# Java 8. Nivel II.

[Sección 1. Conceptos Java. 5](#_Toc141264826)

[1. Encapsulación 5](#_Toc141264827)

[1.1 Encapsulación 5](#_Toc141264828)

[**1.2 Buenas prácticas.** 5](#_Toc141264829)

[1.2 Tipos de clases 5](#_Toc141264830)

[2. Herencia con modificadores de acceso y composición 5](#_Toc141264831)

[2.1 Herencia de clases 5](#_Toc141264832)

[2.2 Herencia de interfaces 6](#_Toc141264833)

[2.3 Asociación de composición 6](#_Toc141264834)

[3. Polimorfismo 6](#_Toc141264835)

[4. Sobrescribir los métodos equals, hashcode y toString 7](#_Toc141264836)

[4.1 Herencia de Object 7](#_Toc141264837)

[4.2 Sobrescritura de equals 7](#_Toc141264838)

[4.3 Sobrescritura de hashCode 7](#_Toc141264839)

[4.4 Sobrescritura de toString 7](#_Toc141264840)

[5.1 Atributos de objeto y de clase 7](#_Toc141264841)

[5.2 Métodos estáticos 7](#_Toc141264842)

[5.3 Constantes 7](#_Toc141264843)

[6. Singleton y clases inmutables 8](#_Toc141264844)

[6.1 Objetos Singleton 8](#_Toc141264845)

[6.2 Objetos inmutables 8](#_Toc141264846)

[7. Abstract 9](#_Toc141264847)

[7.1 Abstract 9](#_Toc141264848)

[7.2 A nivel de clase 9](#_Toc141264849)

[7.3 A nivel de método 9](#_Toc141264850)

[7.4 Clases abstractas que implementan interfaces 9](#_Toc141264851)

[8. Modificador final 10](#_Toc141264852)

[8.1 Clases final 10](#_Toc141264853)

[8.2 Métodos final 10](#_Toc141264854)

[8.3 Variables final 10](#_Toc141264855)

[9. Clases anidadas, internas y locales. Clases anónimas. 10](#_Toc141264856)

[9.1 Clases dentro de otras clases 10](#_Toc141264857)

[9.2 Clases internas 10](#_Toc141264858)

[9.3 Clases locales 11](#_Toc141264859)

[9.4 Clases anónimas 11](#_Toc141264860)

[10. Tipos enumerados 11](#_Toc141264861)

[11. Clases genéricas. 11](#_Toc141264862)

[11.1 Clases genéricas 11](#_Toc141264863)

[11.2 Instanciación y operador diamond 12](#_Toc141264864)

[11.4 Genéricos con tipos comodín 12](#_Toc141264865)

[12. API de colecciones. 12](#_Toc141264866)

[12.1 API de colecciones 12](#_Toc141264867)

[12.2 Tipos de colecciones 13](#_Toc141264868)

[12.3 Intefaz List 13](#_Toc141264869)

[12.4 Interfaz Set 13](#_Toc141264870)

[12.5 Interfaz Map 13](#_Toc141264871)

[13. Comparable y comparator 13](#_Toc141264872)

[13.1 Comparable 13](#_Toc141264873)

[13.2 Comparator 13](#_Toc141264874)

[14. Interfaces funcionales 14](#_Toc141264875)

[14.1 Interfaces funcionales 14](#_Toc141264876)

[15. Predicate. Consumer. Function. Supplier 14](#_Toc141264877)

[15.1 Predicate<T> 14](#_Toc141264878)

[15.2 Consumer<T> 14](#_Toc141264879)

[15.3 Function<T, R> 15](#_Toc141264880)

[15.4 Supplier<T> 15](#_Toc141264881)

[ANEXOS. 15](#_Toc141264882)

[Paradigma de programación funcional 15](#_Toc141264883)

[Expresión lambda 15](#_Toc141264884)

[Interfaz funcional 15](#_Toc141264885)

[Colecciones y el método forEach 16](#_Toc141264886)

[Acceso a métodos 16](#_Toc141264887)

[API Stream 16](#_Toc141264888)

[Sección 2. Stream API. 16](#_Toc141264889)

[16.1 Introducción 16](#_Toc141264890)

[16.2 Características de un Stream 16](#_Toc141264891)

[16.3 Algunos subtipos de streams 16](#_Toc141264892)

[16.4 Formas de obtener un stream 16](#_Toc141264893)

[16.5 Operaciones intermedias 17](#_Toc141264894)

[16.6 Operaciones terminales 17](#_Toc141264895)

[17.1 Métodos de búsqueda 17](#_Toc141264896)

[18.1 Métodos de datos y cálculo 17](#_Toc141264897)

[18.2 Métodos de reducción 18](#_Toc141264898)

[18.3 Métodos de resumen 18](#_Toc141264899)

[18.4 Métodos de ordenación 18](#_Toc141264900)

[19.1 Uso de map 18](#_Toc141264901)

[19.2 Uso de flatMap 18](#_Toc141264902)

[19.3 Collectors 19](#_Toc141264903)

[20.1 Filter 20](#_Toc141264904)

[21.1 Referencias a métodos 20](#_Toc141264905)

[21.2 Tipos de referencias a métodos 20](#_Toc141264906)

[Sección 3. FLUJOS. JAVA I/O Y NI/O2 22](#_Toc141264907)

[23.1 Flujos 22](#_Toc141264908)

[23.2 Tipos de Flujos 22](#_Toc141264909)

[23.3 Flujos de salida 22](#_Toc141264910)

[24.1 Patrón básico de los flujos de entrada 23](#_Toc141264911)

[24.2 Flujos de entrada de bytes 24](#_Toc141264912)

[24.3 Flujos de entrada de caracteres 24](#_Toc141264913)

[25.1 Clase File 24](#_Toc141264914)

[25.2 Creación de un File 24](#_Toc141264915)

[26.1 Introducción 24](#_Toc141264916)

[26.2 Clase Path 24](#_Toc141264917)

[27.1 Métodos de la clase File 24](#_Toc141264918)

[Sección 4. Concurrencia. 24](#_Toc141264919)

[30.1 Hilos con la clase Thread 24](#_Toc141264920)

[30.2 Creación de un Thread 25](#_Toc141264921)

[30.3 Ciclo de vida de un Thread 25](#_Toc141264922)

[30.4 Pausar la ejecución de un Thread 25](#_Toc141264923)

[31.1 Runnable 26](#_Toc141264924)

[31.2 Callable<V> y Future<V> 26](#_Toc141264925)

[31.3 Ejecutores y pools de hilos 26](#_Toc141264926)

[32.1 Introducción 27](#_Toc141264927)

[32.2 Interferencia entre hilos 27](#_Toc141264928)

[32.3 Código synchronized 28](#_Toc141264929)

[32.4 Cerrojos y sincronización 28](#_Toc141264930)

[32.5 Variables atómicas 28](#_Toc141264931)

[33.1 Problemas en colecciones de acceso concurrente 29](#_Toc141264932)

[33.2 wait y notify 29](#_Toc141264933)

[33.3 Colecciones concurrentes 29](#_Toc141264934)

[Sección 4. Acceso a Bases de datos 31](#_Toc141264935)

[34.1 Introducción 31](#_Toc141264936)

[34.2 Drivers 31](#_Toc141264937)

[34.3 Interfaces principales 31](#_Toc141264938)

[34.4 Conexión con la base de datos 31](#_Toc141264939)

[35.1 Introducción 32](#_Toc141264940)

[35.2 Statement 32](#_Toc141264941)

[35.3 ResultSet 32](#_Toc141264942)

[35.4 PreparedStatement 32](#_Toc141264943)

[36.1 RowSet 33](#_Toc141264944)

[36.2 ¿En qué se diferencian de un ResultSet? 33](#_Toc141264945)

[36.3 Creación de un RowSet 33](#_Toc141264946)

[36.4 JdbcRowSet 33](#_Toc141264947)

[36.5 CachedRowSet 34](#_Toc141264948)

# Sección 1. Conceptos Java.

# 1. Encapsulación

## 1.1 Encapsulación

Los objetos conocen solamente su estructura, no la de los demás. El trato entre objetos se realiza a través de los métodos. Normalmente, los atributos de un objeto se deben consultar o editar a través de métodos.

**1.2 Buenas prácticas.**

**Para clases**

La mayoría de las clases que se crean son públicas.

Cada fichero .java tendrá solamente una clase pública, con el mismo nombre del fichero.

**Para atributos**

La mayoría de los atributos de una clase serán privados.

Solamente algunas constantes, o casos muy particulares, tendrán otra modificador de acceso.

**Para métodos**

Si una clase tiene atributos, seguramente tenga métodos públicos.

Los métodos privados son interesantes para cálculos auxiliares o parciales (solo se pueden invocar desde la propia clase).

## 1.2 Tipos de clases

Aunque Java tiene solamente una forma de crear clases, los patrones de diseño nos dicen que podemos encontrar diferentes tipos de clases según su cometido:

**Modelo**: representan objetos o hechos de la naturaleza: un coche, un asiento contable, los datos meteorológicos de un día. Suelen tener atributos, getters y setters, equals, hashCode, toString, …

**Servicios**: implementan la lógica de negocio. Suelen tener algunos atributos, pero sobre todo métodos públicos y privados.

**Auxiliare**s: sirven para realizar operaciones auxiliares de cálculo o transformación de datos. Mayoritariamente, sus métodos son estáticos.

**Main:** son el punto de entrada de la aplicación. La mayoría de las ocasiones, solo tienen este método, y si tienen más, suelen ser estáticos.

**Test**: clases orientadas a realizar pruebas de nuestra aplicación. En Java, suelen ser test unitarios con JUnit.

# 2. Herencia con modificadores de acceso y composición

## 2.1 Herencia de clases

Una clase que extiende a otra hereda sus atributos y sus métodos (no constructores). Puede añadir atributos y métodos nuevos. Se trata de una asociación es-un, ya que la clase Hija es-un(a) (sub)tipo de la clase Base. Por ejemplo un Coche es-un Vehiculo, o un Leon es-un Animal.

Si usamos protected en la clase base, tendremos acceso directo a los atributos. En otro caso, tendremos que acceder vía getters/setters. ¡OJO! Los constructores no se heredan aunque sean públicos.

public class Base {

private String nombre;

protected String apellidos;

*//...*

}

public class Hija extends Base {

public void metodo() {

*//this.nombre = "Pepe"; //Imposible acceder. Nos da error*

this.setNombre("Pepe"); *//Funciona perfectamente*

this.apellidos = "Perez";

}

*//...*

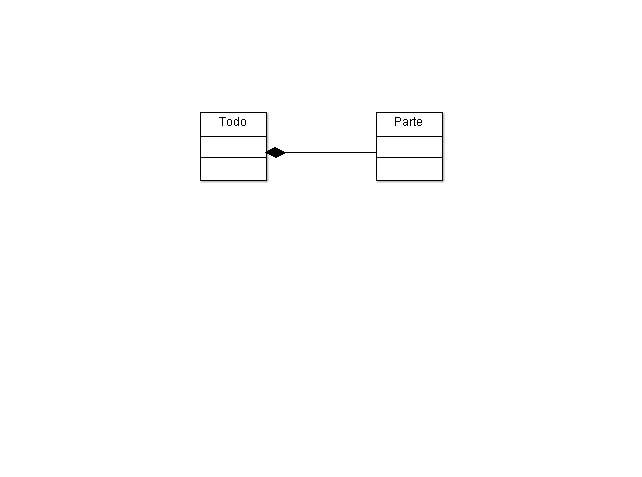
}

## 2.2 Herencia de interfaces

También podemos establecer relaciones jerárquicas entre interfaces. Nos regimos por las mismas reglas que en el caso de las clases.

## 2.3 Asociación de composición

Normalmente, cuando representamos la estructura de un sistema, está formado por muchas clases. En este caso, no solamente importan las clases en sí, sino las asociaciones. Una de ellas es la composición. En UML, se representan de una forma especial.



Dentro de la clase Todo tendremos una referencia a la clase Parte. También es posible que la multiplicidad nos indique que debemos tener una colección (Provincia y Municipio). Normalmente hay dependencia de existencia entre la parte y el todo.

# 3. Polimorfismo

Java nos permite crear instancias de objetos, pero que estos sean referenciados por alguna de sus clases ancestro o alguno de los interfaces que implementa:

Rectangulo r = new Cuadrado();

También permite la ocultación o sobreescritura de métodos por parte de las clases derivadas.

¿Qué sucede en caso de el uso de referencias de clase base, pero instanciación de objetos derivados? Java escoge, en tiempo de ejecución, el tipo de objeto. Si ese tipo ha ocultado un método de la superclase, llama a la concreción. En otro caso, llama al método de la clase base.

Cuadrado [lado=5.39897]

Area de un cuadrado

Perímetro de un cuadrado

Area de la figura: 10,797940 | Perímetro de la figura: 21,595881

Rectangulo [base=10.55045, altura=0.6183386]

Area de un rectángulo

Perímetro de un rectángulo

Area de la figura: 6,523750 | Perímetro de la figura: 22,337578

Esto también sucede cuando una clase implementa una interfaz que forma parte de una jerarquía de herencia:

ClaseQueImplementaInterfaz c1 = new ClaseQueImplementaInterfaz()*;*

c1.saludar("Hola Mundo")*;*

Hija c2 = new ClaseQueImplementaInterfaz()*;*

c2.saludar("Hola Mundo, otra vez")*;*

Base c3 = new ClaseQueImplementaInterfaz()*;*

c3.saludar("Hola Mundo, por tercera vez")*;*

# 4. Sobrescribir los métodos equals, hashcode y toString

## 4.1 Herencia de Object

Todo objeto, de forma directa o indirecta hereda de Object. Esta clase tiene una serie de métodos, entre los que destacan:

equals: nos permite indicar cuando dos objetos son iguales

hashCode: nos devuelve un número “único” asociado a una instancia de una clase

toString: nos devuelve una reperesentación del objeto como una cadena de caracteres.

## 4.2 Sobrescritura de equals

Con tipos primitivos, hemos usado el operador ==. ¿Qué sucede con los objetos? Primero tenemos que definir cuando dos instancias de un objeto son iguales o diferentes.

El método equals nos permite devolver un boolean indicando si un objeto es igual a otro. Nuestro IDE lo autogenera, junto con hashCode. La representación actual es muy verbosa y no aprovecha las capacidades de Java 7. La actualización está en desarrollo.

## 4.3 Sobrescritura de hashCode

Devuelve un número asociado a la clase. Sirve como posición de memoria en hexadecimal. Por definición, si dos objetos son iguales (equals), su hash code también debe serlo. Si sobrescribimos el método equals, también tenemos que sobrescribir hashCode para que se cumpla esa propiedad.

## 4.4 Sobrescritura de toString

Devuelve una representación en String del objeto. Por defecto, devuelve el tipo (la clase) y su hashCode. Lo podemos sobrescribir para que represente los valores. Dos objetos iguales deben tener la misma representación.

### 5. Atributos de objeto y de clase. Métodos estáticos. Constantes

## 5.1 Atributos de objeto y de clase

Java nos permite crear cuantos objetos queramos de una misma clase. Estos objetos tienen una copia propia de los atributos. Sin embargo, en ocasiones, puede interesarnos tener un atributo común a todos los objetos. Estos son los atributos static.

Están asociados a la clase, y compartidos para todas las instancias. Pueden ser manipulados por cualquier objeto, o incluso sin crear una instancia de esa clase.

## 5.2 Métodos estáticos

Se rigen por los mismos principios de los atributos estáticos. Para invocar: Clase.metodoEstatico(...);

Podemos acceder a una variable estática desde un contexto estático o no estático. También podemos crear métodos estáticos como métodos de operaciones auxiliares (recordemos todos los métodos de la clase java.util.Arrays).

## 5.3 Constantes

Se pueden definir como static.

static final PI = 3.141592653589793;

5.4 Clases estáticas

Este tipo de clases solamente tienen sentido cuando trabajemos con clases internas (lo haremos en capítulos posteriores). Nos permiten, entre otras cosas, agrupar código, definiendo clases que tendrán solo sentido si están envueltas por otra.

Para crear una instancia de la clase interna, si esta es estática, no necesitamos una instancia de la clase externa:

public class EjemploClaseStatic {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

Persona p = new Persona("Pepe");

System.out.println(p);

}

public static class Persona {

//...

}

public class OtroEjemplo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

EjemploClaseStatic.Persona p = new EjemploClaseStatic.Persona("Pepe");

System.out.println(p);

}

}

# 6. Singleton y clases inmutables

## 6.1 Objetos Singleton

Singleton es uno de los patrones de diseño propuesto por Gang of Four (GOF), y sirve para poder tener una clase de la cual solamente querremos tener una instancia (manejadores, servicios, …).

Para implementarla, podemos seguir los siguientes pasos:

Definir un único constructor, como privado, para evitar instanciaciones innecesarias.

Obtener siempre la instancia a través de un método estático.

public class MiServicioSingleton {

//Una instancia del objeto que va a existir

private static MiServicioSingleton instance = null;

//Evitamos así la instanciación directa

private MiServicioSingleton() { }

public static MiServicioSingleton getInstance() {

if (instance == null)

instance = new MiServicioSingleton();

return instance;

}

}

## 6.2 Objetos inmutables

Son objetos cuyo estado no puede ser modificado una vez se haya inicializado. No son constantes ya que estas se definen en tiempo de compilación, y los inmutables en tiempo de ejecución. Un ejemplo de clase inmutable que ya hemos utilizado es String.

Algunas recomendaciones para crear objetos inmutables:

Definir todas las propiedades como final private

No añadir métodos setter

Evitar que extiendan la clase, añadiéndole el modificador final a la definición.

# 7. Abstract

## 7.1 Abstract

Es una palabra reservada, que puede usarse a nivel de método o de clase. Sirve para indicar la obligación de implementar un método o de extender una clase completa.

## 7.2 A nivel de clase

Son clases que no se pueden instanciar. Puede tener métodos con implementación y atributos, y también métodos abstractos.

public abstract class ObjectoGrafico implements Transformable {

protected int x, y;

public void moverA(int nuevaX, int nuevaY) {

this.x = nuevaX;

this.y = nuevaY;

}

abstract public void dibujar();

abstract public void cambiarTamanio(int factorAumento);

}

## 7.3 A nivel de método

Los métodos definidos como abstract deben estar en una clase abstracta. Definen la firma del método, pero sin implementación. Sus subclases se comprometen a implementarlo. Si no lo hacen, también deben ser abstractas. Pueden convivir con métodos normales.

## 7.4 Clases abstractas que implementan interfaces

Una clase que implementa una interfaz tiene obligación de implementar todos sus métodos. Sin embargo, una clase abstract puede dejar métodos sin implementación, obligando a quienes la extiendan a hacerlo.

public interface Transformable {

public void rotar();

public void voltearHorizontal();

public void voltearVertical();

}

public abstract class ObjectoGrafico implements Transformable {

protected int x, y;

public void moverA(int nuevaX, int nuevaY) {

this.x = nuevaX;

this.y = nuevaY;

}

abstract public void dibujar();

abstract public void cambiarTamanio(int factorAumento);

}

# 8. Modificador final

Se puede utilizar en diferentes contextos:

Clases final

Métodos final

Variables final

En todos los casos, indica que de una u otra forma, el ámbito sobre el que aplica no podrá ser modificado.

## 8.1 Clases final

Son clases que no se pueden extender. En una jerarquía de herencia, son el último nodo. Se pueden instanciar y tratar con normalidad.

## 8.2 Métodos final

Se definen en clases susceptibles de ser extendidas. Nos permiten indicar que un método no se va a poder sobrescribir. En la clase extendida habrá que usar, obligatoriamente, la implementación de la clase base.

## 8.3 Variables final

Básicamente indican que aquella variable a la que afectan no se puede modificar. Podemos diferenciar entre:

Tipos primitivos: serán valores inmodificables, constantes. Suele usarse junto con static.

Objetos: si declaramos una referencia como final, estamos diciendo que esa referencia no podrá asignarse a otro objeto. Sin embargo sí que podemos modificar el estado del objeto con sus propios métodos. Lo mismo sucedería en el caso de arrays.

# 9. Clases anidadas, internas y locales. Clases anónimas.

## 9.1 Clases dentro de otras clases

Java permite definir clases dentro de otras clases. A estas clases se le llaman anidadas. Pueden ser de dos tipos, estáticas o no estáticas. No se trata de composición de clases, sino anidamiento. En algunos casos, pueden acceder a los atributos de la clase que le envuelve.

Las razones para su uso son varias:

Agrupamiento lógico de clases que se utilizan en un solo lugar. Por tanto hay mayor cohesión.

Aumento de la encapsulación.

Código más legible y fácil de mantener.

## 9.2 Clases internas

Se llaman así a las clases anidadas no estáticas. Solo pueden existir en el marco de una instancia de la clase externa. Pueden acceder a sus miembros (de la clase externa).

Si definimos una variable miembro en la clase interna, con el mismo nombre otra de la clase externa, la interna oculta a la externa. A esto se le llama shadowing

## 9.3 Clases locales

Son clases que se definen dentro de un bloque, normalmente el cuerpo de un método. Sirven para afinar aún más la cohesión del código.

## 9.4 Clases anónimas

Permiten definir e instanciar una clase a la vez. Son como clases locales sin nombre. Sirven para ser usadas una vez.

Las podemos definir a partir de otra clase o de una interfaz. Podemos crearlas en el cuerpo de un método, de una clase, o como argumento de un método.

# 10. Tipos enumerados

Son un tipo de dato especial. Indica que una variable tendrá como valor uno de entre un conjunto cerrado, como por ejemplo Direccion (Norte, Sur, Este, Oeste).

public enum Direccion {

NORTE, SUR, ESTE, OESTE

}

En Java, los tipos enumerados son más potentes que en otros lenguajes. Para Java son un tipo de clase, que pueden incluir métodos y otros atributos. De hecho, el compilador añade métodos especiales (values), que incluso nos permite recorrer todas las instancias. Podemos pensar en que tenemos un conjunto cerrado de instancias de una clase.

# 11. Clases genéricas.

## 11.1 Clases genéricas

Java permite desde sus orígenes usar clases genéricas, utilizando referencias de tipo Object. Sin embargo, estas pueden producir problemas en tiempo de ejecución.

public class Box {

private Object object;

public void set(Object object) {

this.object = object;

}

public Object get() {

return object;

}

}

Desde Java SE 5, podemos crear clases cuyo tipo se indica en tiempo de compilación

public class Box<T> {

private T object;

public void set(T object) {

this.object = object;

}

public T get() {

return object;

}

}

Podemos utilizar más de un tipo diferente a la vez:

public class Par<T, S> {

private T obj1;

private S obj2;

//Resto de la clase

}

Nomenclatura con los tipos:

E (element, elemento)

K (key, clave)

N (number, número)

T (type, tipo)

V (value, valor)

S, U, V, … (2º, 3º, 4º, … tipo)

## 11.2 Instanciación y operador diamond

Hasta Java SE 6, para instanciar un objeto genérico, tenemos que indicar los tipos dos veces.

Par<String, String> pareja2 = new Par<String, String>("Hola", "Mundo");

Desde Java SE 7, tenemos el operador **<> diamond**:

Par<String, String> pareja2 = new Par<>("Hola", "Mundo");

**11.3 Clases genéricas con tipos cerrados**

Podemos acotar el tipo parametrizado, para que sea uno en particular o sus derivados:

public class NumericBox<T extends Number> {

private T object;

//resto de la clase

}

Se puede indicar más de un tipo. Uno de ellos (y solo uno) se corresponderá con una clase; el resto deben ser interfaces. La clase a extender debe ser la primera de la lista:

public class A {

//resto de la clase

}

public interface B {

//resto de la interfaz

}

public class StrangeBox <T extends A & B> {

//resto de la clase

}

## 11.4 Genéricos con tipos comodín

Los tipos comodín nos permiten relajar el tipo concreto de una clase genérica a un subtipo. Son muy útiles en el caso de trabajar con colecciones (las trataremos en los próximos capítulos).

public static double sumOfList (**List<? extends Number**> list) {

double s = 0.0;

for (Number n : list)

s += n.doubleValue();

return s;

}

# 12. API de colecciones.

## 12.1 API de colecciones

Actualmente tiene

* Interfaces: tipos de datos
* Implementaciones: concreciones de los diferentes interfaces.
* Algoritmos: para realizar operaciones como ordenación, búsqueda, …
* Actualmente, todas las colecciones están definidas como genéricas.

## 12.2 Tipos de colecciones

Java propone diferentes tipos de colecciones, a través de varias interfaces. Nosotros trabajaremos con 3:

* **List:** Se trata de una estructura lineal, con posibilidad de orden y de repetidos.
* **Set:** es una colección que no soporta duplicados, y con posibilidad de orden.
* Map: es una estructura de tipo clave, valor, con posibilidad de orden de los elementos (por la clave)

## 12.3 Intefaz List

Los elementos tienen siempre una posición, y permite duplicados. También permite búsqueda e iteraciones. Las implementaciones más conocidas son **ArrayList y LinkedList**. Si no sabemos cual escoger, utilizaremos siempre ArrayList.

Para construir una instacia, desde Java SE 7 podemos usar el operador diamond:

List<String> cars = new ArrayList<>();

## 12.4 Interfaz Set

Se trata de una colección que no puede contener repetidos. Java propone tres implementaciones: **HashSet, TreeSet y LinkedHastSet:**

* HashSet es la más eficiente, pero no nos asegura nada sobre el orden.
* TreeSet utiliza un árbol Red-Black, y ordena según el valor.
* LinkedHashSet es un HashSet ordenado por orden de inserción.

## 12.5 Interfaz Map

No es un subtipo de Collection (List y Set sí que lo son). Cada elemento tiene estructura clave, valor. La clave sirve para acceder directamente al valor. Las implementaciones **son HashMap, TreeMap y LinkedHashMap**. Las consideraciones son análogas a Set.

# 13. Comparable y comparator

Muchas operaciones entre objetos nos obligan a compararlos: buscar, ordenar, … Si bien los tipos primitivos y algunas clases ya implementan su orden (natural, lexicográfico), para nuestras clases (modelo) tenemos que especificar el orden con el que las vamos a tratar.

## 13.1 Comparable

Comparable es un interfaz propuesto por Java, y su definición es sencilla:

**public interface Compararable<T> {**

**public int compareTo(T o);**

**}**

Recibe un objeto del mismo tipo que la clase que lo implementa. El valor de retorno del método compareTo será:

* 0 si ambos objetos son iguales,
* negativo si el objeto es menor,
* positivo si es mayor.

Nos sirve para indicar el orden principal de una clase.

## 13.2 Comparator

Comparator también es un interfaz propuesto por Java, y su definición también es sencilla:

**public interface Comparator<T> {**

**public int compare(T o1, T o2);**

**}**

Recibe dos argumentos, y su valor de retorno es análogo al de comparable.

Comparator nos servirá para indicar un orden diferente al orden natural definido con Comparable (no es necesario haber definido un orden con Comparable para poder utilizar Comparator, aunque sí es recomendable).

# 14. Interfaces funcionales

**Una interfaz** es un contrato que compromete a la clase que lo implementa a dar cuerpo a una serie de métodos abstractos. Además, se pueden utilizar como referencias a la hora de crear objetos (que implementen esa interfaz, claro está):

List<String> lista = new ArrayList<>();

Desde Java SE 8, las interfaces pueden incluir la implementación de algunos métodos, en particular, los métodos anotados con default y static.

public interface Interfaz {

public void metodo(); //método abstracto

**default** public void metodoPorDefecto() {

System.out.println("Este es uno de los nuevos

métodos por defecto");

}

**public static** void metodoEstatico() {

System.out.println("Método estático en

un interfaz");

}

}

## 14.1 Interfaces funcionales

Una interfaz funcional será una interfaz que solamente tenga la definición de un método abstracto. Estrictamente hablando, puede tener varios métodos abstractos, siempre que todos menos uno sobrescriba a un método público de la clase Object. Además, pueden tener uno o varios métodos por defecto o estáticos.

Normalmente, son interfaces que implementamos mediante una clase anónima. Muchos de los interfaces que conocemos, como por ejemplo Comparator, son interfaces funcionales:

Collections.sort(lista, new Comparator<String>() {

//Ordenamos la cadena por su longitud

@Override

public int compare(String str1, String str2) {

return str1.length()-str2.length();

}

});

Java SE 8 incorpora también la anotación **@FunctionalInterface** que permite al compilador comprobar si una interfaz cumple con las características de ser funcional o no (Eclipse nos proporciona dicha funcionalidad en directo, a la par de escribir el código).

Las interfaces funcionales y las expresiones lambda están altamente ligadas, de forma que allá donde se espere una instancia de una clase que implemente una interfaz funcional, podremos utilizar una expresión lambda.

Collections.sort(lista, (str1, str2)-> str1.length()-str2.length());

# 15. Predicate. Consumer. Function. Supplier

## 15.1 Predicate<T>

El método abstracto es: boolean test(T t);

Comprueba si se cumple o no una condición. Se utiliza mucho junto a expresiones lambada a la hora de filtrar:

//...

.filter((p) -> p.getEdad() >= 35l)

Se pueden componer predicados más complejos con sus métodos and, or y negate.

## 15.2 Consumer<T>

El método abstracto es: void accept(T t);

Sirve para consumir objetos. Uno de los ejemplos más claros es imprimir.

//...

.forEach(System.out::println)

Adicionalmente, tiene el método andThen, que permite componer consumidores, para encadenar una secuencia de operaciones.

## 15.3 Function<T, R>

El método abstracto es: R apply(T t);

Sirve para aplicar una transformación a un objeto. El ejemplo más claro es el mapeo de objetos en otros.

//...

.map((p) -> p.getNombre())

Adicionalmente, tiene otros métodos:

* andThen, que permite componer funciones.
* compose, que compone dos funciones, a la inversa de la anterior.
* identity, una función que siempre devuelve el argumento que recibe

## 15.4 Supplier<T>

El método abstracto es: T get();

Sirve para devolver un valor.

Tiene algunos interfaces especializados para tipos básicos:

* IntSupplier
* LongSupplier
* DoubleSupplier
* BooleanSupplier

# ANEXOS.

## Paradigma de programación funcional

A la hora de programar, existen más paradigmas (formas de entender la programación) que la orientación a objetos: imperativa, lógica, orientada a aspectos, … Uno de ellos, que está muy de moda, es la **programación funcional**. Su origen está en un lenguaje matemático formal, y si lo tenemos que resumir en una frase, podríamos decir que en él, la salida de una función depende solo de los parámetros de entrada.

Los lenguajes funcionales son más expresivos (es decir, hacen más con menos código) y más elegantes. Este tipo de expresiones ya estaban presentes en muchos lenguajes de programación, y era algo que la comunidad demandaba a Java, que lo ha incluido en su versión Java SE 8.

## Expresión lambda

Una expresión lamba no va a ser más que una función anónima, un método abstracto. Su sintaxis es sencilla:

() -> expresión

(p1, p2, ...) -> expresión

(p1, p2, ...) -> { sentencia1; sentencia2; .....;}

Algunos ejemplos de expresiones lambda podrían ser:

() -> new ArrayList<>()

(int a, int b) -> a+b

(a) -> {

System.out.println(a);

return true;

}

## Interfaz funcional

Se trata de cualquier interfaz que tenga un solo método (además de métodos default, static, equals, …). Están muy ligadas a las expresiones lambda ya que allá donde se espere una instancia de un objeto que implemente una interfaz funcional, podremos utilizar una expresión de este tipo.

## Colecciones y el método forEach

En Java SE 8 las colecciones nos proveen de un método, .forEach, que acepta una instancia de un objeto que implementa una interfaz funcional, llamada Consumer<T>, que nos permitirá facilmente recorrer el bucle usando expresiones lambda.

lista.forEach(System.out::println);

## Acceso a métodos

**El operador ::** nos permite abreviar más la sintaxis, de forma que podemos acceder a métodos directamente:

Por ejemplo: x -> System.out.println(x) sería equivalente a System.out::println

## API Stream

Se trata de una nueva funcionalidad, que nos permite trabajar con colecciones como si se trataran de flujos de datos. Esto nos dará gran versatilidad, junto con las expresiones lambda, para filtrar, transformar, ordenar, agrupar y presentar datos.

List<Integer> lista = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

lista

.stream()

.filter((x) -> x >= 5)

.forEach(System.out::println);

# Sección 2. Stream API.

## 16.1 Introducción

*El API Stream es una de las grandes novedades de Java SE 8, junto con las expresiones lambda. Permite realizar operaciones de filtro/mapeo/reducción sobre colecciones de datos, de forma secuencial o paralela.*

Un Stream es una secuencia de elementos que soporta operaciones para procesarlos

* Usando expresiones lambda
* Permitiendo el encadenamiento de operaciones (para producir así un código que se lee mucho mejor y es más conciso)
* De forma secuencial o paralela

En Java, los streams vienen definidos por el interfaz java.util.stream.Stream<T>.

## 16.2 Características de un Stream

* Las operaciones intermedias retornan un Stream (permitiendo así el encadenamiento de llamadas a métodos).
* Las operaciones intermedias se encolan, y son invocadas al invocar una operación terminal.
* Solo se puede recorrer una vez; si lo intentamos recorrer una segunda vez, provocará una excepción.
* Utiliza iteración interna en lugar de iteración externa; así nos centramos en qué hacer con los datos, no en como recorrerlos.

## 16.3 Algunos subtipos de streams

En el caso de que vayamos a utilizar un stream de tipos básicos (int, long y double), Java nos proporciona las interfaces IntStream, LongStream y DoubleStream.

## 16.4 Formas de obtener un stream

* Stream.of(...): retorna un stream secuencial y ordenado de los parámetros que se le pasan.
* Arrays.streams(T[] t): retorna un stream secuencial a partir del array proporcionado. Si el array es de tipo básico, se retorna un subtipo de Stream.
* Stream.empty(): retorna un stream vacío.
* Stream.iterate(T, UnaryOperator<T>): devuelve un stream infinito, ordenado y secuencial. Lo hace a partir de un valor y de aplicar una función a ese valor. Se puede limitar el tamaño con limit(long).
* Collection.stream() y Collection.parallelStream(): devuelve un stream (secuencial o paralelo) a partir de una colección.
* Collection.generate: retorna un stream infinito, secuencial y no ordenado a partir de una instancia de Supplier (o su correspondiente expresión lambda).

## 16.5 Operaciones intermedias

Son operaciones que devuelven un Stream, y por tanto, permiten encadenar llamadas a métodos. Sirven, entre otras funcionalidades, para filtrar y transformar los datos.

### 16.5.1 Operaciones de filtrado

* filter(Predicate<T>): nos permite filtrar usando una condición.
* limit(n): nos permite obtener los n primeros elementos.
* skip(m): nos permite *obviar* los m primeros elementos.

### 16.5.2 Operaciones de mapeo

* map(Function<T,R>): nos permite transformar los valores de un stream a través de una expresión lambda o una instancia de Function.
* mapToInt(...), mapToDouble(...) y mapToLong(...) nos permite transformar a tipos básicos, obteniendo IntStream, DoubleStream o LongStream, respectivamente.

## 16.6 Operaciones terminales

Provocan que se ejecuten todas las operaciones intermedias. Las hay de varios tipos:

* Para consumir los elementos (por ejemplo, forEach)
* Para obtener datos de un stram (agregación)
* Para recolectar los elementos y transformarlos en otro objeto, como una colección.

## 17.1 Métodos de búsqueda

Son un tipo de operaciones terminales sobre un stream, que nos permiten:

* Identificar si hay elementos que cumplen una determinada condición
* Obtener (si el stream contiene alguno) determinados elementos en particular.

Algunos de los métodos de búsqueda son:

* allMatch(Predicate<T>): verifica si todos los elementos de un stream satisfacen un predicado.
* anyMatch(Predicate<T>): verifica si algún elemento de un stream satisface un predicado.
* noneMatch(Predicate<T>): opuesto de allMatch(…)
* findAny(): devuelve en un Optional<T> un elemento (cualquiera) del stream. *Recomendado en streams paralelos.*
* findFirst() devuelve en un Optional<T> el primer elemento del stream. *NO RECOMENDADO en streams paralelos.*

## 18.1 Métodos de datos y cálculo

Los streams nos ofrecen varios tipos de métodos terminales para realizar operaciones y cálculos con los datos. Durante el curso trabajaremos con tres tipos:

* Reducción y resumen (en esta lección)
* Agrupamiento
* Particionamiento

## 18.2 Métodos de reducción

Son métodos que reducen el stream hasta dejarlo en un solo valor.

* reduce(BinaryOperator<T>):Optional<T> realiza la reducción del Stream usando una función asociativa. Devuevle un Optional
* reduce(T, BinaryOperator<T>):T realiza la reducción usando un valor inicial y una función asociativa. Se devuelve un valor como resultado.

## 18.3 Métodos de resumen

Son métodos que resumen todos los elementos de un stream en uno solo:

* count: devuelve el número de elementos del stream.
* min(...), max(...): devuelven el máximo o mínimo (se puede utilizar un Comparator para modificar el orden natural).

## 18.4 Métodos de ordenación

Son operaciones intermedias, que devuelven un stream con sus elementos ordenados.

* sorted() el stream se ordena según el orden natural.
* sorted(Comparator<T>) el stream se ordena según el orden indicado por la instancia de Comparator

## 19.1 Uso de map

map es una de las operaciones intermedias más usadas, ya que permite la transformación de un objeto en otro, a través de un Function<T, R>. Se invoca sobre un Stream<T> y retorna un Stream<R>. Además, es muy habitual realizar transformaciones sucesivas.

lista

.stream()

.map(Persona::getNombre)

.map(String::toUpperCase)

.forEach(System.out::println);

## 19.2 Uso de flatMap

Los streams sobre colecciones de un nivel permiten transformaciones a través de map pero, ¿qué sucede si tenemos una colección de dos niveles (o una dentro de objetos de otro tipo)?:

public class Persona {

private String nombre;

private List<Viaje> viajes = new ArrayList<>();

*//resto de atributos y métodos*

}

Para poder trabajar con la colección interna, necesitamos un método que nos unifique un Stream<Stream<T>> en un solo Stream<T>. Ese es el cometido de flatMap.

lista

.stream()

.map((Persona p) -> p.getViajes())

.flatMap(viajes -> viajes.stream())

.map((Viaje v) -> v.getPais())

.forEach(System.out::println);

También tenemos disponibles las versiones primitivas flatMapToInt, flatMapToLong y flatMapToDouble:

Arrays

.stream(numeros)

.flatMapToInt(x -> Arrays.stream(x))

.map(IntUnaryOperator.identity())

.distinct()

.forEach(System.out::println);

## 19.3 Collectors

Los collectors nos van a permitir, en una operación terminal, construir una collección mutable, el resultado de las operaciones sobre un stream.

### **19.3.1 Colectores “básicos”**

Nos permiten operaciones que recolectan todos los valores en uno solo. Se solapan con algunas operaciones finales ya estudiadas, pero están presentes porque se pueden combinar con otros colectores más potentes.

* counting: cuenta el número de elementos.
* minBy(…), maxBy(…): obtiene el mínimo o máximo según un comparador.
* summingInt, summingLong, summingDouble: la suma de los elementos (según el tipo).
* averagingInt, averagingLong, averagingDouble: la media (según el tipo).
* summarizingInt, summarizingLong, summarizingDouble: los valores anteriores, agrupados en un objeto (según el tipo).
* joinning: unión de los elementos en una cadena.

### **19.3.2 Colectores “grouping by”**

Hacen una función similar a la cláusula GROUP BY de SQL, permitiendo agrupar los elementos de un stream por uno o varios valores. Retornan un Map.

Map<String, List<Empleado>> porDepartamento =

empleados

.stream()

.collect(groupingBy(Empleado::getDepartamento));

Se pueden usar en conjunción con los colectores básicos, o con otro colector grouping by:

Map<String, Long> porDepartamentoCantidad =

empleados

.stream()

.collect(groupingBy(Empleado::getDepartamento, counting()));

Map<String, Map<Double, List<Empleado>>> porDepartamentoYSalario =

empleados

.stream()

.collect(groupingBy(Empleado::getDepartamento, groupingBy(Empleado::getSalario)));

También tenemos los colectores partitioning, que nos agrupan los resultados dos conjuntos, según si cumplen una condición:

Map<Boolean, List<Empleado>> salarioMayorOIgualque32000 =

empleados

.stream()

.collect(partitioningBy(e -> e.getSalario() >= 32000));

### **19.3.3 Colectores “Collection”**

Pronducen como resultado una colección: List, Set y Map.

Set<Empleado> setEmpleados = empleados.stream().collect(Collectors.toSet());

List<Empleado> listEmpleados = empleados.stream().collect(Collectors.toList());

Map<String, Double> mapEmpleados = empleados.stream().distinct()

.collect(Collectors.toMap(Empleado::getNombre, Empleado::getSalario));

## 20.1 Filter

filter es una operación intermedia, que nos permite eliminar del stream aquellos elementos que no cumplen con una determinada condición, marcada por un Predicate<T>.

personas

.stream()

.filter(p -> p.getEdad() >= 18 && p.getEdad() <= 65)

.forEach(persona -> System.out.printf("%s (%d años)%n", persona.getNombre(), persona.getEdad()));

Es muy combinable con algunos métodos como findAny o findFirst:

Persona p1 = personas

.stream()

.filter(p -> p.getNombre().equalsIgnoreCase("Andrés"))

.findAny()

.orElse(new Persona());

Y se puede usar también en streams sobre colecciones tipo Map.

Map<Integer, Persona> personas = new HashMap<>();

*//Inicialización*

personas.entrySet()

.stream()

.filter(map -> map.getKey() >= 5)

.collect(Collectors.toMap(p -> p.getKey(), p -> p.getValue()))

.forEach((key, value) -> System.out.printf("%d: %s%n", key, value.getNombre()));

## 21.1 Referencias a métodos

Las referencias a métodos son una forma de hacer nuestro código aun más conciso:

public class Persona {

//…

public static int

compararPorEdad(Persona a, Persona b) {

return a.fechaNacimiento

.compareTo(b.fechaNacimiento);

}

}

//...

List<Persona> personas = //...

// De menos a más "conciso"

personas.sort((Persona p1, Persona p2) -> {

return p1.getFechaNacimiento()

.compareTo(p2.getFechaNacimiento());

});

personas.sort((p1, p2) -> p1.getFechaNacimiento()

.compareTo(p2.getFechaNacimiento()));

personas.sort(Persona::compararPorEdad);

## 21.2 Tipos de referencias a métodos

* Clase::metodoEstatico: referencia a un método estático.
* objeto::metodoInstancia: referencia a un método de instancia de un objeto concreto.
* Tipo::nombreMetodo: referencia a un método de instancia de un objeto arbitrario de un tipo en particular.
* Clase::new: referencia a un constructor.

# Sección 3. FLUJOS. JAVA I/O Y NI/O2

## 23.1 Flujos

Son un canal de comunicación de las operaciones de entrada salida. Este esquema nos da independencia para poder trabajar igual tanto si estamos escribiendo en un fichero, como en consola, o si estamos leyendo de teclado, o de una conexión de red.

## 23.2 Tipos de Flujos

Dependiendo de su destino, tenemos:

* Flujos de entrada: sirven para introducir datos en la aplicación.
* Flujos de salida: sirven para sacar datos de la aplicación.

Dependiendo del contenido del flujo:

* Flujos de bytes: manejan datos en crudo.
* Flujos de caracteres: manejan caracteres o cadenas.

## 23.3 Flujos de salida

### **23.3.1 Patrón básico de los flujos de salida**

Abrir el flujo

Mientras hay datos que escribir

Escribir datos en el flujo

Cerrar el flujo

### **23.3.2 Flujos de salida de bytes**

Algunas de las clases que podemos usar son:

* OutputStream: clase abstracta, padre de la mayoría de los flujos de bytes.
* FileOutputStream: flujo que permite escribir en un fichero, byte a byte.
* BufferedOutputStream: flujo que permite escribir grupos (buffers) de bytes.
* ByteArrayOutputStream: flujo que permite escribir en memoria, obteniendo lo escrito en un array de bytes.

### **23.3.3 Flujos hacia otros flujos**

Solo FileOutputStream tiene un constructor que acepta una ruta (entre otras opciones). El resto reciben en sus constructores un tipo de OutputStream. ¿Por qué? Porque podemos construir flujos que escriben en flujos (encadenados).

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("fichero.dat"));

### **23.3.4 Flujos de salida de caracteres**

* Writer: clase abstracta, padre de la mayoría de los flujos de caracteres.
* FileWriter: flujo que permite escribir en un fichero, caracter a caracter.
* BufferedWriter: flujo que permite escribir líneas de texto.
* StringWriter: flujo que permite escribir en memoria, obteniendo lo escrito en un String
* OutputStreamWriter: flujo que permite transformar un OutputStream en un Writer.
* PrintWriter: flujo que permite escribir tipos básicos Java.

Diagrama

Descripción generada automáticamente

## 24.1 Patrón básico de los flujos de entrada

Abrir el flujo

Mientras hay datos que leer

Leer datos del flujo

Procesar los datos

Cerrar el flujo

## 24.2 Flujos de entrada de bytes

Algunas de las clases que podemos usar son:

* InputStream: clase abstracta, padre de la mayoría de los flujos de bytes.
* FileInputStream: flujo que permite leer de un fichero, byte a byte.
* BufferedInputStream: flujo que permite leer grupos (buffers) de bytes.
* ByteArrayInputStream: flujo que permite leer de memoria (de un array de bytes).

## 24.3 Flujos de entrada de caracteres

Algunas de las clases que podemos usar son:

* Reader: clase abstracta, padre de la mayoría de los flujos de caracteres.
* FileReader: flujo que permite leer de un fichero, caracter a caracter.
* BufferedReader: flujo que permite leer líneas de texto.
* StringReader: flujo que permite leer desde la memoria.
* InputStreamReader: flujo que permite transformar un InputStream en un Reader.

## 25.1 Clase File

Esta clase fue fundamental para el manejo de ficheros y directorios hasta Java SE 6. A partir de Java SE 7 tenemos a nuestra disposición las clases y métodos de Java NIO.2, que estudiaremos en próximas lecciones.

## 25.2 Creación de un File

File f = new File("./", "nuevo.txt");

f.createNewFile();

## 26.1 Introducción

Un path o ruta es una forma de identificar (y acceder) un fichero o un directorio dentro del sistema de ficheros.

Decimos que una ruta es **absoluta** cuando contiene el elemento raíz y la lista de directorios completa para localizar el fichero. Por contrario, decimos que es **relativa** cuando necesita ser combinada para acceder al fichero o directorio.

## 26.2 Clase Path

Se introdujo en Java SE 7, como parate de Java NIO.2. Representa una ruta en el sistema de ficheros, y permite manejar diferentes sistemas de ficheros (Windows, Linux, Mac, …). Contiene el nombre del fichero, así coo la lista de directorios usada para construir la ruta.

## 27.1 Métodos de la clase File

En esta sección vemos la clase flies, que tiene decenas de métodos estáticos para hacer múltiples operaciones con ficheros y directorios.

Ver

# Sección 4. Concurrencia.

## 30.1 Hilos con la clase Thread

Java tiene una clase, Thread, para poder crear y manejar hilos diferentes de ejecución al hilo principal.

Sus métodos básicos son:

* public void run(): contiene el código que queremos ejecutar en el hilo. **No se debe invocar nunca directamente**.
* public void start(): lanza la ejecución del hilo.

El resto de métodos los podemos consultar en la documentación del API. <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>

## Ciclo de Vida30.2 Creación de un Thread

Una primera forma de crear un hilo es extendiendo la clase Thread, y aportando el cuerpo del método run:

public class PrimoThread extends Thread {

*//propiedades y constructores*

@Override

public void run() {

long n = minimo;

while(!testPrimalidad(n)) {

System.out.printf("%d no es primo %n", n);

++n;

}

System.out.printf("El número primo es %d %n", n);

}

public static boolean testPrimalidad(long n) {

*//cuerpo del método*

}

}

Para ejecutarlo, tenemos que instanciar la clase, y usar el método start:

PrimoThread pt = new PrimoThread(135);

pt.start();

## 30.3 Ciclo de vida de un Thread

1. Se llama al constructor de Thread para crear el nuevo hilo.
2. Se llama al método start para designarlo como ejecutable
3. El planificador lo ejecuta en cuanto el procesador está disponible.
4. El hilo puede pasar a bloqueado por diferentes razones, y no vuelve hasta que pasa de nuevo a ejectuable.
5. Si se utiliza el método wait, se pone en estado de espera, y permanece ahí hasta que se ejecuta notify o notifyAll.
6. El hilo termina cuando finaliza la ejecución de su método run.

## 30.4 Pausar la ejecución de un Thread

Lo podemos hacer usando el método estático sleep. Recibe como argumento un número de *milisegundos*.

Thread.sleep(1000)*;*

## 31.1 Runnable

Es un interfaz que nos permite crear tareas para ser ejecutadas en *hilos secundarios*. Si nuestra clase ya hereda de otra, no puede heredar a la vez de Thread. Runnable nos permite solventar esa dificultad. Thread tiene un constructor que recibe como argumento un *runnable*.

Podemos utilizar las clases anónimas o las expresiones lambda para crear código más conciso:

//Este trozo de código puede ser muy común si

//el proceso de lanzamiento del hilo está dentro

//de un manejador de eventos, como el click de un

//botón.

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("Hello from Runnable+Thread!!");

}

}).start();

//Este código es como el anterior, pero aprovecha la potencia

//de las expresiones lambda.

new Thread(() -> System.out.print("Hello from Lambda Runnable+Thread")).start();

## 31.2 Callable<V> y Future<V>

Runnable no permite devolver ningún dato a otro hilo, salvo que utilicemos complejos mecanismos de sincronización programados por nosotros mismos. Para facilitar el trabajo, Java nos ofrece el interfaz Callable<V> que es, básicamente, un Runnable que devuelve un valor.

Para recoger el valor de un Callable, necesitamos un Future. Se trata de otra interfaz, que representa el resultado de una *computación* asíncrona. Nos permite operaciones como comprobar si la computación ha terminado, cancelarla, esperar a que termine, tomar el valor si ha terminado, … El método get nos permite obtener el valor de la ejecución de un Callable. Existen diversas formas de obtener un Future a partir de un Callable, pero siempre nos *invita* al uso de ejecutores.

## 31.3 Ejecutores y pools de hilos

Hasta ahora, el programador tenía que definir y lanzar los hilos de ejecución según su necesidad. Este esquema es válido para aplicaciones pequeñas. Para grandes aplicaciones, hay que separar la administración de los hilos del resto de la aplicación. Esto lo podemos realizar mediante ejecutores (*Executors*).

Java nos ofrece tres tipos de ejecutores (cada uno hereda del anterior)

* Executor: soporta el lanzamiento de nuevas tareas, bajo demanda.
* **ExecutorService: añade a la anterior características que permiten administrar el ciclo de vida.** Este será el más utilizado.
* ScheduledExecutorService: añade a la anterior la posibilidad de ejecutar tareas periódicas.

La mejor forma de crear un ejecutor, es utilizando los métodos de los que nos provee Executors, a través de un pool de hilos. Este no será más que un conjunto de *worker threads* que estarán siempre dispuestos a ejecutar tareas de tipo Runnable o Callable.

Los pools pueden ser de tres tipos:

* *Single*: Con un solo hilo de ejecución disponible. Si le pedimos (submit) más de una tarea a la vez, las pone en cola.
* *Fixed*: Indicamos, en el momento de su creación, el número de hilos. Si dispone de n hilos, y enviamos n+1 tareas, las pone en cola.
* *Cached*: Crea hilos conforme enviamos tareas. Reutiliza los hilos cuyas tareas han finalizado, para ejecutar tareas nuevas.

## 32.1 Introducción

La extensión del texto de esta lección es algo más larga, intentando ofrecer un cierto soporte sobre la sincronización y sus causas. Java ofrece una serie de herramientas de sincronización; su buen uso depende de que el programador maneje conceptos sobre programación concurrente, algunos de los cuales escapan al alcance de este curso.

## 32.2 Interferencia entre hilos

Los hilos se comunican, principalmente, compartiendo el acceso a los atributos y los objetos a los que hacen referencia. Esta comunicación es extremadamente eficiente, pero hace posible dos tipos de errores: errores de interferencia entre hilos y errores de consistencia de memoria. La herramienta necesaria para evitar estos errores es la sincronización.

Supongamos la siguiente clase:

class Counter {

private int c = 0;

public void increment() {

c++;

}

public void decrement() {

c--;

}

public int value() {

return c;

}

}

Counter está diseñada para que cada invocación de increment se sume 1 a c, y con cada invocación de decrement se reste 1 de c. Sin embargo, si un objeto contador es referenciado desde varios hilos, la interferencia entre los hilos puede hacer que esto no suceda como se esperaba.

La interferencia ocurre cuando dos operaciones, que se ejecuta en diferentes hilos, pero que actúa sobre los mismos datos, se intercalan. Esto sucede porque las dos operaciones consisten en múltiples pasos, y las secuencias de pasos se superponen.

No parece posible que las operaciones en las instancias de Counter se intercalen, ya que ambas operaciones en c son, simples sentencias individuales. Sin embargo, incluso las sentencias simples pueden traducirse en múltiples pasos por la máquina virtual. No vamos a examinar los pasos específicos que la máquina virtual realiza -es suficiente para saber que la única expresión de c++ se puede descomponer en tres pasos:

1. Recuperar el valor actual de c.
2. Incrementar el valor recuperado por 1.
3. Guardar el nuevo valor incrementado en c.

La expresión de c— se puede descomponer de la misma manera, excepto que el segundo paso es un decremento en lugar de un incremento. Supongamos que un hilo A invoca increment más o menos al mismo tiempo que un hilo B invoca decrement. Si el valor inicial de c es 0, sus acciones intercaladas pueden seguir esta secuencia:

1. Hilo A: Recuperar c.
2. Hilo B: Recuperar c.
3. Hilo A: Incremento de valor recuperado; resultado es 1.
4. Hilo B: Disminuir el valor recuperado; resultado es -1.
5. Hilo A: Almacena el resultado en c, c es ahora 1.
6. Hilo B: Almacena el resultado en c, c es ahora -1.

El resultado del hilo A se pierde, sobrescrito por el hilo B. Esta secuencia de ejecución es solamente una de las múltiples posibilidades. En otras circunstancias podría ser el resultado del hilo B el que se perdiera, o podría no haber ningún error en absoluto. Debido a que son impredecibles, los errores de interferencia entre hilos pueden ser difíciles de detectar y corregir.

## 32.3 Código synchronized

El lenguaje de programación Java proporciona dos métodos básicos sincronización: métodos sincronizados y declaraciones sincronizadas. Para que un método sincronizado, sólo tiene que añadir la palabra clave synchronized a su declaración:

public class SyncCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

Si count es una instancia de SyncCounter, el hacer estos métodos sincronizados tiene el efecto de que no es posible que dos invocaciones a métodos sincronizados sobre el mismo objeto se intercalen. Cuando un hilo está ejecutando un método sincronizado para un objeto, todos los demás hilos que invocan métodos sincronizados para el mismo objeto se bloquean (suspenden su ejecución), hasta que el primer hilo termina con el objeto.

32.4 Cerrojos y sincronización

La sincronización se construye alrededor de una entidad interna conocido como el cerrojo intrínseco o monitor de bloque. (La especificación de la API a menudo se refiere a esta entidad simplemente como un “monitor”).

Cada objeto tiene un cerrojo intrínseco asociado con él. Por convención, un hilo que tiene acceso exclusivo y constante a los campos de un objeto tiene que adquirir el cerrojo intrínseco del objeto antes de acceder a ellos, y luego liberarlo cuando termine con ellos. Un hilo se dice que es dueño del cerrojo intrínseco entre el tiempo que se haya adquirido el cerrojo y el que lo haya liberado. Mientras un hilo posee un cerrojo intrínseco, ningún otro hilo puede adquirir el mismo cerrojo. El otro hilo se bloquea cuando intenta adquirir el cerrojo.

32.5 Variables atómicas

El paquete java.util.concurrent.atomic define clases que soportan operaciones atómicas en variables simples. Todas las clases tienen los métodos get y set, que funcionan como lectores y escritores en variables volátiles.

Algunas de las clases más usuales son:

* AtomicInteger
* AtomicLong
* AtomicBoolean
* LongAdder
* DoubleAdder

## 33.1 Problemas en colecciones de acceso concurrente

Las colecciones no están exentas de problemas de concurrencia. Uno de los problemas más clásicos es el conocido como productor- consumidor, donde tenemos una colección compartida de tamaño finito, y varios hilos que producen datos y otros hilos que consumen datos. Dados los problemas de condición de carrera, podemos tener que los consumidores pueden correr más que los productores, y dejar la colección vacía, o que los productores vayan más rápido que los consumidores, y la colección se llene. Necesitamos un mecanismo de espera y notificación.

## 33.2 wait y notify

Son método de la clase Object, disponibles en cualquier clase Java que implementemos. Nos permiten poner en espera a un hilo, hasta que se cumpla una determinada condición. Posteriormente, podemos notificarle que ya puede continuar con su ejecución, ya que la condición bloqueante ha dejado de cumplirse. Por ejemplo: si la colección se llena, los productores deben esperar a que los consumidores tomen elementos, para poder seguir produciendo.

public synchronized String take() {

*// Esperamos a que esté disponible*

while (empty) {

try {

wait();

} catch (InterruptedException e) {}

}

*// Modificamos el estado*

empty = true;

*// Notificamos al productor que el estado ha cambiado*

notifyAll();

return message;

}

public synchronized void put(String message) {

*// Esperamos a que el mensaje haya sido leído*

while (!empty) {

try {

wait();

} catch (InterruptedException e) {}

}

*// Cambiamos el estado*

empty = false;

*// Almacenamos el mensaje*

this.message = message;

*// Notificamos al consumidor que el estado ha cambiado.*

notifyAll();

}

## 33.3 Colecciones concurrentes

Nos evitan reinventar la rueda, programando soluciones como la del productor–consumidor. Existen diferentes tipos, entre los que están:

* **BlockingQueue: estructura FIFO que bloquea si la cola se queda llena o vacía.**
* ConcurrentMap: Map con operaciones atómicas.
* ConcurrentNavigableMap: NaviagleMap con búsquedas aproximadas.

### **33.3.1 BlockingQueue**

Se trata de una estructura FIFO: first-in-first-out (primero en entrar, primero en salir). conocida en castellano como cola. Java nos ofrece diferentes implementaciones:

* ArrayBlockingQueue: debemos construirla con una capacidad inicial fija.
* LinkedBlockingQueue: su capacidad inicial es Integer.MAX\_VALUE (el mayor valor entero que podemos almacenar en un int).
* PriorityBlockingQueue: permite ordenar los elementos según prioridad.
* …

# Sección 4. Acceso a Bases de datos

## 34.1 Introducción

Java nos provee de soporte para conectar con bases de datos relacionales. Dicho soporte lo hace a través de la tecnología JDBC (Java DataBase Connectivity). Java SE 8 nos ofrece la versión 4.2 (JSR 211).

JDBC es un conjunto de interfaces, es decir, un contrato de nuestra forma de trabajar. Posteriomente necesitaremos un driver, es decir, un conjunto de clases que implementen esas interfaces, para darnos conexión con un sistema gestor de base de datos concreto.

## 34.2 Drivers

JDBC nos ofrece soporte para muchos SGBD comerciales a través de sus drivers. La lista completa la podemos encontrar en <http://www.oracle.com/technetwork/java/index-136695.html>. En estos capítulos nosotros trabajaremos con Mysql, cuyo driver lo podemos descargar en <https://dev.mysql.com/downloads/connector/j/5.1.html> .

## 34.3 Interfaces principales

* Connection: es el que permite mantener la conexión con la base de datos.
* Statement, PreparedStatement: nos permiten ejecutar consultas.
* ResultSet: juego de resultados de una consulta ejecutada.

## 34.4 Conexión con la base de datos

JDBC nos ofrece dos posibilidades para conectar con una base de datos.

* Usar la clase DriverManager, que nos permite conectar a través de una url jdbc, y soporta varias operaciones.
* Usar el interfaz DataSource, que es más avanzado, y que permite ser transparente a nuestra aplicación. Es más complejo que DriverManager.

**NUESTRA ELECCIÓN**

A lo largo de estos capitulos utilizaremos DriverManager. Aunque no es obligatorio, el uso de DataSource está orientado a proyectos Java EE. DriverManager será suficientemente potente para pequeños proyectos. Para otros más grandes, sería recomendable usar un sistema de persistencia como JPA.

### **34.4.1 URLs JDBC**

Se trata de una cadena de texto con los datos de conexión a la base de datos. Dependerá del driver (base de datos) que vayamos a utilizar.

Por ejemplo, para conectar con Mysql, una URL tipo sería:

jdbc:mysql://hostname/database

### **34.4.2 Pasos para conectar, lanzar consultas y desconectar**

Aunque las consultas las trabajaremos en profundidad en el siguiente capítulo, estos son los pasos tipo para conectar, lanzar consultas, procesar los resultados y desconectar.

1. Cargar el driver JDBC (< 4.0)
2. Establecer datos de conexión
3. Conectar obteniendo un objeto Connection.
4. Crear un objeto Statement y ejecutar consultas SQL
5. Los resultados se almacenan en un objeto ResultSet, donde se pueden consultar.
6. Cerrar los objetos (ResultSet, Statement y Connection).

## 35.1 Introducción

JDBC nos provee de varios objetos para lanzar consultas, y uno para poder procesar los resultados.

* Statement: nos permite lanzar consultas y recoger el resultado. Es la forma más básica de realizarlo.
* PreparedStatement: permite lanzar consultas a las que podemos asignarle los valores de los parámetros mediante métodos convenientes.
* CallableStatement: nos permite lanzar la ejecución de procedimientos almacenados y recoger sus resultados.
* ResultSet: es la clase que nos permite recoger el resultado de la ejecución de una consulta realizada con alguno de los interfaces anteriores.

## 35.2 Statement

Nos provee de método para ejecutar consultas en una base de datos relacional. Recibe la consulta como un String, y devuelve (generalmente) un objeto de tipo ResultSet.

Tiene, entre otros, 3 métodos para ejecutar consultas:

* execute: permite obtener más de un ResultSet
* executeQuery: permite obtener un ResultSet
* executeUpdate: Devuelve un entero que representa el número de filas afectadas. Se usa con consultas INSERT, UPDATE o DELETE

## 35.3 ResultSet

Se trata del objeto que nos permite recoger los resultados de una consulta SELECT. Tiene la estructura de un cursor, es decir, un juego de resultados y un puntero que nos permite navegar resultado resultado. Además, nos provee de métodos getXXX, por cada tipo de dato que podemos rescatar. Hay dos implementaciones por cada tipo de dato, una que recibe el índica de la columna, y otra que recibe el nombre.

Para recorrerlo de forma completa, lo normal suele ser utilizar el método next() con la siguiente estructura:

while (rs.next()) {

String coffeeName = rs.getString(1);

int supplierID = rs.getInt(2);

float price = rs.getFloat(3);

int sales = rs.getInt(4);

int total = rs.getInt(5);

*//…*

}

Tiene otros métodos para navegar. Además, en algunos casos (depende de la implementación del driver), tiene la posibilidad de trabajar en modo escritura, si bien no lo recomendamos desde aquí, ya que es preferible trabajar con otros esquemas, como el uso de PreparedStatement y el patrón de diseño DAO.

## 35.4 PreparedStatement

Se trata de una extensión de Statement que nos permite asignar los tipos de dato de los parámetros y sus valores, de forma conveniente.

Con un Statement, si quisiéramos buscar a todos los empleados con un sueldo mayor a un valor, tendríamos que hacerlo así:

float valor = 1235.34f;

String sql = "SELECT \* FROM empleados WHERE SUELDO >= " + valor;

Frente a este esquema, que puede suponer problemas, PreparedStatement nos permite indicar, allí donde vamos a insertar un valor, indicar un interrogante, y darle valor después:

float valor = 1235.34f*;*

String sql = "SELECT \* FROM empleados WHERE SUELDO >= ?"*;*

PreparedStatement ps = conn.prepareStatement(sql)*;*

ps.setFloat(1, valor)*;*

## 36.1 RowSet

Un RowSet no es más que un objeto que nos permite manejar información tabular de forma mucho más rápida y eficiente que un ResultSet. Existen 5 tipos de referencias y 2 tipos:

* **CONECTADOS**
  + JdbcRowSet
* **DESCONECTADOS**
  + CachedRowSet
  + JoinRowSet
  + FilteredRowSet
  + WebRowSet.

## 36.2 ¿En qué se diferencian de un ResultSet?

Básicamente en que pueden funcionar como *JavaBeans*, y por ende, beneficiarse del mecanismo de notificación que tienen estos (sistema de eventos y manejadores de los mismos), así como del uso de *properties*.

Por otro lado, con los ResultSet podíamos tener la posibilidad de hacer scroll (libre, no solo hacia adelante) o utilizarlos en modo escritura, pero solo en algunos tipos de drivers. RowSet nos asegura el poder realizarlo con cualquier sistema gestor de bases de datos.

## 36.3 Creación de un RowSet

Hasta Java SE 6, teníamos diferentes mecanismos para realizarlos. Java SE 7 nos incorpora uno más *limpio*, utilizando una factoría.

RowSetFactory myRowSetFactory = null;

JdbcRowSet rowSet = null;

myRowSetFactory = RowSetProvider.newFactory();

rowSet = myRowSetFactory.createJdbcRowSet();

La factoría nos permite crear cualquiera de las 5 interfaces que ofrece JDBC.

## 36.4 JdbcRowSet

Este tipo de RowSet es el único que mantiene siempre la conexión de la base de datos abierta. Esto tiene como ventaja la velocidad e inmediatez de los cambios. Como desventaja, dicho esquema de trabajo suele ser muy costoso (hablando en términos de recursos, sobre todo en el servidor de bases de datos).

Nos permite realizar operaciones como recorrer los resultados, actualizarlos, insertar nuevos, borrar, …

*//...*

myRowSetFactory = RowSetProvider.newFactory();

rowSet = myRowSetFactory.createJdbcRowSet();

rowSet.setUrl(DBConnection.JDBC\_URL);

rowSet.setUsername(DBConnection.USERNAME);

rowSet.setPassword(DBConnection.PASSWORD);

rowSet.setCommand("SELECT \* FROM empleados");

rowSet.execute();

*// Nos vamos al último registro de nuevo, y le subimos el sueldo un 10%*

rowSet.last();

rowSet.updateFloat("sueldo", rowSet.getFloat("sueldo") \* 1.1f);

rowSet.updateRow();

*// Insertamos un nuevo registro*

rowSet.moveToInsertRow();

rowSet.updateString("nombre", "Joaquín");

rowSet.updateString("apellidos", "Cañadas Casas");

rowSet.updateDate("fecha\_nacimiento", Date.valueOf(LocalDate.of(1970, 5, 18)));

rowSet.updateFloat("sueldo", 1400.0f);

rowSet.insertRow();

*//..*

## 36.5 CachedRowSet

Se trata del interfaz base de todos los RowSet de tipo desconectado (FilteredRowSet, JoinRowSet y WebRowSet heredan de este interfaz). La principal diferencia es que solo abre la conexión para rellenar el RowSet de datos, y posteriormente, para enviar las modificaciones a la base de datos.

Al inicializarlo, tenemos que indicar que (índices de) columnas forman parte de la clave (primaria). Y para aceptar los cambios, necesitamos una conexión que tenga el modo auto-commit en false.