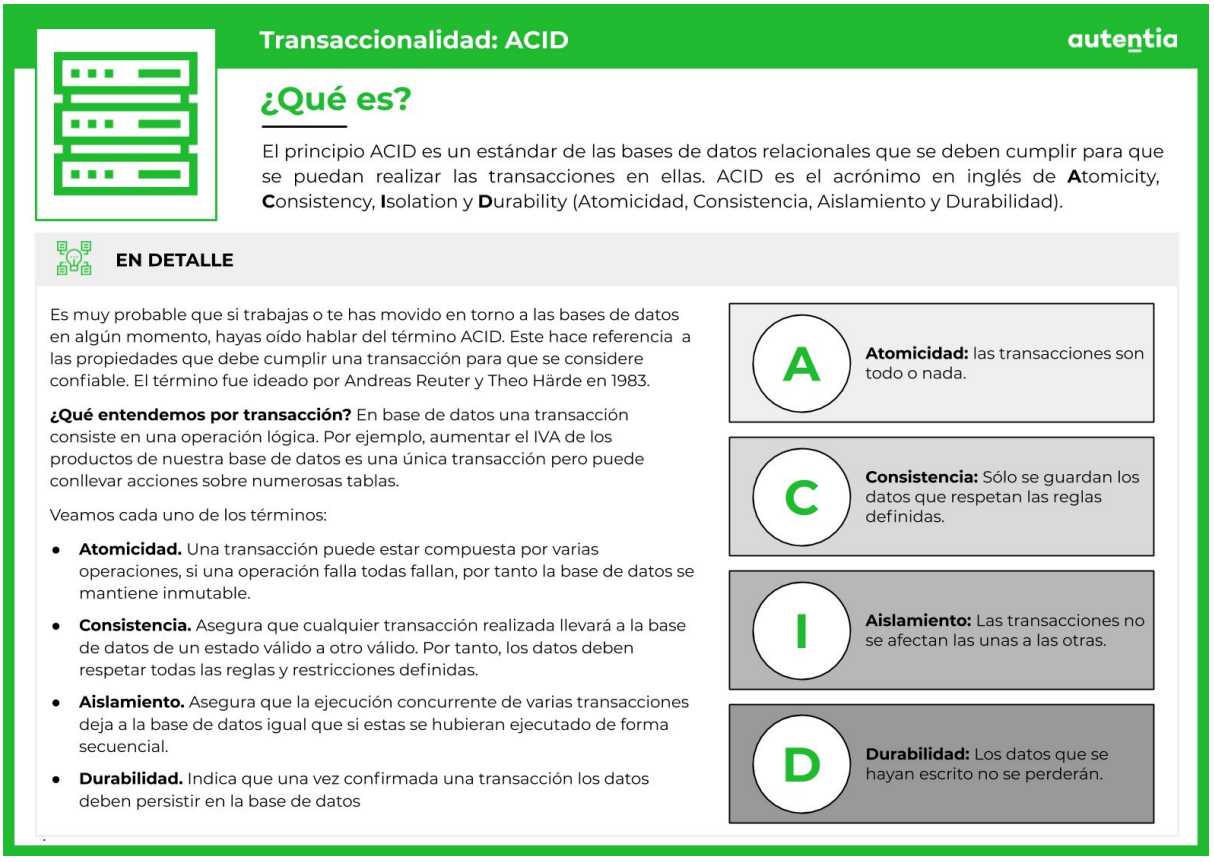
tiene beneficios en términos de rendimiento.

3. **RollbackFor**: se usa para indicar qué excepciones deben hacer un rollback de la transacción, por ejemplo:

```
@Transactional(rollbackFor = {UserNotFoundException.class})
```

4. **NoRollbackFor**: Se usa para indicar qué excepciones no deben hacer un rollback de la transacción.

Todas las opciones de configuración se pueden consultar en la documentación de @Transactional de Spring.



Connection Pooling

Connection Pooling es una manera de acceder a una base de datos de forma que se reduzca la carga de trabajo al realizar conexiones u

operaciones de lectura y/o escritura con la base de datos.

La razón de utilizar este patrón es que los pasos implicados en una conexión a una base de datos corriente, comprenden en su conjunto una operación bastante costosa, desde abrir la conexión y un socket, pasando por leer y/o escribir los datos, hasta el cierre de la conexión y el socket.

Connection Pooling vendría a ser una implementación de una caché de conexión de base de datos, de la cual podríamos reutilizar una cantidad determinada de conexiones existentes. Esto nos permitirá ahorrar una cantidad de carga de trabajo considerable, consiguiendo con ello un incremento en el rendimiento de nuestra aplicación.

¿Qué ofrece Spring respecto a Connection Pooling?

La manera en la que Spring se conecta a una base de datos es a través de un *DataSource*.

DataSource es una clase que se encarga de crear conexiones generalizadas, y es además, parte de la especificación de JDBC. De esta manera, Spring oculta la administración de conexiones y transacciones del resto de código de la aplicación.

La forma recomendada de crear una *DataSource* en Spring es mediante la clase *DataSourceBuilder*, dentro de una clase con la anotación de @Configuration. Veamos un ejemplo:

```
@Configuration
public class AplicationConfig {
      @Bean
      public DataSource buildDataSource() {
            DataSourceBuilder dsb = DataSourceBuilder.create();
            dsb.driverClassName("org.h2.Driver");
            dsb.url("jdbc:h2:file:/test/db");
            dsb.username("user");
            dsb.password("pass");
```

```
            return dsb.build();
        }
}
```

Además, Spring nos ofrece más implementaciones y clases de utilidad como las siguientes:

- **DataSourceUtils**: nos provee métodos para obtener conexiones desde JNDI y cerrar conexiones si es necesario.
- **SmartDataSource**: es una interfaz usada para implementarse en clases que se conectan a una base de datos relacional, y que nos da la posibilidad de saber si la conexión debe ser cerrada después de una transacción.
- **AbstractDataSource**: si quiere escribir su propia implementación, debería extender de esta clase, contiene todo el código común a las implementaciones.
- **SingleConnectionDataSource**: es una implementación para una sola conexión que no se cierra después de cada uso.
- **DriverManagerDataSource**: es una implementación que configura una interfaz básica de JDBC. Es muy útil para propósitos de testing.
- **TransactionAwareDataSourceProxy**: se trata de un proxy para un origen de datos, el cual añade el conocimiento que tiene Spring sobre las transacciones que administra.
- **DataSourceTransactionManager**: es una implementación para una sola fuente de datos.

Spring MVC

Introducción

Spring MVC es un framework web basado en el patrón de arquitectura Modelo-Vista-Controlador (MVC).

Su diseño gira en torno al **DispatcherServlet**, elemento que actúa de *Front Controller*, procesando solicitudes HTTP entrantes y redirigiéndolas al resto de controladores de la aplicación. Se ayuda de la configuración de Spring para saber sobre qué componentes puede delegar.

El DispatcherServlet ha de ser declarado y configurado en Spring. Existen, entre otras, dos opciones de configuración principales:

- **A través del archivo web.xml:** este archivo guarda la configuración de los servlets que van a componer una aplicación web en Java. En este caso en concreto, un ejemplo del contenido del archivo sería el siguiente:

```
<servlet>
  <servlet-name>dispatcherServlet</servlet-name>
  <servlet-class>
     org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcherServlet</servlet-name>
  <url-pattern>/</url-pattern>
<servlet-mapping>
```

Se crea un servlet llamado “dispatcherServlet” que será una instancia de la clase `org.springframework.web.servlet.DispatcherServlet`.

Mediante “<servlet-mapping>” se indica qué URLs van a gestionar el front controller: en este caso, todas las peticiones que comiencen por “/”.

- **A través de la configuración de Java:** sin necesidad de configurar ningún archivo gracias a `WebApplicationInitializer`.

```
public class MyDispatcherServlet implements
WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
      AnnotationConfigWebApplicationContext context =
        new AnnotationConfigWebApplicationContext();
      context.register(AppConfig.class);

      container.addListener(new ContextLoaderListener(context));

      AnnotationConfigWebApplicationContext dispatcherContext =
        new AnnotationConfigWebApplicationContext();
      dispatcherContext.register(DispatcherConfig.class);

      ServletRegistration.Dynamic dispatcher =
        container.addServlet("dispatcherServlet",
        new DispatcherServlet(dispatcherContext));
      dispatcher.setLoadOnStartup(1);
      dispatcher.addMapping("/");
    }
 }
```

Primero se carga el contexto de Spring registrando una clase de configuración mediante `AnnotationConfigWebApplicationContext`. Tras ello, se añade el listener ContextLoaderListener al ServletContext para gestionar el ciclo de vida del contexto de Spring. Finalmente, se registra el DispatcherServlet y se mapean las URLs deseadas.

La configuración del DispatcherServlet era obligatoria mediante el archivo

web.xml hasta Spring 3.0. Tras esa versión, se fueron introduciendo distintas configuraciones para poderlo declarar desde el código.

Modelo, vista, controlador

El DispatcherServlet no sólo actúa como Front Controller escogiendo un controlador al que enviar peticiones entrantes, sino que también se tiene que comunicar con el resto de elementos que componen la arquitectura del sistema. Más adelante se explicará en profundidad cómo se procesa una petición entrante y el recorrido que realiza por toda la aplicación, pero en esta sección se van a explicar qué elementos componen la arquitectura de Spring MVC y cómo se declaran.

- **Modelo:** elemento que se encarga de encapsular los datos de la aplicación. Normalmente, vienen representados por objetos POJO, que definen una entidad.
- **Vista:** representa los datos del modelo en un formato específico, generalmente .html. Representa lo que va a ver el usuario a través de la interfaz del navegador.
- **Controlador:** actúa de puente entre la vista y el modelo. Procesa las peticiones entrantes, forma una respuesta apoyándose en el modelo y envía la respuesta a la vista para que esta se encargue de renderizarla.

Para identificar cada elemento con mayor facilidad veamos un ejemplo en código:

```
@Controller
public class HomeController {

 @RequestMapping(value ="/home", method = RequestMethod.GET)
 public String home(Model model) {
   String msg = "Hello World";
   model.addAttribute("msgInView", msg);
   return "home";
```

```
 }
}
```

Mediante la anotación **@Controller** se indica que la clase es un controlador. Cuando se llame a la URL "/home", se ejecutará el método `home()`, el controlador gestiona la petición creando un modelo que contiene un String con el mensaje "Hello World" y lo asocia a la variable "msgInView". Este modelo se envía a la vista "home", que es lo que devuelve el método. En la aplicación habrá un template .html con el nombre "home.html" que representa la vista. El contenido es el siguiente:

```
<html>
<head>
   <title>Home</title>
</head>
<body>
  <p>${msgInView}</p>
</body>
</html>
```

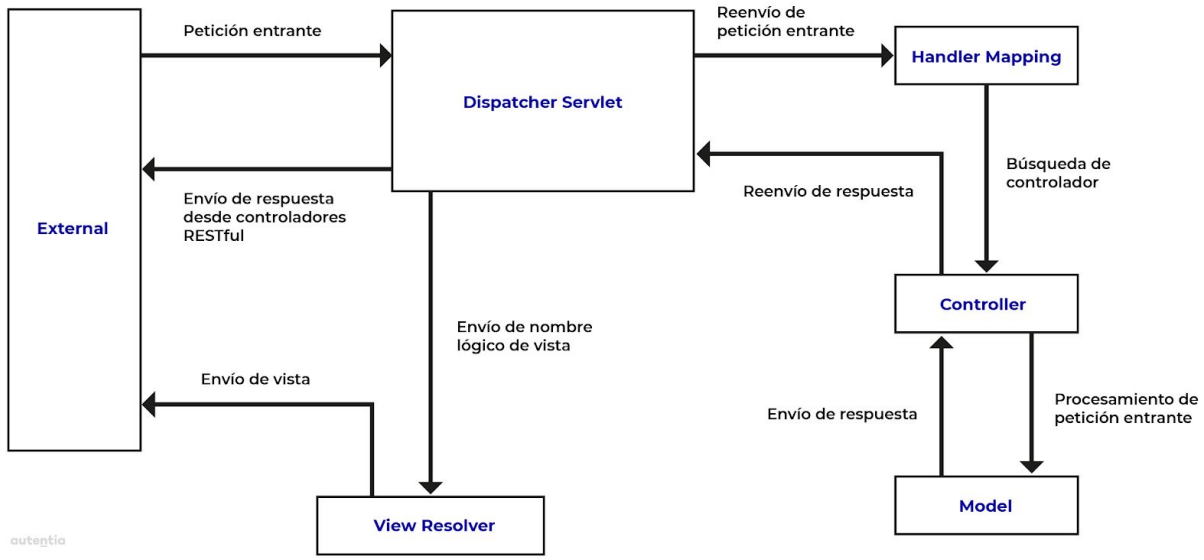
Esto mostrará una interfaz con el mensaje "Hello World" en el navegador.

En Spring MVC existen distintas clases y anotaciones para indicar que un componente es un controlador. **@Controller** es la notación clásica, pero a partir de Spring 4.0 se introdujo **@RestController**, que es la fusión de @Controller y @ResponseBody. Tradicionalmente, Spring MVC cumplía estrictamente con lo que es la arquitectura MVC: un controlador que crea un modelo que va a ser expuesto en la vista tras un proceso de renderizado. Sin embargo, con la irrupción de los servicios REST, los controladores en Spring MVC comenzaron a tener otro papel y se eliminó esa concepción de renderizar las vistas, convirtiéndose en componentes que simplemente **devuelven datos,** ya sea a través de un archivo JSON, a través de un archivo XML o simplemente, a través de una variable, sin

proceso de renderizado.

Flujo de ejecución

Vamos a analizar con exactitud cómo se procesa una petición HTTP según la arquitectura que provee Spring MVC, siguiendo el flujo de ejecución desde que se realiza la petición hasta que se recibe la respuesta a la misma.



Una petición HTTP llega al **DispatcherServlet** y éste decide a qué controlador enviar esta petición en base a una consulta a la interfaz **HandlerMapping**. HandlerMapping es la interfaz encargada del mapeo entre una solicitud HTTP y un determinado controlador. Spring MVC proporciona algunas implementaciones de esta interfaz o bien se puede crear una personalizada.

Dicha petición llega al controlador elegido y éste se encarga de gestionarla, creando un modelo. A parte de mapear el modelo, también se encarga de devolver el **nombre lógico** de la vista que se va a mostrar por la interfaz y que va a representar los datos mapeados en el modelo.

Ambas, modelo y nombre de vista, se envían de vuelta al **DispatcherServlet**

y éste se encarga de enviarlos a otro componente, el **ViewResolver**. La clase ViewResolver se encarga de mapear el nombre lógico de las vistas a vistas físicas, representadas normalmente por templates (.html, .xls...). Una vez que se tiene la vista física, se muestra su contenido, constituyendo la respuesta a la solicitud HTTP.

Este es el flujo de ejecución tradicional pero si se está desarrollando una API REST puede que no se quiera que el controlador mapee el modelo y devuelva una vista, si no que únicamente se quiere devolver una variable o un archivo JSON. En este flujo de ejecución alternativo **no** se consulta el **ViewResolver** y directamente, el controlador RESTful envía la respuesta de la petición al DispatcherServlet y este la devuelve al elemento externo que la ha solicitado.

Rest

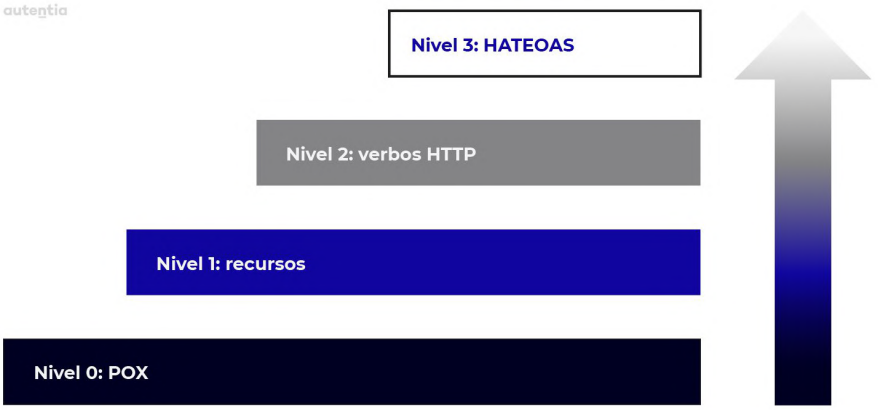
Un servicio REST ofrece operaciones CRUD (creación, lectura, actualización y borrado) sobre recursos (items de información) del servidor web, el cual se aprovecha de todos los aspectos del protocolo http y dicha información se intercambia en formato JSON o XML.

Para implementar una API REST con Java se puede usar:

- JAX-RS
    - Estándar JEE.
    - Java API for RESTful Web Services.
- Spring MVC
    - Framework Spring (no estándar).
    - Mismo sistema usado para aplicaciones web.

En los siguientes ejemplos usaremos este último.

Niveles de cumplimiento de los principios REST



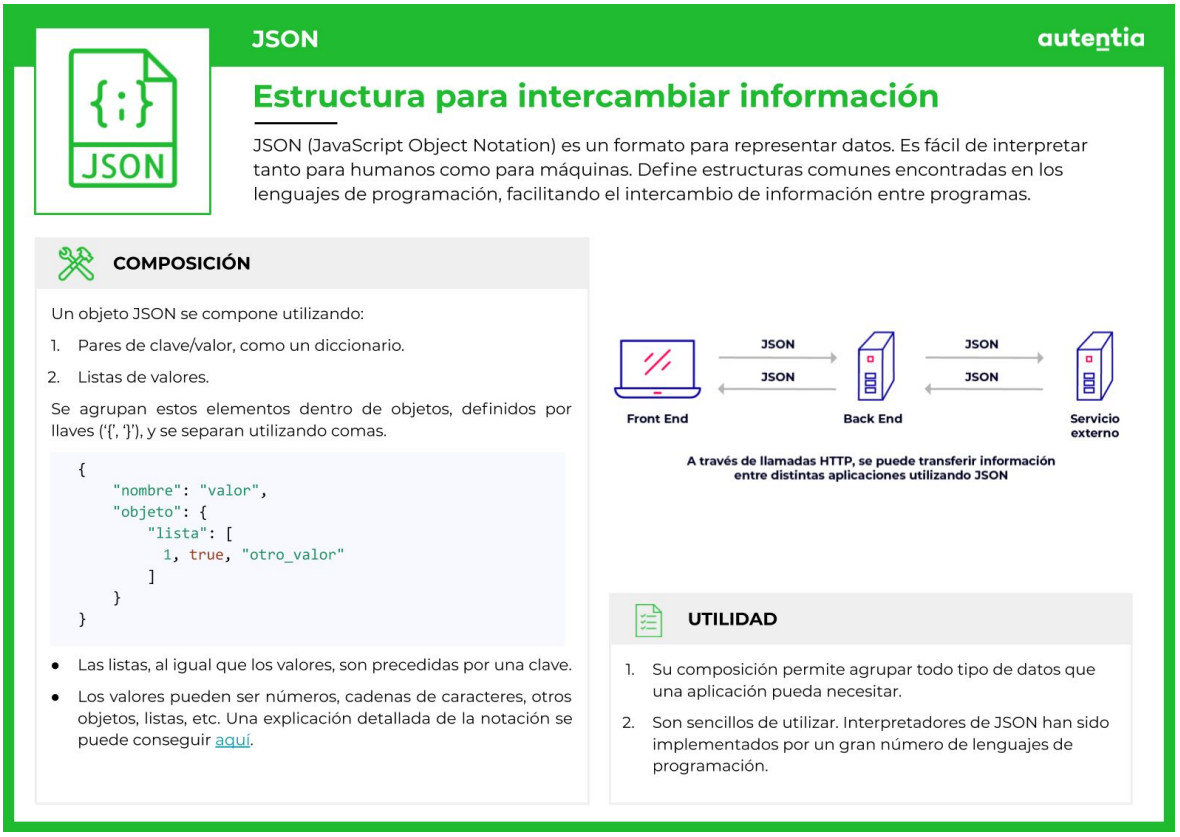
El enfoque más habitual en los servicios REST es el nivel 2.

- Los recursos se identifican en la URL. Una parte de la URL es fija y la otra apunta al recurso concreto.

```
https://www.autentia.com/servicio/agilidad/
```

- Las operaciones que se quieren realizar con ese recurso son los métodos del protocolo HTTP.

- La información se devuelve codificada en JSON.



- Se usan los códigos de estado http para notificar errores.

API REST con SpringMVC

Se crea un controlador con la anotación **@RestController**. Esta clase será la encargada de gestionar las peticiones que se hagan a nuestra API e indica que los datos devueltos por cada método se escribirán directamente en el cuerpo de la respuesta.

Se implementa un método en la clase por cada URL de la API REST:

- Se anota con **@RequestMapping** para indicar la URL y el verbo HTTP correspondiente (GET, POST, etc.) se encarga de representar los endpoints de nuestra API.
- El método devuelve el objeto que quiere enviar al cliente.

Para la prueba de concepto crearemos un API REST con un **endpoint** que devuelva un listado de servicios con la información en formato JSON.

```
@RestController
public class ServiceController {

   @RequestMapping(value="/servicios", method= RequestMethod.GET)
   public List<Service> serviceListing() {
       // TODO
   }
}
```

Cuando todas las URLs de un controlador empiezan de forma similar, se puede poner la anotación @RequestMapping a nivel de clase con la parte común. Por defecto el verbo es GET. Si refactorizamos el URL mapping en el controller queda tal que así:

```
@RestController
@RequestMapping("/servicios")
public class ServiceController {
```

```
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public List<Service> serviceListing() {
        // TODO
    }
}
```

En caso de querer insertar información en la URL nos encontramos con:

- **URL con parámetros**

  Es habitual que se incluya información en la URL para que esté disponible en el servidor cuando el usuario pulsa el enlace.

  - Los parámetros se incluyen al final de la URL separados con `?` (query).
  - Los parámetros se separan entre sí con `&`
  - Cada parámetro se codifica como `nombre=valor`

  ```
  https://www.autentia.com/servicio?name=agilidad
  ```

Para acceder a la información se usa **@RequestParam**.

```
    @RequestMapping(value="/", method=RequestMethod.GET)
    public Service get(@RequestParam String name) {
        // TODO
    }
```

- **Como parte de la propia URL**

  La información también se pueden incluir como parte de la propia URL, en vez de cómo parámetros:

  ```
  https://www.autentia.com/servicio/agilidad/
  ```

El “name” del recurso se codifica en la ruta y se accede a él usando un

@PathVariable.

```
@RequestMapping(value="/{name}", method=RequestMethod.GET)
public Service get(@PathVariable String name) {
    // TODO
}
```

En caso de querer añadir un nuevo recurso, indicamos que el método atiende peticiones POST.

```
@RequestMapping(value = "/", method = RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
public Service add(@RequestBody @Valid Service service) {
    // TODO
}
```

La anotación **@ResponseStatus** (HttpStatus.CREATED) indica que se devuelva el código de estado 201 al cliente si todo ha salido bien.

La anotación **@RequestBody** se utiliza para indicar que el objeto service vendrá en el cuerpo de la petición del cliente. La anotación **@Valid** se utiliza para lanzar las validaciones del estándar JSR 303 Bean Validation.

Nuevas anotaciones

Normalmente, si queremos implementar el controlador de URL usando la anotación tradicional sería de la siguiente manera:

```
@RequestMapping(value="/{name}", method = RequestMethod.GET)
```

En cambio, un nuevo enfoque hace posible simplificarlo:

```
@GetMapping("/{name}")
```

@RequestMapping se aplica a nivel de método y de clase. En cambio

@GetMapping solo podemos aplicarlo a nivel de método.

Actualmente, Spring permite cinco tipos de anotaciones para manejar los diferentes tipos de métodos de solicitud HTTP. Estas anotaciones son:

- @GetMapping.
- @PostMapping.
- @PutMapping.
- @DeleteMapping.
- @PatchMapping.

En el próximo apartado se habla de cómo dotar de flexibilidad a la respuesta HTTP haciendo uso de la clase **ResponseEntity**.

Clase ResponseEntity<T>

En ciertos momentos se necesita enviar respuestas HTTP desde nuestro backend hacia el cliente. Una de las maneras en la que Spring trabaja con ello es usar ResponseEntity para manejar el cuerpo, cabeceras y el estado de las respuestas.

Permitiéndonos total libertad de configurar la respuesta que queremos que se envíe desde nuestros endpoints.

```
@RequestMapping(value="/{name}", method = RequestMethod.GET)
  public ResponseEntity<Service> get(@PathVariable String name) {

      // Si está el servicio con name...
      return new ResponseEntity<>(service, HttpStatus.OK);

      // Si no existe
      return new ResponseEntity<>(HttpStatus.NOT_FOUND);
  }
```

Si el recurso existe se devuelve, y si no, se devuelve 404 NOT FOUND. Por eso el método devuelve un ResponseEntity.

Finalmente, ResponseEntity provee dos clases anidadas de tipo interface: BodyBuilder y HeadersBuilder.

Solucionar problema CORS en nuestra API

Por razones de seguridad, los navegadores prohíben las llamadas AJAX a recursos fuera del origen actual. El intercambio de recursos de origen cruzado (CORS) es una especificación W3C, implementada por la mayoría de navegadores, que permite especificar qué tipo de solicitudes están autorizadas.

Para solventar este problema es suficiente con añadir la siguiente configuración:

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
```

```
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedMethods("*")
                .allowedOrigins("*")
                .maxAge(3600);
    }
}
```

Para habilitar las solicitudes de origen cruzado es necesario tener alguna configuración CORS explícitamente declarada. El `HandlerMapping` de Spring MVC proporciona soporte para el CORS. Cada HandlerMapping puede ser configurado individualmente. Después de mapear con éxito una petición a un handler, se comprueba la configuración del CORS, la cual es interceptada y validada.

Se puede combinar la configuración CORS global a nivel de HandlerMapping con una configuración más fina a nivel de clase o de método, por ejemplo, con la anotación **@CrossOrigin**.

En el ejemplo anterior, se hace uso de una configuración global de CORS. Aclarar que, con `allowedOrigins` se debe establecer uno o más dominios específicos. Para simplificarlo se hace uso de *, pero no es lo adecuado.

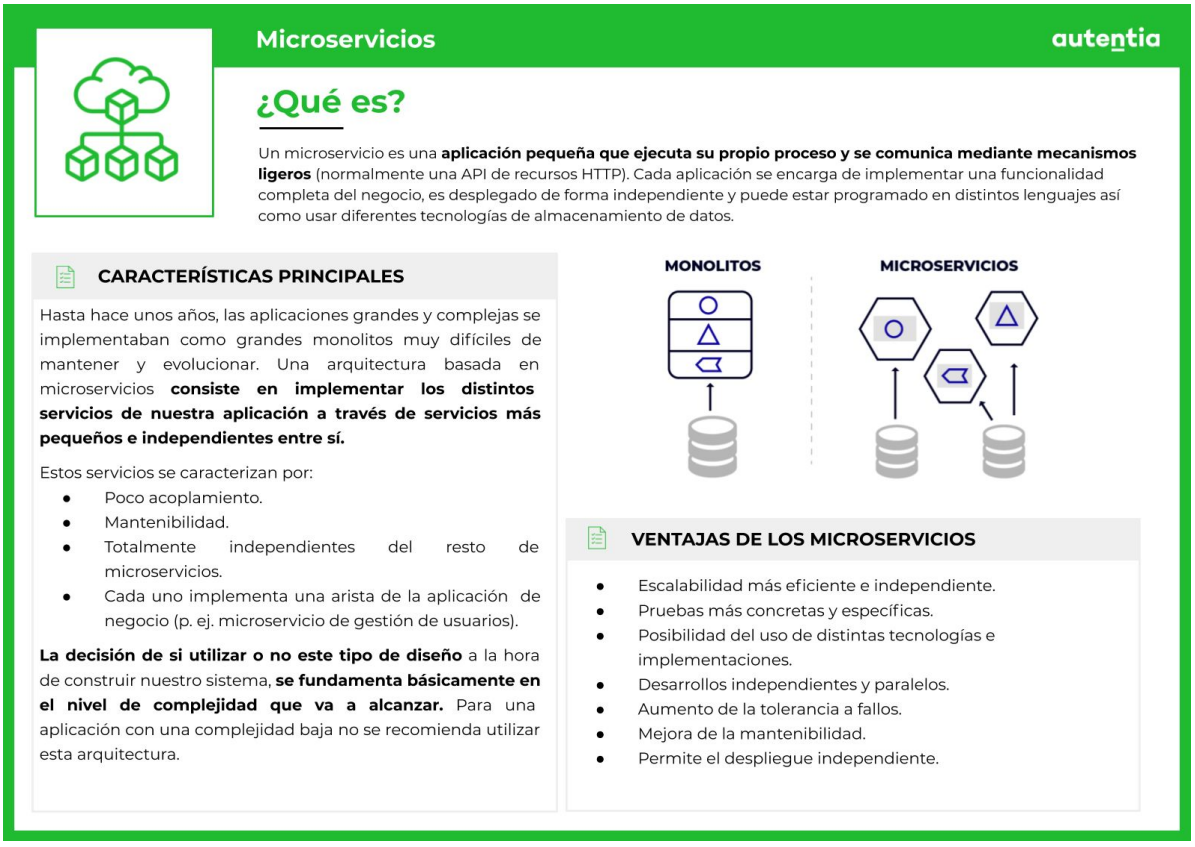
De esta sencilla forma permitimos consumir nuestros servicios desde otros dominios.

Introducción al desarrollo de microservicios

Qué son

La arquitectura de microservicios (en inglés, **M**icro**s**ervices **A**rchitecture, MSA) consiste en construir una aplicación como un conjunto de pequeños servicios, los cuales se ejecutan en su propio proceso y se comunican con mecanismos ligeros (normalmente una API de recursos HTTP). Cada microservicio se encarga de implementar una funcionalidad completa del negocio y es desplegado de forma independiente pudiendo estar programado en distintos lenguajes y usar diferentes tecnologías de almacenamiento de datos.

La arquitectura de microservicios es una manera de construir sistemas de software descomponiendo los modelos de dominio de negocio en contextos más pequeños, consistentes y delimitados, implementados por los servicios.



Estos servicios son aislados y autónomos, pero se comunican para proporcionar alguna funcionalidad de negocio. Los microservicios suelen ser implementados y operados por pequeños equipos con suficiente autonomía para que cada equipo y servicio pueda cambiar sus detalles de implementación interna con un impacto mínimo en el resto del sistema.

Estos equipos se comunican a través de promesas, que son una forma en que un servicio puede publicar intenciones para otros componentes o sistemas que pueden desear utilizar el servicio. Especifican estas promesas con interfaces de sus servicios o mediante wikis que documentan sus servicios.

Los microservicios nos permiten:

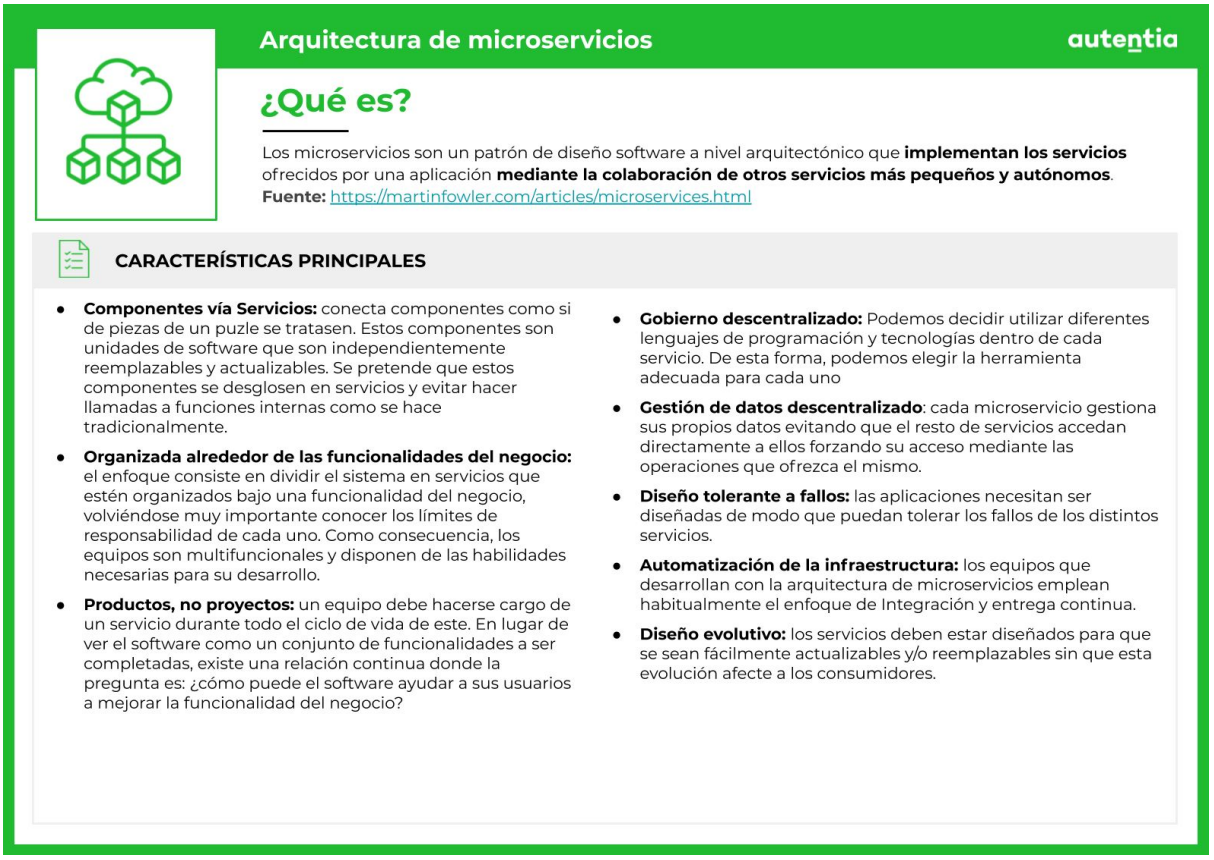
- Comprender lo que hace el servicio sin enredarse con otras preocupaciones de una aplicación más grande.
- Construir rápidamente el servicio localmente.

- Elegir la tecnología adecuada para el problema concreto.
- Testear el servicio concreto de manera aislada.
- Construir/implementar/lanzar el servicio cuando sea necesario para el negocio, que puede ser independiente de otros servicios.
- Identificar y escalar horizontalmente partes de la arquitectura donde sea necesario.
- Mejorar la resiliencia del sistema en su conjunto.

Los microservicios ayudan a desacoplar nuestros servicios y equipos para escalarlos rápidamente. Permiten a los equipos concentrarse en brindar el servicio y realizar cambios cuando sea necesario y hacerlo sin costosos puntos de sincronización.

Los microservicios tienen muchos beneficios pero vienen con sus propios inconvenientes:

- Requieren más recursos.
- La complejidad operativa es mucho mayor.
- Es más difícil depurar los problemas.
- Es difícil comprender el sistema de manera integral.
- Es imprescindible diseñar el sistema de gestión de errores.



Patrones de los microservicios

Como hemos visto, las arquitecturas de microservicios ofrecen importantes ventajas pero también plantean retos que deben ser resueltos. Para ayudarnos en una correcta implementación de los mismos, han ido surgiendo diferentes patrones o recetas que nos ayudan a no cometer los mismos errores que otros han sufrido previamente, a la hora de abordar los problemas comunes que suelen aparecer. Existen multitud de patrones pero hemos seleccionado algunos de los más relevantes para este documento.

Service Discovery

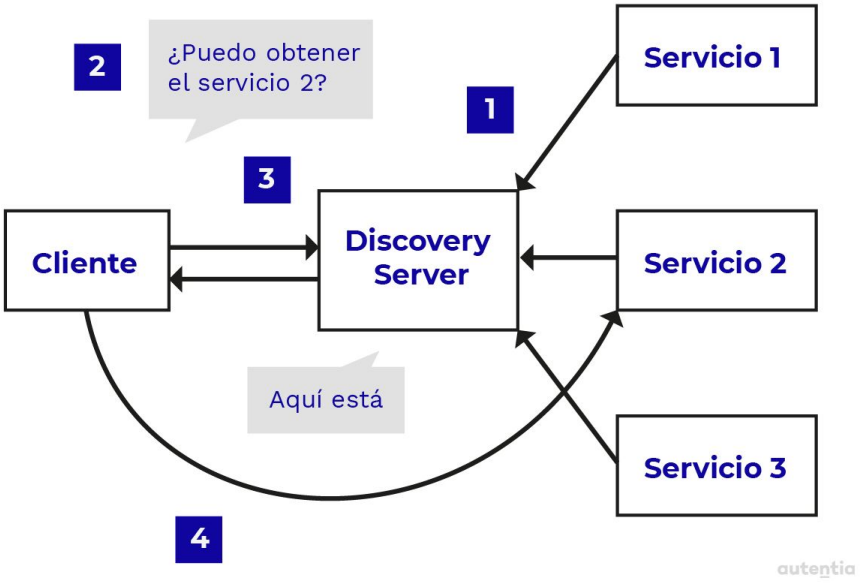
Actualmente, gran parte de las aplicaciones requieren de un API para

funcionar, dicha API ofrece endpoints para que la aplicación pueda interactuar con el backend. Uno de los grandes problemas, dada la gran cantidad de servicios, es conocer donde está alojado dicho servicio, ya que cada uno responde a una dirección y puerto específico y esto se acentúa en arquitecturas Cloud donde pueden cambiar dinámicamente de IP o puerto, ya sea por fallos o por gestionar réplicas según la demanda de nuestra aplicación; es aquí donde entra en juego el patrón Service Discovery.

Entonces, ¿por qué se considera una mala práctica las *hard-coded* URLs?

- Los cambios requieren modificaciones del código.
- Al hacer un despliegue, por ejemplo en Heroku, te encontrarás con URLs dinámicas que van cambiando continuamente.
- Si un servicio tiene mucha demanda se puede replicar, cada uno tendrá su propia URL entonces necesitas un mecanismo que se encargue del balanceo de carga.
- Despliegues en múltiples entornos complican el manejo de diferentes URLs.

Por todas estas razones, necesitamos el **Service Discovery**, un patrón para microservicios que nos permite invocar servicios sin conocer su ubicación física.



Imagínate que disponemos de tres servicios que se van a consumir. El primer paso es añadir una capa de abstracción, **Discovery Server**, que se encargará de saber donde están alojados dichos servicios y así proveer al cliente la URL para posteriormente hacer la llamada. Así, el cliente conocerá en todo momento la existencia de estos:

1. Cada servicio que quiera ser descubierto se registra en el Discovery Server.
2. El cliente solicita la URL de un servicio a consumir.
3. El Discovery Server provee la dirección de dicho servicio.
4. El cliente realiza la llamada.

Existen dos modelos de descubrimiento de servicios:

- Si de la tarea de descubrir el servicio se encarga mayoritariamente el cliente, estaremos hablando de **Client side discovery** (modelo visto anteriormente). Éste es el modelo que utiliza Spring Cloud.
- En cambio, una alternativa es que sea el cliente quien pase un mensaje al Discovery server y éste sea el encargado de transmitirlo al servicio adecuado.

Ambos modelos son válidos, cada uno tiene sus propias ventajas y

desventajas.

En caso de que un microservicio deba acceder a otro, lo ideal sería que de alguna manera pudiera saber en qué direcciones están las instancias de ese otro microservicio funcionando.

Para ello, en Spring se utiliza **Eureka Server** del paquete Spring Cloud Netflix. Estos últimos crearon un montón de librerías aplicables a nuestro ecosistema como pueden ser Ribbon, Hystrix, Zuul, Feign, etc.

Debemos especificar a los clientes que no se guarden en su caché local las direcciones de las diferentes instancias mediante la propiedad `eureka.client.fetch-registry=false`, esto es para que consulte al servidor Eureka cada vez que necesite acceder a un servicio. En un entorno de producción, a menudo se pone a *true* para agilizar las peticiones.

Por otra parte, los microservicios serán **Eureka clients** registrados en el Eureka Server.

Además, si utilizamos los paquetes **Ribbon** y **Feign** conseguiremos que nuestra aplicación sea capaz de encontrar las diferentes instancias de un microservicio y balancear las peticiones de carga. En el siguiente enlace podréis consultar la implementación de un caso práctico sobre Feign.

Circuit Breaker

Para entender este apartado, vamos a dar unas pinceladas a dos conceptos esenciales a la hora de trabajar con microservicios:

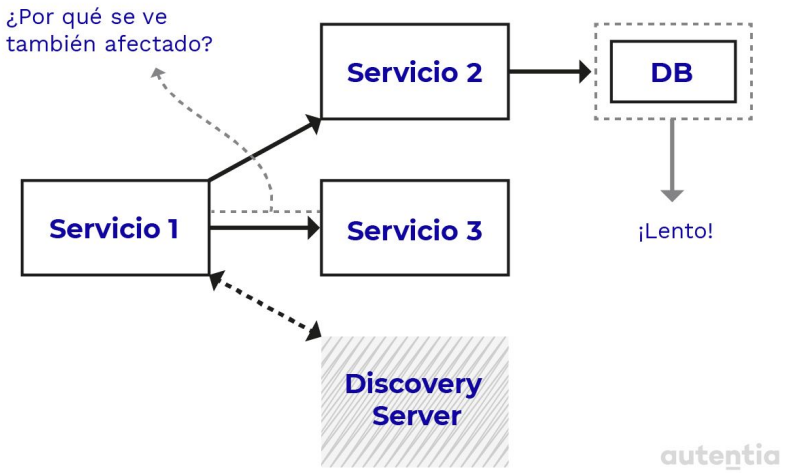
- Entender los desafíos de la alta disponibilidad.
- Conseguir microservicios resilientes y tolerantes a fallos.

Antes de ver cómo hacer los microservicios resilientes, es importante entender cuáles son los problemas a los que te puedes enfrentar: ¿Cuántos fallos puede tolerar un sistema?, ¿qué ocurre cuando se cae un

microservicio?, ¿y cuándo va lento?...

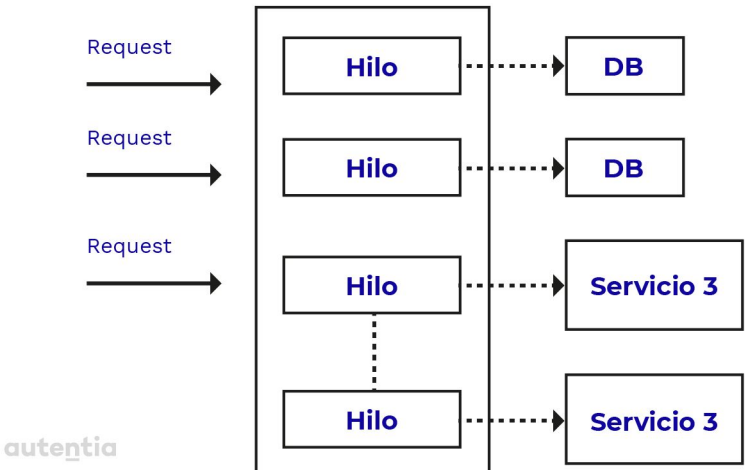
En el caso de que se caiga la instancia de un microservicio, una posible solución será generar múltiples instancias duplicadas, gracias al equilibrio de carga del lado del cliente mediante la lógica proporcionada por el algoritmo *Round-Robin*.

Cuando un servicio va **lento** es un problema mucho más complejo. En el siguiente ejemplo, imaginemos que el microservicio DB tarda en responder, como es esperable, los servicios que dependan de él también se verán afectados. Pero es muy probable que también el Servicio 3 se vea afectado pese a no tener nada que ver con la otra ruta. ¿Cómo es posible?



Debido a los **threads**. Cuando a un servidor web le llega una petición, éste crea un thread para procesar dicha petición y así devolver una respuesta. Si la frecuencia de peticiones es más elevada de lo que el hilo es capaz de procesar, excederá el límite máximo. Lo que termina sucediendo es que aumenta el consumo de los recursos hardware disponibles.

Esa es la razón por la que un servicio independiente puede verse afectado.



Por ejemplo, cuando cientos de usuarios pulsan “recargar", lo que hacen es multiplicar el número de *request*. Entonces, la solución no es incrementar los recursos disponibles, una posible solución son los **timeouts**. Básicamente, liberar los hilos cuando estos tardan demasiado tiempo. Podremos configurar estos timeouts con Spring **RestTemplate**, en el siguiente caso con un timeout de 3 segundos:

```
@Bean
@LoadBalanced
public RestTemplate getRestTemplate() {
   HttpComponentsClientHttpRequestFactory clientHttpRequestFactory
= new HttpComponentsClientHttpRequestFactory();
   clientHttpRequestFactory.setConnectTimeout(3000);
   return new RestTemplate(clientHttpRequestFactory);
}
```

Pero esto no resuelve del todo el problema, ya que **solo lo soluciona cuando la frecuencia de las peticiones entrantes son menores a los threads que se liberan**.

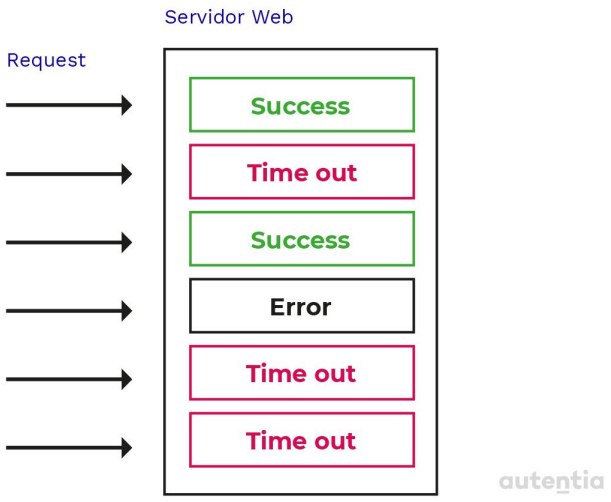
Una solución más eficiente sería detectar qué servicio en particular está siendo lento y evitar que reciba nuevas peticiones por un determinado tiempo. Después, lo intenta de nuevo y si está recuperado continúa recibiendo nuevas peticiones.

Nos encontramos ante un patrón de tolerancia a fallos llamado **Circuit Breaker**, el cual implica:

- Primero, detectar que algo está mal.
- Tomar medidas temporales para evitar que la situación empeore.
- Desactivar el componente problemático para que no afecte a los componentes posteriores.

La ventaja de este patrón es que puede ser reiniciado manual o automáticamente para reanudar el correcto funcionamiento de nuestro entorno.

Digamos que tenemos el siguiente flujo de peticiones en el servidor web y se dan los siguientes casos:



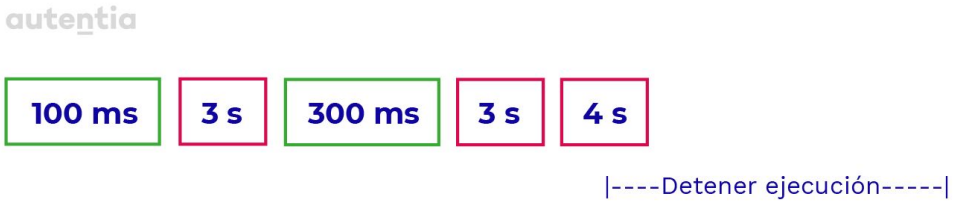
La primera request llega y tiene éxito, en la segunda se produce un timeout y así sucesivamente, ¿ahora qué haces? Esto es un poco complicado ya que los timeouts pueden ocurrir y que ocurran alguna vez, no es suficiente para romper el circuito.

Ahora, nuestra tercera request tiene éxito, la cuarta falla y en las sucesivas se producen timeouts.

Para resolver esta problemática, necesitamos establecer una **lógica** que permita establecer cuándo queremos que se produzca un “cortocircuito”.

Es decir, necesitas configurar los **parámetros** para que tu circuito se rompa. Veámos estos parámetros con un pequeño ejemplo:

- Las últimas n requests a considerar para tomar la decisión, por ejemplo, las últimas 5 requests y ver cuántas de ellas fallan.
- ¿Cuántas de ellas deberían fallar?: 3.
- Duración del timeout: 2 segundos.
- Por último, ¿cuánto tiempo dar de margen para volver a intentarlo? (sleep window): 10 segundos.



Ahora te preguntarás cuál es la mejor combinación posible, pues bien, es una combinación muy difícil de obtener ya que la gran mayoría de veces depende de dos factores principales, de cuántas solicitudes llegan (frecuencia) y cuán grande es el thread pool, así podrás hacerte a la idea de qué parámetros configurar para no desbordar la capacidad de tu aplicación.

Por último, ¿qué ocurre con las peticiones entrantes? En caso de que un microservicio vaya lento, otro microservicio que dependa de él necesita un **fallback**, cuando se produce un cortocircuito también se debe controlar que retornar. Se nos presentan las siguientes posibilidades:

- Lanzar un error.
- Un enfoque mejor es retornar una respuesta por defecto en el fallback.
- Una opción más interesante es guardar las respuestas anteriores en **caché** y utilizarlas cuando sea posible.

Hystrix

Afortunadamente, existe una librería open source desarrollada por Netflix que nos facilita la vida: **Hystrix**. Las ventajas de usarla es que implementa el circuit breaker, tan solo tenemos que configurar los parámetros y lo mejor de todo, funciona extremadamente bien con Spring Boot.

Veamos los pasos para añadir hystrix a nuestro microservicio:

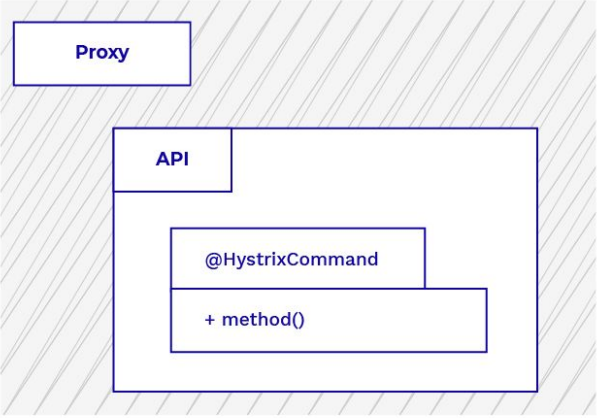
1. Añadir la dependencia: `spring-cloud-starter-netflix-hystrix`
2. Añadir la anotación **@EnableCircuitBreaker** a la clase Application.
3. Añadir la anotación **@HystrixCommand** a los métodos que necesiten romper el circuito.
4. Configurar el comportamiento de Hystrix.

```
@GetMapping
@Cacheable(cacheNames="services")
@HystrixCommand(fallbackMethod = "getFallbackServiceListing")
public List<Service> serviceListing() {
   // TODO
}

public List<Service> getFallbackServiceListing() {
   return Arrays.asList(new Service());
}
```

Además, debemos tener cierta **precaución** a la hora de elegir dónde colocar ese @HystrixCommand, ya que por ejemplo, en el caso de que un controlador haga uso de dos servicios, si implementamos esa anotación en el método principal de la API, con que tan solo uno de los dos falle, ya se llama al fallback. En cambio, con un correcto reparto de granularidad podemos conseguir que un servicio que no ha caído pueda devolver datos válidos.

Destacar que, si en el flujo de ejecución de un método este hace uso de dos microservicios, no será posible controlar ambos métodos fallback debido a que la instancia de la clase API en realidad es implementada por

un intermediario **Proxy**.



Lo que Hystrix hace es envolver la clase API en un proxy. Así que, cuando se realiza una llamada realmente se hace uso de la instancia de ese proxy. En este caso, Hystrix no es capaz de gestionar la llamada a ambos microservicios, así que la única manera de resolver esto es llevar esa funcionalidad a otra clase (granularidad).

Todavía tendremos que configurar las propiedades de Hystrix.

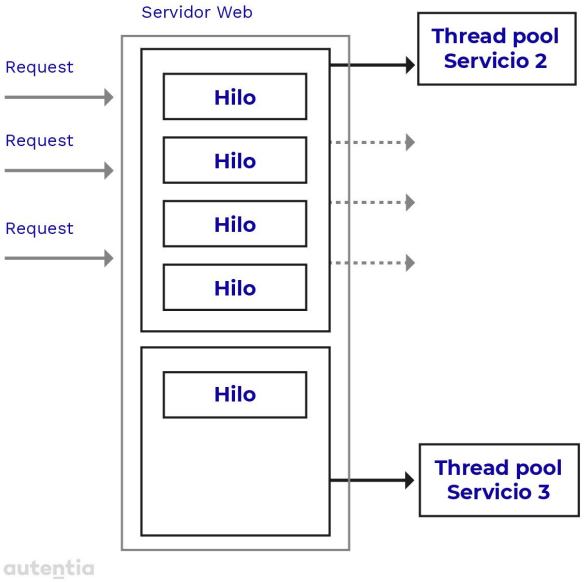
```
@GetMapping
@Cacheable(cacheNames="services")
@HystrixCommand(fallbackMethod = "getFallbackServiceListing"),
        commandProperties = {
                @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value =
"2000"),
                @HystrixProperty(name =
"circuitBreaker.requestVolumeThreshold", value = "5"),
                @HystrixProperty(name =
"circuitBreaker.errorThresholdPercentage", value = "50"),
                @HystrixProperty(name =
"circuitBreaker.sleepWindowInMilliseconds", value = "5000"),
        }
)
public List<Service> serviceListing() {
   // TODO
}
```

Nos encontramos con `timeoutInMilliseconds`, básicamente el tiempo para establecer el timeout. El siguiente parámetro, `requestVolumeThreshold`, es el encargado de evaluar las últimas n peticiones. Después, el parámetro `errorThresholdPercentage` es el porcentaje de fallos admisibles antes de fallar. Por último, `sleepWindowInMilliseconds` establece el tiempo de detención de la ejecución del circuit breaker antes de reintentar la petición.

Bulkhead pattern

Nos encontramos ante la tercera forma de gestionar caídas de los servicios. Se trata del patrón **Bulkhead**, un tipo de diseño de aplicaciones que es tolerante a errores. En una arquitectura Bulkhead, los elementos de una aplicación se aíslan en grupos para que, en caso de fallo, los demás sigan funcionando.

Su nombre proviene de las particiones en las secciones del casco de un barco. Si el casco de un barco corre peligro, solo se llenará de agua la sección dañada, lo que evita que el barco acabe hundiéndose. En el contexto de microservicios, nuestro ejemplo quedaría de la siguiente manera:



En los anteriores casos, las peticiones entrantes y en consecuencia sus respectivos hilos se apilan. Si uno de estos servicios se ralentiza, ese servicio consumirá todos los hilos disponibles así que no habrá hilos disponibles para los siguientes servicios a consumir, lo que genera problemas.

La clave es separar dichos servicios en particiones independientes (**Thread pools**), cada una con su propia configuración. Por ejemplo, en caso de que el servicio 2 vaya lento, el servicio 3 podrá seguir con total **normalidad**.

Configurar los bulkheads es bastante sencillo y muy similar a la configuración del circuit breaker. Veámoslo con un ejemplo:

```
@GetMapping
@Cacheable(cacheNames="services")
@HystrixCommand(
        fallbackMethod = "getFallbackServiceListing",
        threadPoolKey = "serviceInfoPool",
        threadPoolProperties = {
            @HystrixProperty(name = "coreSize", value = "20"),
            @HystrixProperty(name = "maxQueueSize", value = "10")
})
public List<Service> serviceListing() {
```

```
    // TODO
}

public List<Service> getFallbackServiceListing() {
    return Arrays.asList(new Service());
}
```

La primera propiedad es la `threadPoolKey`, básicamente se crea una partición separada llamada *serviceInfoPool*. El siguiente paso es configurar ese bulkhead: primero, mediante la propiedad `coreSize` se establece cuántos hilos simultáneos quieres permitir en esa partición. Y mediante la segunda propiedad, `maxQueueSize`, se establece cuántas peticiones esperan en la cola antes de que puedan acceder al hilo.

Externalized Configuration

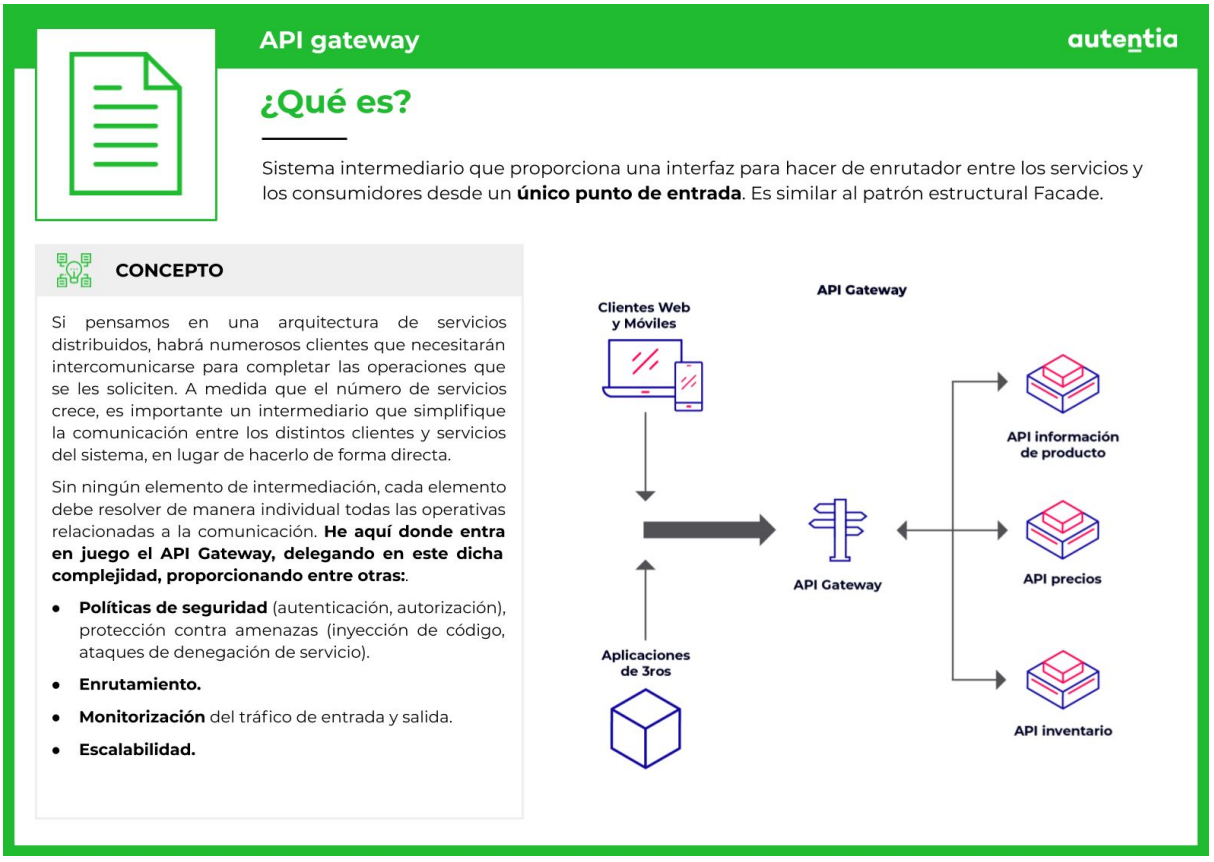
En una arquitectura de microservicios, tenemos varios de ellos, cada uno con su propia configuración, en cada entorno correspondiente (desarrollo, producción, etc.). En sistemas grandes o incluso a nivel internacional, el gestionar la configuración de cada uno de estos microservicios puede ser extremadamente complejo.

Es en este caso donde entra en juego Central Configuration, que se encarga de centralizar todas las configuraciones en un solo lugar. Cuando un microservicio necesita su configuración, lanza un ID y el servidor encargado, busca entre todas las configuraciones y le asigna la correspondiente al microservicio.

API Gateway

Un API gateway es una capa que se encuentra entre los clientes y los servicios que consumen. Actúa como un único punto de entrada para los clientes, recibiendo todas las solicitudes y delegando al servicio

correspondiente para responder a esa solicitud. Es en esta capa donde se gestionan las políticas de seguridad, enrutamiento, monitorización del tráfico de entrada y salida, etc.



Distributed Tracing y Central Log Analysis

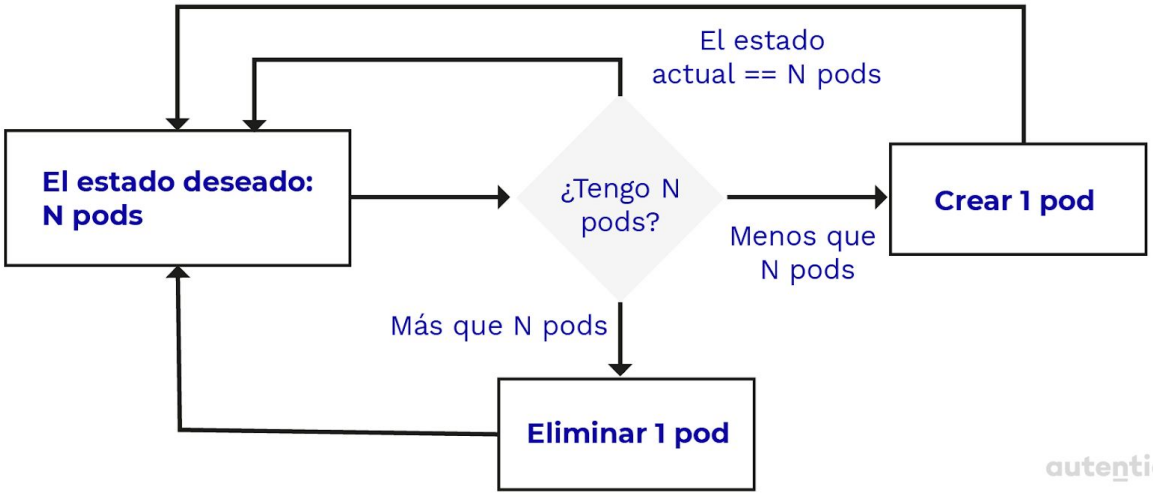
En una arquitectura de microservicios, la llamada a un servicio suele desencadenar invocaciones a otros servicios, cada uno con su propio sistema de trazas, lo que hace que sea complicado conocer qué es lo que ocurre cuando surgen los problemas. Para poder rastrear la ejecución de la traza tenemos que incluir en los logs un identificador único y global por cada petición que nos permita, posteriormente, identificar toda la secuencia de acciones que se ha desencadenado y en el orden adecuado.

Además, para facilitar el análisis de lo ocurrido es fundamental agregar

todas estas trazas en un sistema centralizado. Esta agregación debe realizarse de manera que no penalice el rendimiento de las peticiones y que incluya únicamente la información relevante para aliviar el peso de la información almacenada. Otros criterios a tener en cuenta son la privacidad de los datos que se almacenan.

Control loop

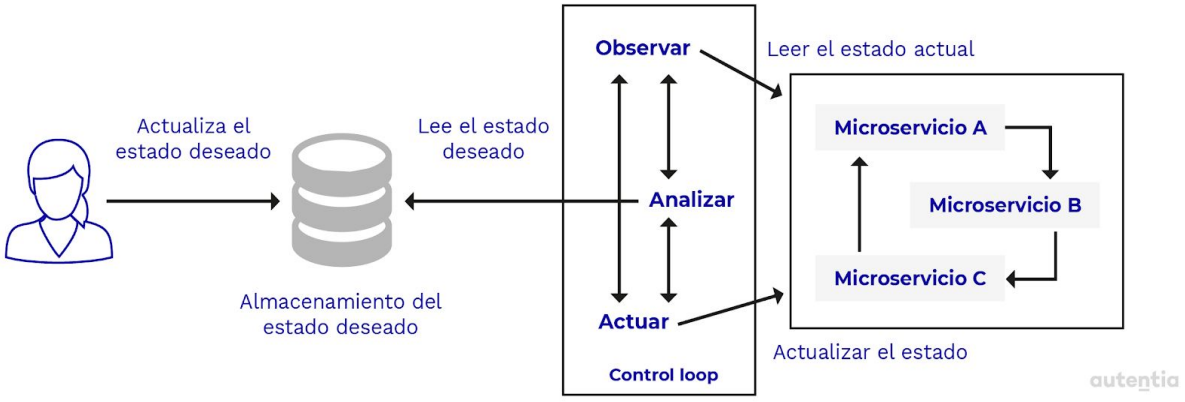
En robótica y automatización, un bucle de control (control loop) es un bucle sin terminación que regula el estado de un sistema. Por ejemplo, un termostato en una habitación.



En otros sistemas como Kubernetes, se utilizan los *controladores* que son bucles de control que vigilan el estado de un clúster (un conjunto de nodos o máquinas) y luego realizan o solicitan cambios cuando sea necesario. Cada controlador intenta acercar el estado actual del clúster al estado deseado.

Un sistema autogestionado debe monitorearse a sí mismo y al entorno, analizar las señales producidas por el monitoreo y aplicar acciones en respuesta, quizás modificándose a sí mismo. Estos pasos se repiten indefinidamente como un bucle de control. IBM sistematizó este bucle proponiendo un modelo de referencia para bucles de control automático

llamado **MAPE-K** (Monitor (Supervisar), Analyze (Analizar), Plan (Planificar), Execute (Ejecutar) y Knowledge (Conocimiento).



En el entorno de microservicios, el patrón Control Loop soluciona el problema de monitorización de muchos microservicios a la vez. Es casi imposible detectar y resolver problemas como una instancia caída o bloqueada en un entorno que tenga muchas instancias de microservicios

La solución de este problema es introducir el nuevo componente Control loop que monitoriza el estado actual del sistema, compara con el estado deseado (que está especificado previamente) y actúa si no coinciden.

En el mundo de los contenedores, normalmente *el container orchestrator* como Kubernetes implementa este patrón.

Centralized Monitoring

En los microservicios, poder monitorearlos en los diferentes entornos de ejecución, es clave. Este patrón de microservicios consiste en un sistema que se encargue de recopilar información desde los diferentes servicios y

que capture diferentes métricas como estado del sistema, solicitudes, respuestas, etc.

Dicho sistema necesita también tener la capacidad de enseñar los resultados mediante una interfaz de visualización.

Existen diferentes herramientas y/o sistemas que nos permiten esto, como:

- **Grafana**: una herramienta open source que nos permite la visualización de diferentes métricas.
- **Prometheus**: al igual que Grafana, es open source y nos permite monitorizar y recibir alertas.

Spring Boot

Introducción

Spring Boot es una herramienta que nace con la idea de ayudar al desarrollo de aplicaciones basadas en Spring. La creación de aplicaciones mediante Spring requiere de un conjunto de pasos que pueden resultar tediosos: primero, añadir en el pom.xml las dependencias necesarias según la aplicación que se vaya a desarrollar; tras ello, desarrollar el código junto con los respectivos archivos de configuración para finalmente desplegar la aplicación en un servidor.

Con Spring Boot lo que se busca es que la creación y el despliegue de aplicaciones mediante Spring se haga de manera mucho más sencilla, sin necesidad de gestionar tantos archivos de configuración. Esto se consigue mediante una serie de aspectos clave que veremos a continuación.

Convención frente a configuración

El principio de convención frente a configuración es uno de los más

importantes en el ámbito del desarrollo software. Su objetivo es reducir al máximo posible la existencia de archivos de configuración, haciendo que el programador se centre sólo en desarrollar software. La configuración sólo será necesaria cuando se haga algo **realmente** distinto.

Con la aparición del framework Spring los programadores se dieron cuenta de que desarrollaban aplicaciones con una infraestructura parecida entre sí, pero que seguían creando esos archivos de configuración una y otra vez. Spring Boot lo cambió todo: provee configuraciones por defecto para que sólo haya que configurar lo que es necesario. Por ejemplo, si queremos utilizar Spring MVC con Spring Boot, la aplicación desarrollada se ejecutará por defecto en el puerto 8080 a no ser que queramos modificarlo.

Starters

Para hacer más fácil la gestión de dependencias en el pom.xml, Spring Boot introdujo los denominados *starters*. Los starters no son más que una herramienta que permite agrupar muchas dependencias en una única. Antes de Spring Boot, si por ejemplo se querían incluir test unitarios en la aplicación, había que incluir la dependencia de Spring Test, de JUnit, de Mockito, etc. Sin embargo, con el *Spring boot starter test* ya están incluidas todas esas dependencias en una.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

Existen más de 30 starters disponibles, entre otros se encuentran el *Spring boot starter web* para crear aplicaciones WEB incluyendo servicios REST, *Spring boot starter data JPA* para usar Spring Data JPA con Hibernate...

Es importante recalcar que los starters vienen con una configuración

establecida por defecto pero si se quiere cambiar alguno de esos valores, no hay más que modificar el archivo de configuración.

Fat Jar file

Otro de los objetivos por los que nació Spring Boot fue el de simplificar el despliegue de las aplicaciones. ¿Cómo lo hace?, permitiendo empaquetar las aplicaciones en un único archivo **.jar,** el cual contiene un servidor embebido integrado que es levantado al iniciar la aplicación. De esta manera no es necesario desplegar la aplicación a parte en un servidor como por ejemplo Tomcat, como se haría con el tradicional archivo **.war**.

Esto facilita el despliegue de nuestras aplicaciones adaptándolo a distintos entornos. Por ejemplo, Spring Boot se utiliza frecuentemente en el desarrollo de microservicios como veremos más adelante, puesto que cada archivo .jar se puede desplegar de manera fácil en un contenedor Docker.

El término “fat jar file” hace referencia a que el archivo .jar va a contener todas las dependencias, las clases del proyecto e incluso el servidor embebido. De ahí el adjetivo *fat*.

@SpringBootApplication

La autoconfiguración es otro de los pilares más importantes que da Spring Boot frente a Spring. En Spring, de una manera u otra hay que registrar manualmente todos los beans que van a ser cargados en el contexto de Spring. Sin embargo, Spring Boot mediante la anotación @SpringBootApplication habilita la autoconfiguración del contexto de Spring, adivinando los beans que se van a necesitar gracias a las dependencias incluidas en el proyecto.

Si vamos a la documentación oficial de la anotación, vemos que la misma se puede desglosar en tres anotaciones equivalentes:

- **@EnableAutoConfiguration:** habilita la autoconfiguración del contexto de Spring.
- **@ComponentScan:** por defecto escanea recursivamente desde el paquete donde se encuentra la clase que incluye @SpringBootApplication. Se puede modificar el paquete a escanear.
- **@Configuration:** explicada anteriormente en la sección Anotaciones.

Microservicios con Spring Boot

Spring Boot no nace con la idea de ser una tecnología referente en el desarrollo de microservicios, pero con el paso del tiempo, ha acabado convirtiéndose en ello. Construir aplicaciones basadas en una arquitectura de microservicios significa construir sistemas pequeños, autónomos y flexibles, y Spring Boot facilita este desarrollo gracias al archivo .jar y a la inclusión del servidor embebido.

Para ahondar en esa facilidad en el desarrollo de microservicios, nace **Spring Cloud.**

Spring Cloud

Spring Cloud es una herramienta provista por Spring cuyo objetivo es facilitar a los desarrolladores la aplicación de patrones o elementos habituales en sistemas distribuidos, como pueden ser API Gateway o Control Bus. Al igual que Spring Boot, contiene una serie de starters que facilitan la inclusión de dependencias en el pom.

Spring Cloud tiene muchas herramientas para ser incluidas en el desarrollo de sistemas distribuidos, pero en este documento nos centraremos en **Spring Cloud Netflix,** pensada para las arquitecturas de microservicios. Sin embargo, se puede consultar la documentación oficial para ver todos los módulos que contiene.

Spring Cloud Netflix

Spring Cloud Netflix integra en Spring Boot la solución provista por Netflix OSS a los problemas que surgen implementando sistemas distribuidos. Netflix OSS, cuyas siglas corresponden a Netflix Open Source Software Center, es un proyecto open source liderado por Netflix en el cual se liberan para los desarrolladores librerías y frameworks que ayudan a resolver problemas de escalado y construcción de sistemas distribuidos.

Nos vamos a centrar en explicar dos soluciones que da Spring Cloud Netflix para la construcción de una arquitectura de microservicios mediante Spring Boot. Antes de nada, es importante recordar la sección de Patrones de los microservicios, donde se explican los patrones. En este apartado nos vamos a centrar en cómo se configuran estas herramientas provistas por Netflix en Spring Boot:

- **Service Discovery:** mediante el servidor **Eureka**, que representa el servidor donde se almacenan todas las configuraciones de los microservicios que componen el sistema, considerados *eureka clients*.

  Lo primero, como siempre, es añadir la dependencia correspondiente en el pom:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

  Una clase será registrada como Eureka Server gracias a la anotación **@EnableEurekaServer.**

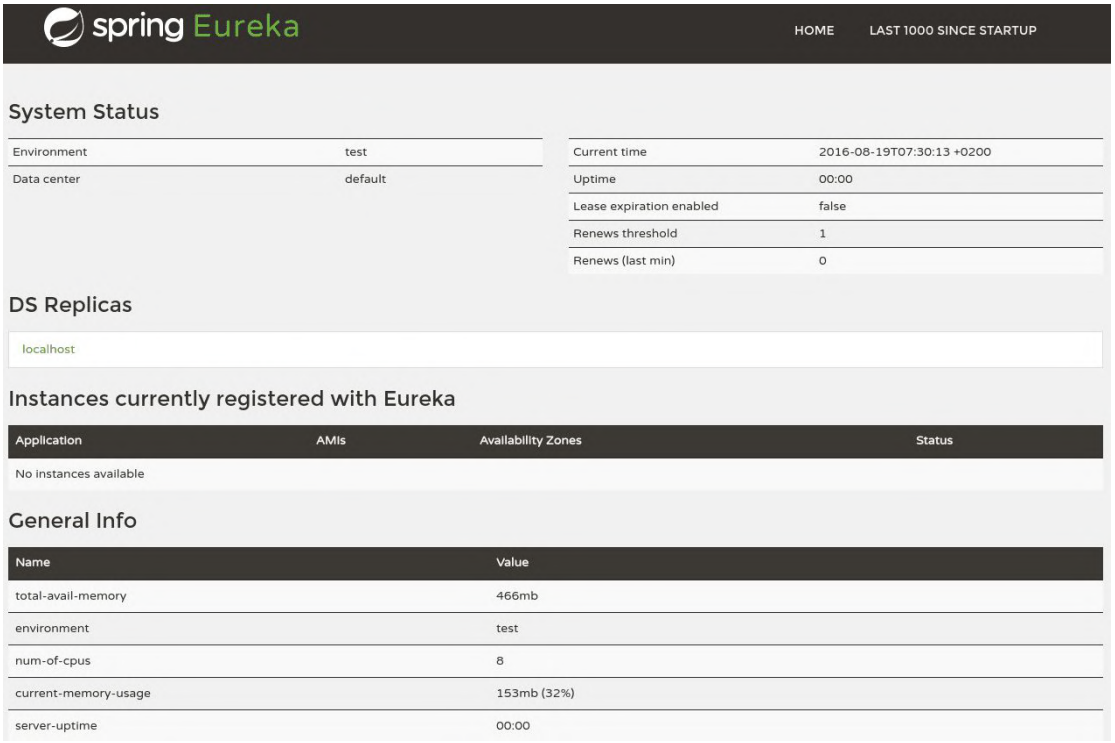
```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServer {
```

```
	public static void main(String[] args) {
SpringApplication.run(EurekaServer.class, args);
	}
}
```

En el archivo de configuración *application.properties* se indicará el puerto donde se va a ejecutar el eureka server. Al ejecutarse, buscará por defecto dicho archivo para fijar las propiedades de configuración.

```
server:
  port: 3333
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

Se puede comprobar que el eureka server se está ejecutando introduciendo https://localhost:3333 y accediendo, ahí veremos el *dashboard,* donde más tarde saldrán registrados los microservicios del sistema.



Una vez que está bien configurado el eureka server, hay que registrar un microservicio como eureka client.

```
@SpringBootApplication
@EnableEurekaClient
public class EurekaClient
    public static void main(String[] args) {
        SpringApplication.run(EurekaClientApplication.class,
args);
    }
}
```

La anotación **@EnableEurekaClient** es **opcional** si en el pom.xml está añadida la dependencia *spring-cloud-starter-netflix-eureka*, aunque es recomendable incluirla para añadir legibilidad a las clases. Hay que configurar otro archivo .yaml para añadir las propiedades del microservicio.

```
spring:
  application:
```

```
    name: microservice-eureka-client
server:
  port: 4444
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:333/eureka
```

El microservicio con nombre “microservice-eureka-client” va a ejecutarse en el puerto 4444. Es muy importante indicarle la URL del eureka server mediante la propiedad `eureka.client.serviceUrl.defaultZone` ya que esto va a permitir la comunicación entre ambos componentes y que toda la arquitectura funcione.

- **Enrutamiento y filtrado de peticiones:** mediante la herramienta **Zuul** que va a actuar de proxy inverso, reenviando peticiones entrantes a los microservicios que componen el sistema. Esta herramienta está ligada al patrón API Gateway. Para incluir Zuul únicamente hay que añadir las dependencias correspondientes en el pom:

```
<dependency>
  <groupId> org.springframework.cloud </groupId>
  <artifactId> spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

Una vez incluida la dependencia hay que crear una clase que actúe como Zuul y añadirle la anotación **@EnableZuulProxy.** En el fichero de configuración .yaml de Zuul hay que indicar hacia qué microservicio y, por lo tanto, hacia qué ruta hay que reenviar la petición entrante. Por ejemplo, supongamos que tenemos un microservicio que gestiona el carrito de compra en una tienda online. El microservicio se registra en el Service Discovery bajo el nombre “cart” y se ejecuta en el puerto 4444.

Por otra parte, tendremos el archivo de configuración de Zuul:

```
server:
  port: 8080

zuul:
  routes:
    cart:
      path: /cart/**
      url: https://localhost:4444
```

Esto significa que, cuando tengamos ambas aplicaciones ejecutándose, si a Zuul le llega una petición HTTP se la va a reenviar al microservicio *cart* a la url `http://localhost:4444`.

A parte del enrutamiento de peticiones, con Zuul también existe la posibilidad de añadir filtros a las mismas. Existen cuatro tipos de filtros: *pre,* para ejecutar antes de que la petición se enrute; *route*, para manejar el enrutamiento actual de la petición; *post*, para ejecutar después de que la petición se ejecute; y finalmente, *error*, para gestionar errores ocurridos en el transcurso del manejo de una petición.

Otra solución sería Hystrix para resolver el Circuit Breaker, Ribbon para el balanceo de carga o Spring Cloud Configuration para aplicar Central Configuration. En la documentación oficial podemos consultar todas las herramientas.

Micronaut

Introducción

Micronaut es un framework similar a Spring que nos permite crear microservicios con Java, Groovy o Kotlin. Se ha diseñado con el objetivo de orientarlo a microservicios, es muy ligero y reactivo, está basado en Netty y una de sus principales características es el Ahead of Time Compilation (AOT), es decir, la creación de objetos del contexto se realiza en tiempo de compilación lo que ofrece un tiempo de inicio más rápido y significativamente menos consumo de memoria que Spring, ya que no utiliza ni reflection ni runtime-proxies.

GraalVM

GraalVM es una máquina virtual que nos permite ejecutar y/o interactuar con diferentes lenguajes de programación, es decir, no solo podremos ejecutar código Java, como ocurre con la Java Virtual Machine, sino que también podremos ejecutar código de Python, Javascript, Scala, etc.

Con GraalVM podremos crear una imagen nativa de nuestro microservicio, lo que nos permitirá optimizar su tiempo de arranque.

Para ello, tendremos primero que crear una imagen nativa con Docker (en el fichero docker-build.sh del proyecto tendremos más información):

```
docker build . -t my-microservice
```

De esta forma compilamos nuestro microservicio con native-image, una

propiedad perteneciente a GraalVM. A través de este proceso se producen todas las optimizaciones necesarias que dan como resultado una imagen Docker de nuestro microservicio. Dichas optimizaciones están basadas en AOT (Ahead of Time Compilation).

Diferencias con Spring

La API de Micronaut está inspirada en Spring y Grails, lo que proporciona un entorno familiar si hemos tenido experiencia con estos frameworks. Aunque también tiene algunas características nuevas que lo diferencian:

1. Micronaut proporciona de serie soporte para microservicios sin necesidad de que instalemos dependencias de terceros.
2. Tiene un tiempo de inicio más rápido y significativamente menor consumo de memoria que Spring, gracias a AOT.
3. Como el tiempo de inicio es reducido, Micronaut se puede usar para

aplicaciones serverless.

4. Ofrece de serie soporte para la programación reactiva con RxJava 2.
5. Micronaut soporta GraalVM, una máquina virtual que puede compilar el código de diversos lenguajes de programación a código nativo de la máquina en la que se está ejecutando, lo que permite acelerar el arranque de la aplicación aún más. El equipo de Spring también está trabajando con GraalVM para liberar una versión estable con ella en un futuro.

Micronaut ofrece una serie de dependencias que se pueden instalar para hacer más fácil la integración con Spring. La documentación se puede consultar en Micronaut for Spring.

Micronaut CLI

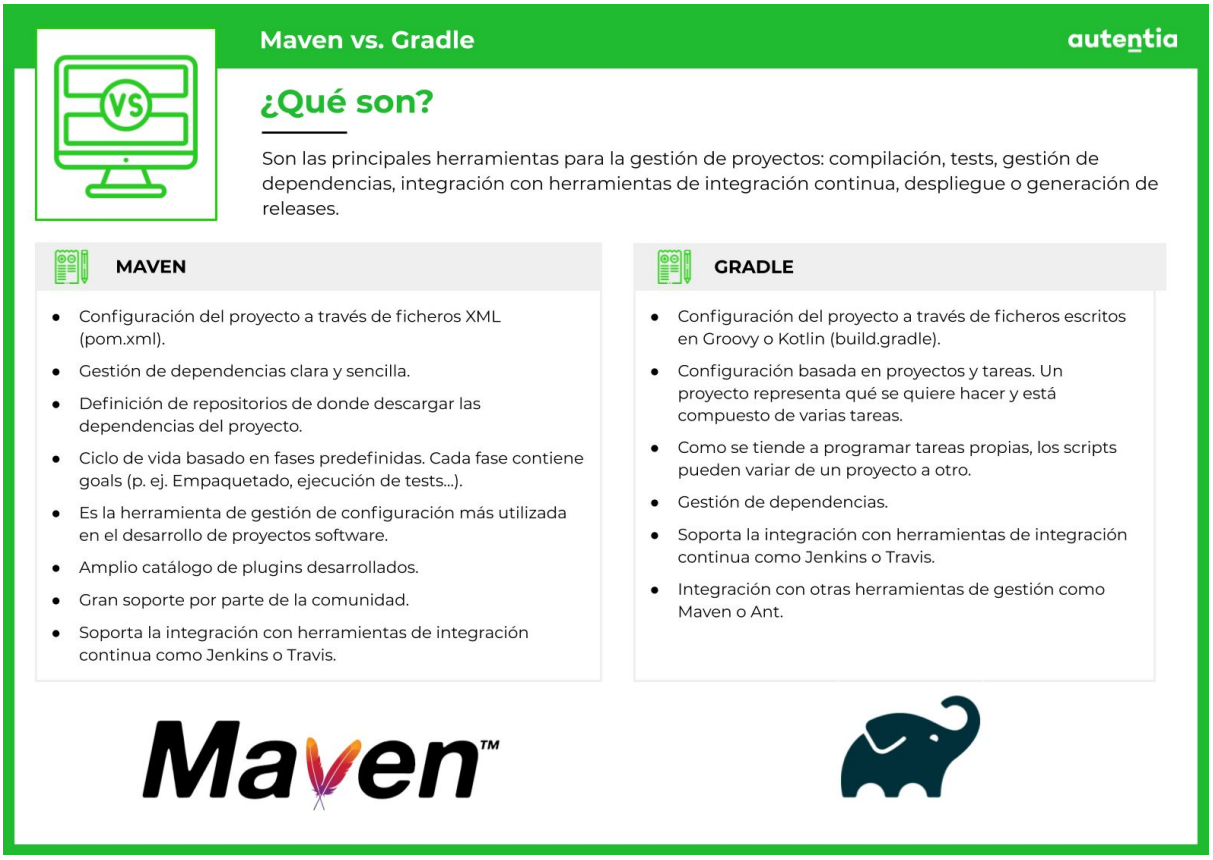
El Command Line Tool de Micronaut, por sus siglas, CLI nos permite crear el esqueleto de nuestra aplicación desde el terminal, no es obligatorio para crear una aplicación, pero sí recomendable y muy útil.

El comando para poder crear una aplicación basada en Micronaut desde la terminal es el siguiente:

```
mn create-app com.example.myapp
```

Si no especificamos nada, la gestión de dependencias por defecto será con Gradle, si queremos que sea con Maven, debemos añadirle el siguiente atributo:

```
-build maven
```



Utilizando la opción `-f (--features)` podemos definir qué características usar. Por ejemplo, con el siguiente comando, la aplicación es preparada para que se compile como imagen nativa.

```
mn create-app com.example.myapp -f graal-native-image -b maven
```

Si queremos listar todas las características o dependencias que podemos añadir a nuestro proyecto, lo podemos hacer a través del siguiente comando:

```
mn create-app --list-features
```

Una vez tengamos nuestra aplicación creada, podremos abrir el proyecto con un IDE y empezar a programar.

Inyección de dependencias en tiempo de compilación

Una de las principales características de Micronaut y que lo diferencia de otros frameworks como Spring, es que Micronaut tiene AOT (Ahead of Time Compilation) lo que le permite generar toda la metainformación que necesita el framework para funcionar en tiempo de compilación, de forma que aunque tengamos 10, 20 o 50 beans, no tiene que leerlos porque que esa información ya existe cuando arranca, a diferencia de Spring que lo hace en tiempo de arranque. Esto permite que el despliegue de un microservicio con Micronaut use poca memoria y a su vez sea veloz.

Reactive IO

Micronaut está implementado sobre Netty, que es un framework basado en eventos y un modelo de entrada/salida no bloqueante. Gracias a ello, implementar nuestras APIs reactivas es muy sencillo, dejando el control a Micronaut para decidir si una petición debe tratarse de manera síncrona o asíncrona.

RxJava 2 es la biblioteca de programación reactiva que viene de serie instalada en Micronaut, aunque también tiene soporte para otras bibliotecas como RxJava 3 o Project Reactor.

¿Cómo sabe Micronaut si la petición es bloqueante o no? Lo puede saber a través de la respuesta del *controller*:

- Si el método del controller devuelve un **tipo no bloqueante** como por ejemplo, Observable de RxJava, la petición se servirá de manera asíncrona usando el event loop de Netty.
- Si el método del controller devuelve **un tipo bloqueante**, por ejemplo, un String, la petición es bloqueante y será tratado por el pool de hilos creado al iniciarse la aplicación. El pool de hilos se puede

configurar.

Cloud Native Features

Micronaut fue diseñado para poder construir microservicios en la nube. Provee herramientas para gestionar muchas de las características más demandadas en sistemas cloud y además se encuentran directamente en Micronaut para su uso.

Algunos de las características incluidas son:

- Distributed Configuration.
- Service Discovery.
- Client-Side Load-Balancing.
- Distributed Tracing.
- Serverless Functions.

Algunos de los servicios en la nube, como por ejemplo Eureka o Consul para gestionar el Service Discovery, se pueden iniciar localmente para desarrollo o pruebas a través de Kubernetes o Docker. A su vez, para el Client-Side Load-Balancing, Micronaut utiliza una distribución round-robin.

En cuanto al monitoreo de la aplicación, Micronaut ofrece varios endpoints que se pueden configurar y activar individualmente:

- **/beans**: Información sobre los componentes cargados.
- **/health**: Disponibilidad de la aplicación.
- **/info**: Información adicional desde configuración y componentes que implementen InfoSource.
- **/metrics**: Metricas a través de Micrometer.
- **/refresh**: Recargar los componentes.
- **/routes**: Información del routing de la app.
- **/loggers**: Información sobre mensajes de logs.

A través de la API de OpenTracing integrando “Zipkin” o “Jaeger”, podremos visualizar informes agregados de latencia, dependencia y errores.

Con los perfiles de función o función-aws, es bastante sencillo desarrollar y desplegar funcionalidades individuales para infraestructuras *serverless*, es tan sencillo como hacerlo a través del siguiente comando:

```
mn create-function “ejemplo”
```

Esto nos daría algo como lo siguiente:

```
@FunctionBean("ejemplo")
public class EjemploFunction
             implements Function<User, Single<Event>> {

    @Inject EjemploService service;

    @Override
    public Single<Event> ejemplo(User user) {
```

```
        return service.ejemplo(user).singleOrError();
    }
}
```

Estos son sólo unos ejemplos de las muchas herramientas que se pueden integrar de manera muy sencilla en nuestros sistemas cloud a través de Micronaut. Muchas de estas características están inspiradas en Spring, dado que fueron desarrolladas por personas que habían trabajado en el proyecto de Spring Cloud. Si quieres ampliar información sobre estas características, puedes encontrar más información aquí.

Parte 4

Kotlin

Introducción a Kotlin

Kotlin es un lenguaje de programación multiplataforma, con tipado estático, de propósito general y open source. Kotlin se ejecuta principalmente, sobre la máquina virtual de Java (JVM) aunque también puede ejecutarse sobre Javascript o a lenguaje nativo (binarios según arquitectura).

Kotlin es un lenguaje orientado a objetos que contiene una ingente cantidad de elementos propios de programación funcional. Además, está preparado para soportar una total compatibilidad con Java, pudiendo tener un proyecto con código en Java y en Kotlin y que interactúen entre sí. Esto permite adoptar Kotlin de forma gradual y ganar adeptos sin mucho esfuerzo.

Para el agrado de todos, Kotlin es un lenguaje muy expresivo, conciso y puede reducir la cantidad de código boilerplate (como por ejemplo: los métodos getters/setters, los punto y coma para separar instrucciones, etc.). Esto nos aporta que el código sea más rápido de escribir, de leer y que además, sea menos propenso a tener errores.

Kotlin también nos aporta seguridad al gestionar los nulos de forma más eficaz que Java. Por defecto, en Kotlin los tipos no aceptan valores nulos. Es el desarrollador el responsable de incluirlos en su código.

Si ya sabes Java estás de enhorabuena: la curva de aprendizaje con Kotlin no es muy alta y si prosigues la lectura acabarás por dominarlo.

Un poco de historia

Kotlin fue creado en 2011 por JetBrains, empresa que te sonará si alguna vez has utilizado alguno de sus IDEs como IntelliJ o Webstorm. A partir de febrero de 2012, todo el código de Kotlin se hizo open source.

La inspiración de JetBrains para desarrollar Kotlin fue la falta de funcionalidades que ellos buscaban en la mayoría de lenguajes de programación a excepción de Scala. Lo malo que tenía Scala era el lento tiempo de compilación. Para diferenciarse de este lenguaje, uno de los primeros objetivos de Kotlin fue el de «compilar tan rápido como Java».

El nombre de Kotlin está basado en la isla de Kotlin cerca de la ciudad de San Petersburgo, y es un evidente guiño a Java, que también es una isla pero en Indonesia.
La primera versión de Kotlin se liberó en febrero de 2016 y se consideró la primera versión estable que garantizó la retrocompatibilidad. En 2017 Google sorprendió a la comunidad desarrolladora informando del soporte de Kotlin en Android, al mismo nivel de importancia que Java. Finalmente, en 2019 anunció que Kotlin sería el lenguaje idóneo para desarrollar en Android, relegando a Java a un segundo nivel.

¿Por qué usar Kotlin?

Cada pocos años aparecen nuevos lenguajes de programación que ambicionan cambiar el modo de trabajar de los desarrolladores. Sin embargo, la mayoría se pone de moda pero no acaba llegando a su objetivo. Desde que Kotlin fue creado en 2011 ha ido poco a poco formando parte de muchos repositorios de código.

Si todavía no te hemos convencido de usar Kotlin, hay algo que es clave: Kotlin ofrece muy buenas funcionalidades que te habilitan ser más conciso y entendible que Java sin sacrificar el rendimiento o la seguridad. De hecho, una vez que aprendes Kotlin y vuelves a programar en Java, acabas echando de menos toda la velocidad de desarrollo de Kotlin.

La brevedad es clave en la productividad del día a día de un desarrollador. Imaginemos por un momento que hacemos un viaje en el tiempo al pasado y que añadimos una nueva instrucción al final de nuestro programa en

ensamblador. La limitación del lenguaje nos obliga a retener demasiada información (como registros, direcciones de memoria, etc.) en nuestra cabeza para no equivocarnos.

En el presente ya no es necesario almacenar tanta información en la cabeza pues disponemos de nombres, objetos, funciones... ¡y además sabemos lo que hacen o lo que pueden o podrán contener! Se llaman lenguajes de programación de alto nivel y te permiten poner más ideas por cada línea de código. Siguiendo el símil, Kotlin está incluso a un nivel superior y te permite multiplicar tu expresividad.

Debes tener en cuenta que Kotlin no busca sacrificar la comprensión del código por hacer las cosas de forma más breve.

Instalación de Kotlin

Hay muchas maneras de instalar y usar Kotlin. Vamos a ver algunas de las más usadas normalmente. Todas las instrucciones están en esta página.

Instalación de Kotlin localmente

Para instalar Kotlin en tu máquina puedes descargar el ZIP con el compilador de Kotlin desde esta página GitHub Releases. Luego tienes que descomprimir el zip en la carpeta donde quieras que esté Kotlin y opcionalmente, añadir la carpeta *bin* al variable de ruta del sistema operativo. La carpeta bin contiene los scripts que se necesitan para compilar y ejecutar el código de Kotlin.

Otra opción sería utilizar el administrador de paquetes SDKMAN. Puedes leer sobre este administrador de paquetes en este tutorial de adictosaltrabajo.com. Si no lo tienes instalado puedes instalarlo con el comando:

```
$ curl -s https://get.sdkman.io | bash
```

Luego puedes instalar Kotlin así:

```
$ sdk install kotlin
```

Si tienes Mac/Linux también puedes usar otro famoso administrador de paquetes HomeBrew. Puedes instalar Kotlin a través de Homebrew utilizando el siguiente comando:

```
$ brew update
$ brew install kotlin
```

Si usas Ubuntu 16.04 o más nuevo, puedes instalar el compilador de Kotlin con la ayuda de Snap.

```
$ sudo snap install --classic kotlin
```

Siempre puedes comprobar si todo es correcto comprobando la versión instalada del compilador de Kotlin.

```
> kotlin -version
    Kotlin version 1.3.50-release-112 (JRE 13+33)
```

Ejecución de Kotlin en local

Kotlinc

Para ejecutar el código de Kotlin se usa el comando *kotlinc*. *Kotlinc* es un alias para el comando *kotlinc-jvm*. Si quieres compilar el código de Kotlin a Javascript tienes que usar *kotlinc-js*. Primero, tenemos el fichero con el código de Kotlin con la extensión *.kt*. El típico ejemplo de HelloWorld en el caso de Kotlin sería así:

```
fun main() = println("Hello World!")
```

Vamos a compilar nuestro código que está en el fichero hello.kt:

```
> kotlinc hello.kt
```

Este comando produce el fichero HelloKt.class que a su vez se puede ejecutar con el comando kotlin.

```
> kotlin HelloKt
```

Si quieres incluir Kotlin runtime puedes añadir el argumento *-include-runtime*. De esta manera, crearás un JAR file que sea autónomo y ejecutable con el comando jar.

```
kotlinc-jvm hello.kt -include-runtime -d hello.jar
java -jar hello.jar
```

Si conoces el mundo Java ya te has dado cuenta que los comandos *kotlinc* y *kotlin* son iguales a los comandos *javac* y *java* para los ficheros de Java.

Script

Si quieres utilizar Kotlin como language para tu script, también necesitarás el comando kotlinc. Cambiando la extensión de tu fichero a .kts, puedes compilarlo como script.

```
kotlinc -script hello.kts
```

El script no tiene la función *main* y todo el código se ejecuta consecuentemente, desde la primera línea hasta el final.

REPL

Para tener una experiencia más interactiva existe una herramienta que se llama Kotlin REPL (Read Eval Print Loop). Es una herramienta que compila el código de Kotlin en un terminal en tiempo real. Una vez hayas escrito el código en el terminal, puedes ver los resultados inmediatamente. Se llama con el comando *kotlinc* sin argumentos.

```
▸ kotlinc
Welcome to Kotlin version 1.3.50 (JRE 11.0.4+11) Type :help for
help, :quit for quit
```

```
>>> println("Hello, World!")
Hello, World!
>>> var name = "Antonio"
>>> println("Hello, $name!")
Hello, Antonio!
>>> :help
Available commands:
:help
:quit
:dump bytecode
:load <file>

>>> :quit
```

REPL es una forma rápida y fácil de evaluar expresiones de Kotlin sin iniciar un IDE completo. Úsalo si no quieres crear un proyecto completo o si necesitas hacer una demostración rápida.

Ejecución de Kotlin en remoto

Puedes ejecutar tu código de Kotlin en la siguiente página Kotlin Playground. *Kotlin Playground* te da una manera fácil de experimentar con Kotlin, explorar funciones que no has utilizado o simplemente ejecutar Kotlin en sistemas que no tienen un compilador instalado. Te da acceso a la última versión del compilador, junto con un editor basado en web que te permite compilar el código sin instalar Kotlin localmente.

Simplemente, escribe tu código y haz clic en el botón Play para ejecutarlo. El botón Settings (el icono de rueda dentada) te permite cambiar las versiones de Kotlin, decidir en qué plataforma ejecutar (JVM, JS, Canvas o JUnit) o agregar argumentos al programa.

Características de Kotlin

Conciso

No es necesario punto y coma

Kotlin, a diferencia de otros lenguajes de programación como C, C++ o Java, no necesita incluir al final de cada sentencia de código un punto y coma ";". Aunque se pueden seguir usando y el código compilará sin ningún tipo de problema, el entorno de desarrollo integrado (IDE) te avisará sugiriendo que ya no son necesarios.

Con este detalle, Kotlin quiere ofrecer a los desarrolladores una manera más limpia de escribir código, quitando elementos sin valor de nuestro código fuente.

Inferencia de tipos

La inferencia de tipos en Kotlin significa que el compilador deducirá el tipo cuando le sea posible.

En el siguiente ejemplo vemos como se hace con Java, donde sí es necesario decir de qué tipo es la variable:

```
String helloWorld = "Hola Mundo!"
```

En cambio, en Kotlin:

```
var helloWorld = "Hola Mundo!"
```

Esto es gracias a la inferencia de tipos. También hay que tener en cuenta que el tipo de variable se define una vez, por lo que si se intenta cambiar su tipología en otro punto del programa, se obtendrá un error de compilación.

```
var helloWorld = "Hola Mundo!"

helloWorld = 42 //Error de compilación
```

Tipos en Kotlin

En Kotlin no existen los tipos primitivos como tal, dado que todo los tipos son objetos. De esta forma, dotamos a estos tipos de funciones y propiedades, permitiéndonos operar de una forma más simple con ellos.

Numbers

Para los números enteros contamos con los tipos *Byte*, *Short*, *Int* y *Long*.

Para los números con coma flotante disponemos de los tipos *Float* y *Double*, que difieren en cuántos decimales son capaces de almacenar, acorde al estándar IEEE 754.

Characters

Los caracteres están representados por el tipo *Char* y van entre comillas simples. Los caracteres especiales se pueden usar a través de un backslash, como vemos en los siguientes casos: `'\t'`, `'\b'`, `'\n'`, `'\r'`, `'\''`, `'\"'`, `'\\'` y `'\$'`. Para codificar cualquier otro carácter, podemos usar la sintaxis Unicode: `'\u007C'`.

Booleans

El tipo *Boolean* representa a los booleanos en Kotlin. Este tipo de dato puede tener dos posibles valores: `true` o `false`.

Arrays

Los arrays en Kotlin están representados por la clase *Array*, la cual tiene las funciones `get()` y `set()`, el atributo `size` y varias funciones de gran utilidad, como la función `iterator()`.

Para crear un array podemos usar la función `arrayOf()` y pasar los valores que queremos que almacene. `var miArray = arrayOf('a','b','c') -> miArray = ['a','b','c']`.

Otra opción es usar el constructor de la clase *Array*, el cual toma dos argumentos: el tamaño de tipo *Int* e init, que se trata de una función lambda que se invoca por cada índice cuando estamos creando el array, como podemos ver en el siguiente ejemplo:

```
var squares = Array(5) { i -> (i * i).toString() }
// squares = ["0", "1", "4", "9", "16"]
```

Conversión explícita entre tipos

Una de las sorpresas que Kotlin aporta a los desarrolladores de Java es que los tipos más cortos no son convertidos automáticamente a tipos más largos. Por ejemplo, en Java es perfectamente normal escribir el código:

```
int quantity = 3
long balance = quantity
```

En Kotlin no sería posible asignar valores de tipo Byte a una variable del tipo Int sin una conversión explícita. Esto se debe a que los tipos pequeños

no son subtipos de los tipos grandes.

La forma correcta de convertir un valor pequeño a un tipo más grande es usando las conversiones explícitas que ofrece cada tipo.

Por ejemplo, para el caso anterior lo haríamos de la siguiente manera:

```
var quantity: Int = 3
var balance: Long = quantity.toFloat()
```

Diferencia entre val y var

Kotlin cuenta con dos palabras reservadas para la declaración de variables, val y var. Val se usa para la declaración de constantes ya que solo se inicializa una vez y no permite otras asignaciones, lo que hace que sea una variable inmutable. Para los que vienen del mundo Java, el uso de val se asemeja al final. En cambio, la palabra reservada var sí que permite realizar múltiples asignaciones convirtiéndose en una variable mutable en Kotlin.

En conclusión, las variables definidas con val son sólo de lectura y por otro lado, las que están definidas con var sirven tanto de lectura como de escritura.

String templates

Los Strings Templates nos permiten ejecutar piezas de código en una cadena de caracteres y el resultado de esa pieza se concatena dentro de este. Es útil dado que nos ayuda a componer una cadena de la manera que deseemos. Por ejemplo:

```
var person: String = "Tony Stark"
var superHero: String = "Iron Man"

println("$person is $superHero")
// prints Tony Stark is Iron Man

println("$superHero has ${superHero.length} characters")
// prints Iron Man has 8 characters
```

Programación orientada a objetos en Kotlin

La programación orientada a objetos en Kotlin se parece bastante a la de Java a grandes rasgos, pero tiene muchos matices interesantes que tenemos que tener en cuenta para mejorar nuestra productividad.

Clases

Si repasamos los conceptos básicos, decimos que una clase es una plantilla preparada para crear nuevos objetos. Para la creación de dichos objetos utilizamos constructores. En Kotlin puede haber un constructor primario y uno o más secundarios. Veamos un ejemplo de una clase y la creación de un objeto:

```
class Persona constructor(nombre: String)

fun main() {
    val jose = Persona("Jose")
}
```

Hemos creado un objeto llamando al constructor primario que forma parte de la cabecera de la clase. Nótese que el parámetro *nombre* es solamente del constructor primario y no lo guardamos en la instancia que creamos.

La palabra reservada *constructor* la podemos eliminar para el constructor

primario siempre y cuando éste no tenga anotaciones o modificadores de visibilidad. Si no hay modificador de visibilidad, aquello que estés definiendo (clase, constructor, objeto...) será público. Ten en cuenta que tampoco hemos utilizado la palabra *new*, pues no existe en Kotlin.

Podemos incluso no definir constructor alguno como vemos en el siguiente ejemplo:

```
class Vacia

fun main() {
    val variable = Vacia()
}
```

Antes de seguir, veamos el mismo ejemplo en Java:

```
public class Vacia {}
public static void main(String[] args) {
    Vacia variable = new Vacia();
}
```

La mejora es sorprendente, ¿no? Hemos declarado una clase sin llaves, sin paréntesis y sin repetir términos redundantes.

Constructores

En los anteriores ejemplos no hemos necesitado realizar ninguna acción tras construir un objeto. Pese a ser lo normal al crear objetos, en algunas ocasiones es preciso realizar algunas operaciones. En Kotlin podemos ejecutar un bloque de código después de crear el objeto con el constructor primario:

```
class Resta(minuendo: Int, sustraendo: Int) {
    private val narrador = "Resta entre: $minuendo y $sustraendo"
    private var resultado = 0;
```

```
    init {
        println(narrador)
        resultado = minuendo - sustraendo
    }
    fun resultado() = resultado
}

fun main() {
    val resta = Resta(5, 2)
    println(resta.resultado()) // Imprime 3
}
```

En el bloque *init* imprimimos lo que estamos haciendo y calculamos el resultado. Además, permitimos devolver dicho resultado declarando una función miembro de la clase. Recuerda que *minuendo* y *sustraendo* sólo son parámetros del constructor primario mientras que *narrador* y *resultado* sí son atributos de la clase.

Como en otros lenguajes de programación, en Kotlin también se puede hacer referencia a otros constructores:

```
class Constructor {

    init {
        println("Código primer constructor")
    }
    constructor(nombre: String) : super() {
        println("Código constructor con parámetro string $nombre")
    }
    constructor(nombre: String, numero: Int) : this(nombre) {
        println("Código constructor con parámetro int $numero")
    }
}
fun main() {
    val objeto = Constructor("nuevo", 3) // ejecuta el código de
los 3 constructores
}
```

Podemos referenciar al constructor primario con *super()* o a otros secundarios con *this(..)* y encadenarlos todos como en el ejemplo.

Una clase podría no poderse crear si declaramos su constructor primario como privado:

```
class NoCreable private constructor()
```

Propiedades

En una clase podemos definir atributos o propiedades de sólo lectura (*val*) o mutables (*var*) a través del constructor primario. Tras su instanciación, podremos acceder a las propiedades y modificar su valor.

```
class Animal(
    val nombre: String,
    val especie: String,
    var crias: Int, // Kotlin permite comas en el último parámetro
)
fun main() {
    val ciervo = Animal("Bambi", "ciervo", 0)
    ciervo.crias = 1
    println(ciervo.crias) //  Imprime 1
}
```

Una variable *val* en Kotlin es similar a una variable *final* en Java. Una vez que asignemos un valor no podremos reasignarlo. Como también vemos en el ejemplo, Kotlin nos permite e invita a que dejemos una coma en el último parámetro.

En Kotlin podemos definir valores por defecto en el constructor primario y también invocar nombrando los parámetros de forma desordenada:

```
class Gato(
    val nombre: String,
```

```
    val especie: String = "Felis silvestris catus",
    var vidas: Int = 7
)
fun main() {
    val garfield = Gato("Garfield") // obtiene el nombre de
especie y las vidas por defecto
    val doraemon = Gato("Doraemon", "gato cósmico") // obtiene las
vidas por defecto
    val manchitas = Gato("Manchitas", "gato", 2)
    val copito = Gato("Copito", vidas = 5) // nombre de especie
por defecto
    val bigotes = Gato(vidas = 5, especie = "Gato", nombre =
"Bigotes")
    val gloton = Gato(vidas = 5, especie = "Gato", "Glotón") //
Error compilación
}
```

Accesores

Hemos hablado de propiedades que se declaran directamente en el constructor primario pero no es la única manera. También podemos declararlas en el cuerpo de la clase y podremos redefinir sus métodos de acceso y modificación (o getter y setter):

```
class Clase {
    var variable: String = "variable"
        get() = "$field-get"
        set(value) {
            field = "$value-set"
        }

    var noModificable: String = "invariante"
        private set
}
fun main() {
    val clase = Clase()
    println(clase.variable) // variable-get
```

```
    clase.variable = "otra"
    println(clase.variable) // otra-set-get
    println(clase.noModificable) // invariante
    clase.noModificable = "otro" // Error de compilación, setter
privado
}
```

Los getters y setters siempre son opcionales pero admiten modificaciones y cambiar su visibilidad como en la mayoría de lenguajes de programación orientada a objetos.

Kotlin también introduce para una propiedad el concepto de *lateinit*. Esto significa que dicha propiedad debe inicializarse antes de que se acceda a ella. En caso contrario, se devolverá una excepción *UninitializedPropertyAccessException* en tiempo de ejecución:

```
class Ejemplo {
    lateinit var asignada: String
    lateinit var perezosa: String
    init {
        asignada = "si"
    }
}
fun main() {
    val ejemplo = Ejemplo()
    println(ejemplo.asignada) // si
    println(ejemplo.perezosa) // Excepción en tiempo de ejecución
    ejemplo.perezosa = "también"
    println(ejemplo.perezosa) // "también"
}
```

Herencia

Específica de la programación orientada a objetos, la herencia se define como una clase que se crea a partir de otra existente. La nueva clase

contiene las propiedades y funciones de la clase padre. En Kotlin todas las clases son *final* por defecto, esto quiere decir que no se pueden extender o declarar clases hijas. Para permitirlo debemos añadir la palabra reservada *open* en la cabecera de la clase o método sobre el que permitimos la herencia. Veamos un ejemplo:

```
open class Animal(nombre: String) {
    open fun comer() {}
    fun desplazarse() {
        println("Me desplazo como animal")
    }
}
class Omnivoro(nombre: String) : Animal(nombre) {
    override fun comer() {
        println("Como de todo")
    }
    override fun desplazarse() { } // Error compilación. No se
puede sobreescribir funciones no 'open'
    fun buscarAlimento() {
        super.desplazarse()
        println("Encuentro alimento")
    }
}
```

Como podemos observar, no podemos sobreescribir métodos que no sean *open*. Por otra parte, tal y como pasa en otros lenguajes de programación como Java, podemos referenciar a la implementación de la clase padre mediante *super*.

En Kotlin también podemos definir clases abstractas con el término *abstract* que no pueden ser instanciadas. A diferencia de otros lenguajes de programación, un método abstracto no puede tener cuerpo y debe ser implementado por otra clase que lo herede. En el término *abstract* viene implícito el término *open*.

```
open class Forma {
    open fun dibujar() {}
```

```
}
abstract class Poligono : Forma() {
    abstract override fun dibujar()
}
class Rectangulo : Poligono() {
    override fun dibujar() {
        println("Dibujo como rectángulo")
    }
}
```

Por defecto, todas las clases heredan de la clase *Any*, que implementa por defecto los métodos *equals()*, *hashCode()* y *toString()*. *Any* es similar a *Object* en Java, veamos el ejemplo:

```
fun main() {
    val ejemplo = Vacia()
    println(ejemplo is Any) // true

    val oso = Omnivoro("Oso")
    println(oso is Any) // true
    println(oso is Animal) // true
    println(oso is Omnivoro) // true
}
```

El operador *is* determina si el objeto es instancia de una clase. Operador muy similar a *instanceof* en Java.

Interfaces

Las interfaces pueden contener métodos abstractos y también pueden tener implementaciones de los mismos. A diferencia de las clases abstractas, una interfaz no puede almacenar estado, es decir, puede contener propiedades pero éstas deben ser abstractas o tener definido el acceso correspondiente (getter). Veamos un ejemplo:

```
interface Bipedo {
    fun andar() {
        println("Ando como Bipedo")
    }
    fun correr()
}
interface Humanoide : Bipedo {
    val pulgares: Boolean
    override fun andar() {
        super.andar()
        println("Y también como Humanoide")
    }
}
class Persona : Humanoide {
    override val pulgares: Boolean
        get() = true
    override fun correr() {
        println("Corro como una persona")
    }
}
```

Como vemos en el ejemplo, también puede haber herencia entre interfaces. La propiedad *pulgares* en la interfaz *Humanoide* es abstracta y requiere que sea implementada en aquellas que la extiendan, como *Persona* en el ejemplo. También disponemos del término *super* para poder invocar a la interfaz que estamos implementando.

Como curiosidad, Kotlin soporta herencia múltiple y para hacer referencia a métodos de clases padre o de clases contenedoras, usamos el término *super* de otra manera:

```
interface A {
    fun foo() { println("A-Foo")}

    fun bar()
}
```

```
interface B {
    fun foo() { println("B-Foo")}

    fun bar() { println("B-Bar")}

    fun foobar() { println("B-FooBar")}
}

class C : A {
    override fun bar() { println("C-bar")}
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
        println("D-foo")
    }

    override fun bar() {
        super<B>.bar() // =super.bar()
        println("D-bar")
    }
}
```

Si nos fijamos en la clase C sólo se ha definido el método *bar()* por no haber sido definida anteriormente. Sin embargo, para la clase D se nos obliga a redefinir todos los métodos heredables que entran en conflicto.

Niveles de acceso

Los niveles de acceso o visibilidad se pueden aplicar a clases, objetos, constructores, propiedades y sus getters/setters. Existen los siguientes niveles de acceso:

- **public**: es la visibilidad por defecto. Es visible desde cualquier lugar.
- **private**: solo es visible en la clase que se declara, incluyendo todos

sus miembros.

- **protected**: a diferencia de Java, Kotlin no incluye visibilidad de paquete aquí. Por tanto, el término protected es igual a private pero dando visibilidad también a las subclases. Si sobreescribimos un método protected en una subclase sin hacer explícito su nivel de acceso, tendrá visibilidad protected.
- **internal**: es visible para cualquiera que esté dentro del mismo módulo. Un módulo es un conjunto de ficheros de Kotlin que han sido compilados juntos, por ejemplo, un módulo de un multiproyecto Maven.

En Kotlin no existe la visibilidad de paquete (visibilidad por defecto o también protected en Java) por no proveer de encapsulamiento real. La visibilidad *internal* es suficiente para proteger la información dentro de un mismo proyecto sin exponerla hacia fuera.

Extensiones

Si tenemos una dependencia con un tercero, ya sea una biblioteca o una función propia de Java o Kotlin, pero echamos en falta una función que nos facilite la vida, podremos decorarla de la siguiente manera. Veamos un ejemplo:

```
fun List<String>.palindromos(): List<String> {
    return this.filter {
      palabra ->
      StringBuilder(palabra).reverse().toString().equals(palabra,
ignoreCase = true)
    }
}

fun main() {
    val nombres = listOf("Ana", "Juan", "Antonio")
    val palindromos = nombres.palindromos()
    println(palindromos) // ["Ana"]
```

```
    val numeros = listOf(1,2,3,4,5)
    val capicuas = numeros.palindromos() // Error de compilación,
sólo se declaró para String
}
```

Ten en cuenta que las extensiones no sobreescriben las clases que extienden y que se resuelven de manera estática. Con otro ejemplo lo entenderás:

```
open class Animal
class Perro : Animal()
fun Animal.comer() {
    println("Comer como animal")
}
fun Perro.comer() {
    println("Comer como perro")
}
fun resolucionEstatica(animal: Animal) {
    animal.comer()
}
fun main() {
    resolucionEstatica(Perro()) // Comer como animal
}
```

Data Classes

Al programar es necesario crear clases para almacenar información. Por ello que en Kotlin se pueda marcar una clase como *data* para mejorar nuestra productividad.

```
class Suplente(val nombre: String)
data class Futbolista(val nombre: String, val edad: Int, val
goles: Int)

fun main() {
```

```
    val suplente = Suplente("Suplente")
    val futbolista = Futbolista("Futbolista", 33, 719)
    println(suplente) // com.autentia.data.Suplente@3ac3fd8b
    println(futbolista) // Futbolista(nombre=Futbolista, edad=33,
goles=719)
    println(suplente.equals(Suplente("Suplente"))) // false
    println(futbolista.equals(Futbolista("Futbolista", 33, 719)))
// true
    println(futbolista.equals(futbolista.copy())) // true
    println(futbolista.component2()) // 33
}
```

El compilador automáticamente genera métodos utilizando las propiedades que hemos declarado, como *nombre*, *edad* y *goles* en el caso del ejemplo. Estos métodos autogenerados son:

- *equals()/hashCode()*
- *toString()*
- Funciones *componentN* correspondientes al orden de los parámetros.
- *copy()*

No obstante, las clases *data* tienen algunas restricciones:

- No pueden ser *abstracts*, *open*, *sealed* or *inner*.
- El constructor primario debe tener al menos un parámetro.
- Los parámetros del constructor primario deben ser marcados como *val* o *var*.

Sealed Classes

Parecidas a las clases *abstract*, con el término *sealed* se restringe el número de las subclases a aquellas que son declaradas en el mismo fichero. En ocasiones, es necesario representar una jerarquía limitada y cerrada. Al estar limitado el número de subclases que la pueden extender, el compilador se dará cuenta y nos obligará en tiempo de compilación a

cubrir todos los casos posibles. Veamos un ejemplo:

```
sealed class Fruta(numero: Int) {
    class Platano(numero: Int) : Fruta(numero)
    class Manzana(numero: Int, color: String) : Fruta(numero)
}
fun comerPiel(fruta: Fruta) = when(fruta) {
    is Fruta.Platano -> false
    is Fruta.Manzana -> true
}
```

Si añadimos otra subclase a *Fruta*, estaríamos obligados a cubrir el caso de la nueva subclase añadida en la función *comerPiel*.

Las clases *sealed* se asemejan a los enumerados pero tienen dos cosas a su favor:

- Se pueden establecer jerarquías dentro de una clase *sealed.*
- Puede haber más de una instancia por subclase .

Por otro lado, también se asemeja a las clases *abstract* pero limitando las clases que las pueden extender.

Genéricos

Los genéricos son tipos parametrizados que sirven para mejorar la productividad en aquellas clases o funciones donde saber el tipo no es relevante. Un ejemplo importante es el orden de una lista: independientemente del tipo contenido podemos ordenarla sin tener constancia del tipo sobre el que se ordena. Hacer esto mismo sin genéricos daría lugar a más líneas de código y a un código menos legible.

Veamos un simple ejemplo que se queda con las posiciones pares de una lista:

```
data class ListaPares<T>(private val list: List<T>) {
```

```
    fun pares(): List<T> {
        return list.filterIndexed { index, _ -> index % 2 == 0 }
    }
}
fun main() {
    var listaNumber: List<Number> = listOf(1, 2, 3)
    var listaEnteros: List<Int> = listaNumber // Error de
compilación. Int no es subtipo de Number.
    println(ListaPares(listaNumber).pares()) // [1, 3]
    var listaString: List<String> = listOf("a", "b", "c", "d",
"e")
    println(ListaPares(listaString).pares()) // [a, c, e]
}
```

En el ejemplo tenemos un tipo T que puede ser un tipo *Number* o un *String*, pero no afecta al comportamiento que queremos en la función. Con los genéricos empezamos a tener problemas en el momento en el que queremos transformar un tipo T a otro, siendo este subtipo de otro o hermanado. Para ello, Kotlin utiliza el término de *variance* que tiene bastantes matices añadidos a los que explicaremos aquí. Si quieres tener un conocimiento más profundo sobre todo esto, puedes consultar la documentación oficial de Kotlin.

Comodines para tipos genéricos

Para entenderlo bien es preciso mencionar la nomenclatura. Repasemos el ejemplo siguiente:

```
var lista: List<Any> = listOf(1, 2, 3)
```

El tipo producido en tiempo de ejecución es un *List<Int>*, es decir, hemos *escrito* en memoria una lista de números. Por otra parte, el tipo declarado y el que lo consume es *List<Any>*, esto quiere decir que es una lista que lee cualquier tipo. Profundizando un poco más vemos la definición de la clase

List en Kotlin con la siguiente firma:

```
public interface List<out E> : Collection<E> {...}
```

Observamos un término *<out E>*, cuyo homólogo en Java sería *<? extends E>*. Veamos en Java la clase *List* para el método *addAll*:

```
boolean addAll(Collection<? extends E> c);
```

Comodín out

El comodín *<out E>* sirve para admitir cualquier subtipo de E. Este comodín afecta al tipo que escribimos, producimos o que devolvemos. Veamos un ejemplo donde lo vamos a entender mejor:

```
class Ejemplo<T>

class OtroEjemplo<out T>

fun main() {
    val ejemplo1: Ejemplo<Any> = Ejemplo<String>() // Error de
compilación. Un tipo String no es un tipo Any (aunque sea subtipo)
    val ejemploOutAny: Ejemplo<out Any> = Ejemplo<String>() //
String es un subtipo de Any
    val ejemploOutString: Ejemplo<out String> = Ejemplo<Any>() //
Error de compilación. Un tipo Any no es subtipo de String
    val otroEjemplo: OtroEjemplo<Any> = OtroEjemplo<String>()
}
```

Si no declaramos el comodín, estaremos obligados a convertir explícitamente entre tipos donde uno es subtipo del otro como en el *ejemplo 1*. Si lo declaramos no hará falta hacerla. Esta capacidad de transformación que otorga el comodín *out* al tipo T se conoce como *covariant*.

Comodín in

De forma análoga al anterior comodín, el comodín *in* afecta al que lee, recibe o consume. Veamos que el término *<in T>* tiene el homólogo *<? super T>* en *Java* en la clase *TreeSet*:

```
public TreeSet(Comparator<? super E> comparator) {...}
```

El comodín *<in E>* sirve para admitir cualquier supertipo de *E*. Este comodín afecta al tipo que declaramos, leemos o consumimos. Veamos un ejemplo:

```
open class Hormiga
class Obrera : Hormiga()
class Caja<T>
class CajaIn<in T>
fun main() {
    val val1: CajaIn<Obrera> = CajaIn<Hormiga>() // Obrera es un
subtipo de Hormiga
    var val2: Caja<in Obrera> = Caja<Hormiga>()
    var val3: CajaIn<Hormiga> = CajaIn<Obrera>() // Error
compilación problema de tipos: Hormiga no es subtipo de Obrera
    var val4: CajaIn<out Hormiga> =
        CajaIn<Obrera>() // Error compilación. No se puede
declarar la proyección out si ya tenemos una proyección in
}
```

De igual forma que con el comodín anterior, si no ponemos *<in T>*, estaremos obligados a convertir explícitamente. Esta capacidad de transformación que otorga el comodín *in* al tipo *T* se conoce como *contravariant*.

Veamos otro ejemplo con funciones:

```
fun sustituirObrera(hormiga: Caja<Hormiga>): Caja<out Hormiga> {
    val obrera: Caja<in Obrera> = hormiga
    descansa(obrera)
    return Caja<Obrera>();
```

```
}
```

Otro ejemplo de aplicación de estos comodines sería en las *lambdas* pues éstas son *contravariant* para los tipos de sus argumentos y *covariant* con el tipo que devuelven.

También existe el comodín <*> para permitir que los dos comodines que hemos estado viendo puedan utilizarse a la vez:

```
val val6: Caja<in Nothing> = Caja<Obrera>()
val val7: Caja<out Any?> = Caja<Obrera>()
val val8: Caja<*> = Caja<Obrera>()
```

Al definir el tipo genérico *T*, también podemos declarar que el tipo *T* deba ser un subtipo de otra clase porque vamos a hacer uso de sus métodos:

```
interface Printable<T>
data class MiNumero(val numero: Int) : Printable<MiNumero>
fun <T : Printable<T>> imprimeEspaciado(list: List<T>) {
    list.map { e -> print(" $e ") }
}
fun main() {
    val numeros = listOf(1, 2, 3, 4) // List<Int>
    imprimeEspaciado(numeros) // Error compilación. Int no es un
subtipo de Printable<T>
    val misNumeros = listOf(MiNumero(1), MiNumero(2))
    imprimeEspaciado(misNumeros) // MiNumero(numero=1)
MiNumero(numero=2)
}
```

Clases anidadas

Las clases anidadas son clases que se declaran dentro de otras clases. Estas clases pueden ser interfaces dentro de clases, clases dentro de

interfaces o interfaces dentro de interfaces. Veamos un ejemplo:

```
class Facturacion(val año: Int, val bruto: Float) {
    inner class Factura(val id: Int, val empresa: String, val
bruto: Float) {
        inner class Impuesto(val porcentaje: Float) {
            fun impuesto() : Float{
                // la propiedad bruto sólo es accesible si la
clase se declara como inner
                return bruto * porcentaje // bruto del ámbito más
cercano, en este caso Factura
                return this@Facturacion.bruto * porcentaje //
explícitamente le indicamos que use bruto de la clase Facturación
            }
        }
    }
}
fun main() {
    val factura = Facturacion(2020, 100000F).Factura(1, "Empresa
S.A", 2000F)
    val impuesto = factura.Impuesto(0.21F)
}
```

Podemos utilizar el término *this* con una label como *@Facturacion* para indicar el ámbito al que hacemos referencia en caso de conflicto o necesidad.

Clases Enumeradas

En Kotlin cada tipo enumerado es un objeto inicializado. Al ser una clase, también aprovechamos el constructor primario. De igual forma que en Java, podemos tener funciones asociadas a los tipos enumerados. Veamos un ejemplo:

```
enum class Semana(val laborable: Boolean) {
    LUNES(true),
    ...
```

```
    DOMINGO(false); // necesario punto y coma si definimos
funciones para el enumerado
    fun horasOcio() = if (laborable) 2F else 5F
}
fun main() {
    val lunes = Semana.LUNES
    Semana.values().iterator().forEach { dia -> print(dia) }
    val miDia = "LUNES"
    val miDiaSemana = Semana.valueOf(miDia)
    println(miDiaSemana.horasOcio()) // 2
}
```

Fácil, ¿no? Veamos que tenemos una clase y que podemos implementar interfaces. Vamos ahora a llevarnos la función *horasOcio* a una interfaz para mostrar otro ejemplo:

```
interface Tiempo { fun horasOcio(): Float }
enum class Semana(val laborable: Boolean) : Tiempo {
    LUNES(true),
    (...)
    DOMINGO(false);
    override fun horasOcio() = if (laborable) 2F else 5F
}
```

Si no pudiéramos sobreescribir el método *horasOcio* porque las horas de ocio dependieran de lo que se hace durante el día, tendríamos que declarar una clase anónima para instanciar los objetos enumerados. Más adelante hablaremos de esto con más detalle.

```
enum class Semana(val laborable: Boolean) : Tiempo {
    LUNES(true) {
        override fun horasOcio(): Float {
            return 0.5F;
        }
    },
    (...)
    DOMINGO(false) {
        override fun horasOcio(): Float {
            return 18F;
```

```
        }
    };
}
```

Objetos anónimos

Una clase anónima es una clase sin nombre que se define una sola vez. Se declara y se usa en el mismo lugar. El objeto es la instancia de dicha clase. Veamos un ejemplo:

```
val coordenada = object {
        val x: Int = 1
        val y: Int = 2
    }
    println(coordenada.x + coordenada.y)
}
```

Además, como estamos creando un objeto de una clase, podemos heredar o implementar otra como explicamos a continuación:

```
interface Interfaz {
    fun metodo()
}
val objeto = object : Interfaz {
    override fun metodo() { print("algo") }
}
```

Los tipos de clases anónimas sólo se pueden usar para ámbitos locales o privados. Si un objeto anónimo tiene visibilidad pública, Kotlin va a inferir el supertipo más alto que es *Any*, salvo que se le defina algún tipo. Los objetos anónimos se pueden declarar como singleton tal como se muestra en el ejemplo:

```
object Coordenada {
```

```
    val x: Int = 1
    val y: Int = 2
}
fun main() {
    val otraCoordenada = Coordenada
    println(otraCoordenada.x) // 1
}
```

Hay ciertas limitaciones y es que esta declaración no es una expresión y no puede estar en el término de la derecha de una asignación. Además, esta declaración tampoco puede ser local, como por ejemplo dentro de una función o dentro de una clase anidada. Por otra parte, la inicialización de un objeto declarado es thread-safe y se realiza cuando se accede por primera vez.

Como Kotlin no puede tener métodos estáticos, ha introducido el concepto de *companion object* donde se permite declarar algo parecido. Veamos un ejemplo:

```
class Idioma {
    companion object {
        val defecto = "Español"
        fun traduce(texto: String, otroIdioma: String) {
            println("Traduciendo $texto a $otroIdioma")
        }
    }
}

val idioma = Idioma("inglés")
println(Idioma.defecto) // Español
Idioma.traduce("Texto", "Alemán")
```

Puedes profundizar más sobre cómo se inicializan estos objetos en la documentación oficial.

Type Aliases

Tal cual como suena: Kotlin nos permite crear un alias para aquellos tipos que son demasiado largos. Veamos un ejemplo muy sencillo:

```
typealias MiMapa = List<Map<String, String>>

fun main() {
   val miListaDeMapas: List<Map<String, String>> =
listOf<Map<String, String>>(mapOf("1" to "2"))
   val acortado: MiMapa = listOf(mapOf("1" to "2"))
}
```

Control de flujo

El control de flujo en Kotlin se hace de forma parecida a otros lenguajes como C/C++ o Java. Pero tiene sus diferencias.

If / else

En Kotlin, *if* es una expresión que retorna valor, así que se puede usar como un operador ternario.

```
val maximum = if (x > y) x else y
```

La última expresión de cada rama de if es el valor de bloque, que se puede asignar a una variable.

```
val max = if (x > y) {
    print("Choose x")
    x
} else {
    print("Choose y")
    y
}
```

En este caso (si utilizamos *if* como expresión) es necesario también definir el caso de *else*.

For

El bucle *for* se utiliza para iterar los objetos que tengan *el iterador* definido. Es un análogo de *foreach* que existe en otros lenguajes.

```
for (item: Object in objects) {
    // ...
}
```

El iterador es una función *iterator()* que devuelve un tipo que tenga dos funciones definidas *next() y hasNext()*. La función *hasNext()* devuelve el tipo boolean. Las 3 funciones tienen que estar marcadas como *operator.*

Si necesitas iterar sobre un array se pueden usar diferentes formas:

```
for (i in array.indices) {
    println(array[i])
}
```

También se puede utilizar la función *withIndex()*

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

While / Do-While

El uso del bucle *while* es muy parecido a otros lenguajes de programación.

```
while (y > 0) {
    y--
}
```

```
do {
    val z = retrieveData()
} while (z != null) // se puede acceder a z desde aquí!
```

When

La expresión *when* es un variante de *switch*, pero mucho más flexible. La expresión *when* compara el argumento con todas las condiciones hasta que encuentre una que sea verdadera. La condición *else* se ejecuta cuando no se cumple ninguna condición. Igual que sucede con el if, el resultado de la expresión when se puede asignar a una variable. En este caso, se usa el valor de la última expresión del bloque que tenga la condición que sea verdadera.

```
val roman = when (number) {
      1 -> "I"
      2 -> "II"
      3 -> "III"
       else -> "?"
    }

    print ("Number $number is $roman")
```

Se pueden combinar diferentes casos.

```
when (count) {
    1,2 -> print("count == 1 o count == 2")
    else -> print("x no es ni 1, ni 2")
}
```

Además, se pueden usar las expresiones o comprobar si el argumento está en el rango especificado en las condiciones.

```
when (count) {
    in 1..10 -> print("count está en el rango entre 1 y 10")
```

```
    convert(x) -> print("convert(x) es igual a count")
    else -> print("no podemos hacer nada")
}
```

El compilador de Kotlin sabe rastrear las comprobaciones de tipo *is* y las conversiones de tipo explícitas, de modo que a menudo no es necesario convertir las clases explícitamente. El compilador inserta conversiones de tipo (*smart cast*) automáticamente cuando es necesario.

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) //x se convierte automáticamente en String
    }
}
```

Gracias al uso de *smart cast* puedes acceder a los métodos de un tipo sin comprobaciones adicionales.

```
fun hasPostfix(x: Any) = when(x) {
    is String -> x.endsWith("postfix")
    else -> false
}
```

A partir de Kotlin 1.3 se pueden utilizar las expresiones incluso en el argumento de la expresión:

```
fun Request.getBody() =
        when (val response = runRequest()) {
            is Success -> response.body
            is Fail -> throw HttpException(response.status)
        }
```


Kotlin - Gestión de nulos
autentia
¿En qué se diferencia?
El sistema de tipos de Kotlin está hecho para eliminar el peligro de encontrarse de repente con una referencia nula en el código. Salvo que se exprese explícitamente, los nulos en Kotlin no existen. Pero existen los tipos nullables y los tipos no nullables.
Los tipos nullables
En Kotlin para poder asignar a un objeto el valor null hay que declararlo como un tipo nullable utilizando el operador ?.
var s1: String = "Hola, mundo"
s1 = null // el error!!!
var s2: String? = null //es un tipo nullable
Kotlin utiliza la misma JVM que Java, por eso también lanza la excepción NullPointerException. Las posibles causas son: llamada explícita para lanzar NullPointerException, uso del operador Not Null Assertion (!!), interoperaciones con el código Java etc.
Operador de acceso seguro (?.)
Si intentamos acceder a una variable nullable tendremos un error de compilación, porque el compilador exige usar con las variables nullables llamadas seguras..
var s1: String? = "Hola, mundo"
println(s1.length) // error!!
getUserData()?.parseToAnotherObject()?.execute()
También reduce la necesidad de usar if/else para comprobar si la variable es nula y ejecuta una acción sólo cuando la referencia tiene un valor no nulo.
Operador Not Null Assertion (!!)
El operador Not Null Assertion (!!) convierte cualquier valor en un tipo no nulo y lanza una excepción NullPointerException si el valor es nulo.
var str : String? = "Alejandro"
println(str!!.length)
str = null
str!!.length //error!!!
Operador Elvis (?:)
Se utiliza para devolver un valor predeterminado si la variable es nula. Operador Elvis hace que el código sea más compacto.
var name = firstName?: "Desconocido"
val name = firstName ?: throw
IllegalArgumentException("Introduce el nombre correcto")

Nulos en Kotlin

El sistema de tipos de Kotlin está hecho para eliminar el peligro de una referencia nula en el código. Los programas lanzan *NullPointerException* en tiempo de ejecución y, a veces, provocan fallos o bloqueos del sistema. Si has programado en Java u otro lenguaje que tiene el concepto de referencia nula, entonces tienes que haber experimentado la NullPointerException en el código. El runtime de Kotlin usa la misma JVM, por lo tanto, también lanza esta excepción si encuentra alguna referencia nula. Por tanto, las posibles causas de las excepciones NullPointerException son las siguientes: llamada explícita para lanzar NullPointerException, uso del operador Not Null Assertion (!!), interoperaciones con el código Java, intentos de acceder a un miembro en una referencia nula, tipo genérico con nulabilidad incorrecta, etc.

Tipos nullables

El sistema de tipos de Kotlin tiene dos tipos de referencias que pueden contener nulos (referencias que aceptan valores *null*) y las que no (referencias no nula). Una variable de tipo String no puede contener nulos. Si intentamos asignar un valor nulo a esta variable, nos da un error de compilación.

```
var s1: String = "Geeks"
s1 = null // el error de compilación
```

Para permitir que una variable sea nula, podemos declarar una variable como nullable.

```
var s1: String? = null
```

Si intentamos acceder a una variable nullable tendremos un error de compilación, porque el compilador exige usar con las variables nullables llamadas seguras o el operador Not Null Assertion !!.

```
var s2: String? = "GeeksforGeeks"
println(s2.length)
Error:(3, 15) Kotlin: Only safe (?.) or non-null asserted (!!.)
calls are allowed on a nullable receiver of type String?
```

Usando Safe Calls

Kotlin tiene un operador de llamada segura ?. que reduce la necesidad de usar if/else para comprobar si la variable es nula y ejecuta una acción sólo cuando la referencia tiene un valor no nulo. Nos permite combinar una verificación de anulabilidad y una llamada a un método en una sola expresión.

```
object?.parseToAnotherObject()
```

Si no existiera este operador, tendríamos que escribir el código muchísimo menos elegante:

```
if(object != null)
 object.parseToAnotherObject()
else
 ...
```

Operador Elvis

El operador Elvis se utiliza para devolver un valor no nulo o un valor

predeterminado cuando la variable original es nula. En otras palabras, si la expresión izquierda no es nula, el operador Elvis la devuelve. De lo contrario, devuelve la expresión derecha. La expresión del lado derecho se evalúa solo si el lado izquierdo es nulo.

```
val name = firstName ?: "Unknown"
```

Es igual a una expresión de if/else así:

```
val name = if(firstName!= null) firstName else "Unknown"
```

Además, también podemos usar expresiones *throw* y *return* en el lado derecho del operador Elvis. Por lo tanto, podemos lanzar una excepción en lugar de devolver un valor predeterminado.

```
val name = firstName ?: throw IllegalArgumentException("Enter
valid name")
```

Operador Not Null Assertion

El operador Not Null Assertion (!!) convierte cualquier valor en un tipo no nulo y lanza una excepción *NullPointerException* si el valor es nulo.

```
var str : String?  = "FirstName"
println(str!!.length)
str = null
str!!.length
```

```
Exception in thread "main" kotlin.KotlinNullPointerException
    at HelloKt.main(helloapp.kt:4)
```

Any y Nothing

Any es una clase abierta y, por defecto, la superclase para todas las clases de Kotlin, ya sea que la definamos explícitamente o no. Esto es similar a la clase Object en Java, que es la superclase para todas las clases de Java.

```
public open class Any {
   public open operator fun equals(other: Any?): Boolean
   public open fun hashCode(): Int
   public open fun toString(): String
}
```

La función *equals* comprueba si el objeto es igual al objeto pasado por parámetro. Hay diferentes criterios que la función *equals* usa para verificar la igualdad pero podemos sobreescribir esta función en cualquier clase.

hashCode() devuelve un número entero único para todos los objetos de clase. Esta función también está abierta y podemos sobreescribirla según nuestro caso de uso.

La última es *toString()* que se utiliza para representar el objeto en forma de String. Igual que otras, podemos definirla como queramos.

Nothing es una clase que se utiliza para representar un valor que nunca existirá. Tiene un constructor privado y tampoco se puede extender.

```
/**
* Nothing has no instances. You can use Nothing to represent "a
value that never exists": for example,
* if a function has the return type of Nothing, it means that it
never returns (always throws an exception).
*/
public class Nothing private constructor()
```

Se utiliza para definir un tipo de retorno de función que siempre generará

una excepción.

```
fun willAlwaysThrowException(): Nothing =  throw Exception("Always
Exception")
```

El posible caso de uso de *Nothing* es una función TODO que no está definida de momento.

```
public fun TODO(): Nothing = throw NotImplementedError()
```

Funciones

La declaración de funciones en Kotlin es muy parecida a otros lenguajes de programación. Se usa la palabra *fun* para declarar una función.

```
//declaración de una función
fun power(x: Int): Int {
  return x * x
}
```

El tipo del argumento se coloca después del nombre. Si la función no devuelve nada, el tipo del return puede omitirse, aunque todas las funciones en Kotlin siempre retornan un tipo. Cuando no se ha especificado ninguno entonces la función retorna *Unit*.

```
power(10) //la llamada a la función
```

Las llamadas y los parámetros de las funciones se utilizan de la misma manera como en C o Java. Si te apetece puedes usar *trailing coma (coma final)* al declarar los argumentos pero es totalmente opcional.

```
class Person constructor (
    val firstName: String,
    val lastName: String,
    val age: Int, // coma final
)
```

A diferencia de Java, en Kotlin las funciones pueden declararse fuera de una clase. Esto añade la flexibilidad a la estructura del código permitiendo ahorrar bastantes líneas.

```
//se puede declarar la función main fuera de cualquier clase
fun main() {
    println("Hello, World")
}
```

Toda función declarada fuera de una clase es estática.

Block body y el tipo de retorno opcionales

Si la función se puede definir solo con una expresión, se pueden omitir las llaves sin declarar el cuerpo de la función.

```
fun power(x: Int) = x*x
```

Además, recuerda que se puede omitir el tipo de retorno de la función declarando el resultado así:

```
fun greetings() = "Hello, World"
```

Es importante acordar que las funciones con los cuerpos (bodies) siempre tienen que tener un tipo de retorno. Porque el compilador no puede deducir el tipo de retorno al tener una lógica complicada dentro del cuerpo de la función.

El tipo Unit

Si la función no devuelve nada, se puede utilizar como el tipo de retorno *Unit*. Pero su declaración es opcional. Se puede omitir sin problemas. Además, se puede omitir el retorno de Unit en el código también.

```
//`fun makeGreetings(personName: String?): Unit` es opcional
fun makeGreetings(personName: String?) {
    if (personName != null)
```

```
        println("Hola $personName")
    else
        println("Hola extraño!")
    // `return Unit` or `return` es opcional
}
```

Vararg y spreads

El parámetro de la función (normalmente es el último) puede llevar la palabra *vararg* permitiendo así pasar diferentes cantidades de parámetros a la función.

```
fun <T> printAll(vararg ts: T) {
    ts.forEach { println(it) }
}
```

Al llamar a la función con el parámetro *vararg* pasamos simplemente, todos los argumentos que queramos, da igual cuántos.

```
printAll("A","B","C","D")
```

Dentro de la función el parámetro *vararg* es un array. Solo un parámetro puede ser *vararg*. Si usamos *vararg* conjuntamente con otros parámetros tenemos que utilizar sus nombres al llamar a la función.

```
fun createUser(vararg roles: String, username: String, age: Int) {
    // ...
}
createUser("admin", "user", username = "me", age = 42)
```

A veces ya tenemos una instancia de Array y queremos pasarla a una función con el parámetro *vararg*. En este caso se utiliza el operador *spread* *.

```
fun sum(vararg xs: Int): Int = xs.sum()

val numbers = intArrayOf(1, 2, 3, 4)
val summation = sum(*numbers)
print(summation) //es igual a 10
```

Destructuring

Kotlin te permite desestructurar un objeto de una manera muy fácil.

```
val (name, age) = person
```

Si en la clase Person tienes las funciones declaradas *component1()* y *component2()* Kotlin les llama para asignar el resultado de cada una de ellas a nuestras variables *name* y *age*.

Puedes declarar n veces las funciones tipo *componentN(),* lo importante es que lleven la palabra *operator* en su declaración.

Si utilizas las data classes, el compilador de Kotlin te las crea por defecto.

```
data class Result(val result: Int, val status: Status)
fun getResult(...): Result {
    // cálculos
   return Result(result, status)
}
// llamada a la función
val (result, status) = getResult(...)
```

También podríamos usar la clase estándar *Pair* y hacer que la función getResult() devuelva *Pair<Int, Status>*, pero es mejor tener los datos nombrados explícitamente.

Con el mapa sucede lo mismo. Kotlin ya te da la posibilidad de

desestructurarlo, por ejemplo en el bucle for.

```
for ((key, value) in map) {
    // ...
}
```

Si necesitas el valor solo del segundo componente Kotlin a partir de la versión 1.1 te permite omitir el nombre de la variable no deseada en la declaración.

```
val (_, status) = getResult()
```

Lambdas

Las lambdas de Kotlin se parecen a las lambdas de Java. Son funciones anónimas que se utilizan directamente sin declarar. La sintaxis de una expresión lambda en Kotlin es la siguiente:

```
val sum = { x: Int, y: Int -> x + y }
```

Una expresión lambda siempre está entre llaves, las declaraciones de parámetros están al principio de la lambda y tienen anotaciones de tipos, aunque es algo opcional. El cuerpo va después del signo ->. Si el tipo de retorno de la lambda no es *Unit*, la última expresión (o posiblemente la única) dentro del cuerpo de la lambda se trata como el valor de retorno.

En Kotlin, si el último parámetro de una función es a la vez una función, entonces podemos pasar una expresión lambda como argumento correspondiente y se puede colocar fuera del paréntesis:

```
val summ = numbers.fold(5) { acc, e -> acc * e }
```

Esta sintaxis se llama *trailing lambda.*

Si la lambda tiene solo un parámetro, se permite omitir su declaración y utilizar el nombre implícito del único parámetro *it*

```
numbers.filter { it > 0 }
```

Normalmente, se devuelve el valor de la última expresión de la lambda.

```
//devolvemos el valor de la última expresión (shouldFilter)
numbers.filter {
    val shouldFilter = it > 0
    shouldFilter
}
```

Pero existe la posibilidad de usar la sintaxis de retorno calificado (*return qualified*). En este caso usamos un label (marcador) que marca el punto de retorno para la expresión de return.

```
//igual que el ejemplo anterior
numbers.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

No podemos usar la expresión "*return shouldFilter*" como si fuera una función normal. Tenemos que usar el marcador (label) que marca la función a la que queremos ir. Si usamos sólo "return" volvemos al llamador de la función *getResult()*:

```
fun getResult() {
  val numbers = intArrayOf(1, 2, 3, 4)
  numbers.filter {
    return //volvemos al llamador de la función de getResult()
  }
```

```
  print("Nunca se va ejecutar")
}
```

Podemos cambiar este comportamiento utilizando la sintaxis de funciones anónimas. Su sintaxis se parece muchísimo a la de funciones normales, pero las funciones anónimas no tienen nombre.

```
numbers.filter(fun(item) = item > 0)
```

Son muy parecidas a lambdas pero funcionan diferente con las expresiones “return”. En este sentido son “funciones normales”. Con el return normal volvemos a un nivel anterior.

```
fun getResult() {
  val numbers = intArrayOf(1, 2, 3, 4)
  numbers.filter(fun(item: Int): Boolean {
    val shouldFilter = item > 0
    return shouldFilter // volvemos a la función getResult()
  })
  println("Ahora sí!!")
}
```

Scope Functions

La biblioteca estándar de Kotlin contiene varias funciones cuyo único propósito es ejecutar un bloque de código dentro del contexto de un objeto. Cuando llamas a una función de este tipo, utilizando un objeto con una expresión lambda proporcionada, creas un ámbito temporal. En este ámbito, puedes acceder al objeto sin su nombre. Estas funciones se denominan *scope functions*. Hay cinco de ellas: *let*, *run*, *with*, *apply*, and *also*.

La función *let* se puede usar para invocar una o más funciones con los resultados de las cadenas de llamadas. Por ejemplo, el siguiente código imprime los resultados de los operaciones en una colección:

```
val names = mutableListOf("Juan", "José", "Francisco", "Javier",
"Pelayo")
names.map { it.length }.filter { it > 4 }.let {
    println(it)
    encode(it)
    save(it)
    //otras operaciones
}
```

El parámetro *it* es el propio objeto (contexto) sobre que vamos a ejecutar las operaciones.

La función *with* es una función útil cuando queremos ejecutar acciones sobre algún objeto.

```
with(names) {
    println("podemos acceder a los nombres utilizando $this")
    println("tiene $size elementos")
}
```

La función *run* se utiliza cuando queremos inicializar nuestro objeto dentro de la lambda.

```
val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}
```

Se puede usar *run* como una función independiente. En este caso podemos ejecutar diferentes comandos y usar su resultado asignando a una variable.

```
val key = run {
    calcularSecretKey()
}
```

La función *apply* se usa para configurar los miembros de un objeto.

```
val employee = Employee("José").apply {
    age = 36
    city = "Madrid"
}
```

Devuelve el propio objeto, así que se puede usar fácilmente en las cadenas de llamadas para configurar un objeto y luego usarlo. El objeto de contexto es accesible como *this* dentro de la función *apply*.

La función *also* se usa para ejecutar las acciones con el mismo objeto. Se puede acceder a él utilizando *it*.

```
fun main() {
   val numbers = mutableListOf(1, 2, 3)
   numbers.also { it.add(4) }
   .run {
      print(this)
   }
}
```

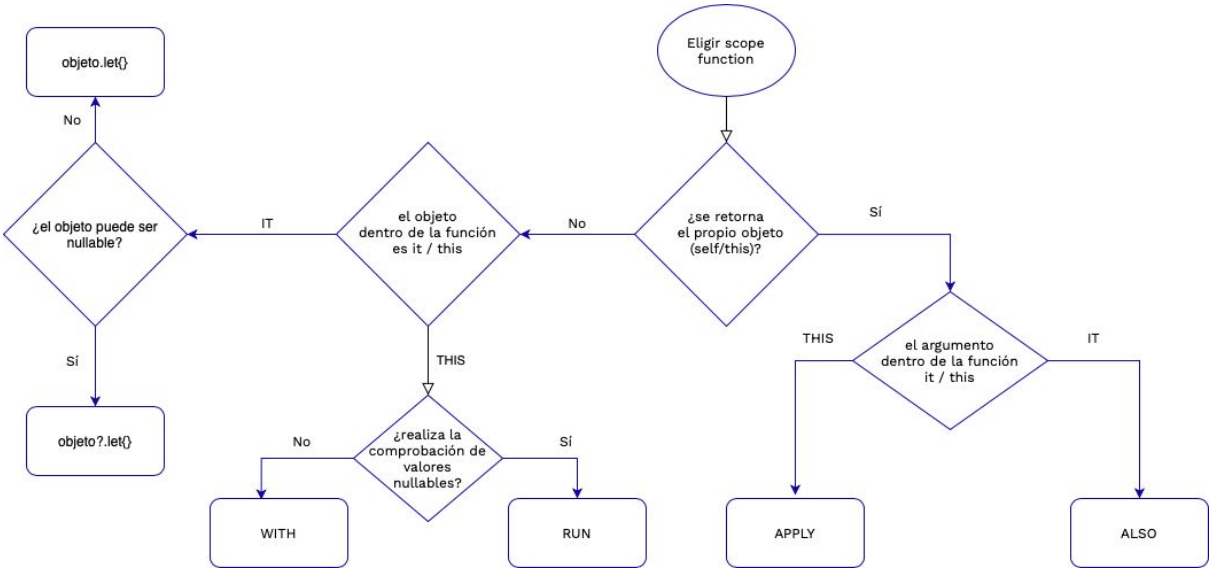
De este modo podemos combinar la función *also* con otras funciones (run o let) o con las funciones de nuestro objeto.

Las funciones de alcance (scope funciones) se parecen entre ellas, por eso vamos a resumir toda la información sobre ellas:

- Si quieres ejecutar una lambda con objetos no nulos: *let*.
- Para crear una expresión como una variable en el ámbito local: *let*.
- Para configurar un objeto: *apply*.
- Para configurar un objeto y calcular el resultado: *run*.
- Ejecución de expresiones: *run independiente (no de extensión)*.
- Si quieres aplicar algunos efectos adicionales: *also*.
- Agrupar llamadas a un objeto: *with*.

Función	La referencia de objeto	El valor de retorno	¿Es la función de extensión?
let	it	Resultado de lambda	Sí
run	this	Resultado de lambda	Sí
run	-	Resultado de lambda	Se puede utilizar sin objeto
with	this	Resultado de lambda	El objeto es en argumento
apply	this	El objeto	Sí
also	it	El objeto	Sí

Un esquema para entender qué función tenemos que usar en un caso concreto:



Inline Functions

En Kotlin, las funciones de orden superior o expresiones lambda se almacenan como un objeto, por lo que la asignación de memoria, tanto para los objetos de función como para las clases y las llamadas virtuales, pueden introducir una sobrecarga en el tiempo de ejecución. Podemos eliminar la sobrecarga de memoria utilizando funciones *inline* que pide al compilador que no asigne memoria y simplemente copie el código de esa función en el lugar de llamada.

```
fun higherfunc(str : String, myCall :(String)-> Unit) {
    //invocamos la función myCall con el argumento str
    myCall(str)
}

// main function
fun main(args: Array<String>) {
    print("Hola, Pepito") // la función print normal
    higherfunc("Hola, Pepito",::print)
}
```

Si miramos el bytecode de la función *main* veremos *mycall.invoke(str)* la llamada a la función *myCall* con el argumento str.

```
public static final void higherfunc(@NotNull String str, @NotNull
Function1 mycall) {
     Intrinsics.checkNotNullParameter(str, "str");
     Intrinsics.checkNotNullParameter(mycall, "mycall");
     mycall.invoke(str);
  }

  public static final void main(@NotNull String[] args) {
     Intrinsics.checkNotNullParameter(args, "args");
     String var1 = "Hola, Pepito";
     boolean var2 = false;
     System.out.print(var1);
     higherfunc("Hola, Pepito", (Function1)null.INSTANCE);
  }
```

Si tuviéramos muchos parámetros-funciones o las funciones que pasáramos fueran complicadas, tendríamos un gran impacto en la memoria. Para lidiar con eso, podemos declarar la función *higherfunc()* como *inline*.

```
inline fun higherfunc( str : String, mycall :(String)-> Unit){
    mycall(str)
}
```

Entonces el bytecode sería distinto:

```
 public static final void main(@NotNull String[] args) {
  Intrinsics.checkNotNullParameter(args, "args");
  String str$iv = "Hola, Pepito";
  boolean $i$f$higherfunc = false;
  System.out.print(str$iv);
  str$iv = "Hola, Pepito";
  $i$f$higherfunc = false;
  int var4 = false;
  boolean var5 = false;
  System.out.print(str$iv);
}
```

El compilador inserta ya la función *print(“Hola, Pepito”)* sin hacer ninguna llamada. El uso de funciones *inline* puede hacer que el código generado crezca. Sin embargo, si lo hacemos de una manera razonable (es decir, evitando la inserción de funciones grandes), dará sus frutos en rendimiento, especialmente dentro de los bucles con las llamadas *megamórficas* (las llamadas a las funciones que tienen muchas implementaciones).

Colecciones

Las colecciones nos sirven para almacenar objetos. Al igual que en otros lenguajes de programación, en Kotlin contamos también con List, Set y Map. Cada uno tiene un propósito distinto y en el siguiente enlace se explica la diferencia entre ellos.

Obtención de elementos

Un único elemento

Cuando trabajamos con colecciones, queremos saber cómo podemos acceder a los elementos que contienen para poder realizar alguna operación con estos. Los más básicos son `get()` y `elementAt()` que pasándole un índice por parámetro de entrada, nos devolverá el valor que se encuentra en esa posición. También contamos con `first()` y `last()` que nos devolverán el primer elemento y último que cumpla con la condición descrita.

```
val coins = listOf(1, 2, 5, 10, 20, 50)
println(coins.first{it > 10} ) // 20
println(coins.last{it + 10 == 15}) // 5
```

Existen también funciones que nos ayudan a controlar de una forma sencilla las excepciones cuando por ejemplo, vamos a obtener un elemento que no se encuentra en la colección. Por eso, disponemos de las funciones `elementAtOrNull()` y `elementAtOrElse()`.

Múltiples elementos

Cuando estamos trabajando con grandes estructuras de datos, a veces necesitamos quedarnos con un subconjunto de elementos que se hallan dentro de la colección. Kotlin ofrece una solución a este problema haciendo uso de las siguientes funciones:

- Slice: devuelve una colección correspondiente al rango que se le pasa por parámetro. Dicho rango hace referencia a los índices del conjunto de datos.

```
val days = listOf("monday", "tuesday", "wednesday", "thursday",
"friday", "saturday", "sunday" )
println(days.slice(1..4)) //[tuesday, wednesday, thursday, friday]
println(days.slice(0..5 step 2)) //[monday, wednesday, friday]
println(days.slice(setOf(3, 5, 0))) //[thursday, saturday, monday]
```

- Take and drop: nos ayudan a coger los primeros elementos o los últimos y dependiendo de la función que usemos, empezará desde el inicio o desde el final de la colección. Como parámetros de entrada se le pueden pasar un entero o también, pueden recibir una condición.

```
val days = listOf("monday", "tuesday", "wednesday", "thursday",
"friday", "saturday", "sunday" )
println(days.take(3)) // [monday, tuesday, wednesday]
println(days.takeLast(3)) // [friday, saturday, sunday]
println(days.drop(3)) // [thursday, friday, saturday, sunday]
println(days.dropLast(5)) // [monday, tuesday]
```

```
val days = listOf("monday", "tuesday", "wednesday", "thursday",
"friday", "saturday", "sunday" )

println(days.takeWhile { !it.startsWith('f') })
// [monday, tuesday, wednesday, thursday]
```

```
println(days.takeLastWhile { it != "friday" })
// [saturday, sunday]

println(days.dropWhile { it.length != 8 })
// [thursday, friday, saturday, sunday]

println(days.dropLastWhile { it.contains('s') })
// [monday, tuesday, wednesday, thursday, friday]
```

Transformaciones

Las transformaciones son útiles para alterar una colección a través de unas reglas dadas. Disponemos de distintas técnicas para realizar esa modificación, como *mapping*, *zipping* y *association*.

- Mapping: cuando queremos realizar la misma operación a todos los elementos de una colección, empleamos la función `map()` que añadirá en un nuevo conjunto de datos el resultado de la lambda.

```
val days = listOf("monday", "tuesday", "wednesday", "thursday",
"friday", "saturday", "sunday" )
println(days.map { it.substring(0,2) })

// [mo, tu, we, th, fr, sa, su]
```

- Zipping: esta transformación es útil cuando queremos relacionar dos colecciones por pares de elementos. El par estará compuesto por los elementos que posean el mismo índice.

```
val days = listOf("monday", "tuesday", "wednesday", "thursday",
"friday", "saturday", "sunday" )
val numbers = listOf("1", "2", "3", "4", "5", "6" , "7")
println(days zip numbers)
```

```
// [(monday, 1), (tuesday, 2), (wednesday, 3), (thursday, 4),
(friday, 5), (saturday, 6), (sunday, 7)]
```

- Association: Consiste en generar un mapa (Key, Value) que partiendo de una colección, los elementos de ésta sean la key del mapa y el valor corresponda a la función lambda que hemos definido.

```
val days = listOf("monday", "tuesday", "wednesday", "thursday",
"friday", "saturday", "sunday")
println(days.associateWith { it.startsWith("t") })
// {monday=false, tuesday=true, wednesday=false, thursday=true,
friday=false, saturday=false, sunday=false}
```

Fold y reduce

Se realizan operaciones entre los elementos de la colección de manera secuencial. La diferencia entre ambos es que a `fold()` se le pasa por parámetro el valor inicial de la operación y en cambio, en `reduce()`, el acumulado inicial es cero.

```
val numbers = listOf(5, 2, 10, 4)

val sum = numbers.reduce{ sum, element -> sum + element }
println(sum) // 21
val sumDoubled = numbers.fold(10){ sum, element -> sum + element }
println(sumDoubled) // 31
```

Filtros

La tarea de filtrado es útil cuando se quiere obtener una colección que cumpla una serie de condiciones, las cuales estarán definidas en la función

lambda.

Las funciones de filtrado más básicas son `filter()` y `filterNot()` cuya diferencia es obtener aquellos valores que cumplen o no cumplen la condición.

```
val days = listOf("monday", "tuesday", "wednesday", "thursday",
"friday", "saturday", "sunday")

val filtered = days.filter { it.length <= 6 }
val filteredNot = days.filterNot { it.length <= 6 }

println(filtered) // [monday, friday, sunday]
println(filteredNot) //  tuesday, wednesday, thursday, saturday]
```

La librería de Kotlin también ofrece otros filtros, uno para tratar con los índices de la colección: `filterIndexed()` y otro para filtrar por tipo: `filterIsInstance()`.

Agrupaciones

Las agrupaciones nos sirven para obtener subconjuntos de la colección, cuyos elementos siguen un patrón común, el cual se ha definido en la función lambda.

```
val days = listOf("monday", "tuesday", "wednesday", "thursday",
"friday", "saturday", "sunday")

println(days.groupBy { it.first().toUpperCase() })
// {M=[monday], T=[tuesday, thursday], W=[wednesday], F=[friday],
S=[saturday, sunday]}
```

Existen algunas operaciones que se pueden hacer cuando se realiza una agrupación en una colección:

- eachCount: cuenta el número de elementos que tiene cada

subconjunto.

- aggregate: realiza operaciones secuenciales dentro de cada subconjunto y devuelve el resultado.

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val aggregated = numbers.groupingBy {it % 2}.aggregate { key,
accumulator: StringBuilder?, number, first ->
    if (first) // first element
      StringBuilder().append(if (key == 0) "Pares" else "Impares")
         .append(" -> ").append(number)
    else
      accumulator!!.append(",").append(number)
}

println(aggregated.values)
// [Impares -> 1,3,5,7,9, Pares -> 2,4,6,8,10]
```

Delegación

La delegación es un patrón de diseño en el que un objeto maneja una petición delegándola a un objeto auxiliar, llamado delegado. El delegado es responsable de manejar la petición en nombre del objeto original y hacer que los resultados estén disponibles para el objeto original. Kotlin facilita la delegación al proporcionar el soporte para delegados de clase y de propiedad e incluso al tener algunos delegados integrados.

Delegados de clases

Supongamos que tenemos una clase de *ArrayList* y queremos añadir la funcionalidad de recuperar su último elemento eliminado. Básicamente, todo lo que necesitamos es la misma funcionalidad ArrayList con una referencia al último elemento eliminado. Una forma de hacer esto es extender la clase ArrayList. Dado que esta nueva clase está extendiendo la clase ArrayList concreta en lugar de implementar la interfaz *MutableList*, está muy acoplada con la implementación concreta de ArrayList. Sería mejor si pudieramos sobreescribir la función *remove()* para mantener una referencia del elemento eliminado y delegar el resto de las implementaciones vacías de *MutableList* a algún otro objeto. Kotlin proporciona una forma de lograr esto al delegar la mayor parte del trabajo a una instancia interna y personalizar su comportamiento. Para hacer esto, Kotlin introduce una nueva palabra clave: *by*. Veamos cómo funciona la delegación de clases. Cuando usa la palabra clave *by*, Kotlin genera automáticamente el código para usar la instancia *innerList* como delegado.

```
class ListWithLastRemovedElementSaved <T>(
    private val innerList: MutableList<T> = ArrayList<T>()
) : MutableCollection<T> by innerList {
    var deletedItem : T? = null
    override fun remove(element: T): Boolean {
        deletedItem = element
        return innerList.remove(element)
    }
    fun recover(): T? {
        return deletedItem
    }
}
```

La frase *by innerList* significa que Kotlin delegue la funcionalidad de la interfaz MutableList a una instancia interna de ArrayList llamada *innerList*. De este modo, la clase *ListWithLastRemovedElement* implementa todas las funciones de la interfaz *MutableList* usando métodos del objeto interno innerList. Además, ahora tiene la posibilidad de agregar su propio comportamiento.

Los delegados de clase son especialmente útiles cuando no se puede heredar de una clase en particular. Con la delegación de clase, la clase que usa los delegados no forma parte de ninguna jerarquía de clases. En cambio, comparte la misma interfaz y usa el objeto interno para llamar al tipo original. Esto significa que puedes cambiar fácilmente la implementación sin romper la API pública.

Delegated properties

Además de la delegación de clases, también puedes utilizar la palabra clave *by* para delegar propiedades. Con la delegación de propiedad, el delegado es responsable de manejar las llamadas a las funciones *get* y *set* de la propiedad. Esto puede ser extremadamente útil si necesitas reutilizar la lógica de *getter/setter* en otros objetos y te permite extender fácilmente la

funcionalidad.

Por ejemplo si tienes una clase de *Person* así:

```
class Person(name: String, var lastName: String) {
    var name: String = name
        set(value) {
            name = value.toLowerCase().capitalize()
            updateCount++
        }
    var updateCount = 0
}
```

Añadimos a la propiedad *name* de esta clase, el setter para cumplir con los requisitos de formato. Cuando se establece el nombre, queremos asegurarnos de que la primera letra esté en mayúscula mientras formatee el resto en minúsculas. Además, al actualizar el nombre, deseamos incrementar automáticamente la propiedad *updateCount*.

El código funciona, pero ¿qué pasa si los requisitos cambian y también queremos incrementar *updateCount* cada vez que cambie el apellido? Puedes copiar/pegar la lógica, pero es una mala práctica que “degrada” tu código.

```
//el código copiado es muy mala práctica
class Person(name: String, lastName: String) {
    var name: String = name
        set(value) {
            name = value.toLowerCase().capitalize()
            updateCount++
        }
    var lastName: String = lastName
        set(value) {
            lastName = value.toLowerCase().capitalize()
            updateCount++
        }
    var updateCount = 0
}
```

Con la delegación de propiedades, podemos reutilizar el código delegando setters y getters a una propiedad. Al igual que con la delegación de clases, tienes que usar la palabra clave *by* para delegar una propiedad y Kotlin generará el código para usar el delegado.

```
class Person(name: String, lastName: String) {
    var name: String by FormatDelegate()
    var lastName: String by FormatDelegate()
    var updateCount = 0
}
```

Con este cambio pedimos a Kotlin que utilice para las propiedades name y lastName la clase *FormatDelegate*. La clase delegada necesita implementar ReadProperty<Any?, String> si necesita delegar solo el getter o ReadWriteProperty<Any?, String> si necesita delegar tanto el getter como el setter. En nuestro caso, FormatDelegate necesita implementar ReadWriteProperty<Any?, String> ya que queremos usar tanto el setter como el getter.

```
class FormatDelegate : ReadWriteProperty<Any?, String> {
    private var formattedString: String = ""

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): String {
        return formattedString
    }

    override fun setValue(
        thisRef: Any?,
        property: KProperty<*>,
        value: String
    ) {
        formattedString = value.toLowerCase().capitalize()
    }
}
```

Es posible que hayas notado que hay dos parámetros adicionales en las funciones getter y setter. El primer parámetro es *thisRef* y representa el objeto que contiene la propiedad. thisRef se puede utilizar para acceder al objeto en sí con el objetivo de comprobar otras propiedades o llamar a otras funciones de la clase. El segundo parámetro es *KProperty <*>*, que se puede utilizar para acceder a los metadatos de la propiedad delegada. De esta manera, hemos conseguido utilizar la misma lógica que está definida en un sitio sin necesidad de copiarla y pegarla.

Hay ciertos tipos comunes de propiedades que, aunque podemos implementarlas manualmente cada vez que las necesitemos, sería muy bueno implementarlas solo una vez. Por ejemplo, propiedades *lazy*: el valor se calcula solo en el primer acceso; propiedades *observables*: los suscriptores son notificados sobre cambios en esta propiedad; almacenar propiedades en un mapa, en lugar de un campo separado para cada propiedad. Para implementar estas funcionalidades, Kotlin también utiliza propiedades delegadas.

La biblioteca estándar de Kotlin proporciona algunos delegados "por defecto". *Lazy* es una función que toma una lambda y devuelve una instancia de *Lazy<T>* que puede servir como delegado para implementar una propiedad *lazy*: la primera llamada a get() ejecuta la lambda pasada a *lazy()* y recuerda el resultado, las llamadas posteriores a get() simplemente, devuelven el resultado recordado.

```
val lazyValue: Double by lazy {
    return complexCalculations()
}
```

Delegates.observable() tiene dos argumentos: el valor inicial y un handler para modificaciones. Se llama al handler cada vez que asignamos a la propiedad el valor nuevo (después de que se haya realizado la asignación). Tiene tres parámetros: la meta info de la propiedad, el valor anterior y el

nuevo:

```
import kotlin.properties.Delegates

class User {
    var surname: String by Delegates.observable("") {
        prop, old, new ->
        println("$old -> $new")
    }
}
```

Si quieres interceptar asignaciones y "cancelarlas", usa *Delegates.vetoable()* en lugar de *Delegates.observable()*. Se llama al handler antes de que se haya realizado la asignación de un nuevo valor de propiedad.

Un caso de uso común es almacenar los valores de las propiedades en un mapa. En este caso, puedes utilizar la propia instancia del mapa como delegado de una propiedad.

```
class Person(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

Y luego puedes inicializar el objeto así:

```
val user = Person(mapOf(
    "name" to "José Fernández",
    "age"  to 45
))
println(user.name) // "José Fernández"
println(user.age)  // 45
```

Corrutinas

Una de las características más populares de Kotlin es el soporte para corrutinas que permite a los desarrolladores escribir código asíncrono como si fuera síncrono. Las corrutinas hacen que sea mucho más fácil escribir código concurrente empleando corrutinas que usan otras técnicas como *callbacks* o *reactive streams*. La programación asíncrona siempre es un desafío, especialmente cuando intentas coordinar múltiples funciones separadas, manejar cancelaciones y excepciones, etc.

La idea detrás de las corrutinas es que se pueden suspender y reanudar. Al marcar una función con la palabra clave *suspend*, le estás diciendo al sistema que puede poner la función en espera temporalmente y reanudarla en otro hilo más tarde, todo sin tener que escribir un código complejo de múltiples *threads* (hilos).

Kotlin, como lenguaje, proporciona solo un mínimo de API de bajo nivel en su biblioteca estándar. A diferencia de muchos otros lenguajes con capacidades similares, *async* y *await* no son palabras clave en Kotlin y ni siquiera forman parte de su biblioteca estándar. Para usar las corrutinas tienes que añadir *kotlinx.coroutines*. Es la biblioteca de corrutinas desarrollada por JetBrains.

El modelo de programación en sí mismo no cambia realmente:

```
fun postData(data: Data) {
    launch {
        val token = getToken()
        val post = submitData(token, data)
        processPost(post)
    }
```

```
}

suspend fun getToken(): Token {
    // lanzar la petición y suspender la corrutina
    return suspendCoroutine { /* ... */ }
}
```

La función *getToken()* es una “función suspendible”, por eso lleva la palabra “suspend” en su declaración, lo que significa que esta función puede estar “suspendida” en su ejecución y luego “reanudada” sin bloquear el hilo donde esté.

Las ventajas que tienen las corrutinas:

- Las firmas de funciones no cambian. Simplemente tienes que añadir la palabra *suspend*.
- El código se escribe como si fuera síncrono sin usar ningún sintaxis especial.
- Se puede usar las mismas APIs, bucles, excepciones, etc.
- El código se escribe de la misma manera para cualquier plataforma (JVM o Javascript).

Para crear las corrutinas se usan *los builders*. Dado que no son funciones “suspendidas”, se pueden utilizar en una función “no suspendida” o con cualquier otro fragmento de código. Actúan como un vínculo entre las partes suspendidas y no suspendidas de nuestro código.

runBlocking() es un builder que suspende el hilo actual hasta que finalicen todas las tareas de la corrutina que crea. Normalmente, ejecutamos runBlocking() para ejecutar pruebas de funciones “suspendidas” o para usarlo dentro del terminal. Mientras ejecutamos las pruebas, queremos asegurarnos de que la prueba no finalice antes de que hagamos el trabajo “pesado” en las funciones suspendidas.

```
import kotlinx.coroutines.delay
```

```
import kotlinx.coroutines.runBlocking

fun main() {
     println("Antes de crear la corrutina")
     runBlocking {
       print("Hola, ")
       delay(2000L)
       println("Mundo!")
     }
     println("Después de terminar la corrutina")
}
```

El output de este programa será:

```
Antes de crear la corrutina
Hola,
Mundo!
Después de terminar la corrutina
```

Como ves, aunque hemos utilizado el delay de 2 segundos, la ejecución del programa es consecuente de acuerdo al orden comandos. Es porque runBlocking() bloquea el hilo y espera hasta que acaben todas las tareas de la corrutina.

launch() es un builder de Kotlin que se usa para ejecutar las tareas asíncronas sin devolver ningún resultado. Esto significa que el builder launch() crea una nueva corrutina que no devolverá ningún resultado a la función que le llama. También permite iniciar una corrutina en segundo plano.

```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
    println("Empezamos..")
```

```
    GlobalScope.launch {
        println(doSomething())
    }
    println("¿hemos terminado?..")
    runBlocking {
      delay(4000L) //para asegurarnos de que JVM esté activa
    }
 }

suspend fun doSomething() : String {
    delay(3000L) // simulación de trabajo lento
    return "Hemos hecho algo"
}
```

Y al ejecutar el programa veremos lo siguiente:

```
Empezamos...
¿hemos terminado?..
Hemos hecho algo
```

async() un constructor de corrutinas que devuelve un valor al llamador. async se puede utilizar para realizar una tarea asíncrona que devuelve algún valor. Este valor en términos de Kotlin es un valor *Deferred<T>*.

Deferred<T> es una *promesa* o *un valor futuro (Promise o Future)* que tendrá algún valor en el futuro. Para obtener este valor, se necesita llamar a la función *await()* de esta promesa.

await() es una función suspendida que llama al builder *async* para obtener el valor de la promesa. La corrutina iniciada por async se suspenderá hasta que el resultado esté listo. Cuando el resultado esté listo, se devuelve y se reanuda la corrutina.

```
import kotlinx.coroutines.*

val a = 10
val b = 20
```

```
suspend fun main() = coroutineScope {
    println("Vamos a calcular la suma de a y b")
    launch {
        val result = async {
            calculateSum()
        }
        println("La suma de a & b es: ${result.await()}")
    }
    println("Procedemos con otra tarea mientras estamos esperando
el resultado de la corrutina")
}

suspend fun calculateSum(): Int {
    delay(2000L) // simulación del trabajo lento y largo
    return a + b
}
```

El output será:

```
Vamos a calcular la suma de a y b
Procedemos con otra tarea mientras estamos esperando el resultado
de la corrutina
La suma de a & b es: 30
```

Así funciona el flujo de la aplicación: *async()* devuelve una promesa como resultado de calculateSum() llamando a la función suspendida *await()*. *await()* nos traerá el resultado que devuelve calculateSum(). Mientras se ejecuta calculateSum(), sucede lo siguiente: el builder *async* suspenderá la corrutina (utilizada para calcular la suma) continuando la ejecución de otras tareas (println). Una vez que calculateSum() devuelve el resultado, la corrutina suspendida se reanudará e imprimirá el resultado calculado.

Se puede evitar la abundancia de async/await en tu código utilizando una función especial *withContext*. Si necesitas usar *async* e inmediatamente obtener el resultado a través de *await* conviene mejor sistituirlas por

withContext.

```
suspend fun retrieve1(url: String) = coroutineScope {
    async() {
        runTask()
}.await() }

suspend fun retrieve2(url: String) = withContext() {
        runTask()
}
```

```
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

suspend fun main() = coroutineScope {
     for(i in 0 until 5){
     launch {
        delay(1000L)
        print("$i")
       }
     }
}
```

Como se puede ver en el ejemplo, hemos utilizado la función *main()* como función suspendida. De este modo, podemos evitar el uso de las funciones de *GlobalScope* o *runBlocking()*. Su uso está muy desaconsejado, si no completamente desaprobado. El problema con esas funciones es que lanzan corrutinas que están vinculadas al alcance de todo la aplicación (digamos, que son globales) y abarcan todo el ciclo de vida de la aplicación. Si no las cancelas, duran para siempre. Por lo tanto, no las utilices a menos que tengas una razón fundamental para hacerlo.

La función *coroutineScope()* es una función suspendida que espera hasta que finalicen todas las corrutinas incluidas antes de terminar. Tiene la ventaja de no bloquear el hilo principal (a diferencia de *runBlocking()*), pero

debe invocarse como parte de una función suspendida.

Así hemos llegado a uno de los principios fundamentales del uso de corrutinas, usarlas dentro de un alcance definido. Cada corrutina tiene que estar definida dentro un alcance que puede ser *GlobalScope* (alcance global) o *CoroutineScope (alcance definido por el programador).* El beneficio de *coroutineScope* es que no tienes que comprobar si las corrutinas están terminadas. *CoroutineScope* automáticamente espera a que se completen todas las corrutinas incluidas.

```
suspend fun <R> coroutineScope(block: suspend CoroutineScope.() -> R):R
```

Según la signatura de la función *coroutineScope*, la función tiene como argumento una lambda (con receptor CoroutineScope) que no tiene argumentos y devuelve un valor genérico. Es una función suspendida, por lo que se debe llamar desde una función suspendida u otra corrutina.

Es un patrón común usar la función *coroutineScope* para establecer el alcance de las corrutinas incluidas, y dentro del bloque puedes usar otros builders como *launch* o *async* para manejar tareas individuales. *CoroutineScope* esperará hasta que se terminen todas las corrutinas antes de salir y si alguna de las corrutinas falla, también cancelará el resto. Esto logra un buen equilibrio de control y manejo de errores sin tener que sondear para ver si se realizan las rutinas y evita fugas de memoria en caso de que una rutina falle.

Las corrutinas es un mundo inmenso y hemos tocado solo una parte de él. Si estás interesado en aprender más profundamente las funcionalidades de las corrutinas puedes hacerlo leyendo la documentación oficial de Kotlin.

Convenciones de Kotlin

Organización del código fuente

Estructura de directorios

En los proyectos de Kotlin, la estructura de directorios recomendada es como la que mostramos a continuación. También hay que tener en cuenta que en los proyectos donde está junto con Java, deberían guardarse los ficheros en sus respectivos directorios.

```
- com.autentia.kotlin
  - ...
  - src
    - main
      - kotlin
        - domain
          - SomeClass.kt
        - MainClass.kt
      - java
        - OtherClass.java
      - resources
    - test
      - kotlin
        - ...
      - java
        - ...
      - resources
```

Nombre de los ficheros

Cuando los ficheros de Kotlin tienen una única clase, el nombre debería ser como el de la clase. En cambio, si el fichero contiene varias clases o simplemente otro tipo de información, se elige un nombre acorde a lo que contiene el fichero. En ambos casos, usamos *upper camelcase*, la primera letra en mayúscula, junto con la extensión *.kt*, por ejemplo, *UserFactory.kt*.

Organización de las clases

Normalmente, el contenido de la clase está organizado de la siguiente manera:

- Declaración de atributos (propiedades) y constructor.
- Constructores secundarios.
- Declaración de métodos.

Reglas de nomenclatura

Los nombres de los paquetes siempre se escriben en minúsculas y sin usar underscores. En el caso de que el paquete esté compuesto por varias palabras (desaconsejable), puedes concatenarlas o usar camelcase.

Para las clases y los objetos, la regla que se sigue es usar camelcase y poner la primera letra en mayúscula. Por ejemplo:

```
open class Booking { /*...*/ }
object BookingFlight : Booking { /*...*/ }
```

En el caso de las funciones, atributos y variables locales la regla a seguir es poner la primera letra en minúscula, usando camelcase y no poniendo underscores para separar las palabras.

```
fun doSomething() { /*...*/}
var myVariable = 42
```

Los nombres de las constantes y los enums suelen regirse bajo el estilo screaming snakecase. Esto quiere decir que el nombre se escribe en mayúsculas y si está compuesto por varias palabras se unen a través de underscores.

```
val CLIENT_NAME  = "Client's name"
enum class Bikes {ROAD, MTB, E_BIKE}
```

En los tests, Kotlin permite usar el nombre de los métodos con espacios en blanco, siempre y cuando el texto esté entre tildes (backticks). También se permite que el nombre del método sea a través del estilo snake camelcase.

```
class MyTestCase {
     @Test fun `should return something`() { /*...*/ }

     @Test fun should_return_something() { /*...*/ }

     @Test fun shouldReturnSomething() { /*...*/ }
}
```

Cuidar el formato

Cuidar el formato nos ayuda a trabajar mejor en equipo, dado que si todos seguimos el mismo estilo se facilita la comprensión del código. Cuando usamos un IDE, como IntelliJ, podemos emplear el plugin de Kotlin para que se encargue de formatear nuestro código automáticamente.

Esta herramienta también nos permite formatear el código a través del shortcut (Cmd + Alt + L) por si vemos que el fichero no tiene el formato acorde a la guía de estilos que marca Kotlin en su documentación.

Evitar sintaxis redundante

Una de las ventajas de Kotlin es que te permite obviar cierta sintaxis que no aporta nada de valor al código. Por ejemplo, evitar poner los puntos y coma al final de cada sentencia, como mencionamos en las primeras secciones.

También se debe omitir Unit cuando la función no devuelve ningún tipo.

```
fun myMethod() /* : Unit */ {
      // TODO
}
```

Y por último, si hacemos uso de los Strings Templates, no es necesario poner llaves <{}> para referenciar a una variable.

```
var stringTemplate = "$rol has ${powers.size()} powers"
println(stringTemplate)
```

Bibliografía

Estas son las fuentes que hemos consultado y en las que nos hemos basado para la redacción de este material:

- Jugando con Optional en Java 8: https://www.adictosaltrabajo.com/2015/03/02/optional-java-8/
- Expresiones Lambda con Java 8: https://www.adictosaltrabajo.com/2015/12/04/expresiones-lambda-con-java-8/
- Documentación de Oracle: https://docs.oracle.com/en/java/index.html
- https://git-scm.com/docs
- https://maven.apache.org/
- https://docs.gradle.org/
- https://docs.spring.io/spring-framework/docs/current/reference/html/
- https://microservices.io/patterns/index.html
- https://guides.micronaut.io/
- https://martinfowler.com/articles/richardsonMaturityModel.html
- https://www.baeldung.com/rest-with-spring-series
- https://spring.io/guides/tutorials/rest/
- https://www.baeldung.com/spring-vs-spring-boot
- https://www.baeldung.com/spring-dispatcherservlet
- https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html

- https://spring.io/projects/spring-cloud-netflix
- https://spring.io/projects/spring-boot
- https://www.adictosaltrabajo.com/2017/07/04/kotlin-primeros-pasos/
- https://kotlinlang.org/docs/reference/
- https://www.oreilly.com/library/view/kotlin-cookbook/9781492046660/
- https://www.soldevelo.com/blog/why-you-should-use-kotlin-instead-of-java-in-your-next-project
- https://blog.danlew.net/2017/05/17/why-kotlin/
- https://medium.com/swlh/generics-in-kotlin-5152142e281c

Lecciones aprendidas con esta guía

Si el front es la piel de nuestra aplicación, aquello por lo que todo el mundo la va a juzgar en primera instancia, **el back es el corazón.** Un corazón robusto, fiable y seguro es fundamental para conseguir un producto de calidad que no se desmorone ante el primer soplido.

En esta guía hemos puesto los cimientos sobre los que construir nuestro castillo, utilizando **Java** como forjado., además de enseñar pinceladas de algunas herramientas y técnicas básicas para que te inicies en el desarrollo profesional de aplicaciones. Hemos expuesto algunos de los frameworks más importantes para el desarrollo backend e introducido la **arquitectura de microservicios**, con sus respectivos patrones y algunos de los frameworks más usados para su desarrollo . Algunos de los puntos más importantes son:

- Conocer los tipos de aplicaciones y **paradigmas de programación**.
- Aprender las bases de Java y Kotlin como lenguaje orientado a objetos multiplataforma y la JVM como entorno de ejecución.
- Usar **clases, interfaces y anotaciones,** y aplicar la herencia, la abstracción y el polimorfismo.
- Dominar el control del flujo y las excepciones en Java.
- Descubrir las **APIs** más utilizadas, desde los tipos básicos, genéricos y opcionales, hasta las colecciones, los streams o la concurrencia.
- Primeros pasos para guardar y sincronizar nuestro trabajo con un repositorio de código distribuido como Git.

- Conocer **Maven** para automatizar las tareas de configuración, empaquetado, verificación de la calidad, gestión de dependencias, entre otras cosas.

- Aprender a **hacer tests** y dar los primeros pasos en la apasionante técnica del diseño de software conocida como TDD.

- Conocer dos de los principales frameworks de desarrollo con Java: **Spring y Micronaut.**

- Introducir los primeros pasos en los servicios REST y su implementación con Spring MVC.

- Exponer la **arquitectura de microservicios**, sus patrones y herramientas más relevantes.

- Comprender los desafíos de la alta disponibilidad y conseguir microservicios resilientes y tolerantes a fallos, establecer una correcta **gestión de threads y pools**, así como analizar su impacto en ellos.

Por supuesto, esto no es más que una pincelada de lo que un lenguaje tan veterano y potente como Java nos ofrece junto con Kotlin, un lenguaje joven que aglutina lo mejor de los lenguajes modernos, las herramientas que tenemos a nuestra disposición para el desarrollo del backend y los diferentes frameworks que tenemos a nuestro alcance, además de los beneficios del desarrollo basado en microservicios. Te animamos a que sigas indagando por tu cuenta en los temas que más llamen tu interés.

Expertos en creación de software de calidad

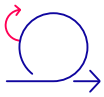
Diseñamos productos digitales y experiencias a medida

SOPORTE A DESARROLLO

Construimos entornos sólidos para los proyectos, trabajando a diario con los equipos de desarrollo.

DISEÑO DE PRODUCTO Y UX

Convertimos tus ideas en productos digitales de valor para los usuarios finales.

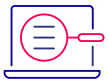


AGILE & CULTURE

Conectamos la estrategia con la ejecución, ayudándote a capitalizar las oportunidades del mercado.

DESARROLLO DE SOFTWARE

Te ayudamos a impulsar tu negocio mediante la construcción de software de calidad que apoye la transformación digital de tu organización.



AUDITORÍA

Analizamos la calidad técnica de tu producto y te ayudamos a recuperar la productividad perdida.

FORMACIÓN

Formamos empresas, con clases impartidas por desarrolladores profesionales en activo.

www.autentia.com

info@autentia.com | T. 91 675 33 06

¡Síguenos en nuestros canales!



Adictos al trabajo | by autentia