CertiDevs

Java Standard Edition

Índice

Índice

Índice

Índice

Índice

Índice

Índice

1. Fundamentos Java

La **tecnología Java** consiste en un lenguaje de programación y también en una plataforma para el desarrollo de software portable entre sistemas operativos.

Surge a comienzos de los años 90 como una iniciativa de la empresa **Sun Microsystems** para cubrir la necesidad de creación de software para dispositivos inteligentes. En 2010 Oracle toma el relevo de su desarrollo adquiriendo la empresa Sun.

1.1. Lenguaje de programación Java

Java es un lenguaje de programación de alto nivel y multiparadigma, es decir, soporta múltiples paradigmas de programación: estructurada, orientada a objetos, funcional.

Es rápido, seguro y fiable, permitiendo a los desarrolladores crear diferentes tipos de aplicaciones software, entre los que destacan:

- **Aplicaciones standalone**: software que puede ser ejecutado sin ser instalado y que puede trabajar offline.
- **Aplicaciones web**: software cliente-servidor accedido comúnmente desde un navegador.
- **Aplicaciones móviles** y de **internet de las cosas (IoT)**: aplicaciones para dispositivos inteligentes.

1.1. Lenguaje de programación Java

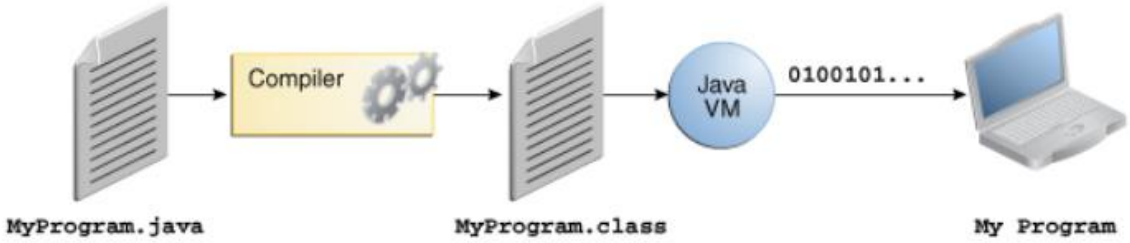
Principales características del lenguaje Java:

- **Fácil de aprender**: con respecto a otros lenguajes como C y C++.
- **Robusto**: tipado fuerte y mecanismos para gestión de excepciones.
- **Arquitectura neutral**: gracias a la Máquina Virtual Java se puede ejecutar sobre cualquier Plataforma.
- **Capacidad multihilo**: el propio lenguaje ofrece la capacidad para crear múltiples hilos para ejecutar diferentes tareas simultáneamente.

1.1. Lenguaje de programación Java

El código se escribe en ficheros con **extensión .java**, que después son compilados a código intermedio en ficheros con **extensión .class** por el compilador javac.

El código intermedio se conoce como **bytecode** y es interpretado y compilado a código nativo del sistema operativo por la Máquina Virtual de Java.



Proceso de compilación y ejecución código Java. Fuente: Oracle.

1.2. Plataforma Java

La plataforma Java es un conjunto de librerías y herramientas para el desarrollo de software.

Se compone de los siguientes elementos:

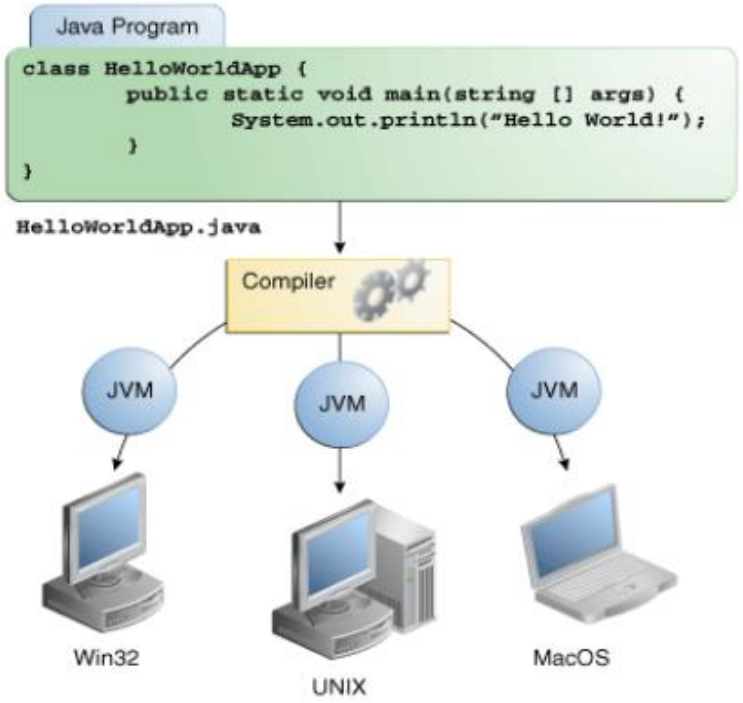
- **Bibliotecas estándar** o Java Application Programming Interface (API): un conjunto de interfaces y clases con funcionalidades comunes ya desarrolladas para ser utilizadas.

- **Máquina Virtual Java** (JVM): compuesta por los registros, la pila, el recolector de basura, etc. Permite interpretar el *bytecode* creado por el compilador de Java.

1.2. Plataforma Java

La **Máquina Virtual Java** es el componente principal de la plataforma Java.

Una de las filosofías principales de Java es la portabilidad entre sistemas. Gracias a la Máquina Virtual Java es posible escribir un código una vez y ser ejecutado en cualquier sistema operativo, transformándose al código nativo correspondiente.



Máquina Virtual Java. Fuente: Oracle.

1.3. Ediciones de la plataforma Java

Existen múltiples ediciones de la plataforma Java según el tipo dispositivo final sobre el que se ejecutará el software desarrollado, siendo las más populares en la actualidad:

- **Java SE o Java Standard Edition**: dirigido a estaciones de trabajo y ordenadores de propósito general.
- **Java EE o Java Enterprise Edition**: ahora conocida como *Jakarta Enterprise Edition*, es un conjunto de librerías añadidas sobre Java SE para incorporar funcionalidades dirigidas a servidores empresariales y entornos distribuidos en Internet, acceso a bases de datos, creación de servicios web, etc.

1.4. Versiones Java

Desde el lanzamiento de su versión 9, Java ha adoptado un ciclo de release en el que cada **6 meses** se libera una nueva versión.

Aquellas versiones establecidas como **LTS** o *Long Term Support* son las más recomendadas por ofrecer mantenimiento y actualizaciones a largo plazo.

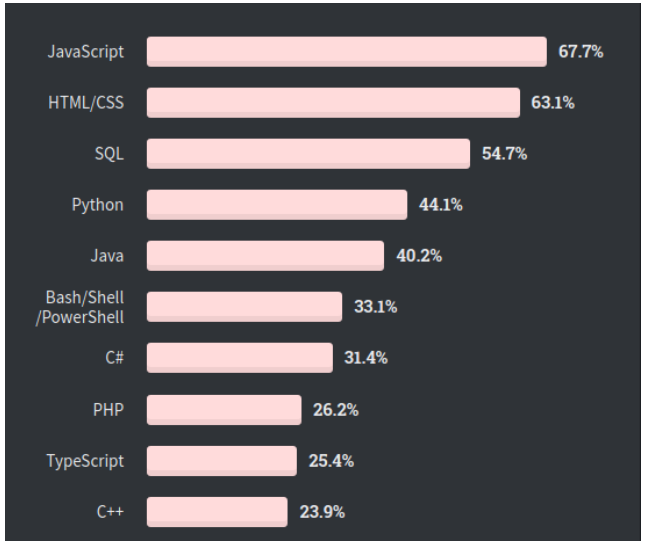
Version	Release date	End of Free Public Updates[1][4][5][6]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	**January 2019 for Oracle (commercial)** Indefinitely for Oracle (personal use) December 2030 for Zulu At least May 2026 for AdoptOpenJDK At least May 2026 for Amazon Corretto	December 2030
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	September 2027 for Zulu At least October 2024 for AdoptOpenJDK At least September 2027 for Amazon Corretto	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	September 2030 for Zulu	TBA

Legend: Old version | Older version, still maintained | **Latest version** | Future release

Versiones Java. Fuente: wikipedia.

1.5. Popularidad

Desde su **aparición en 1995** Java se ha convertido en una de las tecnologías más populares para el desarrollo de software, manteniéndose entre los primeros puestos año tras año.



Stack Overflow 2020 survey.

Jan 2021	Jan 2020	Change	Programming Language	Ratings	Change
1	2	↑	C	17.38%	+1.61%
2	1	↓	Java	11.96%	-4.93%
3	3		Python	11.72%	+2.01%
4	4		C++	7.56%	+1.99%
5	5		C#	3.95%	-1.40%
6	6		Visual Basic	3.84%	-1.44%
7	7		JavaScript	2.20%	-0.25%
8	8		PHP	1.99%	-0.41%
9	18	⇈	R	1.90%	+1.10%
10	23	⇈	Groovy	1.84%	+1.23%

Índice TIOBE 2021.

1.6. Instalación entorno desarrollo

Para desarrollar software con Java en este curso instalaremos un **entorno de desarrollo integrado** (IDE).

- IntelliJ IDEA Community Edition

1.6. Instalación entorno desarrollo

Ver vídeo en el que se instala IntelliJ IDEA Community.

2. Sintaxis

La **sintaxis** de Java se refiere al conjunto de reglas que especifican cómo se debe escribir e interpretar un código en lenguaje Java. Los elementos básicos de la sintaxis de Java son:

- Tipos de datos
- Operadores
- Sentencias
- Estructuras de control
- Estructuras de datos
- Métodos
- Clases y objetos
- Mecanismos de abstracción

2.1. Hola mundo

El siguiente código muestra la estructura de un programa básico *Hola mundo* en sintaxis Java.

```java
package com.company; // declara el paquete al que pertenece la clase

public class Main { // declara el nombre de la clase

    public static void main(String[] args) { // método punto de entrada
        System.out.println("Hola mundo"); // imprime por consola "Hola mundo"
    } // llave cierre del método

} // llave cierre de la clase
```

2.1. Hola mundo

Ver vídeo de cómo crear un nuevo proyecto java desde el entorno IntelliJ IDEA y cómo crear una clase que imprima el texto *Hola mundo*.

2.2. Tipos de datos

Los **tipos de datos** son un conjunto de valores y las operaciones que pueden hacerse con esos valores y las formas en las que pueden almacenarse.

Java es un lenguaje **fuertemente tipado**, con **tipado estático**, lo que quiere decir que cada tipo de dato está predefinido como parte de la sintaxis y se conoce en tiempo de compilación. Java tiene dos tipos de datos:

- **Tipos primitivos**: tipos numéricos, punto flotante, char, boolean.
- **Tipos de referencia**: tipos clase, interfaces, enumeraciones, arrays.

2.2.1. Tipos primitivos

Los **tipos primitivos** en Java incluyen números enteros, números de punto flotante, unidades de código UTF-16 y un tipo booleano:

- Números enteros
  - **byte**: 1 byte de espacio de almacenamiento.
  - **short**: 2 bytes de espacio de almacenamiento.
  - **int**: 4 bytes de espacio de almacenamiento.
  - **long**: 8 bytes de espacio de almacenamiento.
- Números de punto flotante (decimales):
  - **float**: 4 bytes de espacio de almacenamiento, 6-7 decimales.
  - **double**: 1 byte de espacio de almacenamiento, 15 decimales.
- **char**: representa un carácter.
- **boolean**: true o false.

2.2.1. Tipos primitivos

Creación de variables usando **tipos primitivos**.

Se utilizan las palabras reservadas correspondientes a cada tipo de dato, en minúsculas: *byte, short, int, long, float, double, boolean, char*.

```
// tipos enteros
byte number1 = 127;
short number2 = 20500;
int number3 = 30400;
long number4 = 65700;
// tipos decimales
float number5 = 5.8f;
double number6 = 89.77;
// booleanos
boolean check1 = false;
boolean check2 = true;
// carácter
char caracter = 'e';
```

2.2.2. Tipos de referencia

Los **tipos de referencia** implican la creación dinámica de un objeto. Cuando se crea un objeto Java reserva el espacio en memoria requerido para almacenar dicho objeto. Cuando se crean variables de este tipo realmente apuntan a una referencia hacia el objeto (la dirección en memoria), en lugar del objeto en sí.

```
// tipos clase
Coche fiatMultipla = new Coche("Fiat","Multipla", 1455.48);
Coche alfaRomeo = new Coche("Alfa","Romeo", 1655.48);
// tipo array
Coche[] coches = new Coche[2];
coches[0] = fiatMultipla;
coches[1] = alfaRomeo;
```

2.2.3. Conversión de tipos

La conversión de tipos se conoce como casting o casteo, permite transformar un dato de un tipo en otro. Existen 2 formas de casting para los tipos primitivos:

- **Casting de ampliación**: pasamos de un tipo de menor capacidad a uno de mayor.

```
// 1. widening casting o casting de ampliación (es automático)
// byte to short
byte number1 = 127;
short number2 = number1;
// short to int
int number3 = number2;
```

2.2.3. Conversión de tipos

- **Casting de acortamiento**: pasamos de un tipo de mayor capacidad a uno de menor.

```
// 2. Narrowing casting o casting de acortamiento (manual)
// double to float
double number7 = 9.99;
float number8 = (float) number7;
// float to long
long number9 = (long) number8;
// long to int
int number10 = (int) number9;
System.out.println(number10); // 9
```

2.3. Variables

Una **variable** es un espacio en memoria donde se almacenará un valor que podrá cambiar durante la ejecución de un programa. Las variables permiten guardar datos temporalmente para utilizarlos en diferentes operaciones.

No se pueden utilizar variables sin inicializar, los diferentes tipos de variables pueden ser:

- Local
- De instancia
- De clase
- Parámetros

2.4. Constantes

Una **constante** es un espacio en memoria donde se almacenará un valor que no podrá cambiar durante la ejecución de un programa. Las constantes permiten utilizar un valor fijo en diferentes puntos del programa.

```
public static final double IVA = 0.21;
public static final double PROFIT = 0.10;
public static final int MAX_QTY = 10;

public static void main(String[] args) {
    double price = 5.99;
    double totalPrice = price + price * IVA + price * PROFIT;
    System.out.println(totalPrice);
}
```

2.5. Operadores

Con el fin de realizar operaciones con las variables y constantes, se emplean los **operadores**. Los operadores permiten realizar desde operaciones aritméticas hasta comparaciones.

- **Aritméticos**:
  - Suma: **+**
  - Resta: **-**
  - Multiplicación: ***
  - División: **/**
  - Resto: **%**
- **Lógicos, relación y booleanos**:
  - Mayor que: **>, >=**
  - Menor que: **<, <=**
  - Igual que: **==**
  - Distinto que: **!=**
  - Operación and: **&&**
  - Operación or: **||**
  - Negación: **!**
- **Asignación**:
  - **=**
  - **+=, -=, *=, /=, %=**
  - **&=, |=**
- **Incremento**: **++**
- **Decremento**: **--**
- **Concatenación**: **+**

2.5. Operadores

Ejemplos de **operaciones** con **operadores**:

```
int resultado1 = 5 + 4; // operador suma
int resultado2 = 5 - 4; // operador resta
int resultado3 = 5 * 4; // operador multiplicación
int resultado4 = 5 / 4; // operador división
int resultado5 = 5 % 2; // operador resto
resultado5++; // operador incremento
resultado5--; // operador decremento
```

2.6. Estructuras de control

Por defecto la ejecución de las sentencias es secuencial, de arriba abajo. Las **estructuras de control** permiten modificar el flujo de ejecución permitiendo crear saltos y repeticiones. Los diferentes tipos de estructuras de control son:

- **Condicionales**:
  - if
  - if else
  - if anidados
  - switch
- **Repetición**
  - for
  - for each
  - while
  - do while
  - break y continue

2.6.1. Estructuras control condicional

Mediante estructuras **if else anidadas** podemos evaluar múltiples condiciones:

```
int edad2 = 20;
if (edad2 > 50 && edad2 < 80) {
    System.out.println("Entre 51 y 79 años");
} else if (edad2 >= 18 && edad2 <= 50) {
    System.out.println("Entre 18 y 50 años");
} else {
    System.out.println("Menor de edad");
}
```

2.6.1. Estructuras control condicional

Mediante la estructura **switch-case** podemos evaluar múltiples condiciones:

```
    switch (dayNum) {
      case 6:
          dayName = "Sábado";
          break;
      case 7:
          dayName = "Domingo";
          break;
      default:
          dayName = "Entre semana";
  }
```

2.6.2. Estructuras control repetitivo

Las estructuras de control repetitivo permiten repetir un fragmento de código.

- Estructura de control repetitivo **determinada**: *for* y *foreach*

```
// for normal
for (int i = 0; i < 10; i++) {
    System.out.println("Iteración número: " + i);
}
// foreach
int contador = 0;
for (String nombre : nombres) {
    System.out.println("Iteración " + contador + ": " + nombre);
    contador++;
}
```

2.6.2. Estructuras control repetitivo

- Estructura de control repetitivo **indeterminada**: *while*

```
boolean comprobacion = false;
int cont = 0;
while (!comprobacion) {
    if (cont == 5)
        comprobacion = true;
    cont++;
}
```

2.6.2. Estructuras control repetitivo

- Estructura de control repetitivo **indeterminada**: do-*while*

```
// do-while
int contador = 0;
do {
    System.out.println("Hola mundo " + contador);
    contador++;
} while (contador < 10);
```

2.6.2. Estructuras control repetitivo

Las palabra reservada **break** permite romper el flujo de ejecución y salir de un bucle, si se encuentra el nombre “Juan” el bucle se termina:

```
for (int i = 0; i < nombres.length; i++) {

    if (nombres[i] == "Juan" )
        break;

    System.out.println(nombres[i]);
}
```

2.6.2. Estructuras control repetitivo

Las palabra reservada **continue** permite saltar a la siguiente iteración en un bucle, si se encuentra el nombre “Juan” el bucle pasa la siguiente iteración sin ejecutar el resto de código que hay después:

```
for (int i = 0; i < nombres.length; i++) {

    if (nombres[i] == "Juan" )
        continue;

    System.out.println(nombres[i]);
}
```

2.7. Métodos

Un **método** es un bloque de código que se ejecuta cuando es invocado haciendo uso de **paréntesis ().**

Los métodos sirven para **reutilizar código**, permiten ejecutar un mismo código las veces que sea necesario desde diferentes partes de un programa sin necesidad de duplicar dicho código en cada parte donde se necesita.

Los métodos también son llamados **funciones**, pero en el contexto de la POO se les conoce como métodos cuando están asociados a instancias de clases y definen un comportamiento.

2.7.1. Signatura de un método

La **signatura de métodos** en Java hace referencia al encabezado del método, es decir, el nombre del método, tipo de valor devuelto y parámetros que recibe. Con esta información se conoce lo que hace el método, pero no cómo lo hace.

```
public static double toCelsius(double fahrenheit) { // signatura
    // código del método
}
```

2.7.2. Tipo de retorno

El **tipo de retorno** de un método se especifica antes del nombre de dicho método, pudiendo ser cualquier tipo de dato o no devolver nada, en cuyo caso será *void*.

```
public static double toCelsius(double fahrenheit) { // signatura
    return (5.0 / 9.0) * (fahrenheit - 32);
}
public void printCelsius(double fahrenheit){
    System.out.println(toCelsius(fahrenheit));
}
```

2.7.3. Nomenclatura de los métodos

Se deben elegir **nombres únicos** salvo que se esté utilizando *sobrecarga*. El nombre **main** está reservado para la función principal punto de entrada a las aplicaciones java.

Convenciones de nombres:

- **Métodos sin retorno**: realizan acciones. Se nombran con verbos: imprimirInforme(), calcularNomina()…
- **Métodos con valor de retorno**: identifican el valor devuelto. Se utiliza el nombre del valor devuelto: obtenerAltura(), daUnidadMedida(), getHeight()…
- **Métodos que devuelven booleanos**: usan el verbo ser o estar. Ejemplo: isFull(), isEmpty(), hasDiscount()…

2.7.4. Argumentos y parámetros

Cuando se invoca un método y se pasan valores se denominan argumentos, en la signatura del método se conocen como parámetros.

```
static double suma(double num1, double num2) {
    return num1 + num2;
}
```

2.7.5. Alcance

El **alcance** o *scope* define la región en la que son accesibles las variables. Cuando una variable se crea dentro de un método o un bloque de código solamente será accesible dentro del mismo, entre llaves.

```
static void saludo() {
    String nombre = “Ricardo";
    System.out.println(nombre);
}
public static void main(String[] args) {
    saludo();
    // error: la variable nombre no existe dentro de esta región
    System.out.println(nombre);
}
```

3. Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es la evolución natural de la programación estructurada. La **programación estructurada** es un paradigma que organiza los programas de forma jerárquica con estructuras de control. Cuando un programa aumenta de tamaño la programación estructurada hace difícil su evolución y mantenimiento.

La **Programación Orientada a Objetos** permite estructurar los programas acorde a objetos de la vida real donde cada objeto tiene un estado definido a través de atributos y también un comportamiento. De esta forma, los objetos encapsulan datos y métodos, proporcionando mayor flexibilidad y organización en el desarrollo de software.

3.1. Clases

Las **clases** actúan como plantillas o modelos a partir de los cuales se crean las instancias, que son los objetos.

Una clase se crea con la palabra reservada *class* y permite definir los **atributos** y **métodos** que tendrá cada objeto creado a partir de la misma. Una vez definida una clase se podrá crear objetos o instancias de la clase.

La clase Car define un coche en el cual existe:

- **Atributos**: número de ruedas, centímetros cúbicos, peso, ancho, alto, color, etc.
- **Comportamiento**: arrancar, apagar, acelerar, frenar, girar, etc.

3.1.1. Sintaxis clases

Una clase se compone de **partes bien diferenciadas**, por convención siguen este orden:

- Constantes
- Atributos
- Constructores
- Métodos

```
public class Empresa {

    public String nombreLegal;
    public String cif;
    public int numEmpleados;
    public int numProductos;
    public double facturacionAnual;
    public double costeAnual;

    public Empresa() {
    }
}
```

3.1.2. Atributos

Los **atributos** son las características comunes a todos los objetos. Son variables cada una con su tipo de dato correspondiente (*primitivo* o *de referencia*).

Los atributos definen el **estado** de los objetos en cada momento.

```
public class Empresa {

    public String nombreLegal;
    public String cif;
    public int numEmpleados;
    public int numProductos;
    public double facturacionAnual;
    public double costeAnual;
}
```

3.1.3. Constructor

Un **constructor** es un método especial que permite crear objetos o instancias de una clase.

Utilizando el mecanismo de *sobrecarga* una misma clase puede tener más de un constructor, donde cada uno tendrá diferente signatura.

```
public class Empresa {
    // ... atributos ...
    public Empresa() {}
    public Empresa(
        String nombre,
        String cif,
        int numEmpl) {
        this.nombreLegal = nombre;
        this.cif = cif;
        this.numEmpleados = numEmpl;
    }
}
```

3.1.4. Métodos

Los **métodos** definen el **comportamiento** que podrán realizar las instancias de la clase. Siguiendo el principio de una única responsabilidad por clase es posible dividir diferentes métodos complejos en pequeñas clases.

```
public class Empresa {
    // ... atributos ...
    // ... constructores...
    public double calcularBeneficio() {
        return this.facturacionAnual - this.costeAnual;
    }
}
```

3.1.5. La referencia this

La palabra reservada **this** se utiliza para referenciar al objeto actual dentro de un método o constructor.

Se utiliza comúnmente para distinguir entre los atributos de la clase y los parámetros recibidos cuando ambos tienen el mismo identificador.

```
public class Empresa {
    // ... atributos ...
    public Empresa() {}
    public Empresa(
        String nombre,
        String cif,
        int numEmpl) {
        this.nombreLegal = nombre;
        this.cif = cif;
        this.numEmpleados = numEmpl;
    }
}
```

3.1.6. El método toString()

Todas las clases en Java heredan de la clase Object, heredando así el método **toString()**, el cual proporciona una representación textual de un objeto.

```
@Override
public String toString() {
    return "Empresa{" +
    "nombreLegal='" + nombreLegal + '\'' +
    ", cif='" + cif + '\'' +
    ", numEmpleados=" + numEmpleados +
    ", numProductos=" + numProductos +
    ", costeAnual=" + costeAnual +
    '}';
}
```

3.1.7. El modificador static

La palabra reservada **static** indica que un miembro de una clase (atributo, método o clase interna) pertenece a la clase que lo contiene y no a las instancias (objetos). El siguiente método estático puede ser invocado como: *ArrayUtil.arrayContains(…);*

```
public class ArrayUtil {
    public static boolean arrayContains(int[] array, int number) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == number)
                return true;
        }
        return false;
    }
```

3.1.8. La clase String

Un **String** es una secuencia de caracteres y se utiliza para almacenar texto. En Java los strings son tratados como objetos y para ello existe la clase predefinida String. Los objetos String implementan una serie de métodos para la manipulación de texto:

```
String name = "Fernando";
int size = name.length(); // longitud
char caracterF = name.charAt(0); // charAt(int x): obtiene un caracter
char caracterN = name.charAt(3); // charAt(int x): obtiene un caracter
String fullName = name.concat(" García"); // concatenar strings
if (fullName.isEmpty())  // isEmpty devuelve true si está vacío
    System.out.println("Nombre vacío");
```

3.1.9. La clase Math

En java la clase **Math** contiene métodos que permiten realizar todo tipo de operaciones matemáticas, algunos ejemplos:

```
// máximo
System.out.println(Math.max(5.7, 4.6));
// mínimo
System.out.println(Math.min(5.7, 4.6));
// valor absoluto
System.out.println(Math.abs(-6.99));
// raíz cuadrada
System.out.println(Math.sqrt(64));
```

3.1.10. La clase Scanner

La clase **Scanner** se utiliza en el área de I/O o entrada y salida de datos. Permite a los usuarios interactuar con una aplicación sin interfaz gráfica, a través de la consola de comandos.

```
Scanner teclado = new Scanner(System.in);
int numeroLeido = teclado.nextInt();
int sumatorio = 0;
for (int i = 0; i < numeroLeido; i++) {
    if (i % 2 == 0)
        sumatorio += i;
}
System.out.println(sumatorio);
teclado.close();
```

3.1.11. Clases envoltorio

Las **clases envoltorio** o *wrapper* son clases que permiten tratar los tipos primitivos como si fueran objetos. Esto permite crear variables con objetos de este tipo y que puedan ser utilizadas por el *framework Collections* de Java.

```
// Autoboxing - Asignacion directa del tipo primitivo a clase envoltorio
Integer myInt = 5;
Double myDouble = 5.99;
Character myChar = 'A';
Long myLong = 5l;
Boolean myBoolean = true;
Short myShort = 5;
Float myFloat = 4f;
Byte myByte = 100;
```

3.2. Objetos

Un **objeto** es una instancia de una clase. La clase actúa como plantilla que permite crear objetos. Cada objeto creado es independiente del resto de objetos de la misma clase.

Un objeto puede ser utilizado como una estructura de datos que almacena atributos y permite ejecutar comportamientos. Para crear un objeto utilizamos la palabra reservada **new**:

```
Coche coche1 = new Coche(); //sintaxis creación objeto
// asignación valores a sus propiedades/atributos
coche1.marca = "Ferrari";
coche1.modelo = "458 Italia";
coche1.peso = 1437.78;
```

3.2.1. Sintaxis objetos

Cuando se tienen muchos atributos, es más práctico el uso de un constructor con parámetros que permita asignarles valor a todos a la vez:

```
Coche coche1 = new Coche(); //sintaxis creación objeto
// asignación valores a sus propiedades/atributos uno por uno
coche1.marca = "Ferrari";
coche1.modelo = "458 Italia";
coche1.peso = 1437.78;

// asignación por constructor
Coche fiatMultipla = new Coche("Fiat","Multipla", 1755.48);
```

3.2.2. Agrupaciones de objetos

A menudo se trabaja con objetos utilizando estructuras de datos como arrays o listas dinámicas, de forma que se puedan procesar utilizando estructuras de control:

```
Coche[] coches = {coche1, coche2, coche3};
for (int i = 0; i < coches.length; i++)
    System.out.println(coches[i]);
```

3.3. Encapsulación

La **encapsulación** es una técnica que permite ocultar los detalles internos de los objetos. Se utiliza para evitar el acceso directo desde fuera de la clase. Se implementa mediante el modificador de visibilidad *private*.

```
public class Casa {
    private String tipo;
    private int numPlantas;
    private int numHabitaciones;
    private boolean chimenea;
    private boolean garaje;
}
```

3.3. Encapsulación

Cuando se encapsulan atributos es común crear métodos **getters** y **setters** que permitan trabajar con ellos desde fuera, de una forma controlada:

```
// ...getters and setters...
public String getTipo() {
    return tipo;
}
public void setTipo(String tipo) {
    this.tipo = tipo;
}
// ...getters and setters...
```

3.3. Encapsulación

Las **ventajas** que aporta la encapsulación son numerosas:

- **Flexibilidad**: al permitir cambiar partes de una clase sin afectar a otras clases que utilizan sus instancias.
- **Bajo acoplamiento**: dado que se interactúa a través de métodos getter y setter las clases que los utilizan no necesitan conocer su implementación.
- **Seguridad**: se controla qué se accede y qué se modifica, pudiendo añadir restricciones de seguridad.
- **Mayor control**: solo se permite realizar aquellas operaciones que son requeridas.

3.4. Herencia

La **herencia** es un mecanismo de POO que permite la reutilización de código entre clases, en lugar de duplicar atributos y métodos en aquellas clases que son parecidas.

La clase de la que se hereda se conoce como *clase base*, *superclase* o *clase padre*, mientras que las clases que heredan son las *clases derivadas*, *subclases* o *clases hijas*.

3.4.1 La palabra clave extends

La **herencia** en Java se implementa haciendo uso de la palabra reservada **extends,** la clase que extiende hereda todos los atributos y métodos de la clase padre, pudiendo definir los suyos propios también.

```
public class Programador extends Empleado {

    private int numLineasCodigoDia;
    private String lenguaje;
    private boolean backend;
}
```

3.4.2. El modificador protected

El modificador de visibilidad **protected** implica que solamente las subclases y las clases que están en el mismo paquete podrán acceder al miembro que lo posee (atributo, método o constructor).

```
public class Vehiculo {
    // atributos
    protected int numRuedas;
}
```

3.4.3. El método super()

El método **super()** permite invocar al constructor de la clase padre. Se utiliza comúnmente para reutilizar el constructor de la clase padre en el que se asignan los valores de los atributos.

```
public Mecanico(
    String nombre,
    String nif,
    long nss,
    int antiguedad) {
    super(nombre, nif, nss, antiguedad);
}
```

3.5. Sobreescritura

La **sobreescritura de métodos** (*overriding*) en Java consiste en declarar en las clases hijas aquellos métodos que ya existen la superclase con el fin de modificar su comportamiento. De esta forma cuando se llama al método sobre un objeto de la clase hija se invoca el método sobrescrito en lugar del método de la superclase.

```
@Override
public void trabajar() {
    System.out.println("Soy mecánico y estoy "+
    "arreglando coches.");
}
```

3.6. Sobrecarga

Dentro de una clase dos o más métodos pueden compartir el mismo nombre, siempre y cuando sus declaraciones de parámetros sean diferentes, a este mecanismo se le conoce como **sobrecarga**.

La **sobrecarga** permite tener múltiples versiones de un método o constructor lo que proporciona flexibilidad a la hora de ejecutar comportamientos y crear objetos.

3.6. Sobrecarga

Ejemplo de **sobrecarga** en constructores:

```
public Mecanico(
    String nombre, String nif) {
    this.nombre = nombre;
    this.nif = nif;
}
public Mecanico(
    String nombre, String nif, long nss) {
    this.nombre = nombre;
    this.nif = nif;
    this.nss = nss;
}
```

3.7. Abstracción

La **abstracción** es un mecanismo que permite ocultar detalles del funcionamiento de una clase o método. En java es posible implementar la abstracción a través de clases y métodos con el modificador *abstract*, pero también a través de interfaces con la palabra reservada *interface*.

La **arquitectura limpia** promueve el uso de tipos abstractos para obtener un bajo acoplamiento entre las clases, de forma que las clases concretas puedan ser reemplazadas sin necesidad de modificar el código que las utiliza.

3.7.1. Clases abstractas

Una **clase abstracta** no permite crear instancias de la misma. Se emplea cuando tenemos una jerarquía de herencia en la que únicamente se deberían poder instanciar las clases hijas. La clase abstracta puede tener atributos, constructores, métodos implementados y métodos abstractos que solo declaran su signatura.

```
 public abstract class Banco {
    abstract double tipoInteres();
}
```

3.7.2. Interfaces

Una **interfaz** actúa como una clase abstracta en cuando a que solo declara métodos, no los implementa.

La principal diferencia reside en que cualquier clase puede implementar más de una interfaz a la vez, pero no heredar de más de una clase al mismo tiempo.

Una clase implementa una interfaz haciendo uso de la palabra reservada *implements*, lo cual obliga a implementar todos los métodos de la interfaz proporcionándoles un cuerpo de código.

3.7.2. Interfaces

Una **interfaz** es un **contrato** que define lo que se ha de hacer, pero delega en las clases que la implementan la responsabilidad de programar esos métodos.

```
public interface CustomerDAO {

    Customer findOne(int id);

    Customer findOneByFirstName(String firstName);

    List<Customer> findAll();
}
```

3.7.2. Interfaces

La **implementación** de una interfaz se realiza mediante la palabra reservada *implements*.

```
public class CustomerDAOImpl implements CustomerDAO {

    @Autowired
    JdbcTemplate jdbc;

    @Override
    public Customer findOne(int id) {
        return jdbc.queryForObject(...
    }
}
```

3.7.2. Interfaces

Reglas en el uso de interfaces:

- No se pueden instanciar.
- Cuando se implementa una interfaz es obligatorio implementar sus métodos.
- Los métodos son por defecto abstractos y públicos.
- No tienen constructores.

Ventajas del uso de interfaces:

- Bajo acoplamiento al poder cambiar una implementación por otra en cualquier momento.
- Código más fácil de desarrollar y mantener al estar claros los métodos.
- Se reduce la complejidad.

3.8. Polimorfismo

El **polimorfismo** hace referencia a muchas formas y se produce cuando hacemos referencia a una superclase o interfaz en lugar de a su clase derivada o implementación concreta. En este ejemplo se trabaja con instancias de la clase Mecánico y Programador como si fueran instancias de la clase Empleado:

```
Empleado empleado = new Empleado(“Roberto", ...);
Empleado mecanico = new Mecanico(“David", ...);
Empleado programador = new Programador("Rigatuso", ...);

Empleado[] empleados = {empleado, mecanico, programador};
```

3.9. Enumeraciones

Las **enumeraciones** son un tipo de dato enumerado que consta de una agrupación de **constantes**. También pueden contener variables, constructores y métodos.

```
public enum CoffeeSize {
    SMALL, MEDIUM, LARGE, EXTRA_LARGE
}
public class Coffee {
    String name;
    boolean sugar;
    CoffeeSize size;
}
```

4. Gestión de excepciones

Una **excepción** es un suceso en tiempo de ejecución que puede causar que un código no realice su tarea. Una excepción provoca una interrupción del hilo de ejecución normal de un programa si no es tratada adecuadamente.

Java ofrece un **mecanismo de gestión de excepciones** con el cual se puede dar un tratamiento adecuado cuando estas ocurren, como por ejemplo:

- **Reintento**: intentar cambiar las condiciones que condujeron a la excepción y ejecutar el código de nuevo.
- **Fracaso**: limpiar el entorno, terminar la ejecución actual e informar del fallo al nivel anterior.

4.1. Tipos de excepciones

Cuando se ejecuta un código Java se pueden producir diferentes **tipos de errores** según su naturaleza:

- **Errores de compilación**: errores detectados por el compilador: errores sintácticos, errores de tipos, de inicialización, de devolución de valores, etc. El compilador los detecta y el programa no se llega a ejecutar. Son los más fáciles de detectar. Cuando el compilador advierte que se debe gestionar una determinada excepción sobre un fragmento de código entonces se trata de una excepción comprobada o *checked*.

4.1. Tipos de excepciones

Cuando se ejecuta un código Java se pueden producir diferentes **tipos de errores** según su naturaleza:

- **Errores de ejecución**: errores producidos al ejecutarse el programa por condiciones no previstas: referencias nulas, accesos a arrays fuera de los límites, errores derivados de operaciones aritméticas, etc. Un programa es robusto cuando controla este tipo de errores. Son más difíciles de detectar que los anteriores. También se les conoce como excepciones *unchecked* o no comprobadas, el compilador no avisa aunque no se gestionen.

4.1. Tipos de excepciones

Cuando se ejecuta un código Java se pueden producir diferentes **tipos de errores** según su naturaleza:

- **Errores de diseño o lógicos**: errores que hacen que el programa no realice adecuadamente su tarea. Un programa que tiene estos errores no produce ningún error de ejecución pero no hace lo que debe: salidas incorrectas, fallos inesperados, defectos aleatorios, errores ante determinadas casuísticas complejas. Son los más difíciles de detectar.

4.2. Excepciones en POO

En POO las excepciones son objetos creados en el momento en que se produce la excepción.

En Java la clase principal de la cual heredan todas las demás excepciones es **Throwable**. A partir de ella nacen dos ramas: **Error** y **Exception**. Error representa errores de la JVM, desbordamientos de buffer, etc, el programador no debería intentar controlarlos. Exception representa aquellas excepciones que el programador sí gestionará dentro del código.

4.3. Gestión de excepciones

Los mecanismos de gestión de excepciones en Java se emplean mediante el uso de palabras reservadas como: *try*, *catch*, *finally*, *throw*, *throws:*

- **try catch**: permite envolver el código susceptible de lanzar una excepción en un bloque try catch de forma que la excepción pueda ser capturada y gestionada.
- **finally**: si se añade envuelve un bloque de código que se ejecutará siempre, se utiliza para limpiar el entorno y cerrar recursos como conexiones a base de datos.
- **throw**: palabra reservada que permite lanzar una excepción.
- **throws**: se utiliza en la signatura de un método para indicar que no gestiona la excepción y debe gestionarla el nivel anterior que invoca al método.

4.3.1. Try catch

El bloque **try catch** permite capturar excepciones

```
Scanner teclado = new Scanner(System.in);
System.out.println("Introduce un número");
try {
    int number = teclado.nextInt();
    System.out.println("Numero leído: " + number);
} catch (InputMismatchException e) {
    e.printStackTrace();
}
```

4.3.2. Finally

El bloque **finally** es opcional y se utiliza para cerrar recursos:

```
Scanner teclado = new Scanner(System.in);
System.out.println("Introduce un número");
try {
    int number = teclado.nextInt();
    System.out.println("Numero leído: " + number);
} catch (InputMismatchException e) {
    e.printStackTrace();
} finally {
    teclado.close();// Cerrar recursos
}
```

4.3.3. Throw y throws

La palabra **throw** permite lanzar una excepción, la palabra **throws** se añade a la signatura de un método para indicar a quien lo invoca que debe gestionar la excepción:

```
private static void leerNombres() throws NameFormatException {
    Scanner teclado = new Scanner(System.in);
    while (teclado.hasNext()) {
        String nombre = teclado.nextLine();
        if (nombre.length() < 8) {
            teclado.close();
            throw new NameFormatException("Obligatorio 8 caracteres");
        }
    }
}
```

4.4. Excepciones propias

El programador puede definir sus **propias excepciones** heredando de una clase de excepción ya predefinida.

```
public class NameFormatException extends Exception {
    private String mensaje;

    public NameFormatException(String mensaje) {
        this.mensaje = mensaje;
    }
    public String getMensaje() {
        return mensaje;
    }
}
```

5. Colecciones

Una **colección** se conoce en Java como una agrupación de objetos representados como una sola unidad, otro objeto.

Se trata de un objeto que implementa una estructura de datos que permite almacenar, manipular, recuperar y comunicar un conjunto de objetos.

Module java.base

Package java.util

Contains the collections framework, some internationalization support classes, a service
miscellaneous utility classes. This package also contains legacy collection classes and leg

Java Collections Framework

For an overview, API outline, and design rationale, please see:

- **Collections Framework Documentation**

For a tutorial and programming guide with examples of use of the collections framework,

- **Collections Framework Tutorial**

Since:

1.0

5. Colecciones

El framework Collections o marco de colecciones es una arquitectura unificada para representar y manipular colecciones en Java. Incluye:

- **Interfaces**: tipos de datos abstractos, permiten manipular una colección independientemente de su implementación, permitiendo el polimorfismo.
- **Implementaciones**: implementaciones concretas de los tipos abstractos, representan estructuras de datos específicas.
- **Algoritmos**: fragmentos de código reutilizables entre las diferentes implementaciones concretas para realizar operaciones como por ejemplo búsqueda y ordenación.

5. Colecciones

Dentro del **framework Collections** se encuentran las siguientes interfaces más utilizadas: *Collection*, *Set*, *SortedSet*, *List*, *Queue*, *Map*, *SortedMap*. Algunas de las más comunes son:

- La interfaz **List** y la implementación **ArrayList**: representa una lista dinámica.
- La interfaz **Map** y la implementación **HashMap**: un mapa representa un objeto que puede mapear claves únicas a determinados valores.

Nota: es necesario importar del paquete *java.util* las interfaces y clases a utilizar.

5.1. Listas dinámicas: ArrayList

Un **ArrayList** o lista dinámica es un array en el que podemos modificar la longitud, pudiendo añadir o quitar elementos de forma dinámica.

```
ArrayList<String> cars = new ArrayList<>(); // crear ArrayList
cars.add("Volvo"); // Añadir datos al array con el metodo add
cars.add("BMW");
System.out.println(cars.get(0)); // obtener un elemento por su indice
System.out.println(cars.size()); // Obtener la longitud del array
for (String car : cars)  // iterar el array
    System.out.println(car);
```

5.2. Mapas: HashMap

Un **mapa** es una estructura de datos que permite almacenar pares de clave valor (key/value). Existe múltiples formas de iterar un mapa mediante estructuras de control. El siguiente código muestra cómo crear un mapa e insertar valores:

```
HashMap<String, String> personas = new HashMap<>(); // crear mapa
personas.put("David", "García"); // añadir par clave-valor
personas.put("Sergio", "Fernández"); // añadir par clave-valor
String apellidoDavid = personas.get("David"); // obtener un valor
int personasNum = personas.size(); // obtener el número de elementos
```

5.2. Mapas: HashMap

Formas de iterar un mapa:

```
        // 1. iterar sobre las claves
for (String nombre : personas.keySet())
    System.out.println("Nombre: " + nombre);

// 2. iterar sobre los valores
for (String apellido : personas.values())
    System.out.println("Apellido: " + apellido);

// 3. iterar sobre ambos a la vez con un entry
for (Map.Entry<String, String> persona : personas.entrySet())
    System.out.println(persona.getKey() + " " + persona.getValue());
```

Java Standard Edition

Write Once, Run Everywhere

CertiDevs