BACK

INTRODUCCIÓN AL BACKEND Y JAVA

PRIMERA PARTE

autentia

GUIA PARA
DIRECTIVOS Y TÉCNICOS

V.1

Back

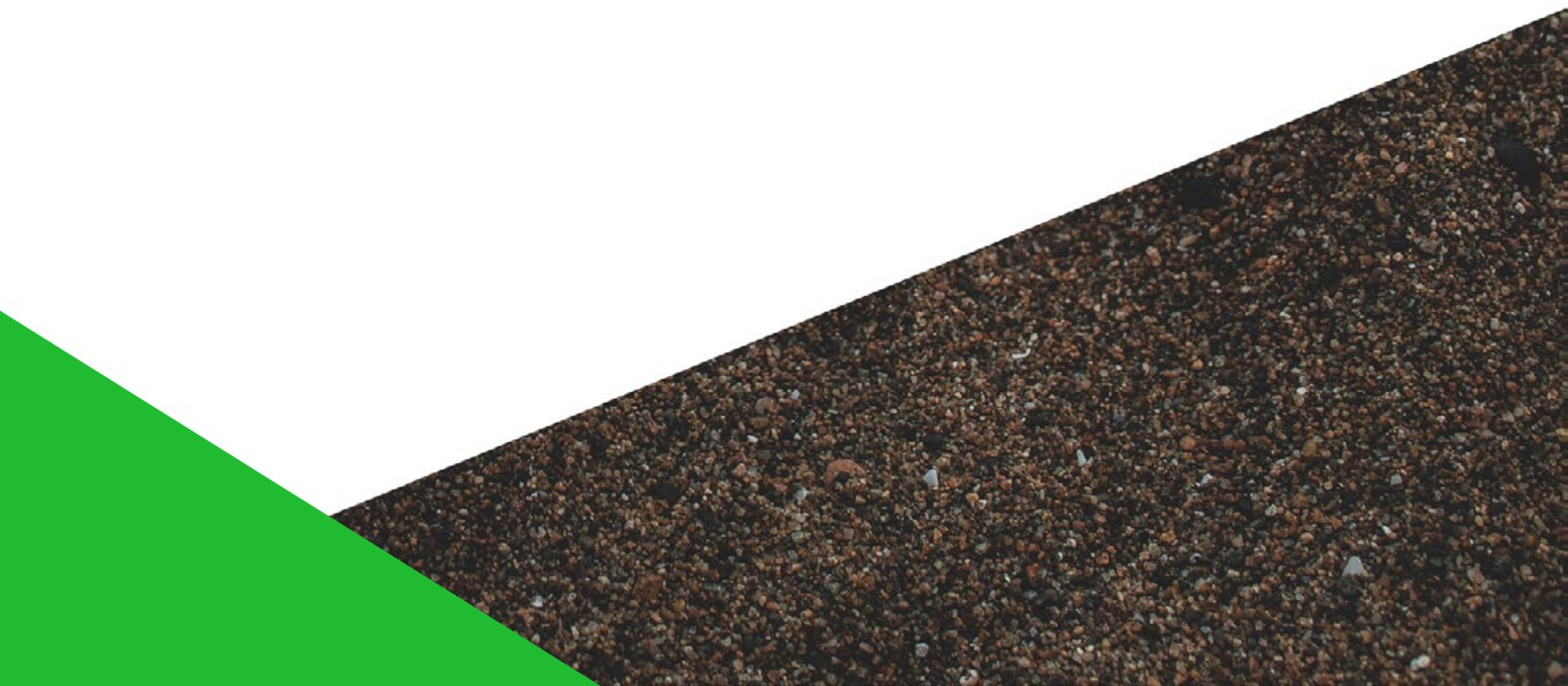
INTRODUCCIÓN AL BACKEND Y JAVA

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 partes que completan este documento, 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

INTRODUCCIÓN AL BACKEND Y JAVA

Índice

    - Execution Engine
- Control de flujo
    - if/else
    - switch
    - for
    - 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
- Bibliografía
- Lecciones aprendidas

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.

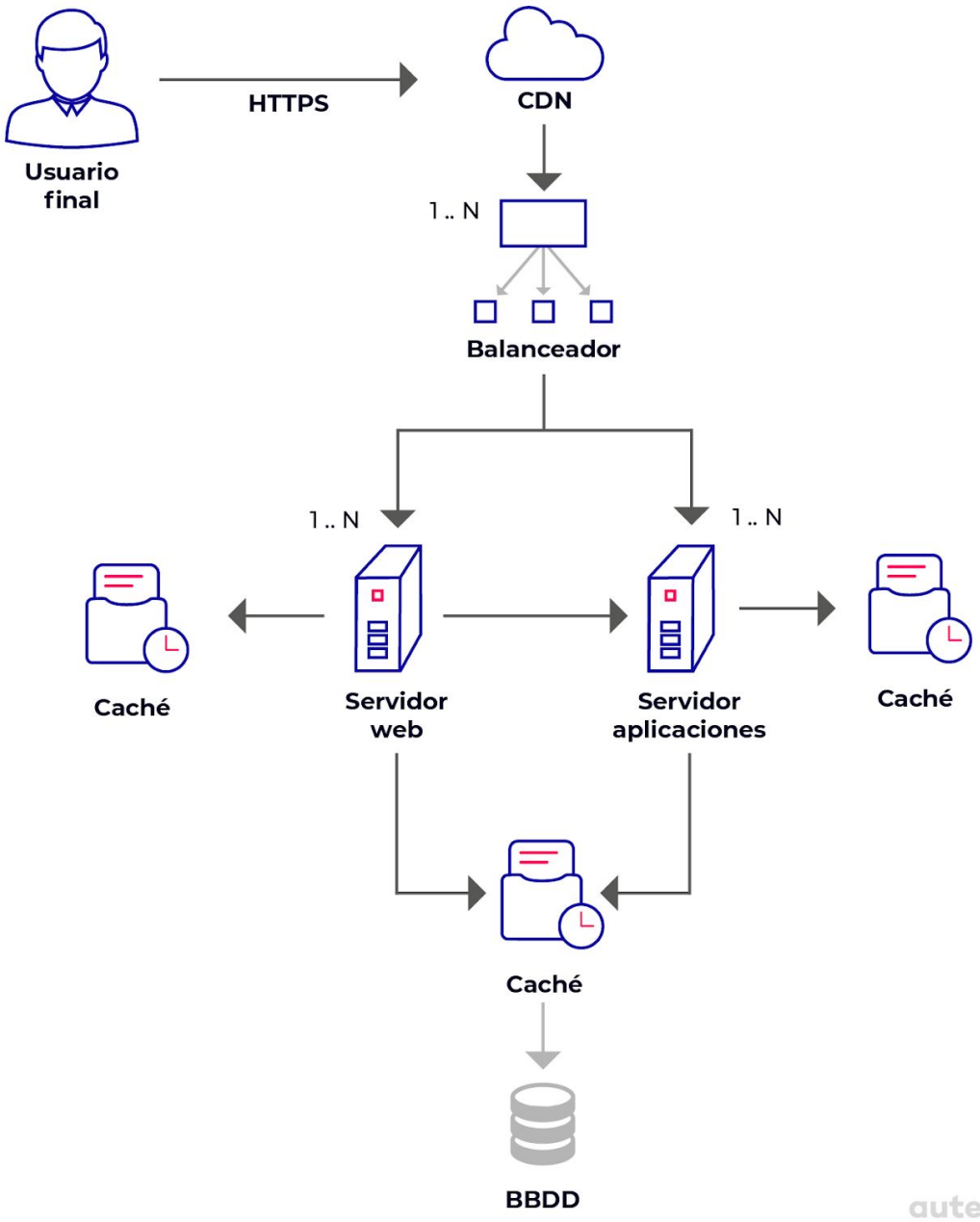
Este tipo de aplicaciones se caracteriza por tener, en mayor o menor grado, una buena parte del procesamiento de los datos en el propio dispositivo. Por tanto, tienen también un lado back instalado. La interfaz puede generarse de muchas formas. Pueden usar componentes del sistema operativo, interfaces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

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.

01100
10110
11110

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.

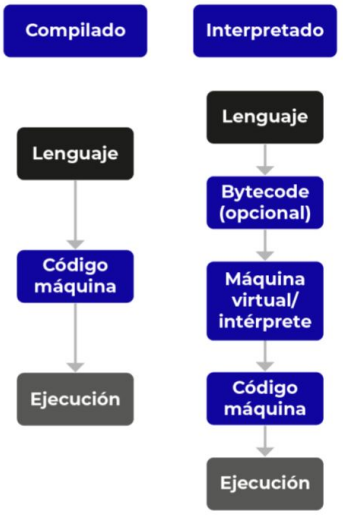
EJEMPLOS

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

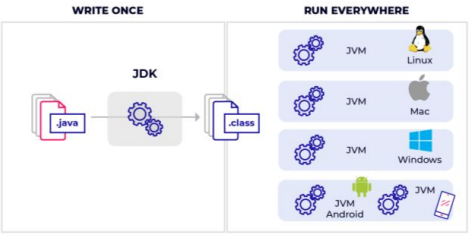
autentia

¿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).

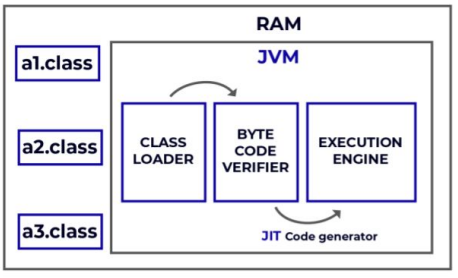
OBJETIVO

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.

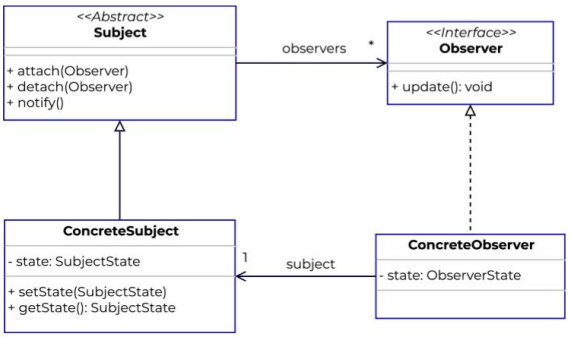
CONCEPTO

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).

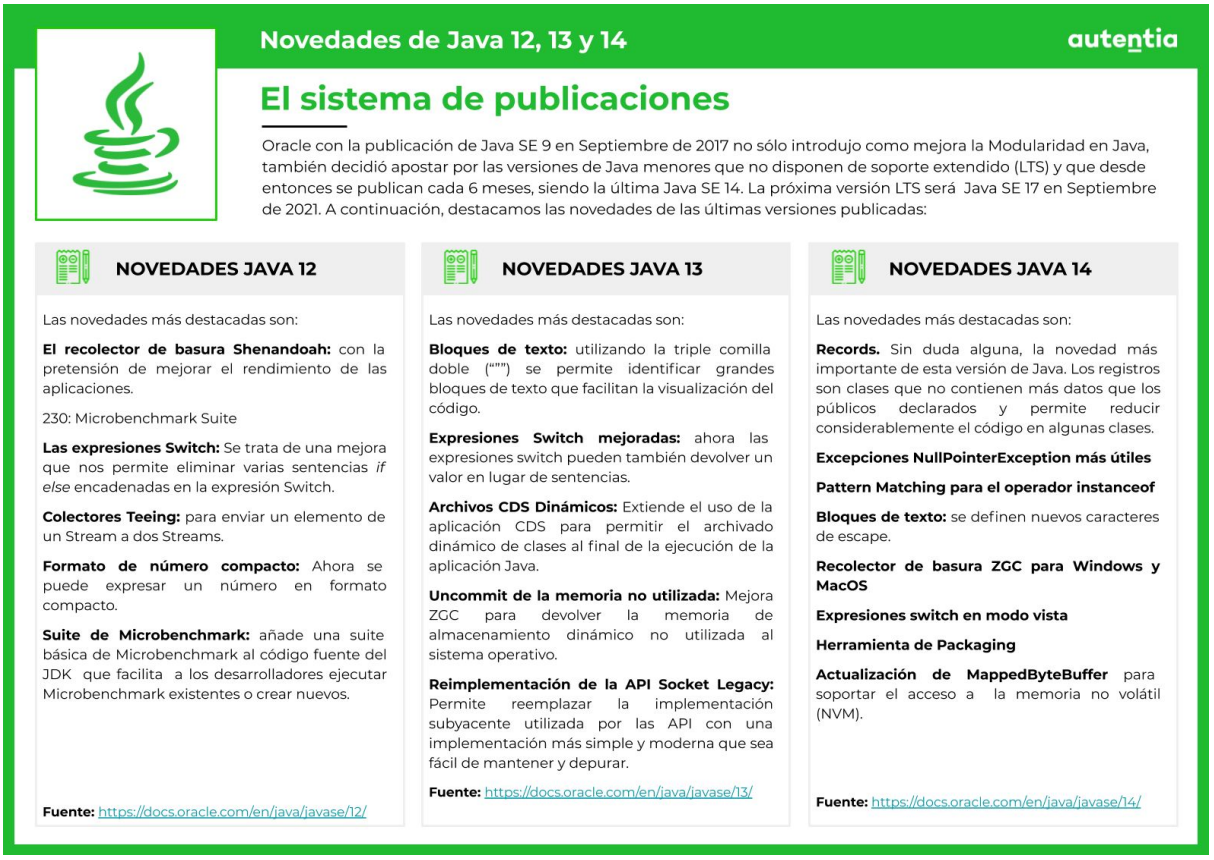


Java

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:

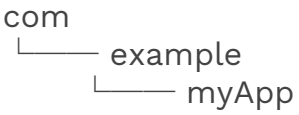
- Mediante la opción -cp en la línea de comandos. Es el método preferido ya que especifica un classpath diferente para cada aplicación, sin que afecte al resto.
- Declarándolo como una variable de entorno.

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
```

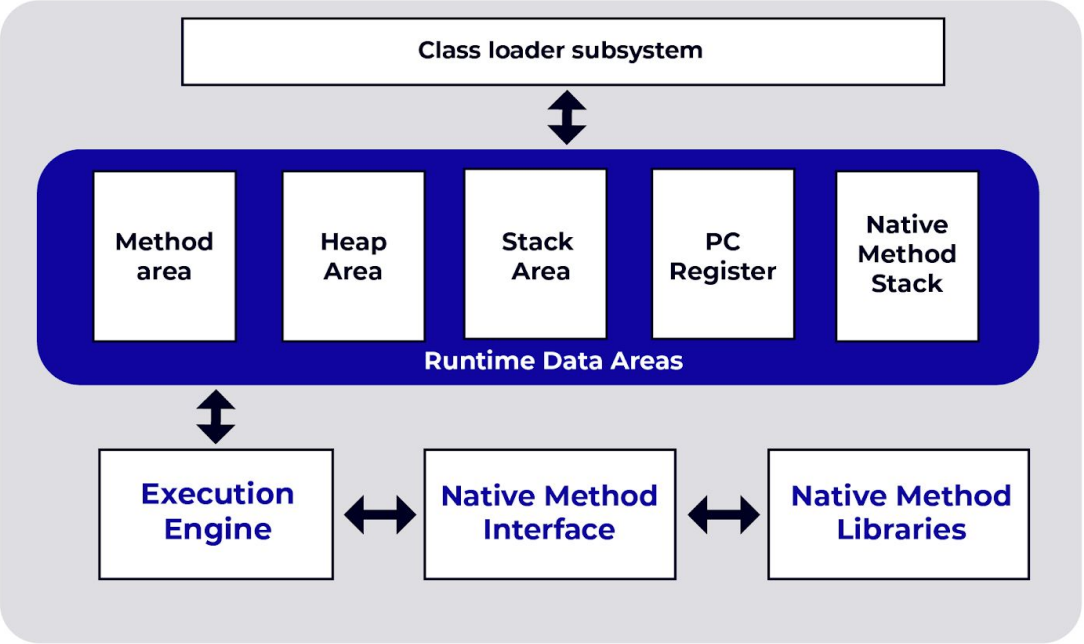
Los archivos JAR también pueden ser agregados al classpath, de forma que las aplicaciones puedan obtener sus dependencias al explorar dentro de su contenido. Es la principal forma de distribución de librerías. Normalmente, cuando descargamos una aplicación Java, esta trae sus propios JAR además de las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

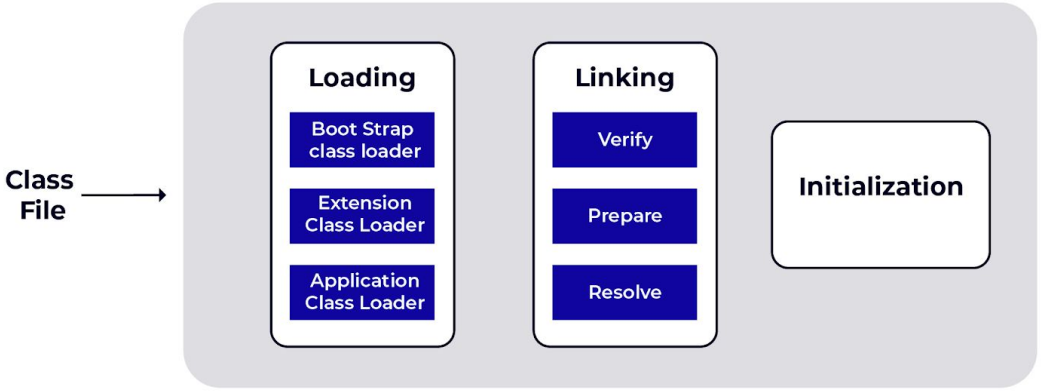
JVM se divide en 3 subsistemas que vamos a explicar a continuación:



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

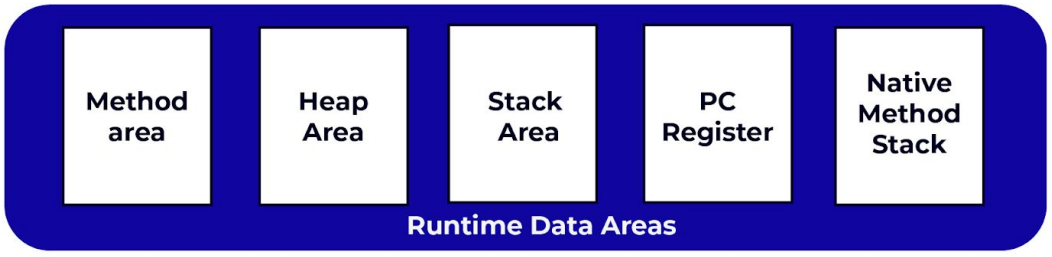
- **Resolve:** JVM localiza las clases, interfaces, campos y métodos referenciados en una tabla llamada *constant pool (CP)* y determina los valores concretos a partir de su referencia simbólica. Cuando se compila una clase Java, todas las referencias a variables y métodos se almacenan en el CP como 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).

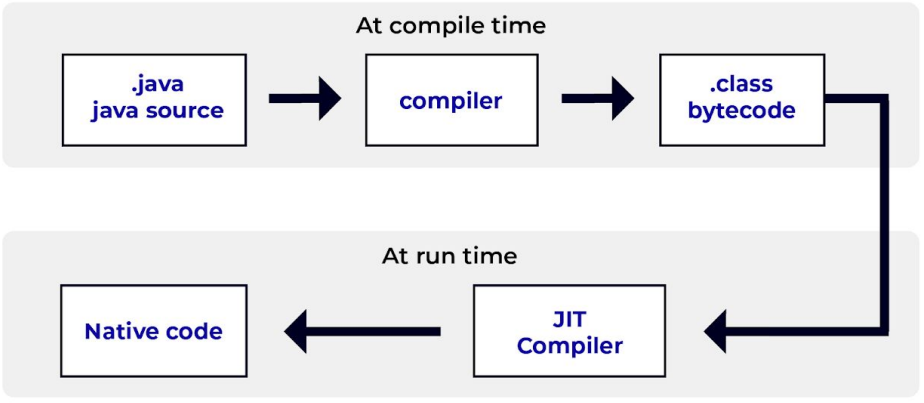


- **Method Area:** es parte de *Heap Area*. Contiene el esqueleto de la clase (métodos, constantes, variables, atributos, constructor, etc.).
- **Heap Area:** fragmento de memoria donde se almacenan los objetos creados (todo lo que se inicialice con el operador *new*). Si el objeto se borra, el *Garbage 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

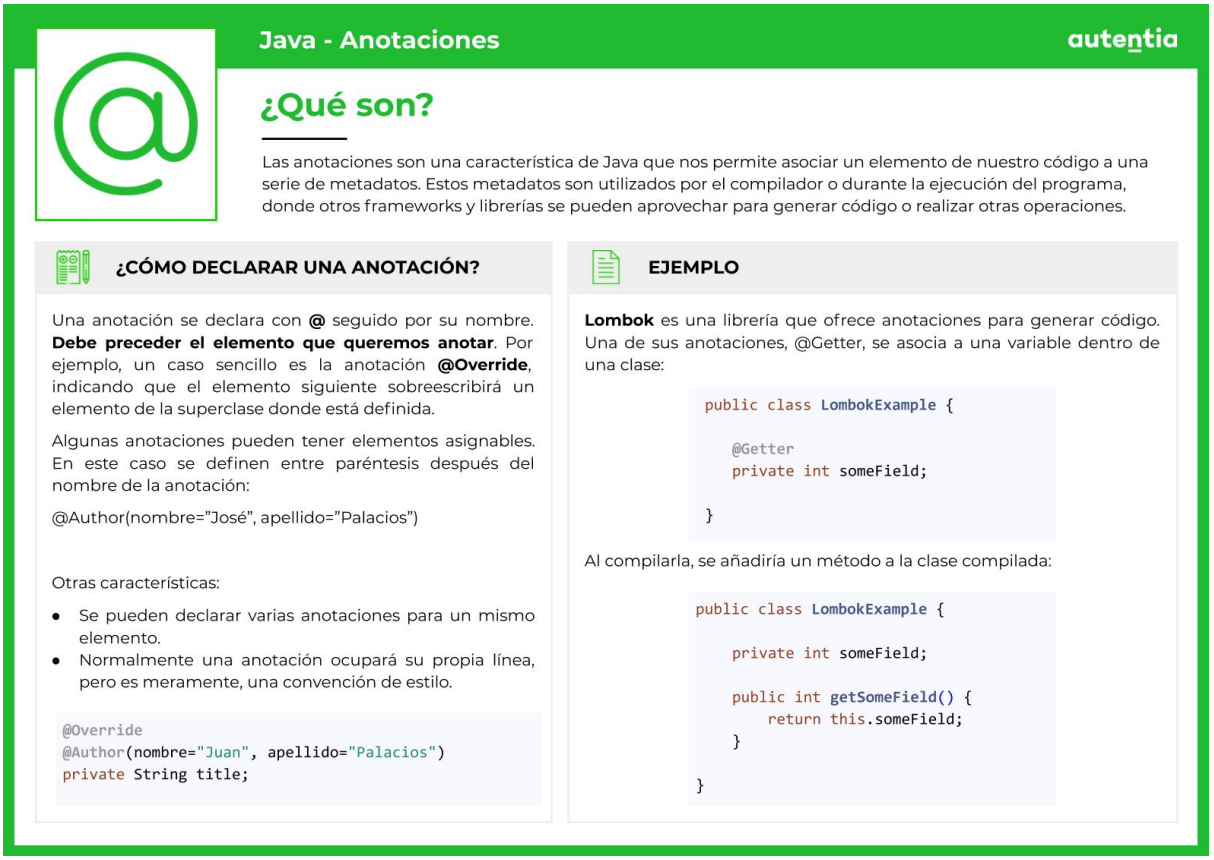
Las anotaciones son una forma de añadir metadatos a los elementos de nuestras aplicaciones. Estos metadatos pueden ser luego utilizados por el compilador, librerías o frameworks para tratar de una forma determinada esas piezas de nuestro código. No afectan de ninguna manera al código en sí.

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.

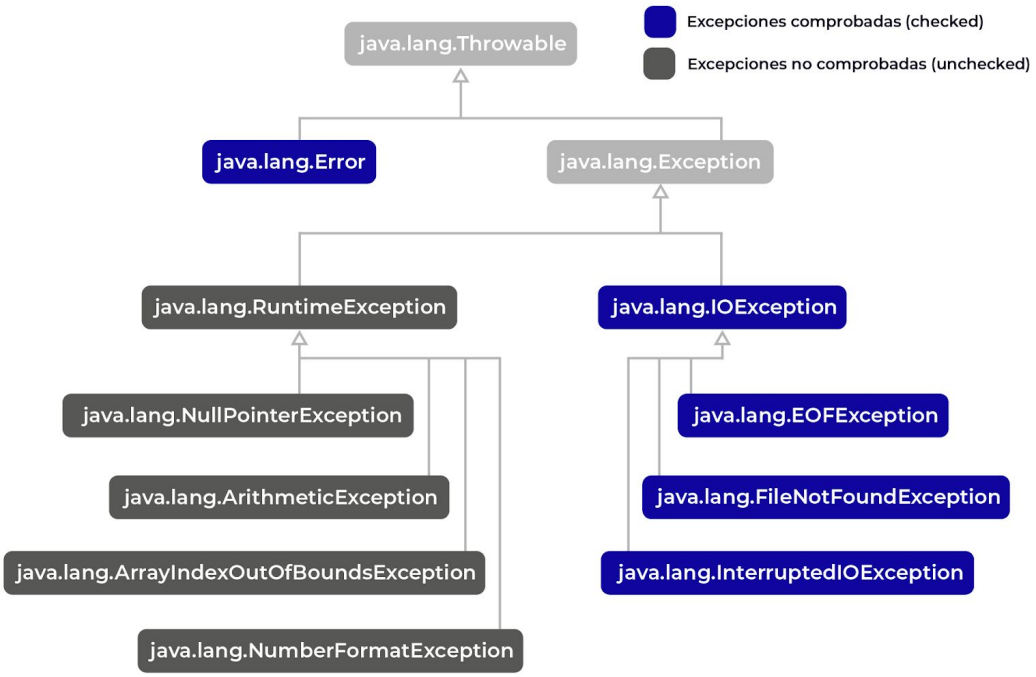
Las anotaciones van precedidas del símbolo @. Ya hemos utilizado la

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.



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.

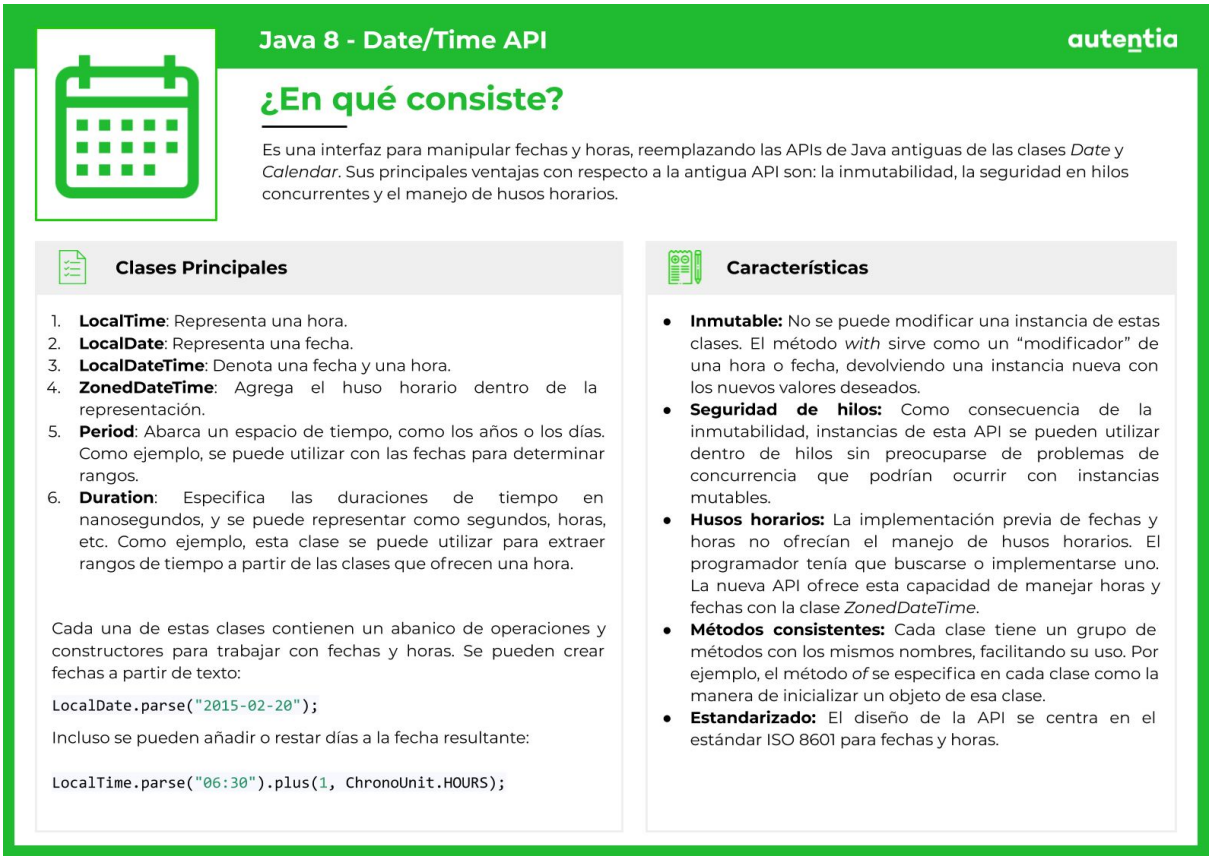
Las clases base son LocalTime, LocalDate y LocalDateTime.

```
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();
...
```

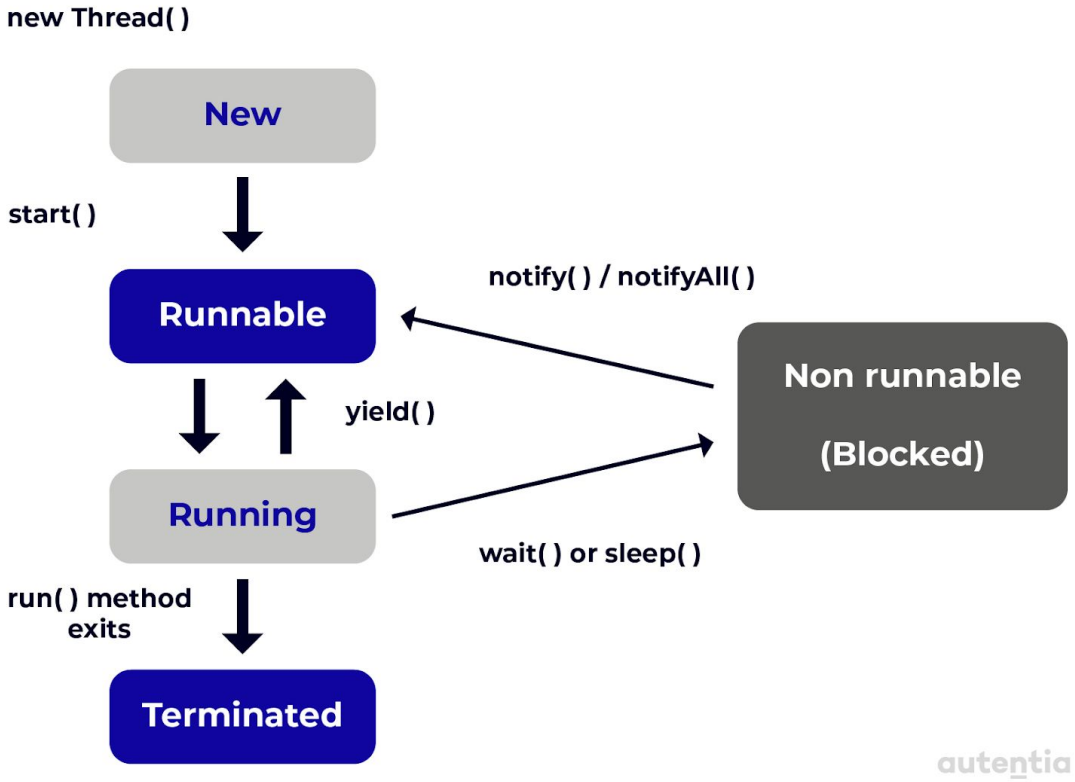
2. **Implementado la Interfaz *Runnable*** y pasando por parámetro dicha instancia a la clase *Thread*. Si nuestra clase no extiende de la clase Thread, nuestra instancia de clase no se tratará como thread. Por este motivo, se crea explícitamente una instancia de la clase Thread en el siguiente ejemplo.

```
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.
```

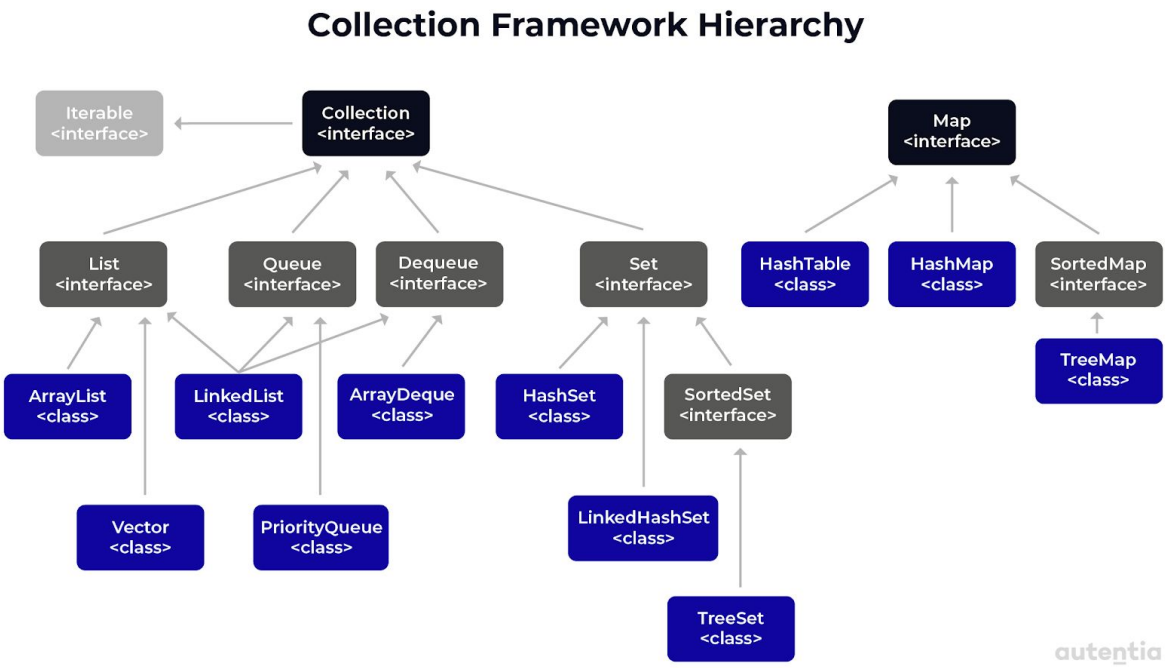
En cambio cuando se usa “? super” nos encontramos con el caso contrario, el compilador de Java sabe que los elementos almacenados son los tipos

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.

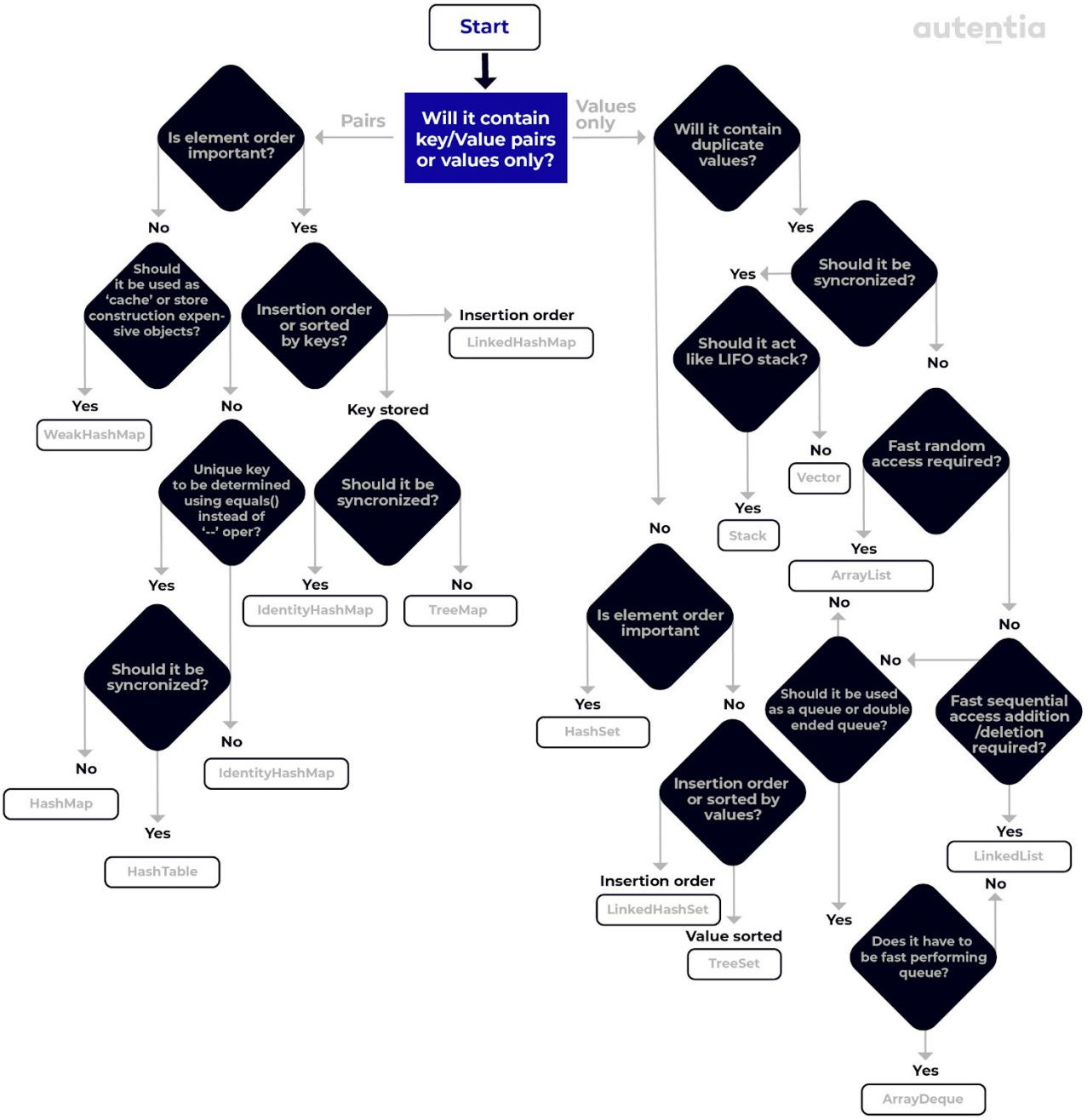
¿Uso ArrayList o LinkedList? Si necesitamos manipular los datos constantemente (insertar o eliminar), LinkedList nos ofrece un mejor rendimiento (O(1) vs. O(n)). En caso de necesitar hacer más operaciones de búsqueda (*get()*) y no tanto de inserción o eliminación, ArrayList ofrece un mejor rendimiento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

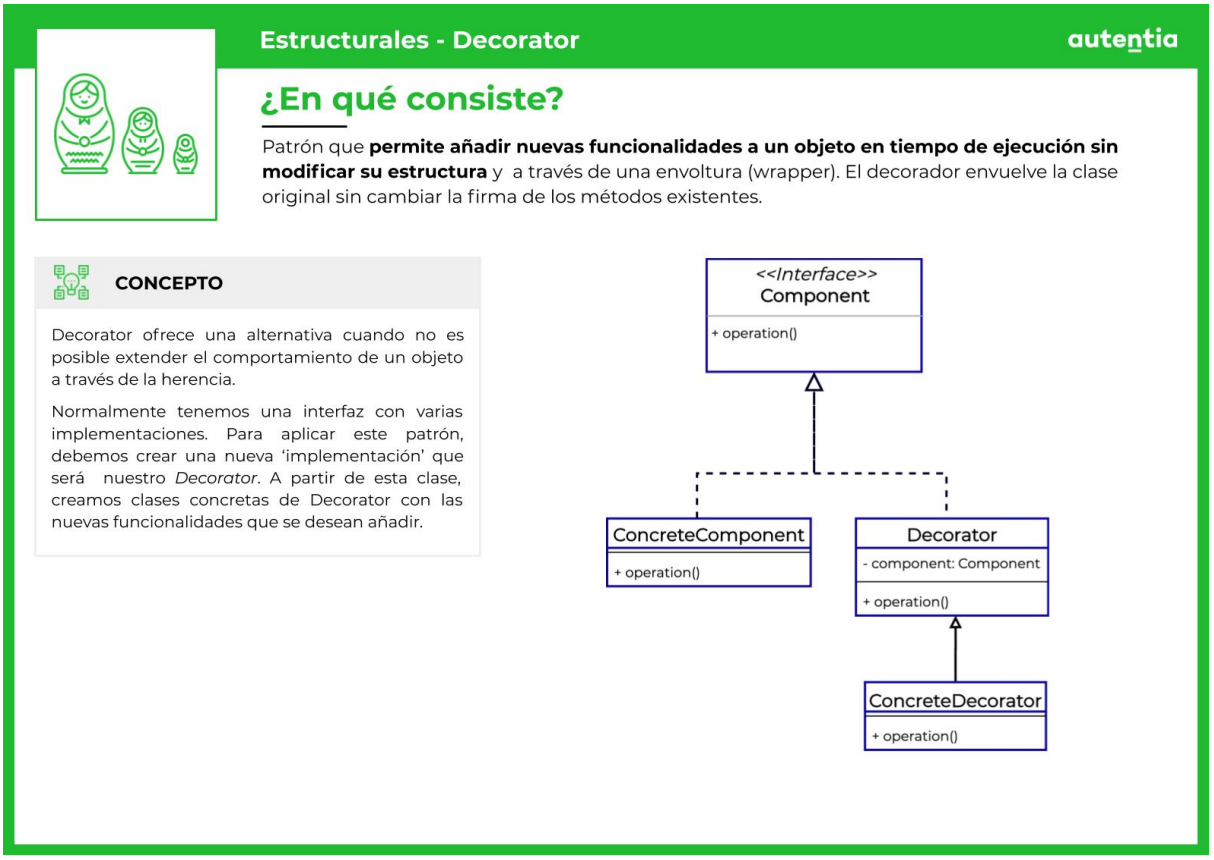
En el siguiente ejemplo, se observa como el BufferedOutputStream recibe por parámetro FileOutputStream y una vez se ha escrito en el buffer (a través de *write()*), se hace un flush para asegurarnos y forzar a que el buffer escriba todos los datos. No debemos olvidarnos de cerrar ambos canales.

```
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;
}
```

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

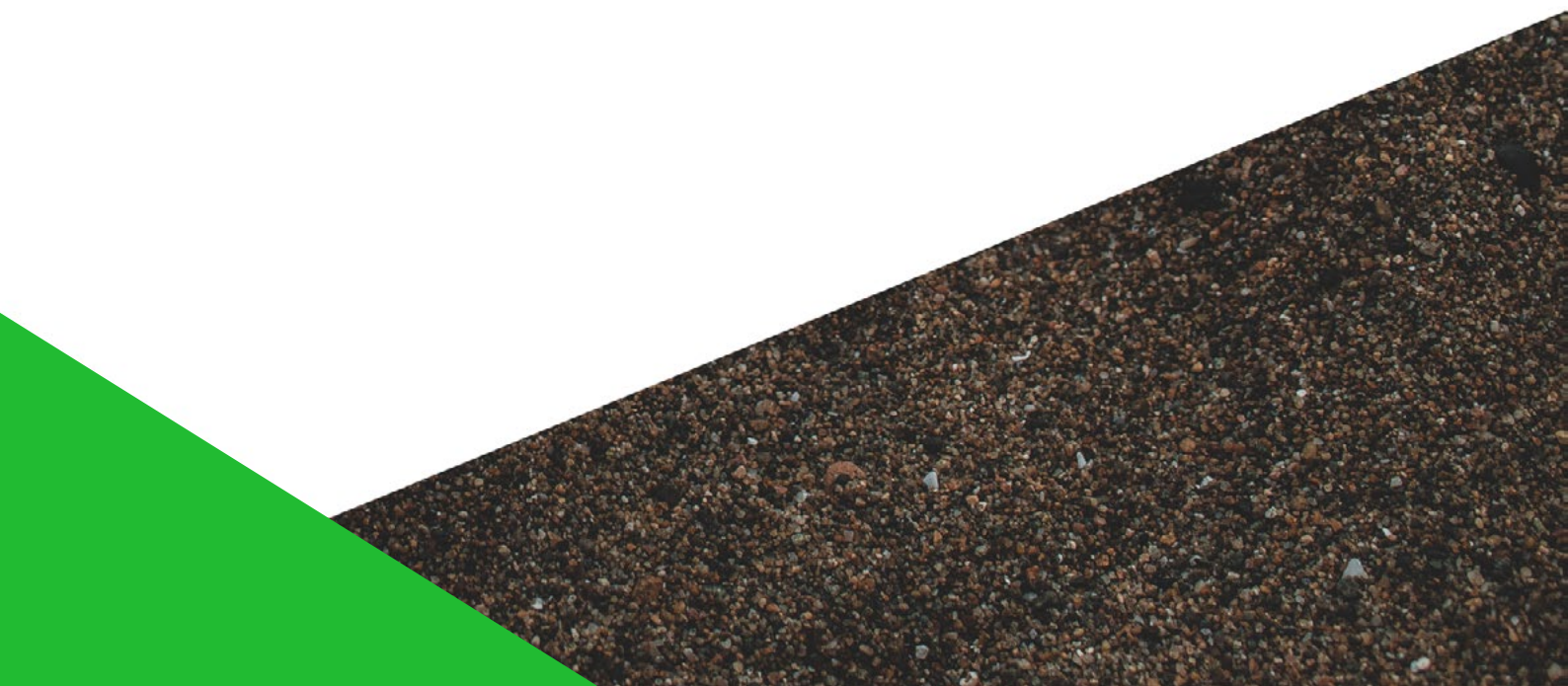
Lecciones aprendidas con esta guía

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. Algunos de los puntos más importantes son:

- Conocer los tipos de aplicaciones y paradigmas de programación.
- Aprender las bases de Java 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.
- Conocer los problemas que debemos prevenir, tanto a nivel de seguridad como de gestión de memoria.
- Descubrir las APIs más utilizadas, desde los tipos básicos, genéricos y opcionales, hasta las colecciones, los streams o la concurrencia.

Por supuesto, esto no es más que una pincelada de lo que un lenguaje tan veterano y potente como Java nos ofrece. Te animamos a que sigas indagando por tu cuenta en los temas que más llamen tu interés.



Expertos en creación de software de calidad

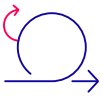
Diseñamos productos digitales y experiencias a medida

SOPORTE A DESARROLLO

Construimos entornos sólidos para los proyectos, trabajando a diario con los equipos de desarrollo.

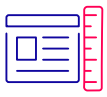
DISEÑO DE PRODUCTO Y UX

Convertimos tus ideas en productos digitales de valor para los usuarios finales.



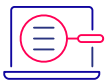
ACOMPAÑAMIENTO AGILE

Ayudamos a escalar modelos ágiles en las organizaciones.



SOFTWARE A MEDIDA

Desarrollamos aplicaciones web y móviles. Fullstack developers, expertos en backend.



AUDITORÍA DE DESARROLLO

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