

# Módulo 3B

## Programación



## ÍNDICE DE CONTENIDOS

<b>6. PROGRAMACIÓN ORIENTADA A OBJETOS (POO). FUNDAMENTOS.....</b>	<b>5</b>
6.1. Introducción a la programación orientada a objetos .....	13
6.2. Definición de objetos y características .....	24
6.3. Tablas de tipos primitivos delante de tablas de objetos.....	26
6.4. Utilización de métodos .....	27
6.5. Utilización de propiedades.....	27
6.6. Utilización de métodos estáticos.....	29
6.7. Constructores .....	29
6.8. Memoria: gestión dinámica delante de gestión estática; posibilidad del lenguaje .....	31
6.9. Destrucción de objetos y liberación de memoria .....	31
<b>7. DESARROLLO DE PROGRAMAS ORGANIZADOS EN CLASES .....</b>	<b>33</b>
7.1. Concepto de clase. Estructura y componentes.....	33
7.2. Creación de atributos .....	37
7.3. Creación de métodos .....	38
7.4. Sobrecarga de métodos .....	41
7.5. Creación de constructores .....	42
7.6. Creación de destructores y/o métodos de finalización .....	43
7.7. Uso de clases y objetos. Visibilidad .....	43
7.8. Conjuntos y librerías de clases .....	45
<b>8. UTILIZACIÓN AVANZADA DE CLASES EN EL DISEÑO DE APLICACIONES.....</b>	<b>59</b>
8.1. Composición de clases .....	59
8.2. Herencia.....	62
8.3. Jerarquía de clases: superclases y subclases .....	65
8.4. Clases y métodos abstractos y finales .....	65
8.5. Sobrescritura de métodos (Overriding) .....	69

8.6. Herencia y constructores/destructores/métodos de finalización .....	71
8.7. Interfaces .....	71
<b>9. APLICACIÓN DE ESTRUCTURAS DE ALMACENAMIENTO EN LA PROGRAMACIÓN ORIENTADA A OBJETOS .....</b>	<b>74</b>
9.1. Estructuras de datos avanzadas .....	74
9.2. Creación de arrays .....	76
9.3. Arrays multidimensionales .....	81
9.4. Cadena de caracteres. <i>String</i> .....	84
9.5. Colecciones e Iteradores .....	86
<b>10. CONTROL DE EXCEPCIONES .....</b>	<b>117</b>
10.1. Captura de excepciones .....	118
10.2. Captura frente a delegación .....	120
10.3. Lanzamiento de excepciones.....	121
10.4. Excepciones y herencia .....	123
<b>11. LECTURA Y ESCRITURA DE INFORMACIÓN .....</b>	<b>127</b>
11.1. Clases relativas de flujos. Entrada/salida.....	127
11.2. Tipos de flujos. Flujos de byte y de caracteres .....	129
11.3. Ficheros de datos. Registros.....	142
11.4. Gestión de ficheros: modos de acceso, lectura/escritura, uso, creación y eliminación.....	143
<b>12. INTERFACES GRÁFICAS DE USUARIO.....</b>	<b>150</b>
12.1. Creación y uso de interfaces gráficas de usuarios simples.....	150
12.2. Paquetes de clases para el diseño de interfaces .....	157
12.3. Acontecimientos (eventos). Creación y propiedades .....	160
<b>13. DISEÑO DE PROGRAMAS CON LENGUAJES DE POO PARA GESTIONAR BASES DE DATOS RELACIONALES .....</b>	<b>166</b>
13.1. Establecimiento de conexiones .....	167
13.2. Recuperación y manipulación de información .....	170

<b>14. DISEÑO DE PROGRAMAS CON LENGUAJES DE POO PARA GESTIONAR BASES DE DATOS OBJETO-RELACIONALES .....</b>	<b>173</b>
14.1. Establecimiento de conexiones .....	173
14.2. Recuperación y manipulación de la información.....	174
<b>15. DISEÑO DE PROGRAMAS CON LENGUAJES DE POO PARA GESTIONAR LAS BASES DE DATOS ORIENTADAS A OBJETOS (BBDDOO).....</b>	<b>176</b>
15.1. Introducción a Bases de Datos Orientadas a Objetos (BBDDOO) .....	176
15.2. Características de las Bases de Datos Orientadas a Objetos (BBDDOO) .....	176
15.3. Creación de Bases de Datos Orientadas a Objetos (BBDDOO) .....	177
15.4. Mecanismos de consulta.....	187
15.5. Tipos de datos objeto. Atributos y métodos .....	188
15.6. Tipos de datos colección .....	189
15.7. Modelo de Datos Orientado a Objetos .....	189
15.8. Relaciones .....	190
15.9. Integridad de las relaciones .....	191
15.10. UML .....	191
15.11. El modelo estándar ODMG.....	193
15.12. Prototipos y productos comerciales del Sistema Gestor de Bases de Datos Orientadas a Objetos (SGBDOO) .....	194
<b>BIBLIOGRAFÍA .....</b>	<b>195</b>
<b>WEBGRAFÍA.....</b>	<b>195</b>

## 6. Programación Orientada a Objetos (POO). Fundamentos

Los elementos que componen un programa son siempre similares. En la mayoría de programas encontramos variables, constantes, sentencias alternativas, repetitivas, etc. La principal diferencia la podemos apreciar entre las distintas palabras reservadas y en cómo se van a definir estas en un lenguaje de programación específico.

### Lenguaje de programación Java

- **Es un lenguaje compilado e interpretado.** El código de Java que creamos se debe compilar para cada sistema operativo en el que deseemos utilizar nuestro aplicativo. Una vez compilado, obtenemos un código que se denomina *bytecode*, interpretable a través de una máquina virtual llamada JRE (*Java Environment Runtime*). Esta máquina está escrita en el código nativo de la plataforma en la cual se ejecuta el programa y se basa en aquellos servicios que ofrece el sistema operativo, que permitirán atender las solicitudes que necesite dicho programa.
- **Es un lenguaje multiplataforma.** El compilador de Java produce un código binario de tipo universal, es decir, se puede ejecutar en cualquier tipo de máquina virtual que admita la versión utilizada.

Java es un tipo de lenguaje denominado *write once* (escribir una sola vez) y *run anywhere* (ejecutar en cualquier parte).

- **Es un lenguaje orientado a objetos.** El lenguaje Java es uno de los que más se acerca al concepto de una programación orientada a objetos. Los principales módulos de programación son las clases. Además, no permite que existan funciones independientes. Cualquier variable o método que se utilice en Java tiene que pertenecer a una clase.

- **Posee una gran biblioteca de librerías.** El lenguaje Java cuenta con una gran colección de clases agrupadas en diferentes directorios. Estas clases sirven al usuario para realizar una tarea determinada sin necesidad de tenerla que implementar.

### ¿Como podemos crear un nuevo proyecto en Java?

Para programar en Java existen multitud de IDE (Entorno Intregado de Desarrollo), cada uno de ellos con su interfaz y sus características propias, aunque el resultado final será independiente del IDE utilizado.

Los más conocidos actualmente son Eclipse, IntelliJ Idea Community y NetBeans.

#### Eclipse

<https://www.eclipse.org/downloads/>

#### IntelliJ Idea

<https://www.jetbrains.com/es-es/idea/download>

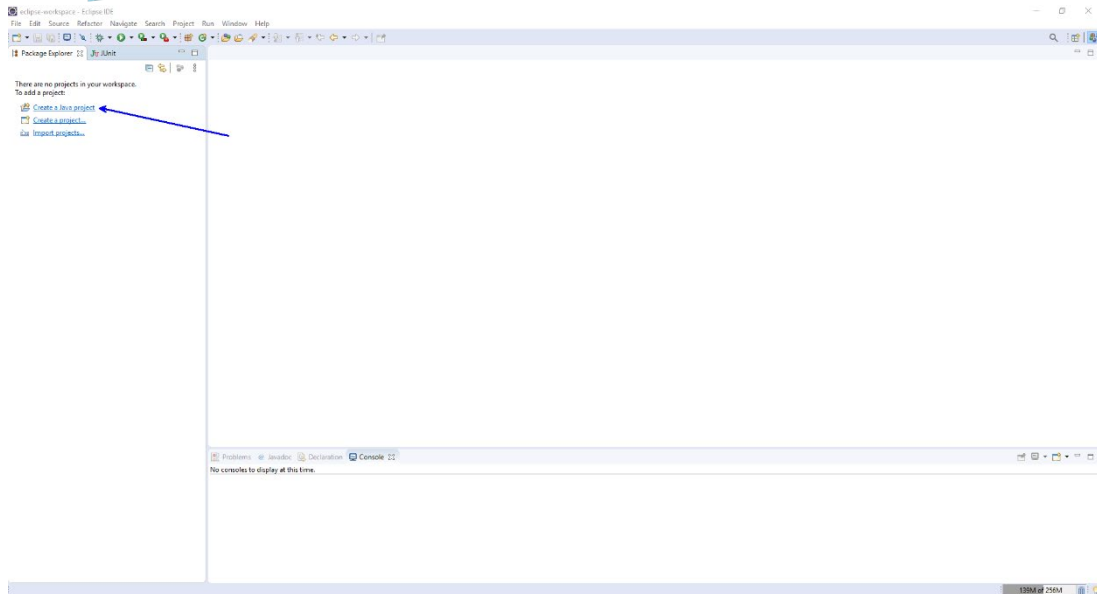
#### NetBeans

<https://netbeans.apache.org/download/index.html>

Además del IDE, necesitaremos unas herramientas de desarrollo llamadas JDK (Java Development Kit), las cuales nos proporcionarán todo lo necesario para escribir y compilar nuestros programas en Java. Hay que tener en cuenta que el código de Java solo se escribirá una vez, pero habrá que compilarlo para cada sistema operativo utilizado.

Una vez instalado el JDK y elegido el IDE, podremos crear nuestro proyecto.

En primer lugar, abriremos el IDE. En este caso emplearemos Eclipse y clicaremos en el menú que indica *Create a Java project*. También podemos dirigirnos a *File > New > Java Project*.



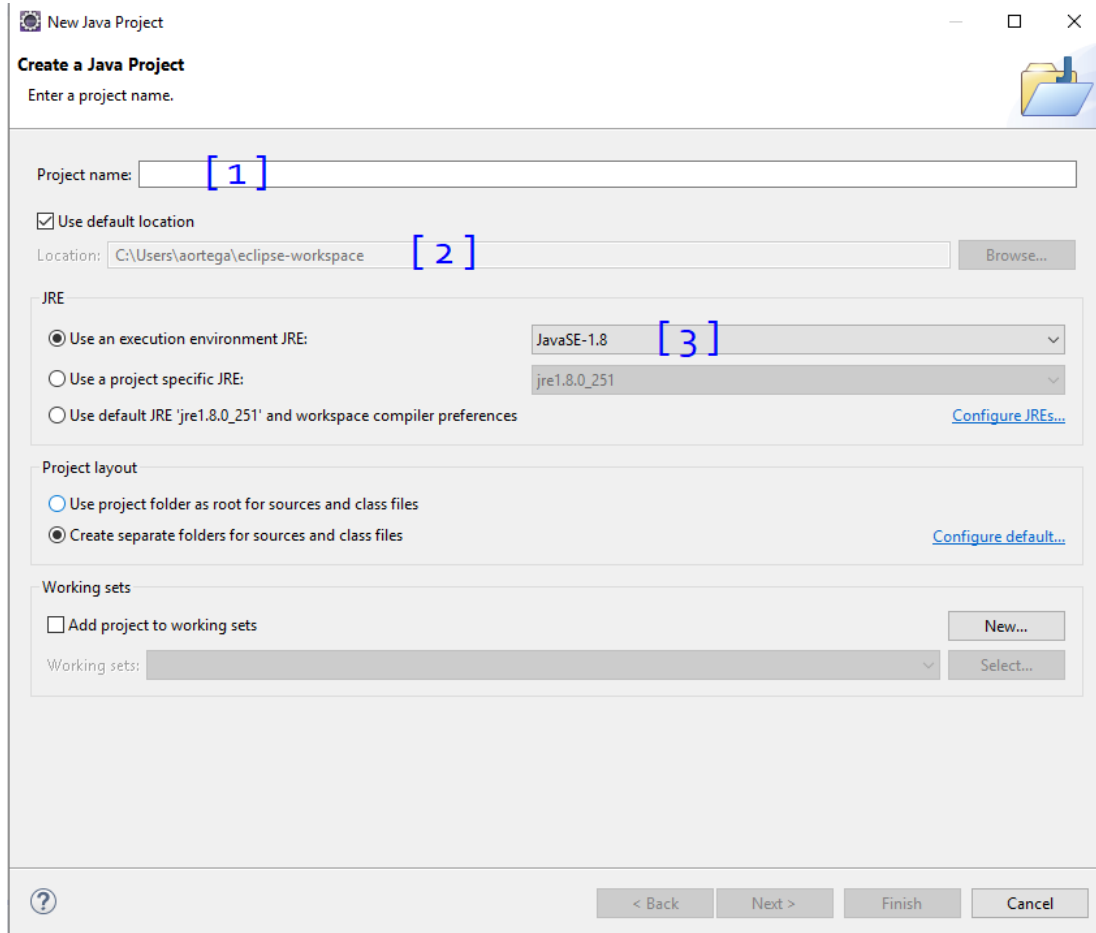
## JDK

Visita la página oficial de Oracle para descargarlo:

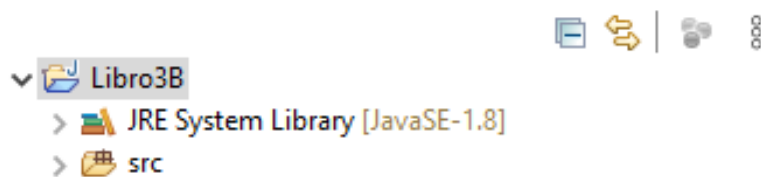
<https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html>

A continuación, hay que establecer unos parámetros para crear correctamente un proyecto:

1. Nombre del proyecto deseado
2. Ubicación dónde se guardará el proyecto
3. Versión del JDK que se va a utilizar en el proyecto



Una vez creado el proyecto con un nombre específico, como por ejemplo “Libro3B”, encontraremos una estructura como la de la siguiente imagen:



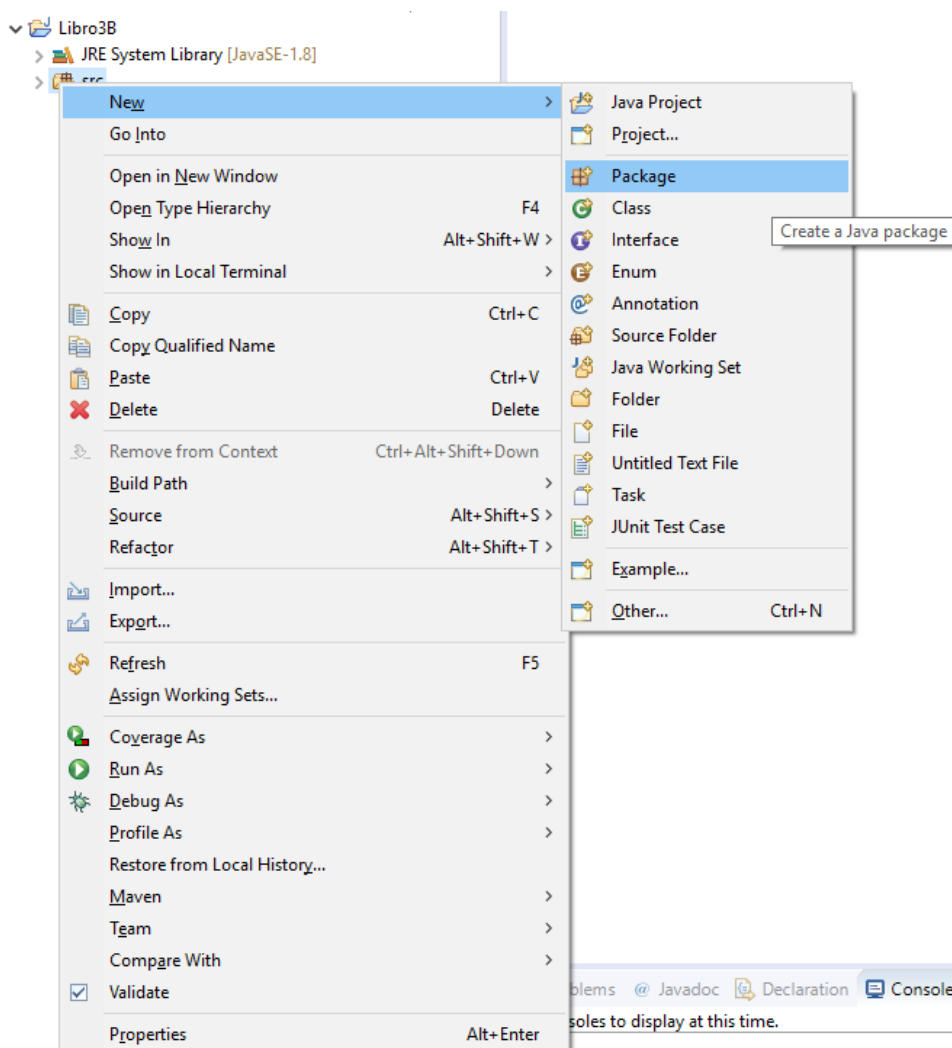
Dentro de la carpeta src, tendremos todos los *packages* y clases de Java que forman nuestra aplicación. Los *packages* corresponderían a los directorios, aunque estos se pueden definir como unos contenedores de clases donde se agruparán aquellas con un mismo propósito. Por otro lado, las clases se corresponderían a los ficheros de código que se tienen que compilar. Estos últimos tienen extensión “.java”.

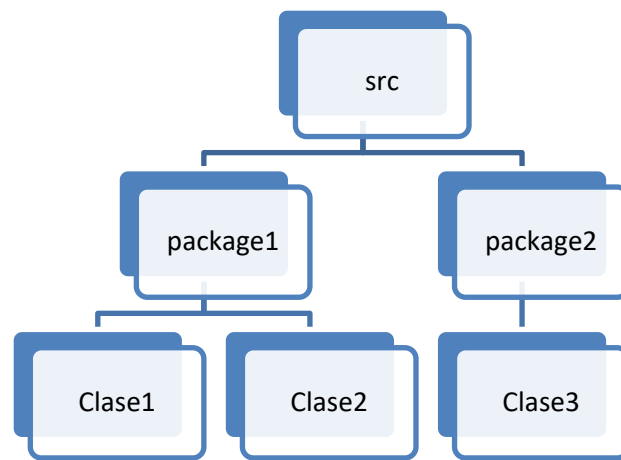


Los ficheros con la extensión “.java” serán los que posteriormente se compilen en código bytecode. El resultado de esta compilación generará unos ficheros con extensión “.class”.

Dependiendo del IDE utilizado, tener un primer package será obligatorio. Por ejemplo, si en Eclipse no creamos ningún package de forma manual, antes de crear una clase se creará un primer package virtual llamado *default package*. Decimos virtual porque ese package por defecto no se creará de forma física sobre nuestro directorio, mientras que para el resto de packages, se creará una carpeta en el directorio del proyecto.

Para crear un package debemos hacer clic con el botón izquierdo sobre la carpeta src y, a continuación, dirigirnos a *new>package*. A este únicamente deberemos ponerle un nombre. Hay que tener en cuenta que un package, a su vez, puede contener otros package, aunque siempre seguirán una estructura arborea.





Para crear una clase daremos los mismos pasos que al crear un package, pero en esta ocasión seleccionaremos la opción *Class*.

**New Java Class**

Java Class  
Create a new Java class.

Source folder: Libro3B/src [1] Browse...

Package: [2] (default) Browse...

☐ Enclosing type: Browse...

Name: [3]

Modifiers: ☒ public ☐ package ☐ private ☐ protected  
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?  
☐ public static void main(String[] args)  
☐ Constructors from superclass  
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))  
☐ Generate comments

Finish Cancel

### Puntos importantes en la creación de una clase:

1. La carpeta que contendrá todos los ficheros de programación, por defecto es `src`.
2. Package en el que se creará la clase.
3. Nombre de la clase.

### Reglas generales sobre el nombre de las clases:

- Debe empezar por una letra mayúscula.
- Debe estar en singular.
- Debe ser descriptivo, con su misión a realizar.
- Utilizan la nomenclatura *CamelCase*, que establece cómo se van a crear palabras compuestas. En el caso de *CamelCase* nos indica que la palabra empezará en mayúscula. A continuación, nos indica que cada una de las diferentes palabras utilizadas se unirán mediante una mayúscula y no con un espacio en blanco o un guion bajo. Por ejemplo, la *CamelCase* “Asiento Avión”, se escribirá “AsientoAvion”.
- No se utilizan caracteres especiales, como las letras ñ o ç y acentos. Y únicamente los símbolos del lenguaje inglés.

Finalmente, debemos indicar que en todo proyecto Java debe existir, como mínimo, una clase de entrada. Esta clase de entrada será la primera en ejecutarse, recibirá los parámetros iniciales e iniciará el flujo del programa a seguir.

La clase que permite la entrada en Java puede tener cualquier nombre, pero contendrá una función llamada *main*. Esta será pública, estática, recibirá un *array* de *Strings* por parámetro y no devolverá ningún dato. En cualquier caso, abundaremos en esta cuestión más adelante.

En la siguiente imagen podemos observar la función de entrada en una clase llamada `Main.java`:

```
public class Main {
    //Función de entrada a nuestro programa
    public static void main(String[] args) {

    }
}
```

Veamos un resumen a modo de ejemplo del código inicial de una clase:

```

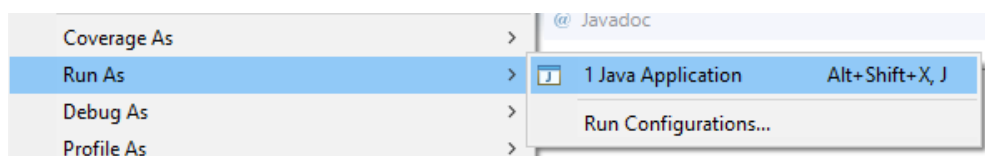
CÓDIGO

package ejemplo;

/**
 * @author ilerna
 */
public class Ejemplo {
    public static void main (String [] args) {
        /** Código de las aplicaciones */
        // Esto es una impresión en consola:
        System.out.println("Hola Mundo!");
    }
}

```

- **package:** el package será la carpeta que contenga los archivos .java y .class que utilizaremos para la realización y ejecución de nuestros proyectos-
- **public class NombreClase { }:** entre llaves incluiremos aquellos atributos y métodos que necesitemos en nuestro código para, posteriormente, realizar las llamadas correspondientes en el método “main()”.
- **Método Main():** cuyo formato de cabecera lo escribiremos de la siguiente forma: **public static void main (String[] args)**. Esta función la utilizaremos para ejecutar una determinada clase, ya sea estática o pública. No devolverá ningún valor (void) y va a utilizar como parámetros un array de cadenas de caracteres. Se necesitan para que el usuario pueda introducir valores a la hora de iniciar el programa. Esta función nos permitirá ejecutar nuestra clase haciendo clic sobre ella con el botón derecho y seleccionando **Run As > Java Application**.



- **Comentarios:** los comentarios sirven para hacer un seguimiento de nuestro programa. Pensemos que, si un código va acompañado de comentarios, facilitará mucho la tarea a la hora de trabajar con él. Escribiremos los caracteres `“//”` para comentarios de una única línea y `“/*”` o `“*/”` para los que contengan más de una

## 6.1. Introducción a la programación orientada a objetos

“A medida que se van desarrollando los lenguajes, se va desarrollando también la posibilidad de resolver problemas cada vez más complejos. En la evolución de cada lenguaje, llega un momento en el que los programadores comienzan a tener dificultades a la hora de manejar programas que sean de un cierto tamaño y sofisticación”. (Bruce Eckel, *“Aplique C++”*, p. 5 Ed. McGraw- Hill).

La Programación Orientada a Objetos (POO) pretende acercarse más a la realidad, de manera que los elementos de un programa se puedan ajustar, en la medida de lo posible, a los diferentes elementos de la vida cotidiana.

La POO ofrece la posibilidad de crear diferentes softwares a partir de pequeños bloques, que pueden ser reutilizables.

Sus propiedades más importantes las podemos dividir en:

- **Abstracción:** cuando utilizamos la programación orientada a objetos nos basamos, principalmente, en qué hace el objeto y para qué ha sido creado, aislando y abstrayendo otros factores, como la implementación del programa en cuestión.
- **Encapsulamiento:** en este apartado se pretende ocultar los datos de los objetos de cara al mundo exterior, de modo que, del objeto solo se conozca su esencia y qué es lo que pretendemos hacer con él.
- **Modularidad:** que la programación orientada a objetos es modular quiere decir que vamos a tener una serie de objetos que van a ser independientes los unos de los otros y podrán ser reutilizados.
- **Jerarquía:** nos referimos a que vamos a tener una serie de objetos que desciendan de otros.
- **Polimorfismo:** nos va a permitir el envío de mensajes iguales a diferentes tipos de objetos. Solo se debe conocer la forma en la que debemos contestar a estos mensajes.

A continuación, vamos a ver las características principales que diferencian el lenguaje Java de los demás:

- **Independencia de la plataforma.** Podemos desarrollar diferentes aplicaciones que pueden ser ejecutadas bajo cualquier tipo de *hardware* o sistema operativo. Inicialmente, se va a generar un bytecode que, después, va a ser traducido por la máquina en el lugar en el que se ejecute el programa.
- **Fácil de aprender.** Java es el lenguaje de programación más utilizado hoy en día en los entornos educativos, ya que viene provisto de unas herramientas que permiten configurarlo en un entorno cómodo y fácil de manejar.
- **Basado en estándares.** A través del proceso Java Community se pueden ir definiendo nuevas versiones y características.

#### Java Community

Visita la página de la comunidad a través del siguiente enlace:

<http://www.icp.org/en/home/index>

- **Se utiliza a nivel mundial.** Java es una plataforma libre que dispone de un gran número de desarrolladores. Estos cuentan, entre otras cosas, con una gran cantidad de información, librerías y herramientas.
- **Entornos *runtime* consistentes.** Su función es intermediar entre el sistema operativo y Java. Está formado por la máquina virtual de Java, las bibliotecas y otros elementos, también necesarios para poder ejecutar la aplicación deseada.
- **Optimizado para controlar dispositivos.** Ofrece un soporte para aquellos dispositivos integrados.
- **Recolector de basura.** Su función principal es eliminar de forma automática aquellos objetos que no hacen referencia a ningún espacio determinado de memoria.

### 6.1.1. Tipos de datos primitivos

Cuando hablamos de tipos de datos primitivos, nos referimos a los que denotan magnitudes de tipo numérico, carácter o lógico. Se caracterizan, principalmente, por su eficiencia, ya que consumen menos cantidad de memoria y permiten la realización de los cálculos correspondientes en el menor espacio de tiempo posible.

Los tipos de datos primitivos tienen una serie de valores que se pueden almacenar de dos formas diferentes: en variables o en constantes.

#### Variables

Las variables en Java son similares a C#. La diferencia está en los tipos de datos que se usan en Java: *int*, *float*, *char*, *boolean*, etc. Los veremos a continuación.

En el siguiente ejemplo vemos cómo declarar un mensaje en una variable llamada "txt" y mostrarlo en pantalla:

#### CÓDIGO

```
public class Ejemplo {
    public static void main (String[] args) {
        String txt;
        txt = 'Hola mundo';
        System.out.println(txt);
    }
}
```

- Todas las variables que se utilicen en un programa deben ser declaradas.
- El valor de una variable declarada previamente, sin asignarle valor, es un valor desconocido. No suponemos que su valor es 0.
- Siempre que declaremos variables, deberemos asignarles un valor.

## Constantes

Las constantes son similares a las variables, con la diferencia de que en las constantes se utiliza la palabra reservada **final**. El uso de esta palabra provoca que la constante no pueda ser modificada. En este ejemplo podemos ver cómo lanzar un mensaje por consola usando una constante:

### CÓDIGO

```
public class Ejemplo {
    public static void main (String [] args) {
        final String txt = "Hola Mundo!";
        System.out.println(txt);
    }
}
```

- **Tipos numéricos enteros**

Para usar tipos numéricos enteros podemos usar: **short**, **int** o **long**. Debemos tener en cuenta que las variables ocupan un espacio en la memoria. Y eso es lo que diferencia las unas de las otras.

En la siguiente tabla, podemos ver el tamaño que ocupa cada tipo:

Nombre	Tamaño (bits)
short	16
int	32
long	64



Por ejemplo, supongamos que tenemos que escribir en un array (los estudiaremos más adelante) las edades de 50 alumnos. Podríamos hacerlo de la siguiente forma:

#### CÓDIGO

```
short edades_short [50]; // Ocupa 50 * 16 = 800 bytes
int edades_int [50]; // Ocupa 50 * 32 = 1600 bytes
```

En este ejemplo que acabamos de ver parece que es preferible utilizar el tipo short, ya que necesita menos espacio en la memoria para almacenar información.

- **Expresiones**

Podemos procesar la información mediante expresiones. Estas son diferentes combinaciones de valores, variables y operadores. A modo de ejemplo, podría ser:

#### CÓDIGO

```
int num1, num2, num3; // Definimos 3 variables de tipo entero
num1 = 2; // Expresión de asignación
num2 = 3; // Expresión de asignación
num3 = num1 + num2; // Expresión compleja
```

Las **expresiones simples** son aquellas en las que interviene un único operador. En este ejemplo, el único operador de asignación que interviene es “=”.

Sin embargo, en **expresiones complejas**, puede aparecer más de un operador. Se pueden evaluar las subexpresiones de forma separada y utilizar sus resultados para calcular un resultado final.

Los diferentes operadores numéricos disponibles en Java pueden ser:

Operador	Significado
+	Suma (enteros y reales)
-	Resta (enteros y reales)
*	Multiplica (enteros y reales)
/	Divide (enteros y reales)
%	Módulo (enteros)

- Tipos numéricos reales**

Los **tipos reales o coma flotante** son aquellos que permiten realizar diferentes cálculos con decimales. Pueden ser **float** o **double**.

Nombre	Tamaño (bits)
float	32
double	64

Se pueden utilizar clases como *BigInteger* y *BigDecimal*, que permiten realizar cálculos (enteros o coma flotante) con precisión arbitraria.

- Operadores aritméticos para números reales**

Estos operadores son los habituales "+", "-", "\*", "/". Es decir, todos excepto el módulo "%", que no existe cuando se utilizan números reales. Vamos a ver un ejemplo en el que se utilicen estas cuatro reglas básicas con los números 2 y 3:

## CÓDIGO

```
publicclassReglas {
publicstaticvoid main (String [] args){
System.out.printf("%d\n", 2+3);
    System.out.printf("%d\n", 2-3);
    System.out.printf("%d\n", 2*3);
    System.out.printf("%d\n", 2/3);
}
}
```

Hemos realizado el programa haciendo uso de expresiones fijas, por lo que siempre vamos a obtener el mismo resultado cuando lo ejecutemos.

Por este motivo, no es muy frecuente su uso y se opta por hacerlo utilizando diferentes variables, ya que estas pueden ir tomando valores diferentes.

Veamos un ejemplo igual que el anterior, pero haciendo uso de variables:

## CÓDIGO

```
publicclassReglas2 {
publicstaticvoid main (String [] args){
int num1;
    int num2;
    num1=2;
    num2=3;
    System.out.printf("%d\n", num1+num2);
    System.out.printf("%d\n", num1-num2);
    System.out.printf("%d\n", num1*num2);
    System.out.printf("%d\n", num1/num2);
}
}
```

Veamos las siguientes palabras reservadas:

- **int** → Abreviatura de *integer*. Tipo de datos entero, adecuado a la hora de realizar cálculos.

**Declaramos** la variable num1 de tipo entero.

CÓDIGO

```
int num1;
```

**Asignamos** a la variable num1 el valor de 2.

CÓDIGO

```
num1 =2;
```

- **Tipo char**

El tipo primitivo denominado *char* se utiliza cuando tenemos que representar un carácter que sigue el formato Unicode. En Java existen varias clases para el manejo de estas cadenas de caracteres y hay métodos que nos facilitan el trabajo con cadenas.

### Tratamiento de caracteres

El tipo char se utiliza para representar un único carácter. Contempla los números del 0 al 9 y las letras, tanto mayúsculas como minúsculas. Su valor irá dentro de unas comillas simples. Veamos cómo podríamos escribirlo:

CÓDIGO

```
char dato; //Declaramos una variable de tipo char, denominada dato  
dato ='a';
```

También tenemos la opción de dar un valor a la variable tipo char en el momento de su declaración:

#### CÓDIGO

```
char dato ='a';
```

#### • Tipo boolean

Es el equivalente al tipo de dato **bool** en C#. El tipo de dato *boolean* solo almacena valores lógicos (*true* o *false*).

#### CÓDIGO

```
boolean fin; //Declaramos una variable de tipo Boolean
fin =true;
```

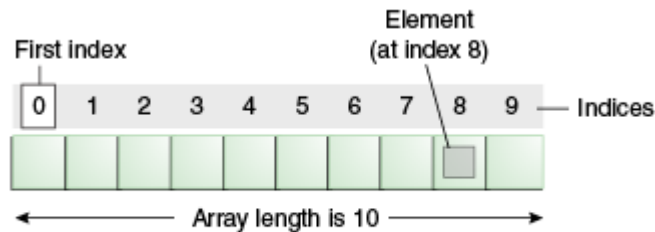
Veamos un ejemplo práctico donde, si se cumple una determinada condición, este realizará una acción dentro de una iteración:

#### CÓDIGO

```
boolean fin; //Declaramos una variable de tipo Boolean
fin =true;
int cont =0;
while(fin){
    System.out.println("Dentro del while: "+ cont);
    cont = cont +1;
    if(cont ==3){
        fin =false;
    }
}
System.out.println("Fuera del while: "+ cont);
```

- **Arrays y tablas en Java**

En Java se pueden crear tablas, matrices y estructuras de una forma bastante simple. Un array puede ser representado como en la siguiente imagen:



An array of 10 elements.

Fuente: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>

Debe cumplir con esta sintaxis:

**CÓDIGO**

```
Tipo[] lista;
lista = new Tipo[tamaño];
```

Algunos ejemplos podrían ser:

**CÓDIGO**

```
int[] vector = new int[5];
float[][] matriz = new float[2][2];
char[][][] cubo_rubik = new char[3][3][3];
```

## Reglas para utilizar arrays y tablas

- En Java, el primer índice siempre va a ser 0, no 1. Los valores van desde 0 hasta el último, n-1. Hay que tener en cuenta que la referencia no es circular y, por tanto, el valor -1 no corresponde al último elemento del array.
- Java hace una comprobación de índices, de forma que, si se quiere acceder a uno que no está, lanza una excepción llamada `IndexOutOfBoundsException`.
- Solo se puede dar valor a un array de forma completa en el momento de su declaración. Una vez que ya esté declarado, solo se pueden asignar valores al array de forma individual.
- En Java es posible hacer una copia de un array por asignación, aunque con limitaciones.

### • Cuadro de tipos primitivos

Recopilando todos los datos que se han ido detallando en los apartados anteriores, en el siguiente cuadro se visualiza cada tipo de dato primitivo con su correspondiente tamaño en bits.

Tipo primitivo	Tamaño (bits)
boolean	-
char	16
byte	8
short	16
int	32
long	64
double	64

## 6.2. Definición de objetos y características

Definimos los **objetos** como un **conjunto de datos (características o atributos) y métodos (comportamientos) que se pueden realizar**. Es fundamental tener claro que estos dos conceptos (atributos y métodos) están ligados formando una misma unidad conceptual y operacional.

Debemos señalar también que estos objetos son casos específicos de unas entidades denominadas clases, en las que se pueden definir las características que tienen en común estos objetos. Los objetos podríamos definirlos como un contenedor de datos, mientras que una clase actúa como un molde en la construcción de objetos.

A continuación, vamos a ver un ejemplo relacionado con la vida cotidiana en el que aclararemos todos estos conceptos:

- Podemos declarar un objeto “coche”.
- Sus atributos pueden ser, entre otros: color, marca, modelo.
- Y algunas de las acciones que este objeto puede realizar son, entre otras: acelerar, frenar y cambiar velocidad.

Recordemos que un objeto va a utilizar estos atributos en forma de variables y los métodos van a ser funciones que se van a encargar de realizar las diferentes acciones.

Además, el nombre de todos los atributos y métodos empezarán en minúscula, ya que el inicio en mayúscula está reservado para el nombre de la clase.

En nuestro ejemplo, tendríamos variables en las que almacenar el color, la marca y el modelo, junto con una serie de funciones que desarrollarán las acciones de acelerar, frenar y cambiar de velocidad.



## CÓDIGO

```
//Clase coche que vaa servir para crear objetos coche
public class Coche {
    private String color;
    private String marca;
    private String modelo;
    public Coche(String color, String marca, String modelo) {
        this.color = color;
        this.marca = marca;
        this.modelo = modelo;
    }
    public void acelerar() { //código del método }
    public void frenar() { //código del método }
    public void cambiar_velocidad() { //código del método }
}

//Clase donde vamos a crear objetos tipo coche
public class Garaje {
    public static void main(String[] args) {
        //Declaración de objetos con sus atributos
        Coche coche1 = new Coche("Azul", "Nissan", "Almera");
        Coche coche2 = new Coche("Negro", "Seat", "Ibiza");
        Coche coche3 = new Coche("Blanco", "Renault", "Megane");
        Coche coche4 = new Coche("Gris", "BMW", "Z3");
        Coche coche5 = new Coche("Rojo", "Ferrari", "Testa rosa");
        //Acciones que pueden realizar los objetos
        coche1.acelerar();
        coche2.frenar();
        coche3.cambiar_velocidad();
    }
}
```

### 6.3. Tablas de tipos primitivos delante de tablas de objetos

		TIPOS PRIMITIVOS			
		NOMBRE	TIPO	OCUPA	RANGO APROXIMADO
		byte	Entero	1 byte	-128 a 127
		short	Entero	2 bytes	-32768 a 32767
		int	Entero	4 bytes	2*10 <sup>9</sup>
		long	Entero	8 bytes	Muy grande
		float	Decimal simple	4 bytes	Muy grande
		double	Decimal doble	8 bytes	Muy grande
		char	Carácter simple	2 bytes	
		boolean	Valor true o false	1 byte	
TIPOS DE DATOS EN JAVA		TIPOS OBJETO			
		<b>Tipos de la biblioteca estándar de Java</b>		String (cadenas de texto) Muchos otros (p.ej. Scanner, TreeSet, ArrayList...)	
		<b>Tipos definidos por el programador/usuario</b>		Cualquiera que se nos ocurra, por ejemplo Taxi, Autobús, Tranvía...	
		<b>Arrays</b>		Serie de elementos o formación tipo vector o matriz. Lo consideraremos un objeto especial que carece de métodos	
		<b>Tipos envoltorio o wrapper</b>		Byte	
				Short	
				Integer	
				Long	
				Float	
				Double	
				Character	
				Boolean	

En el esquema anterior, podemos comprobar los distintos tipos de datos que hay en Java y todas las estructuras que podemos utilizar con estos tipos. Tanto para los tipos primitivos o definidos por el compilador, como para los tipos objetos (creados por el programador de la aplicación mediante la definición de una clase previamente) podemos definir los arrays o estructuras de tablas.

## 6.4. Utilización de métodos

Los métodos son las funciones propias que tiene una clase, capaces de acceder a todos los atributos que tenga definidos. La forma de acceder a los diferentes métodos se debe definir dentro de la clase en cuestión. Aparte de acceder a los atributos, los métodos nos ofrecerán toda aquella funcionalidad esperada de la clase, realizando las diferentes acciones a ejecutar.

- **Métodos de acceso**

La función principal de estos métodos es habilitar las tareas de lectura (llamadas **getter**) y de escritura de los atributos de la clase (llamados **setter**). Se utilizan, entre otras cosas, para reforzar la encapsulación al permitir que el usuario pueda añadir información a la clase o extraer información de ella sin que sea necesario conocer detalles más específicos de la implementación. Cualquier cambio que se realice en la implementación de los métodos no afecta a las clases clientes.

## 6.5. Utilización de propiedades

Para poder utilizar los métodos de una clase, lo primero que debemos hacer es crearnos una clase que sea el esquema, donde definamos tanto los atributos como la declaración e implementación de los métodos que deseamos utilizar. A continuación, en otra clase complementaria crearemos un objeto de la primera clase. De esta manera, podremos utilizar todas sus funciones. Para acceder a estas se deberá poner el nombre del objeto, el carácter punto "." y, finalmente, el nombre del método que se desea invocar.

A continuación, vemos un ejemplo de la creación de un objeto y la llamada a sus métodos. Los conceptos de este ejemplo se detallan más adelante.

## CÓDIGO

```
//Clase alumno que nos servirá para crear objetos tipo alumno
publicclass Alumno {
//Atributos
    private String nombre;
    private String ciclo;
//Constructor
public Alumno(String nombre, String ciclo){ ... }
//Métodos
publicvoid evaluar() { //código del método }
}

-----

//Clase Aula donde utilizaremos los objetos alumnos
publicclass Aula {
publicstaticvoid main(String[] args){
//Creación de objetos Alumno
Alumno alumno_1 =new Alumno("Antonio", "DAW");
//Sintaxis de utilización de los métodos del objeto
alumno_1.evaluar();
}
}
```

## 6.6. Utilización de métodos estáticos

Para poder declarar un atributo o método como estático debemos poner delante de este la palabra reservada *static*. ¿Pero en que consiste la palabra *static*?

Dependiendo de su utilización tendrá una función u otra:

- **Atributos:** cuando la palabra reservada *static* se utiliza en un atributo quiere decir que este atributo se ubica en una zona especial de la memoria y es idéntico para todos los objetos que creemos de la clase. Si se modifica el valor, afecta a todos los objetos que se hayan creado.
- **Métodos:** cuando se usa delante de un método, este pasará a convertirse en un método de la clase. Gracias a esta acción podremos invocar el método sin tener que crear un objeto de la clase.

## 6.7. Constructores

Los constructores son unas funciones especiales que nos permiten crear objetos de la clase en la que se encuentran.

Características de los constructores:

- Generalmente son públicos, aunque en algunas ocasiones pueden ser privados. Por ejemplo, cuando se utiliza una fábrica de objetos. Pero este es un tema más avanzado, del que hablaremos más adelante.
- Tienen el mismo nombre que la clase.
- Pueden existir diversos constructores, dependiendo de la cantidad de parámetros que reciban. Se invocará a uno u otro de forma automática al establecer los valores correspondientes.
- Si no se especifica ningún constructor, existirá uno llamado *por defecto*, que no recibe ningún parámetro. Pero hay que tener en cuenta que si se crea un constructor con parámetros, este deja de funcionar, por lo que si deseamos mantener su funcionamiento deberemos crearlo manualmente.

## CÓDIGO

```
//Clase alumno que nos servirá para crear objetos tipo alumno

publicclass Alumno {

    //Atributos

    private String nombre;

    private String ciclo;

    //Constructor por defecto

    public Alumno(){}

    //Constructor con 1 parámetro

    public Alumno(String nombre){

this.nombre=nombre;

this.ciclo="DAM";

    }

    //Constructor con 2 parámetros

    public Alumno(String nombre, String ciclo){

this.nombre=nombre;

this.ciclo=ciclo;

    }

}
```

## 6.8. Memoria: gestión dinámica delante de gestión estática; posibilidad del lenguaje

En programación existen dos tipos de almacenaje de memoria: estática y dinámica. La memoria estática es aquella en la que, una vez creada, la variable no puede modificar su tamaño. No puede incrementar ni decrementar el espacio que ocupa en memoria. Así, después de crear un array con una longitud determinada, no se puede modificar posteriormente.

Por otro lado, está la memoria dinámica, que será creada y modificada en tiempo de ejecución. En este caso nos encontramos con lo que viene a ser la creación de objetos.

En Java, la memoria estática y dinámica se divide en dos zonas independientes de la memoria:

- *Stack* (o pila): zona de la memoria donde se almacenan variables, referencias, parámetros y valores de retorno.
- *Heap*: zona de la memoria donde se almacenan objetos y variables de instancia.

## 6.9. Destrucción de objetos y liberación de memoria

En algunos lenguajes de programación, a la hora de destruir algún objeto se cuenta con unas funciones especiales que ejecutan de forma automática cuándo se debe eliminar un objeto. Se trata de una función que no devuelve ningún tipo de dato (ni siquiera void), ni recibe ningún tipo de parámetro de entrada a la función. Normalmente, los objetos dejan de existir cuando finaliza su ámbito, antes de terminar su ciclo vital.

También existe la posibilidad del conocido recolector de basura (*garbage collector*) que, cuando hay elementos referenciados, forma un mecanismo para gestionar la memoria y conseguir que estos se vayan eliminando.

En **Java no existen los destructores** como tal, por ser un tipo de lenguaje que ya se encarga de la eliminación o liberación de memoria que ocupa un objeto determinado a través del recolector de basura.

El recolector de basura en Java, antes de “barrer el objeto no usado”, llama al método *finalize()* de ese objeto. Esto significa que se ejecuta primero el método *finalize()* y después el objeto se destruye de la memoria. El método *finalize()* existe para todos los objetos en Java y se utiliza para realizar algunas operaciones finales u operaciones de limpieza en un objeto, antes de que ese objeto se elimine de la memoria.

El método *finalize()* debe tener las siguientes características:

#### CÓDIGO

```
protected void finalize() throws Throwable{
    //Cuerpo del destructor
}
```

En este método se está utilizando la cláusula *throws*, que hace referencia al lanzamiento de una excepción. Este tema se explicará más adelante en el tema 11.



## 7. Desarrollo de programas organizados en clases

### 7.1. Concepto de clase. Estructura y componentes

Las clases en Java van precedidas de la palabra **class** seguida del nombre de la clase correspondiente. Y, normalmente, vamos a utilizar el **modificador *public***, quedando de la siguiente forma:

#### CÓDIGO

```
Modificador_de_acceso class nombre_de_la_clase{
//Propiedades;
//Métodos;
}
```

El comportamiento de las clases es similar al de una estructura donde algunos campos actúan como punteros de una función, definiendo estos punteros de tal forma que poseen un parámetro específico (*this*) que va a actuar como el puntero de nuestra clase. ¿Qué quiere decir esto? En esencia, que si necesitamos acceder en cualquier momento en una clase a uno de sus atributos, podemos utilizar el apuntador *this* para indicarle qué atributo o función de la clase deseamos utilizar.

De esta forma, las funciones que señalan estos punteros (métodos) van a poder acceder a los diferentes campos de la clase (atributos).

Definimos la clase como un molde ya preparado en el que podemos fijar los componentes de un objeto: los **atributos** (variables) y las acciones que van a realizar estos atributos (**métodos**).

En POO podemos decir que *coche* es una instancia de la clase de objetos conocida como *Coche*. Todos los coches tienen algunos estados o atributos (color, marca, modelo) y algunos métodos (acelerar, frenar, cambiar velocidad) en común.

Debemos tener en cuenta que el estado de cada coche es independiente al de los demás. Es decir, podemos tener un coche negro y otro azul, ya que ambos tienen en común la variable *color*. De tal forma que podemos utilizar esta plantilla para definir todas las variables que sean necesarias y sus métodos correspondientes para los coches. Estas plantillas que usaremos para crear objetos se denominan **clases**.

La **clase** es una plantilla que define aquellas variables y métodos comunes para los objetos de un cierto tipo.

Veamos un ejemplo en el que participen todos los conceptos que estamos definiendo. Partimos de nuestra clase *Coche*, en la que debemos introducir datos que tengan sentido (elementos de la vida cotidiana). Establecemos que un coche se caracteriza, entre otras características, por:

- Tener unas ruedas determinadas
- Tener matrícula
- Tener una cantidad de puertas
- Tener un color
- Tener una marca
- Ser de un determinado modelo

Aunque, si a nuestro taller llega un Seat Ibiza de tres puertas, las características serían:

- Ruedas tipo X. Cuatro más una de repuesto
- Matrícula FNR 9774
- Tres puertas
- Negro
- Seat
- Ibiza

De esta forma, tenemos la clase *Coche* y el objeto *SeatIbiza*.

Cuando creamos una clase, definimos sus atributos y métodos:

- **Atributos:** las variables que hacen referencia a las características principales del objeto que tenemos.
- **Métodos:** diferentes funciones que pueden realizar los atributos de la clase.

- **Estructura y miembros o componentes**

A continuación, vamos a ver un ejemplo:

CÓDIGO

```
public class Alumno {
    //Atributos;
    //Métodos;
}
```

- **public** → Palabra reservada que se utiliza para indicar la visibilidad de una clase
- **class** → Palabra reservada que se utiliza para indicar el inicio de una clase
- **Alumno** → Nombre que le asignemos a la clase
- **Atributos** → Diferentes características que van a definir la clase
- **Métodos** → Conjunto de operaciones que van a realizar los atributos que formen parte de la clase

Los **miembros o componentes** que forman parte de una clase (atributos y métodos) se pueden declarar de varias formas:

- **Públicos (*public*)**. Engloba aquellos elementos a los que se puede acceder desde fuera de la clase y package. Todas las clases de nuestro aplicativo tendrán acceso completo a las funciones o atributos del objeto.

- **Privados (*private*).** Aquellos componentes de carácter privado solamente pueden ser utilizados por la propia clase. Para el resto es como si no existiesen.

También hay otros modificadores que se pueden utilizar en determinadas ocasiones, como:

- ***Protected*.** Lo utilizamos cuando trabajamos con varias clases que heredan las unas de las otras, de tal forma que, aquellos miembros que queremos que actúen de forma privada se suelen declarar como *protected*. De este modo, puede seguir siendo privado, aunque permite que lo utilicen las clases que hereden de ella.
- ***Package*.** Podemos utilizarlo cuando tenemos una clase que no tiene modificador y, además, es visible en todo el paquete. De esta forma, aunque la clase no tenga modificador, puede actuar de forma similar sin utilizar package.

En la siguiente tabla podemos ver con detalle el alcance de cada modificador:

Ubicación	<i>Private</i>	<i>package</i>	<i>Protected</i>	<i>public</i>
Misma clase	x	x	x	x
Mismo package	x	x	x	x
Subclase mismo package		x	x	x
Subclase diferente package			x	x
Diferente package				x

A continuación, aparece un listado de palabras reservadas (***key words* o *reserved words***) del lenguaje de programación Java. Estas palabras no se pueden utilizar como identificadores en los programas escritos en Java.

Las palabras reservadas *const* y *goto* no se utilizan actualmente. Las palabras *true*, *false* y *null* se tratan como si fueran palabras reservadas. Sin embargo, son constantes literales y no se pueden usar como identificadores.

PALABRAS RESERVADAS				
abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

## 7.2. Creación de atributos

Gracias a los atributos podemos recoger características específicas de un objeto determinado mediante el uso de variables.

Se expresan de la siguiente forma:

### CÓDIGO

```
//Sintaxis de los atributos
[Modificador_de_acceso]Tipo_datonombre_atributo;
```

Donde:

- **Modificador\_de\_acceso:** se utiliza para definir el nivel de ocultación o visibilidad de los miembros de la clase (atributos y métodos). Estos pueden ser default, protected, private o public, como los más utilizados. También tenemos otros valores como final, static, volatile y transient
- **Tipo\_dato:** un atributo puede ser de cualquier tipo de datos que exista, como int, double, char o algunos más complejos, como estructuras o incluso objetos
- **Nombre\_atributo:** identificador que vamos a utilizar para esa variable

A continuación, vamos a ver un ejemplo donde detallamos la creación de los atributos:

CÓDIGO

```
public class Alumnos {
    //Atributos
    private String nombre;
    private String curso;
}
```

### 7.3. Creación de métodos

Los métodos son las diferentes funciones de una clase y pueden acceder a todos los atributos que esta tenga. Vamos a implementar estos métodos dentro de la propia clase, excepto cuando los métodos sean abstractos (abstract), que se definen en clases derivadas utilizando la palabra extends.

CÓDIGO

```
[Modificador_de_acceso] tipo_devuelto nombre_metodo (parámetros) {
    //sentencias;
}
```

Donde:

- **tipo\_devuelto:** es el tipo de dato que devuelve el método. Para ello, debe aparecer la instrucción *return* en el código. En el caso en el que la función no devuelva ningún valor, utilizaremos la palabra *void*.
- **nombre\_metodo:** nombre con el que vamos a llamar al método.
- **parámetros:** distintos valores que se le pueden pasar a la función para que se puedan utilizar.
- **sentencias:** distintas operaciones que debe realizar el método.

En Java podemos tener los siguientes tipos de métodos:

- **static:** se puede invocar directamente al método, sin tener que crear un objeto de la clase que lo contiene. De la misma forma, también podemos crear atributos estáticos. Cuando utilizamos un método tipo *static*, empleamos las variables estáticas definidas en la clase.
- **abstract:** es más sencillo de comprender después de ver el significado de la herencia. De todas formas, debemos señalar que los métodos abstractos no se declaran en la clase principal, pero sí en las demás que hereden de esta.
- **final:** estos métodos no ofrecen la posibilidad de sobrescribirlos. Se pueden definir como métodos constantes que no admiten modificaciones.
- **native:** métodos implementados en otros lenguajes, pero que deseamos añadir a nuestro programa. Podemos hacerlo incorporando la cabecera de la función en cuestión y sustituyendo el cuerpo del programa por “;” (punto y coma).
- **Synchronized:** utilizado en aplicaciones multihilo.

Vamos a ver un ejemplo siguiendo con la clase *Alumnos*, la cual vamos a desarrollar durante este apartado:

## CÓDIGO

```
//Clase alumnos que nos servirá para crear objetos tipo alumno
publicclass Alumnos {
    //Atributos
    private String nombre;
    private String curso;

    //Métodos GET y SET
    public String getNombre() {return nombre;}
    publicvoid setNombre(String nombre) {this.nombre = nombre;}
    public String getCurso() {return curso;}
    publicvoid setCurso(String nombre) {this.curso = curso;}

    //Métodos creados por el programador
    publicdouble evaluar(double nota) {
        nota = nota *0.7;
        return nota;
    }
}
```

Hemos creado los métodos *get* y *set*. Estos métodos son funcionalidades del programa hechas por el programador. Son muy comunes en Java, ya que estas devuelven los valores de los atributos o nos permiten modificarlos. Los métodos *get* sirven para mostrar los valores de los atributos y los métodos *set* sirven para insertar o modificar los valores de los atributos.

También hemos creado un método que realizará una funcionalidad propia de esta clase, como podría ser evaluar a los alumnos. Según la nota que reciba, este realizará cálculos internos y devolverá el valor del ejercicio en la nota final.



## 7.4. Sobrecarga de métodos

La **sobrecarga de métodos** consiste en crear métodos en la misma clase y con el mismo nombre, pero con distintos parámetros de entrada. La sobrecarga de métodos permite asignar más funcionalidad.

Por ejemplo, veamos diferentes posibilidades que podrían existir para una función denominada *visualizar*:

### CÓDIGO

```
public void visualizar () {
    //código del método visualizar sin parámetros
}
public void visualizar (Objeto X) {
    //código del método visualizar con un parámetro
}
public void visualizar (Objeto X, int num1) {
    //código del método visualizar con dos parámetros
}
```

Podemos comprobar que existen tres funciones que se llaman de la misma forma, aunque cada una de ellas tiene diferentes parámetros. A la hora de realizar la llamada a la función no va a existir ambigüedad

### CÓDIGO

```
visualizar (); //Estamos haciendo referencia a la primera, que no
               //tiene parámetros.
visualizar (dato); //Si dato es de tipo objeto, nos estaremos
                  //refiriendo a la segunda.
visualizar (dato, 5); // Nos referimos a la tercera opción.
```

Podemos ver de forma clara que las tres llamadas se diferencian perfectamente entre sí. El paso de parámetros es el adecuado.

## 7.5. Creación de constructores

Una de las maneras que tenemos de identificar un constructor de una clase es el nombre. Debe llamarse igual que esta. El constructor se ejecuta siempre de forma automática al crearse una instancia de la clase.

Existe la posibilidad de tener varios constructores, cada uno de ellos con diferentes parámetros.

Siguiendo con el ejemplo de los alumnos, vamos a ver cómo crear los constructores para esta clase:

### CÓDIGO

```
public class Alumnos {  
    //Atributos  
    private String nombre;  
    private String curso;  
    //Constructor vacío, constructor por defecto  
    public Alumnos() {  
        this.nombre = "Ilerna";  
        this.curso = "Online";  
    }  
    //Constructor con parámetros  
    public Alumnos(String nombre, String curso) {  
        this.nombre = nombre;  
        this.curso = curso;  
    }  
    //Métodos  
}
```

En este ejemplo hemos creado dos constructores: uno sin parámetros y otro que recibe parámetros.

La llamada al constructor sin parámetros devuelve un valor por defecto del objeto. En el caso de llamar al constructor con parámetros, le podemos indicar los valores que deseemos a este objeto.

## 7.6. Creación de destructores y/o métodos de finalización

Como hemos indicado en apartados anteriores, Java **no utiliza destructores** como tal, por ser un tipo de lenguaje que se encarga de la eliminación o liberación de memoria que puede ocupar un objeto determinado a través de la recolección de basura.

Anteriormente ya hemos comentado que el recolector de basura en Java, antes de “barrer el objeto no usado”, llama al método ***finalize()*** de ese objeto. Este método puede ser implementado por nosotros para realizar acciones antes de la eliminación del objeto.

## 7.7. Uso de clases y objetos. Visibilidad

Como ya hemos visto en apartados anteriores, la definición de las clases y de los objetos sigue una estructura implementada en un programa en Java.

Esta es la sintaxis que debemos seguir a la hora de instanciar un objeto:

### CÓDIGO

```
//Declaración
nombre_clase nombre_variable;

//Creación
nombre_variable=new nombre_clase ();

//Declaración creación
nombre_clase nombre_variable =new nombre_clase ();
```

De la misma forma que se utiliza en otros lenguajes de programación, debemos hacer uso de la palabra **new** para poder reservar un espacio en memoria, de tal forma que, si solo declaramos el objeto, no podremos utilizarlo. Esta instrucción comienza con una expresión para instanciar (crear) una clase, la cual crea un objeto del tipo especificado a la derecha del **new**.

Una vez instanciado el objeto, la forma de acceder a los diferentes miembros de la clase va a ser utilizando el operador punto. En el lenguaje Java vamos a utilizar el operador **this** para poder referenciar la propia clase junto con sus métodos y atributos.

Si necesitamos crear arrays de objetos, debemos inicializar cada objeto de la casilla que le corresponda en la tabla de la clase para cuando llegue el momento de utilizar ese objeto que se encuentra almacenado en un array, antes debe haber sido creado.

#### CÓDIGO

```
//Declaración creación del array
Clase [] nombre_array =new Clase [MAX];
//Creación objetos que se necesiten
for(int i=0; i<MAX; i++){
    nombre_array [i]=new Clase ();
}
//Creación de un objeto determinado para que exista antes de ser
utilizado
nombre_array [pos]=new Clase ();
```

El método *main* proporciona el mecanismo para controlar la aplicación. Cuando se ejecuta una clase Java, el sistema localiza y ejecuta el método main de esa clase.

A continuación, tenemos el ejemplo de la clase *Aula* donde vemos cómo crear los objetos tipo alumno y cómo utilizar los métodos que hemos generado.

## CÓDIGO

```
//Clase Aula donde utilizaremos los objetos alumnos
publicclass Aula {
    publicstaticvoidmain(String[] args){
double nota[] = new double[2];
//Creación de objetos Alumno
Alumno alumno1 = new Alumno("Antonio", "DAW");
Alumno alumno2 = new Alumno("Laura", "DAM");
//Sintaxis de utilización de los métodos del objeto
nota[0] = alumno1.evaluar(6.5);
nota[1] = alumno2.evaluar(8);
}
}
```

## 7.8. Conjuntos y librerías de clases

- **Operadores de entrada/salida de datos en Java.** Hasta el momento hemos estado utilizando aplicaciones de consola y, según estas aplicaciones, las correspondientes entradas y salidas de datos mediante teclado o consola del sistema.
- **Métodos para visualizar la información.** Los métodos de los que disponemos cuando tenemos que escribir información por pantalla siguen esta sintaxis:

## CÓDIGO

```
System.out.metodo();

//Donde método puede ser cualquier método de impresión de
información la //salida estándar.

//En el siguiente ejemplo podemos verlo de una forma más clara:
System.out.println("Este es el texto");

//Esto muestra por pantalla el texto escrito entre comillas
```

Debemos apuntar que, aparte de **println ()**, existen otros **métodos** que detallamos a continuación:

Método	Descripción
println (boolean)	Sobrecarga del método <i>println</i> utilizando diferentes parámetros en cada tipo
println (char)	
println (char [])	
println (double)	
println (float)	
println (int)	
println (long)	
println (java.lang.Object)	
println (java.lang.String)	
print()	Imprime información por pantalla sin salto de línea
printf ( <i>cadena para formato, variables</i> )	Escribe una cadena con los valores de las variables

Los diferentes **caracteres especiales** que se pueden utilizar con el método **printf ()** son los siguientes:

Caracteres especiales	Significa
%a	Real decimal con mantisa y exponente
%b	Booleano
%c	Carácter Unicode
%d	Entero decimal
%e	Real notación científica
%f	Real decimal
%g	Real notación científica o decimal
%h	Hashcode
%n	Separador de línea
%o	Entero octal
%s	Cadena
%t	Fecha y hora
%x	Entero hexadecimal

El **formato de fecha y hora (%t)** tiene distintas variantes que mostramos en la siguiente tabla:

Caracteres especiales	Significa	Resultado
<b>%tA</b>	Día de la semana	jueves
<b>%ta</b>	Día de la semana abreviado	jue
<b>%tB</b>	Mes	febrero
<b>%tb</b>	Mes abreviado	feb
<b>%tC</b>	Parte del año que señala el siglo	20
<b>%tc</b>	Fecha y hora con formato “%ta %tb %td %tT %tZ %tY”	jue feb 01 13:31:41 CET 2018
<b>%tD</b>	Fecha con formato “%tm/%td/%ty”	02/01/18
<b>%td</b>	Día del mes con formato “01-31”	01
<b>%te</b>	Día del mes con formato “1-31”	1
<b>%tF</b>	Fecha con formato ISO 8601	2018-02-01
<b>%tH</b>	Hora del día con formato “00-23”	13
<b>%th</b>	El mismo formato que %tb	feb
<b>%tI</b>	Hora del día con formato “01-12”	01
<b>%tj</b>	Día del año con formato “001-365”	032
<b>%tk</b>	Hora del día con formato “00-23”	13
<b>%tI</b>	Hora del día con formato “01-12”	1
<b>%tM</b>	Minuto de la hora con formato “00-59”	31



<b>%tm</b>	Mes actual con formato "01-12"	02
<b>%tN</b>	Nanosegundos con formato de 9 dígitos "000000000-999999999"	844000000
<b>%tp</b>	Marcador específico "am-pm"	pm
<b>%tQ</b>	Milisegundos desde la época del 1 de enero de 1970 a las 00:00:00 UTC	1517488301844
<b>%tR</b>	Hora actual con formato 24 horas	13:31
<b>%tr</b>	Hora actual con formato 12 horas	01:31:41 PM
<b>%tS</b>	Segundos de la hora actual "00-60"	41
<b>%ts</b>	Segundos desde la época del 1 de enero de 1970 a las 00:00:00 UTC	1517488301
<b>%tT</b>	Hora actual con formato 24 horas	13:31:41
<b>%tY</b>	Año actual con formato "YYYY"	2018
<b>%ty</b>	Año actual con formato "YY"	18
<b>%tZ</b>	Abreviación de zona horaria	CET
<b>%tz</b>	Desfase de zona horaria de GMT	+0100

### Métodos para introducción de información

Utilizamos el objeto **System.in.** para referirnos a la entrada estándar en Java. Al contrario que **System.out**, este no cuenta con demasiados métodos para la recogida de información, por lo que podemos hacer uso del método **read()**, que recoge un número entero, equivalente al código ASCII del carácter pulsado en el teclado.

Veamos un ejemplo para aclarar estos conceptos:

#### CÓDIGO

```
int numero= System.in.read (); //(1)
System.out.printf ("%c\n", (char) num); //(2)
```

- (1) En el primer caso, `System.in.read ()` se va a esperar hasta que el usuario pulse una tecla.
- (2) Mientras que, en el segundo caso, si deseamos mostrar en pantalla el carácter pulsado, tendremos que realizar un casting a la variable.

- **Clase System**

El uso de la clase `System` es algo habitual cuando queremos mostrar datos por la consola de nuestro programa de desarrollo. Esta clase pertenece al package **java.lang** y dispone de varias variables estáticas a utilizar.

Estas variables son **in**, **out** y **err**, que hacen referencia a la entrada, salida y errores, respectivamente.

#### CÓDIGO

```
//Variable estática out con su método println()
System.out.println("Ilerna Online");
```

La clase `System` tiene otros métodos muy útiles, ya que es la encargada de interactuar en el sistema. Por ejemplo, nos permite acceder a la propiedad de Java home, al directorio actual o a la versión de Java que tenemos.

## CÓDIGO

```
System.out.println(System.getProperty("user.dir"));
System.out.println(System.getProperty("java.home"));
System.out.println(System.getProperty("java.specification.version"));
```

Podemos ver el resto de variables de sistema en el JavaDoc de la clase. A continuación, enumeramos otros métodos que se usan habitualmente.

- **arrayCopy():** se encarga de copiar arrays
- **currentTimeMillis():** nos devuelve el tiempo en milisegundos
- **exit():** termina el programa Java

- **Clase Console:**

Facilita el manejo de entrada y salida de datos por la línea de comandos. Para la lectura de datos mediante la clase *Console* con Java necesitamos obtener la consola, que es una instancia única.

Esto lo logramos de la siguiente manera:

## CÓDIGO

```
Console console =null;
console = System.console();
```

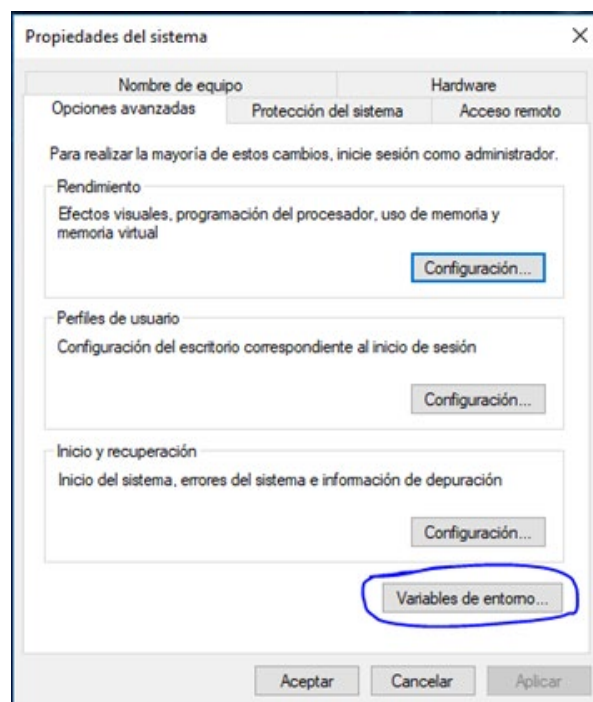
Cuando obtenemos la instancia principal de consola podemos hacer uso de las funciones de input de datos por teclado, como son:

- `readLine()`
- `readPassword()`

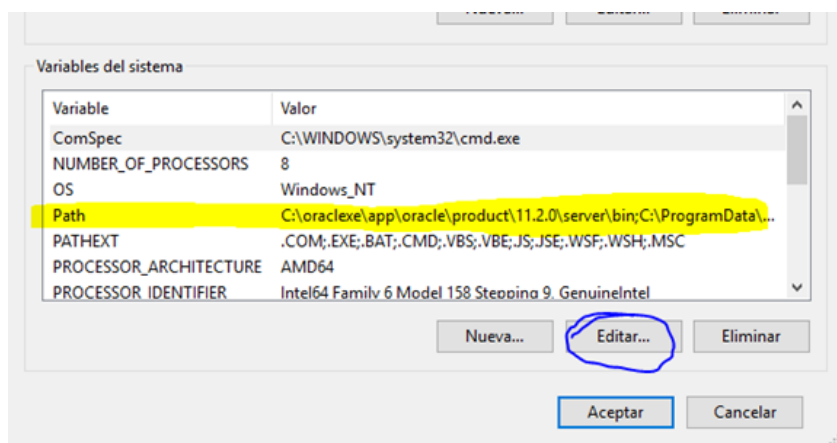
La clase Console funciona en base a la consola de usuario. Vamos a ver cómo configurar la consola de Windows para poder ejecutar un programa con esta función:

Las siguientes capturas de pantalla se han realizado con **Windows 10**. En caso de tener un sistema operativo anterior a este, la modificación de las variables de entorno se realiza añadiendo la ruta nueva mediante una cadena de texto, pero la finalidad es la misma.

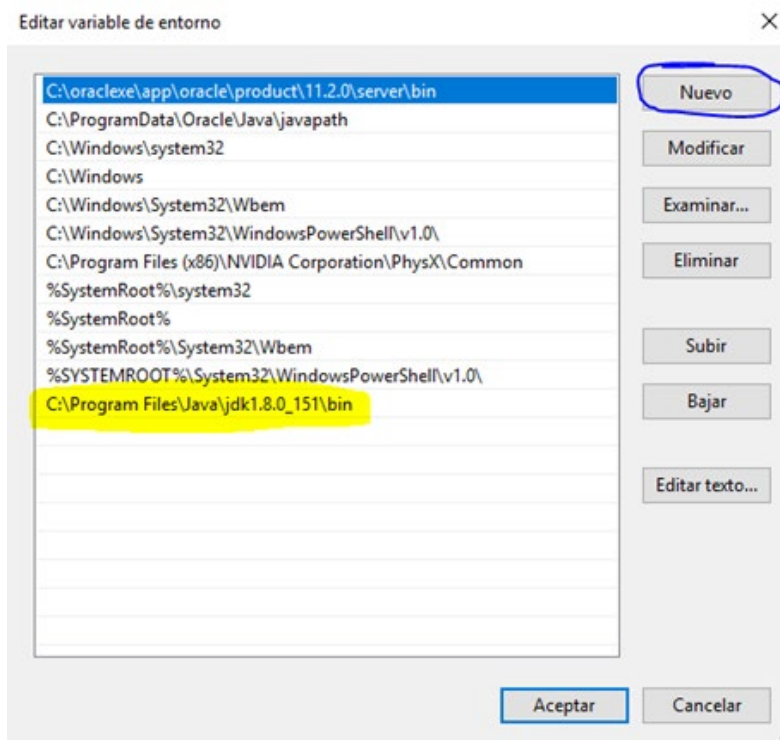
- **Modificar las variables de entorno de Windows:**



- **Modificar la variable Path:**



- **Añadir la ruta de la instalación de Java (C:\Program Files\Java\jdk1.8.0\_151\bin):**



La ruta de instalación puede variar, dependiendo de la versión de Java instalada y la que cada usuario le haya dado (el ejemplo muestra la ruta de instalación por defecto).

Cuando añadimos una ruta en la variable Path, no tenemos que modificar las demás rutas ya establecidas en esta variable.

- **Compilar el programa en el símbolo del sistema (cmd):**

CÓDIGO

```
javac nombre_programa.java
```

- **Ejecutar el programa desde el símbolo del sistema (cmd):**

CÓDIGO

```
java nombre_programa
```

Una vez hemos visto cómo configurar la cmd de Windows para poder ejecutar programas Java, vamos a ver un ejemplo de la clase Console.

CÓDIGO:

```
import java.io.Console;

public class consoleEx {

    public static void main(String[] args) {

        Console c = System.console();

        String name = c.readLine("Nombre usuario:");

        char pwd[] = c.readPassword("Contraseña");

        String upwd = new String(pwd);

        //El usuario es Ilerna y la contraseña Online
        if (name.equals("Ilerna") && upwd.equals("Online")) {

            System.out.println("Usuario y contraseña validos");

        }

        else {

            System.out.println("Usuario o contraseña no validos");

        }

    }

}
```

- **Clase Scanner**

La clase *Scanner* pertenece a la librería **java.util**. Esta clase nos ofrece una forma muy sencilla de obtener datos de entrada del usuario. Scanner posee un método para la lectura de datos.

Cuando utilizamos la clase **Console** en uno de nuestros programas, tenemos que importar la librería "**java.io**" para que este funcione correctamente.

```
import java.io.Console;
```

Dicha clase tiene una característica que la diferencia del resto y es que recibe por parámetro el fichero del cual realiza la lectura de los datos. En el caso más general suele ser la entrada de teclado estándar “**System.in**”, pero nada nos impide establecer un fichero con datos y leerlo de este.

#### CÓDIGO

```
//Se crea el lector
Scanner sc =new Scanner(System.in);

//Se pide un dato al usuario
System.out.println("Por favor ingrese su nombre");

//Se lee el nombre con nextLine() que retorna un String con el dato
String nombre =sc.nextLine();

//Se pide otro dato al usuario
System.out.println("Por favor ingrese su edad");

//Se guarda la edad directamente con nextInt()
int edad =sc.nextInt();

//Imprimimos los datos solicitados por pantalla
System.out.println("Gracias "+ nombre +" en 10 años usted tendrá "+(edad +10)+" años.");
```

Cuando utilizamos la clase **Scanner** en uno de nuestros programas, tenemos que importar la librería “**java.util**” para que funcione correctamente.

```
import java.util.Scanner;
```

Si deseamos observar más ejemplos, podemos revisar la documentación oficial de Java.

#### Java Scanner

Visita la documentación oficial en el siguiente enlace:

<https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html>



Finalmente, vamos a ver el ejemplo completo de las clases Alumno y Aula para comprobar el funcionamiento de todos los puntos vistos en este apartado:

#### CÓDIGO

```
//Clase alumno que nos servirá para crear objetos tipo alumno
publicclass Alumno {
//Atributos
private String nombre;
private String curso;
//Constructor vacío
publicAlumno() {
this.nombre="Ilerna";
this.curso="Online";
}
//Constructor con parámetros
publicAlumno(String nombre, String curso){
this.nombre= nombre;
this.curso= curso;
}
//Métodos GET y SET
public String getNombre() {return nombre;}
publicvoid setNombre(String nombre) {this.nombre = nombre;}
public String getCurso() {return curso;}
publicvoid setCurso(String nombre) {this.curso = curso;}
//Métodos creados por el programador
publicdouble evaluar(double nota) {
nota = nota *0.7;
return nota;
}
}
```

## CÓDIGO:

```
//Clase Aula donde utilizaremos los objetos alumnos
publicclass Aula {
publicstaticvoidmain(String[] args){
double[] nota = new double[2];
//Creación de objetos Alumno
    Alumno alumno1 = new Alumno("Antonio", "DAW");
    Alumno alumno2 = new Alumno("Laura", "DAM");
//Sintaxis de utilización de los métodos del objeto
    nota[0] = alumno1.evaluar(6.5);
    nota[1] = alumno2.evaluar(8);
//Impresión de los Atributos del alumno
    System.out.println("Nombre: " + alumno1.getNombre() +
" Curso " + alumno1.getCurso() +
" Nota " + nota[0]);
    System.out.println("Nombre: " + alumno2.getNombre() +
" Curso " + alumno2.getCurso() +
" Nota " + nota[1]);
}
}
```

## 8. Utilización avanzada de clases en el diseño de aplicaciones

### 8.1. Composición de clases

La composición es el agrupamiento de uno o varios objetos y valores como atributos, que conforman el valor de los distintos objetos de una clase.

La composición crea una relación, de forma que la clase contenida debe coincidir con la vida de la clase contenedor. La **composición** y la **herencia** son las dos formas que existen para llevar a cabo la reutilización de código.

Para formalizar este concepto de composición de clases vamos a utilizar un ejemplo para verlo de una manera más práctica. En este caso, haremos una clase *Empleado* que será la clase contenida en la clase *Empresa*, la cual actuará como clase contenedora.

#### CÓDIGO CLASE EMPLEADO

```
import java.util.Date;
public class Empleado {

    //atributos
    private String dni;
    private String nombre;
    private double sueldo;
    private Date fechaNac;

    //Constructores
    public Empleado () {
        this.dni = "00000000I";
        this.nombre = "Ilerna Online";
        this.sueldo = 1000;
        this.fechaNac = new Date ();
    }
}
```

```
public Empleado(String dni, String nombre, double sueldo, Date fn) {
    this.dni = dni;
    this.nombre = nombre;
    this.sueldo = sueldo;
    this.fechaNac = fn;
}

//GETS AND SETS
public String getDni() {return dni;}
public void setDni(String dni) {this.dni = dni;}
public String getNombre() {return nombre;}
public void setNombre(String nombre) {this.nombre = nombre;}
public Date getFechaNac() {return fechaNac;}
public void setFechaNac(Date fn) {this.fechaNac = fn;}
public double getSueldo() {return sueldo;}
public void setSueldo(double sueldo) {this.sueldo = sueldo;}

//Métodos
public double horasExtras(double horas) {
    double PrecioHora = 11;
    double extras;
    extras = horas * PrecioHora;
    return extras;
}
}
```

Y vamos a ver ahora la clase *Empresa*. Va a tener como miembros varios empleados (emp1, emp2, etc...), que son objetos de la clase *Empleado* y también la dirección y teléfono de esta.

## CÓDIGO CLASE EMPRESA

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class Empresa {
    private Empleado emp;
    private int telefono;
    private String direccion;
    //Constructores
    public Empresa() {
        emp = new Empleado();
        telefono = 900730222;
        direccion = "Turó Gardeny 25003 Lleida";
    }
    public Empresa(Empleado e) {
        this(e, 900730222, "Turó Gardeny 25003 Lleida");
    }
    public Empresa(int tel, String dir) {
        this(new Empleado(), tel, dir);
    }
    public Empresa(Empleado e, int tel, String dir) {
        emp = e;
        telefono = tel;
        direccion = dir;
    }
    //Método calculo horas extras Empleado
    public double horasExtras(double horas) {
        emp.horasExtras(horas);
        return horas;
    }
    //Método que da formato a la fecha
    public Date fechaNac(String fecha) throws ParseException {
        SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yyyy");
        Date data = formato.parse(fecha);
        return data;
    }
}
```

## 8.2. Herencia

La herencia es uno de los términos más importantes en la POO. Podemos definir la herencia entre diferentes clases de la misma forma que lo hacemos en la vida real. Es decir, un hijo puede heredar de un padre el color de sus ojos, los gestos, la constitución, etc.

Por eso, en cuanto a las clases se refiere, se dice que **una clase hereda de otra cuando adquiere características y métodos de la clase *padre***.

Gracias a la herencia está permitido jerarquizar un grupo de clases y podemos aprovechar sus propiedades y métodos sin necesidad de volverlos a crear o implementar.

Podemos diferenciar entre dos tipos diferentes de clase:

- **Clase base:** es la clase desde la que se hereda. En una jerarquía de clases, la clase base es la que está situada más arriba y se pueden aprovechar sus características y funcionalidades. Se la denomina también **clase *padre*** o **superclase**. En Java la clase base de todas las clases se denomina *Object* y nos ofrece un conjunto de acciones para que todo el aplicativo funcione con normalidad.
- **Clase derivada:** es la clase que hereda de otras. Aprovecha la funcionalidad y, aunque sea clase derivada, también puede ser clase base de otras clases.

Y también existen dos tipos distintos de herencia:

- **Simple:** en la que cada clase deriva de una única clase.
- **Compuesta:** Java no soporta este tipo de herencia. Para simularla tiene que hacer uso de las **interfaces**. Sí que la pueden soportar determinados lenguajes de programación, como C++.

La sintaxis que sigue la herencia es la siguiente:

#### CÓDIGO

```
class Base {
//Código clase Base
}
class Derivada extends Base {
//Código clase Derivada
}
```

Utiliza la palabra reservada ***extends*** para crear la relación de herencia.

Debemos tener en cuenta las características que tiene una **clase derivada**. Por ejemplo, solo tiene la opción de acceder a los miembros ***public*** y ***protected*** de la clase base. A los miembros ***private*** no se puede acceder de forma directa, aunque sí podemos hacerlo mediante métodos públicos o protegidos de la clase. Es decir, si un método público accede a otro privado, este sí que tendrá acceso, pero no podremos hacerlo directamente al método privado.

En caso de tener miembros de la clase definidos como privados, la forma de acceder a ellos será con los métodos ***gets*** y ***sets***.

La **clase derivada** debe incluir los miembros que se han definido en la clase base.

A continuación, veamos un ejemplo de estas definiciones:

## CÓDIGO

```

class Empleado {
protected String nombre, apellidos;
protected Double sueldo;
protected String DNI;

//código de clase empleado
}
class Cualificados extends Empleado {
protected String titulacion;
protected Double extra;
protected String departamento;

//código de clase cualificados
}
class Obreros extends Empleado {
private String destino;
private int horas_extra;
private double precio_hora;

//código de clase obreros
}
class Jefe extends Cualificados {
private int total_trabajadores;
private String [] proyectos;
private double extra;
//código de clase jefe
}

```



### 8.3. Jerarquía de clases: superclases y subclases

En la Programación Orientada a Objetos Avanzada, cuando vemos términos como herencia o polimorfismo, podemos simular situaciones mucho más complejas que las de la vida real. Aparece un concepto nuevo, que es la jerarquía de clases.

Una jerarquía es la estructura por niveles de algunos elementos, donde los situados en un nivel superior tienen algunos privilegios sobre los situados en el nivel inferior. También puede contener una relación entre los elementos de varios niveles entre los que también exista una relación.

Por tanto, si esa definición la extrapolamos a la POO, nos referimos a la jerarquía de clases que se define cuando algunas clases heredan de otras.

A la que situamos en el nivel superior la nombramos **superclase** y la que representamos en el nivel inferior pasamos a llamarla **subclase**. Por tanto, la herencia es poder utilizar como nuestros los atributos y métodos de una superclase en el interior de una subclase. Es decir, un hijo toma prestado los atributos y métodos de su padre.

### 8.4. Clases y métodos abstractos y finales

Imaginemos que tenemos una clase *Profesor* de la que van a heredar dos clases: *ProfesorInterino* y *ProfesorOficial*. De tal forma que, todo profesor debe ser o bien *ProfesorInterino* o bien *ProfesorOficial*. En este ejemplo, no se necesitan instancias de la clase *Profesor*. Entonces, ¿para qué hemos creado esta clase?

Una superclase se declara para poder unificar atributos y métodos a las diferentes subclases evitando, de esta forma, que se repita código. En el caso anterior de la clase *Profesor* no se necesita crear objetos, solo se pretende unificar los diferentes datos y operaciones de las distintas subclases. Por este motivo, se puede declarar de una manera especial en Java. Podemos definirla como clase abstracta.

La sintaxis de esta clase abstracta sería:

#### CÓDIGO

```
modificador_de_accesoabstractclass NombreClase {}
```

Si seguimos con el ejemplo de la clase *Profesor* que estamos viendo, lo declaramos:

#### CÓDIGO

```
public abstract class Profesor {
    //Atributos
    protected String nombre;
    protected String dni;
    protected int edad;
    //Métodos
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getDni() {
        return dni;
    }
    public void setDni(String dni) {
        this.dni = dni;
    }
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
    public double pacs(double pac1, double pac2, double pac3) {
        return (pac1 + pac2 + pac3) / 3;
    }
    public double pacs(double pac1, double pac2, double pac3, double
    pac4) {
        return (pac1 + pac2 + pac3 + pac4) / 4;
    }
}
```

Cuando empleamos esta sintaxis no podemos instanciar la clase en cuestión. No podemos crear objetos de ese tipo. Lo que sí está permitido es que siga funcionando como superclase normal, pero sin posibilidad de crear ningún objeto de esta clase.

A parte de este requisito que hemos contemplado, debemos tener en cuenta también unas características que hagan que una clase sea abstracta.

- En la clase **abstract** no se pueden crear objetos directamente (new), es decir, no puede ser instanciada, solo heredada.
- Cuando uno de los métodos de la clase es **abstract**, este obliga a que toda la clase sea **abstract**, en cambio, la clase puede contener métodos no abstractos.
- Los métodos **abstract** no tienen cuerpo, es decir, no llevan los caracteres { }.
- La primera subclase que herede de una clase **abstract** debe implementar todos los métodos de la **superclase**.

Veamos cómo quedaría su sintaxis:

#### CÓDIGO

```
abstractmodificador_de_accesoTipo de retorno(parámetros) ;
```

Ejemplo: `abstractpublicvoid(String a) ;`

#### Notas

- Cuando declaramos un método abstracto, el compilador de Java nos obliga a que declaremos esa clase abstracta. Ya que, si no lo hacemos, tendremos un método abstracto de una clase determinada NO ejecutable. Y eso no está permitido.
- Las clases se pueden declarar como abstractas, aunque no tengan métodos abstractos. Algunas veces estas clases realizan operaciones comunes a las subclases, sin que necesiten métodos abstractos. Otras utilizan métodos abstractos para referenciar operaciones de la clase abstracta.

- Una clase abstracta no se puede instanciar, aunque sí podemos crear subclases determinadas sobre la base de una clase abstracta y crear instancias de estas subclases. Para llevar a cabo esta operación, debemos heredar de la clase abstracta y anular aquellos métodos abstractos existentes (tendremos que implementarlos).

**CÓDIGO**

```
public class ProfesorInterino extends Profesor {
    //Main
    public static void main(String[] args) {
        ProfesorInterino p = new ProfesorInterino();
        System.out.println(p.getNotaMedia(6.5,7,8));
    }
    //Constructor
    public ProfesorInterino() {
        super.dni = "45633254L";
        super.nombre = "Adrián García";
        super.edad = 29;
    }
    //Método
    public double getNotaMedia(double pac1,double pac2,double pac3) {
        double notaMedia;
        notaMedia = super.pacs(pac1,pac2,pac3);
        System.out.println(notaMedia);
        return notaMedia;
    }
}

public class ProfesorOficial extends Profesor {
    //Main
    public static void main(String[] args) {
        ProfesorOficial p = new ProfesorOficial();
        p.getNotaMedia(7.5, 7, 9.3, 8.7);
    }
    //Constructor
    public ProfesorOficial() {
        super.dni = "48566221F";
        super.nombre = "Ana Gómez";
        super.edad = 35;
    }
    //Método
    public double getNotaMedia(double pac1, double pac2,
        double pac3, double pac4) {
        double notaMedia;
        notaMedia = super.pacs(pac1, pac2, pac3, pac4);
        System.out.println(notaMedia);
        return notaMedia;
    }
}
```

Por otro lado, nos encontramos con los métodos finales. Estos métodos vendrán precedidos de la palabra reservada **final**. Esta permitirá tener un método que jamás podrá ser sobrescrito en ninguna clase que herede de esta.

Esto se utiliza para conseguir un código robusto, evitando posibles errores en el momento de programar. Imaginemos que en la clase *Profesor* descrita con anterioridad tenemos un método en el que debemos comprobar el formato del DNI y este método jamás debe ser modificado. Podemos establecer el siguiente código:

#### CÓDIGO

```
public abstract class Profesor {
//Atributos
protected String nombre;
protected String dni;
protected int edad;
//Métodos

public final boolean comprobarDNI(String dni){
//Comprobación del DNI recibido por parametros.
}
}
```

## 8.5. Sobrescritura de métodos (Overriding)

Podemos definir la sobrescritura de métodos como la forma mediante la cual una clase que hereda puede redefinir los métodos de su clase *padre*. Y, así, se pueden crear nuevos métodos que tengan el mismo nombre de su superclase.

Veamos un ejemplo: si tenemos una clase *padre* con el método “ingresar()”, podemos crear una clase *hija* que tenga un método que se denomine también “ingresar()”, pero implementándolo según los requisitos que necesite. Este proceso es el que recibe el nombre de sobrescritura.

Existen una serie de reglas que se deben cumplir para llevar a cabo la sobrescritura de métodos:

- Debemos comprobar que la estructura del método sea la misma a la de la superclase. La firma de los métodos debe de ser idéntica. Denominamos firma al conjunto de los valores nombre, valores de entrada y valor de salida
- Debe tener, no solo el mismo nombre, sino también el mismo número de argumentos y valor de retorno

Adicionalmente, se incluirá la anotación **@Override** encima del método, la cual obligará al compilador a comprobar si se está realizando correctamente la sobrescritura del método. En términos prácticos se trata de conseguir un código mucho más robusto y seguro.

Siguiendo con el ejemplo del profesor, veamos un ejemplo de cómo sobrescribir un método creado en la clase *padre*:

**CÓDIGO**

```
public class ProfesorOficial extends Profesor {
    //Main
    public static void main(String[] args) {
        ProfesorOficial p = new ProfesorOficial();
        p.getNotaMedia(7.5, 7, 9.3, 8.7);
    }
    //Constructor
    public ProfesorOficial() {
        super.dni = "48566221F";
        super.nombre = "Ana Gomez";
        super.edad = 35;
    }
    //Método
    public double getNotaMedia(double pac1, double pac2,
        double pac3, double pac4) {
        double notaMedia;
        notaMedia = pacs(pac1, pac2, pac3, pac4);
        System.out.println(notaMedia);
        return notaMedia;
    }
    //Método sobrescrito de la clase padre
    @Override
    public double pacs(double pac1, double pac2, double pac3) {
        double nota_final;
        nota_final = ((pac1 + pac2 + pac3) / 3) + 0.6;
        return nota_final;
    }
}
```

## 8.6. Herencia y constructores/destructores/métodos de finalización

Cuando utilizamos **herencias** y tenemos algunas clases que heredan de otras, estas pueden heredar los atributos de la clase base en que deben ser inicializados. Si la clase base posee atributos privados, no son accesibles para las clases que heredan, pero sí que podemos hacer un llamamiento a estos atributos mediante sus métodos constructores.

Para ello, debemos tener en cuenta una serie de características:

- Al declarar una instancia de la **clase derivada** debe llamarse de manera automática al constructor que posea la **clase base**.
- Si la **clase base** cuenta con **un constructor**, la clase derivada debe hacer referencia a este en su constructor.
- Si la **clase base** no cuenta con **ningún tipo de constructor**, la clase derivada no tiene la obligación de hacer referencia en su constructor.
- Al crear un **constructor** en la **clase derivada**, debemos añadirle los parámetros necesarios para que pueda inicializar la clase en cuestión y la clase base.

## 8.7. Interfaces

Las **interfaces** están formadas por un **conjunto de métodos que no necesitan ser implementados**. Se podría decir que son clases 100% abstractas. Es decir, que están compuestas por un conjunto de métodos abstractos.

Una interfaz también puede contener unos atributos, pero estos se consideran variables y no pueden ser modificados en ninguna clase que implemente dicha interfaz.

Dentro de una clase podemos implementar una o algunas interfaces. Y la implementación de esta interfaz se basa en desarrollar los métodos que se han definido para tal fin.

Estas sirven para establecer la forma que debe tener una clase. Es decir, son como el molde de una clase. Al definir interfaces, permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos.

Algunas veces, mediante las interfaces se puede representar la herencia múltiple en los programas, ya que C# y Java no las pueden soportar.

La sintaxis que se suele utilizar para crear interfaces es bastante parecida para estos dos lenguajes (C# y Java):

#### CÓDIGO

```
Modificador_de_acceso interface nombre_interface{
    tipo nombre1 (parámetros);
    tipo nombre2 (parámetros);
}
```

Dónde:

- **Modificador\_de\_acceso:** puede ser cualquiera de los utilizados en las diferentes definiciones de clases como **public**, **private**, **protected**, etc.
- **Nombre\_interfaz:** es el nombre que se le asigna a la interfaz
- **Tipo nombre (parámetros):** hace referencia a los diferentes métodos de la interfaz que van a implementar las demás clases

Además, también es posible realizar la herencia entre interfaces, de tal forma que vamos a poder disponer de interfaces base y derivadas.

La definición que se va a llevar a cabo de herencia para interfaces es parecida a la que utilizan las clases, con la diferencia de que va a utilizar la palabra **interface**. Su sintaxis es:

#### CÓDIGO

```
Modificador_de_acceso IDerivada extends IBase{
    //código de la interface
}
```



A la hora de implementarla, necesitaremos hacer uso (en Java) de la palabra *implements* de la siguiente forma:

## CÓDIGO

```
class nombre implements nombreInterface1, nombreInterface2,
nombreInterfaceN {
//código de la clase nombre
Modificador_de_acceso tipo nombre (parámetros){
//código del método
}
}
```

## POO. Librerías de clases fundamentales

### 9. Aplicación de estructuras de almacenamiento en la programación orientada a objetos

#### 9.1. Estructuras de datos avanzadas

Java tiene, desde la versión 1.2, todo un juego de clases e interfaces para guardar agrupaciones (colecciones) de objetos. En esta, todas las entidades conceptuales están representadas por interfaces, y las clases se usan para proveer implementaciones de esas interfaces. Una introducción conceptual debe enfocarse primero en esas interfaces. **La interfaz nos dice qué podemos hacer con un objeto.**

Como corresponde a un lenguaje orientado a objetos, estas clases e interfaces están estructuradas en una jerarquía: a medida que se va descendiendo a niveles más específicos aumentan los requerimientos y lo que se le pide a ese objeto que sepa hacer.

Java cuenta con la interfaz ***Collection*** para el manejo de estructuras avanzadas de datos (colecciones).

- **Conjuntos**

Un conjunto es un grupo de **elementos no duplicados**, es decir, un grupo de valores únicos que, dependiendo del caso en cuestión, pueden estar **ordenados o no**.

- **Listas**

Las listas podemos definirlas como **una secuencia de elementos que ocupan una posición determinada**. Sabiendo la posición que ocupa cada uno, podemos insertar o eliminar un elemento en una posición determinada.

- **Pilas**

Las pilas son similares a las listas, pero añadiendo algunas restricciones. Pueden definirse como una sucesión de varios elementos del mismo tipo, cuya forma para poder acceder a ellos sigue el método de acceder siempre por un único lugar: la cima.

**El primer elemento que entra va a ser el último en salir.**

Sus operaciones principales son:

- Introducir un nuevo elemento sobre la cima (*push*)
- Eliminar un elemento de la cima (*pop*)

### **Ejemplo**

Podemos imaginar que tenemos una pila con varios libros. La forma de tener acceso a alguno de ellos es solo ver el libro que se encuentra arriba del todo, en la cima. Por ese motivo, los demás libros no son accesibles. La pila se vendría abajo.

- **Colas**

Las colas son similares a las listas, pero añadiendo algunas restricciones. Pueden definirse como una sucesión de varios elementos del mismo tipo, cuya forma de poder acceder a ellos sigue el método siguiente:

**El primer elemento que entra es también el primero en salir.**

Sus operaciones principales son:

- Encolar (*enqueue*): para ir añadiendo elementos
- Desencolar (*dequeue*): para eliminar elementos

## Ejemplo

Podemos imaginar que una cola es similar a las colas que se hacen en el supermercado o cuando esperas a que te atiendan en un banco. El primero que llega es el primero en ser atendido.

## 9.2. Creación de arrays

Un array es la agrupación de un conjunto de datos del mismo tipo. Cada uno de los datos de este conjunto es distinguido mediante un índice que nos permite tanto acceder a un elemento concreto como recorrer todos los elementos del array.

Podemos llevar a cabo la sintaxis de un array en Java de la siguiente forma:

### CÓDIGO

```
tipo_dato [] nombre_array; //declaración para el array
nombre_array =new tipo_dato [MAX]; //Reservamos memoria para el array
tipo_dato [] nombre_array =new tipo_dato [MAX];
```

Aplicamos esta definición de array a un ejemplo y así lo vemos de forma más práctica:

### CÓDIGO

```
int[] array =new int[100];
char[] cadena =new char[200];
```

El primer array que nos encontramos en todo programa Java, corresponde a los argumentos de entrada, si nos fijamos la función principal de Java:

CÓDIGO

```
public static void main(String[] args)
```

Recibe por parámetros el array de String y esto corresponderá a cada uno de los valores de entrada que le enviemos a nuestro programa en la ejecución de este y así poder iniciar el programa de una manera u de otra.

Para poder acceder a cada dato, debemos escribir el nombre del array correspondiente, junto con su posición entre corchetes. De esta forma, si queremos acceder al contenido de la posición 5, podremos hacerlo según indicamos a continuación:

Recordemos que la **primera posición** de un array siempre es **cero**:

```
array[0];
```

CÓDIGO

```
array[4];
```

A continuación, vamos a ver un ejemplo de un array. Cómo inicializarlo, cómo darle valor y cómo operar con los valores de sus distintas posiciones:

## CÓDIGO

```
import java.util.Scanner;

public class Ejemplo {
    public static void main(String[] args) {
        //Declaramos el array
        int[] array = new int[3];
        int num = 0;

        Scanner sc = new Scanner(System.in);

        //Inicializamos el primer elemento
        System.out.println ("Introduzca el primer número");
        num = sc.nextInt();

        array [0] = num;

        //Inicializamos el segundo elemento del array
        System.out.println ("Introduzca el segundo número");
        num = sc.nextInt();

        array [1] = num;

        //Realiza la suma de dos posiciones
        array[2] = array[0] + array[1];

        //Mostramos el resultado
        System.out.println (array[0] + " + " + array[1] + " = " + array[2]);
    }
}
```

Tenemos otras opciones a la hora de inicializar los elementos de un array. Podemos inicializar un array cuando lo declaramos:

**CÓDIGO**

```
tipo_dato[] nombre_array =new tipo_dato []{var1, var2, varN};  
tipo_dato[] nombre_array ={var1, var2, varN};
```

Los valores del array van siempre entre corchetes “[]” y se pueden ir asignando, indicando la posición en la que se encuentran. El compilador nos da error si escribimos algún valor entre corchetes cuando creamos el array:

**CÓDIGO**

```
int[] array =newint[4]{2,4,6,8}; //ERROR!!!  
int[] array =newint[] {2,4,6,8}; //CORRECTO  
int[] array ={2,4,6,8}; //CORRECTO
```

En Java existen una serie de métodos ya diseñados que tenemos a nuestra disposición en caso de que necesitemos utilizarlos. En la siguiente tabla podemos ver varios de estos métodos disponibles:

Método	Descripción
length	Obtiene la longitud de la matriz
clone()	Crea y devuelve una copia de este objeto
hashCode()	Devuelve un valor de código hash para el objeto
toString()	Devuelve una representación de cadena del objeto
equals()	Indica si algún otro objeto es igual a este
getClass()	Devuelve la clase de tiempo de ejecución de este objeto
notify()	Despierta un único hilo que está esperando en el monitor de este objeto
notifyAll()	Despierta todos los hilos que están esperando en el monitor de este objeto
wait()	Hace que el subproceso actual espere hasta que otro subproceso invoque el método <i>notify</i> () o el método <i>notifyAll</i> () para este objeto o haya transcurrido un período de tiempo especificado

Por ejemplo, para saber la longitud de una determinada tabla podemos hacer uso del método *length*, que nos devuelve el número de posiciones que tiene un array.

#### CÓDIGO

```
int[] array = new int[10];
array[1] = 5;
array[5] = 2;

System.out.println ("El número total de valores que podemos
introducir en el array es de: "+array.length);
```



### 9.3. Arrays multidimensionales

El lenguaje Java ofrece la posibilidad de trabajar con arrays de más de una dimensión. Presentan una sintaxis muy parecida a los arrays de una dimensión.

Las dimensiones del array vendrán determinadas por el numero de corchetes que le indiquemos en el momento de instanciarlo.

#### CÓDIGO

```
tipo_dato [][] nombre =new tipo_dato [MAX1][MAX2];
tipo_dato [][] [] nombre =new tipo_dato [MAX1][MAX2] [MAX3];
```

En el siguiente ejemplo, vamos a plasmar una matriz de dos dimensiones, ya que su representación gráfica es más sencilla de entender.

#### CÓDIGO

```
publicclass Ejemplo {
    publicstaticvoidmain(String[] args){
        int[][] matriz =newint[2][];
        matriz[0]=newint[]{1,2,3,4,5};
        matriz[1]=newint[]{1,1,1,1,1};
        for(int i =0; i <2; i++){
            System.out.println();
            for(int j =0; j <5; j++){
                System.out.print(matriz[i][j]+" ");
            }
        }
    }
}
```

Si representamos la **matriz** visualmente, lo haríamos de la siguiente forma:

		Segunda dimensión				
		0	1	2	3	4
Primera dimensión	0	1	2	3	4	5
	1	1	1	1	1	1

Ahora vamos a ver un segundo ejemplo de matriz. En este caso, vamos a representar tres dimensiones. En primer lugar, vamos a representar la matriz en 2D:

#### CÓDIGO

```
public class Main {
    public static void main(String[] args) {

        int[][] doble = new int[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                doble[i][j] = i + 1 + j * 3;
            }
        }
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(doble[j][i] + " ");
            }
            System.out.println();
        }
    }
}
```

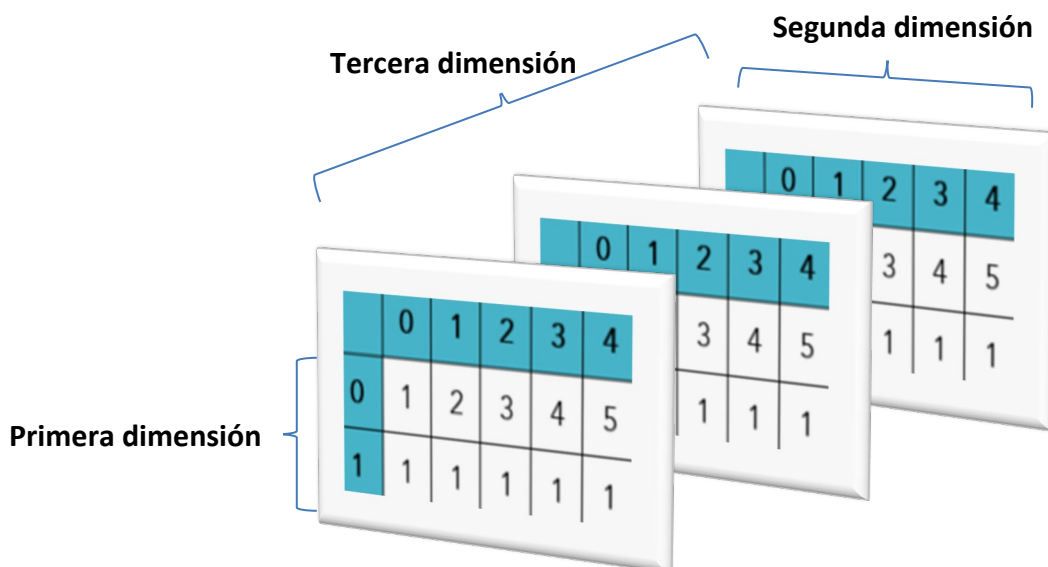
A continuación, vamos a realizar lo mismo, pero en 3D:

CÓDIGO

```
public class Main {
    public static void main(String[] args) {
        int[][][] matriz = new int[3][3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                for (int k = 0; k < 3; k++) {
                    matriz[i][j][k] = i + 1 + j * 3;
                }
            }
        }

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                for (int s = 0; s < 3; s++) {
                    System.out.print(matriz[s][j][i] + " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Si representamos la **matriz** visualmente, lo haremos de la siguiente forma:



## 9.4. Cadena de caracteres. *String*

El lenguaje Java no cuenta con ningún tipo de dato específico para almacenar y procesar las cadenas de caracteres. Por lo que pueden utilizarse los arrays de caracteres aunque, a veces, resulten un poco complicados.

También tenemos la posibilidad de recurrir a diferentes clases que se han diseñado para poder utilizar cadenas y cuentan con métodos que nos permiten trabajar con ellas (clase *String*). La clase *String* es una de las más utilizadas en aplicaciones Java.

**Los Strings son objetos** que pueden ser utilizados para representar caracteres y números. Esto significa que todas las instancias de *String* creadas de un programa Java tienen acceso a los métodos descritos dentro de dicha clase.

Para declarar estas variables de tipo cadena de caracteres, tenemos que instanciar la clase *String* y podemos hacerlo de las siguientes formas:

### CÓDIGO

```
public static void main(String[] args) {
    //Declaración de un array que pasaremos a String
    char[] array = new char[] {'l', 'i', 't', 'e', 'r', 'a', 'l'};
    //Formas de declarar un String
    String forma1 = new String ("literal_cadena_caracteres");
    String forma2 = "literal_cadena_caracteres";
    String forma3 = new String (array);
    String forma4 = new String (forma2);
}
```

Entre las posibles formas que tenemos de declarar un *String*, la más eficiente es la forma en la que se iguala el *String* a su nuevo valor. Esto es debido a que Java guarda una zona especial de la memoria llamada *String Pool*, donde estarán todos los *Strings* creados con anterioridad. Si ya existe el valor deseado, no nos creará otro objeto, lo que nos permitirá ahorrar memoria. Si en cambio utilizamos el operador *new*, en todo momento estaremos creando un nuevo objeto del tipo *String* sin mirar si este ya existe en el pool de *Strings*.

Estos son algunos de los métodos más frecuentes de variables tipo String:

- **char charAt (int indice):** devuelve el carácter que se encuentra en la posición de índice.
- **int compareTo (String cadena):** compara una cadena.

Devuelve:

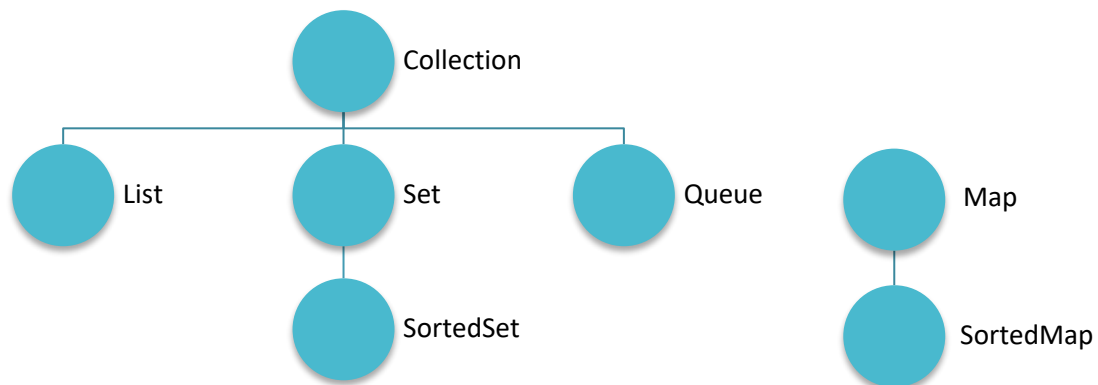
1. Un número entero menor que cero → si la cadena es menor
  2. Un número entero mayor que cero → si la cadena es mayor
  3. Cero → si las cadenas son iguales
- **int compareToIgnoreCase (String cadena):** compara dos cadenas (igual que el anterior), pero no diferencia entre mayúsculas y minúsculas.
  - **Boolean equals (Object objeto):** devuelve *True* si el objeto que se pasa por parámetro y el String son iguales. Si no, devuelve *False*.
  - **int indexOf (int carácter):** devuelve la posición de la primera vez que aparece el carácter en la cadena de caracteres. Como el carácter es de tipo entero, se debe introducir el valor del carácter correspondiente en código ASCII.
  - **boolean isEmpty ():** si la cadena es vacía, devuelve *True*, es decir, si su longitud es cero.
  - **int length ():** devuelve el número de caracteres de la cadena.
  - **String replace (char caracterAntiguo, char caracterNuevo):** devuelve una cadena que reemplaza el valor de *carácterAntiguo* por el valor del *carácterNuevo*.
  - **String [] Split (String expresión):** devuelve un array de String con los elementos de la cadena expresión.
  - **String toLowerCase ():** devuelve un array en el que aparecen los caracteres de la cadena que hace la llamada al método en minúsculas.
  - **String toUpperCase ():** devuelve un array en el que aparecen los caracteres de la cadena que hace la llamada al método en mayúsculas.
  - **String trim ():** devuelve una copia de la cadena, pero sin los espacios en blanco.
  - **String valueOf (tipo variable):** devuelve la cadena de caracteres que resulta al convertir la variable del tipo que se pasa por parámetro.

## 9.5. Colecciones e Iteradores

Una **Colección** representa un grupo de objetos (elementos) que se pueden recorrer (o iterar) y de los que se puede saber el tamaño. Dentro de las colecciones están las anteriormente vistas: conjuntos, listas, colas o pilas.

A partir de la interfaz **Collection** se extienden otras interfaces, imponiendo más restricciones y dando más funcionalidades. Según queramos usar conjuntos, listas, colas, pilas, listas ordenadas, etc., tendremos que emplear una interfaz u otra.

En el siguiente esquema vamos a ver las diferentes interfaces de **Collection**:



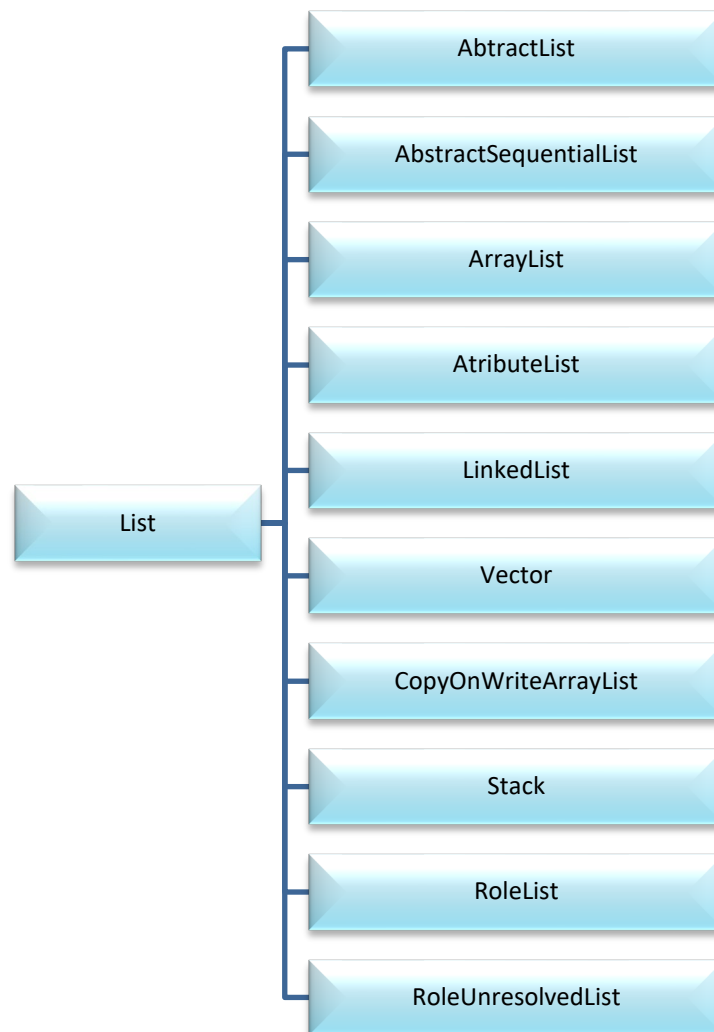
- **List:** pueden estar repetidos, están indexados con valores numéricos
- **Set:** permite almacenar una colección de elementos no repetidos y sin ordenar
- **Queue:** no permite el acceso aleatorio y solo permiten acceder a los objetos del principio o del final
- **Map:** no es un tipo de *Collection*, pero permite crear una colección de elementos repetibles indexados por clave única – valor

A partir de estas interfaces, tenemos una serie de clases que podemos usar, por ejemplo, para tratar con elementos ordenados en las listas. A continuación profundizamos en ello.

- **List**

Las colecciones tipo *List* contienen una agrupación de elementos que pueden ser añadidos en el inicio, al final o en cualquier punto. Sus elementos pueden estar duplicados.

A continuación, vamos a ver un esquema con todas las clases que heredan de List:



- **ArrayList**

Implementa una lista de elementos mediante un array de tamaño variable. El array tendrá un tamaño inicial y se irá incrementando cuando se rebase este tamaño inicial. Se adapta a un gran número de escenarios.

La declaración de la colección tiene la siguiente sintaxis:

#### CÓDIGO

```
//Instancia de tipo Genérico (Cuando no sabemos qué tipo de datos
vamos a introducir en la colección):

ArrayList nombre=new ArrayList ();

//Instancia de colección con tipo específico:

ArrayList <Tipo_de_dato>nombre=new ArrayList<>();
```

En esta colección, los métodos más importantes son los que citamos a continuación:

- **get (int):** obtiene el objeto que se encuentra en la posición pasada por parámetro.
- **indexOf (Object):** devuelve la posición en la que se encuentra el objeto que se pasa por parámetro.
- **isEmpty ():** devuelve un *booleano* que indica si el array está vacío o no.
- **set (int, Objeto):** sobrescribe el elemento que se encuentra en la posición *int* por un objeto que se indica por parámetro.
- **toArray():** devuelve una lista de objetos y, en cada casilla de esta lista, inserta un objeto de *ArrayList*.



A continuación, vamos a ver un ejemplo práctico:

#### CÓDIGO

```
import java.util.ArrayList;

public class Ejemplo {

    public static void main(String[] args) {

        //Instancia de tipo Genérico

        ArrayList array = new ArrayList ();

        array.add(2); //Valor entero
        array.add(4.3); //Valor decimal
        array.add("Ilerna"); //Valor texto

        //Recorremos la colección con el método size()
        for(int i = 0; i < array.size(); i++) {

            System.out.println(array.get(i));

        }

        //Instancia de tipo específico
        ArrayList<Integer> arrayEnteros = new ArrayList<Integer>();

        arrayEnteros.add(2);
        arrayEnteros.add(3);
        arrayEnteros.add(4);

        //Mostrar todo el contenido con el método toString()
        System.out.println(arrayEnteros.toString());

    }

}
```

- **LinkedList**

Se implementa la lista mediante otra lista doblemente enlazada. Cuando realicemos operaciones (inserción, borrado o lectura) en los extremos de la lista, el rendimiento será constante; mientras que, para cualquier operación en la que necesitemos localizar un elemento dentro de la lista, el tiempo será lineal con el tamaño de la lista, ya que esta se tendrá que recorrer de principio a fin.

La declaración de la colección tiene la siguiente sintaxis:

#### CÓDIGO

```
//Instancia de tipo Genérico (Cuando no sabemos qué tipo de datos
vamos a introducir en la colección):

LinkedListnombre=newLinkedList();

//Instancia de colección con tipo específico:

LinkedList<Tipo_de_dato>nombre=newLinkedList<Tipo_de_dato>();
```

Algunos de sus métodos más importantes son:

- **removeFirst():** elimina el primer elemento de la lista enlazada.
- **addFirst():** añade un elemento al principio de la lista.
- **addLast():** añade un elemento al final de la lista.
- **getFirst():** devuelve el primer elemento de la lista.
- **getLast():** devuelve el último elemento de la lista.

Ahora vamos a ver un ejemplo de la clase *LinkedList*. Esta clase hereda de la *interfaceList*:

## CÓDIGO

```
import java.util.LinkedList;

public class Ejemplo {

    public static void main(String[] args) {

        //Instancia de tipo Genérico
        LinkedList listaEnlazada = new LinkedList();

        listaEnlazada.add(3);
        listaEnlazada.add(4.52);

        listaEnlazada.add("Amaia");
        System.out.println(listaEnlazada);
        listaEnlazada.removeFirst();
        listaEnlazada.addFirst("Laura");
        listaEnlazada.addLast(72);
        System.out.println(listaEnlazada);

        //Instancia de tipo específico
        LinkedList<String> listaEnl = new LinkedList<String>();
        listaEnl.add("Pablo");

        listaEnl.add("Carlos");
        listaEnl.add("Ruben");

        System.out.print(listaEnl.getFirst() + " ");
        System.out.println(listaEnl.getLast());
    }
}
```

- **Vector**

El objeto del tipo vector es muy parecido al de ArrayList, aunque dispone de una mayor cantidad de métodos. Dispone de un array de objetos que puede aumentar o disminuir de forma dinámica, según las operaciones que se vayan a llevar a cabo.

La declaración de la colección tiene la siguiente sintaxis:

#### CÓDIGO

```
//Instancia de tipo Genérico (Cuando no sabemos qué tipo de datos
vamos a introducir en la colección):
Vectornombre=newVector();
//Instancia de colección con tipo específico:
Vector<Tipo_de_dato>nombre=newVector<Tipo_de_dato>();
```

Algunos de sus métodos más importantes son:

- **firstElement ()**: devuelve el primer elemento del vector.
- **lastElemento ()**: devuelve el último elemento del vector.
- **capacity ()**: devuelve la capacidad del vector.
- **setSize (int)**: elige un nuevo tamaño para el vector. En el caso de que sea más grande que el que tenía en un principio, inicializa a *null* los nuevos valores. En el caso de que sea menor que el inicial, elimina los elementos restantes.

Vamos a ver un ejemplo práctico de la clase *vector*:

## CÓDIGO

```
import java.util.Vector;

public class Ejemplo {
    public static void main(String[] args) {
        //Instancia de tipo genérico
        Vector vector = new Vector();
        vector.add(3);
        vector.add(5.8);
        vector.add("Ilerna");

        //Mostramos el contenido con el método elementAt()
        System.out.println(vector.elementAt(0));
        System.out.println(vector.elementAt(1));
        System.out.println(vector.elementAt(2));

        //Instancia de tipo específico
        Vector<String> vectorCadenas = new Vector<>();
        vectorCadenas.add("Ilerna");
        vectorCadenas.add("Online");

        //Mostramos el contenido con firstElement y lastElement
        System.out.print(vectorCadenas.firstElement()+" ");
        System.out.print(vectorCadenas.lastElement());
    }
}
```

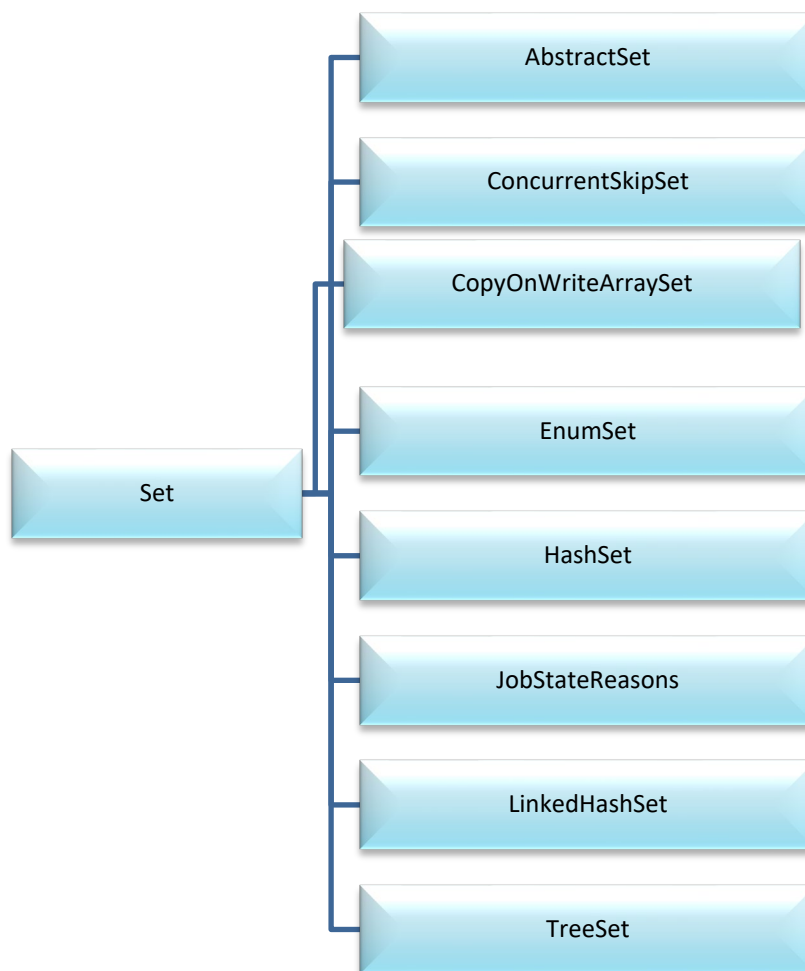
En este apartado hemos comentado las colecciones y métodos más utilizados. Para obtener más información de todas las colecciones, adjuntamos el enlace web a la plataforma de Java, donde explica la colección List y contiene los enlaces a todas las clases que hemos visto en el esquema anterior:

<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>

- **Set**

*Set* es la palabra inglesa para conjunto. La colección *Set* agrega una sola restricción, no puede haber duplicados. Por lo general, en un *Set* el orden no es dato, aunque existen algunas clases heredadas de *Set* que proporcionan una ordenación. No obstante, la interfaz *Set* no tiene métodos para manipular esta funcionalidad. La ventaja de utilizar *Sets*, es preguntar si un elemento está ya en la colección mediante el método *contains()* y es muy eficiente.

A continuación, mostramos el esquema de las clases que heredan de la interfaz *Set*:



- **HashSet**

Permite almacenar los datos en una tabla de dispersión (*hash*). Es rápida en cuanto a operaciones básicas (inserción, borrado y búsqueda) y no admite duplicados. La iteración a través de sus elementos es más costosa, ya que, necesitará recorrer todas las entradas de la tabla y la ordenación puede diferir del orden en el que se han insertado los elementos.

La declaración de la colección tiene la siguiente sintaxis:

#### CÓDIGO

```
//Instancia de tipo Genérico (Cuando no sabemos qué tipo de datos
vamos a introducir en la colección):

HashSetnombre=newHashSet () ;

//Instancia de colección con tipo específico:

HashSet<Tipo_de_dato>nombre=newHashSet<Tipo_de_dato>() ;
```

Algunos de sus métodos más importantes son:

- **isEmpty():** devuelve si el conjunto está vacío o contiene valores con un valor booleano.
- **clone():** devuelve una copia superficial de esta instancia de *HashSet*: los elementos en sí no están clonados.
- **clear():** borra todos los elementos del conjunto.

En el siguiente ejemplo de la clase *HashSet* vamos a ver la implementación de algunos de sus métodos. Este ejemplo es muy sencillo y algunos de los métodos se aplican solo para ver su funcionamiento, sin tener en cuenta el rendimiento del programa:

## CÓDIGO

```
import java.util.HashSet;

public class Ejemplo {

    public static void main(String[] args) {

        //Instancia de tipo Genérico
        HashSet colSet = new HashSet();

        colSet.add(3);
        colSet.add(5.8);

        colSet.add("Ilerna");

        //Como utilizar los métodos isEmpty() y el método clear()
        while(!colSet.isEmpty()){
            System.out.println(colSet);
            colSet.clear();
        }

        //Instancia de tipo específico
        HashSet<Double> colSetInt = new HashSet<Double>();

        colSetInt.add(23.10);
        colSetInt.add(32.00);
        colSetInt.add(83.24);

        //Recorrer la colección con un bucle for each
        for(Double s : colSetInt){
            System.out.println(s);
        }
    }
}
```



- **TreeSet**

Permite almacenar los datos en un árbol. Por lo tanto, el coste para realizar las operaciones básicas será logarítmico, con el número de elementos que tenga el conjunto.

En este marco de trabajo de colecciones también podemos encontrar la interfaz *SortedSet* (extendida de *Set*), que puede ser utilizada en los diferentes conjuntos que tienen sus elementos en orden. La clase *TreeSet* va a implementar la interfaz *SortedSet*.

La declaración de la colección tiene la siguiente sintaxis:

#### CÓDIGO

```
//Instancia de tipo Genérico (Cuando no sabemos qué tipo de datos
vamos a introducir en la colección):
TreeSet nombre=new TreeSet ();

//Instancia de colección con tipo específico:
TreeSet<Tipo_de_dato> nombre=new TreeSet<Tipo_de_dato>();
```

Algunos de sus métodos más importantes son:

- **first():** devuelve el primer elemento actual (el más bajo) en este conjunto.
- **last():** devuelve el último elemento actual (el más alto) en este conjunto.
- **floor():** devuelve el elemento más grande en este conjunto, menor o igual que el elemento dado o nulo, si no hay dicho elemento.
- **tailSet():** devuelve una vista de la parte de este conjunto, cuyos elementos son mayores que o iguales al elemento pasado por parámetro.

Ahora, vamos a ver un ejemplo muy sencillo de la clase *TreeSet*, donde aplicaremos algunos de sus métodos:

## CÓDIGO

```
import java.util.TreeSet;

public class Ejemplo {

    public static void main(String[] args) {

        //Instancia de tipo Genérico
        TreeSet arbolPersonas = new TreeSet();

        arbolPersonas.add(3);
        arbolPersonas.add(45);
        arbolPersonas.add(72);

        //Vemos el funcionamiento del método tailSet()
        System.out.println(arbolPersonas.tailSet(1));

        System.out.println(arbolPersonas.tailSet(20));

        System.out.println(arbolPersonas.tailSet(50));

        //Instancia de tipo específico
        TreeSet<String> arbolPer = new TreeSet<String>();

        arbolPer.add("Sandra");

        arbolPer.add("Amanda");

        arbolPer.add("Diana");

        //Recorremos los resultados con un bucle for each
        for (String s : arbolPer) {

            System.out.println(s);

        }

    }

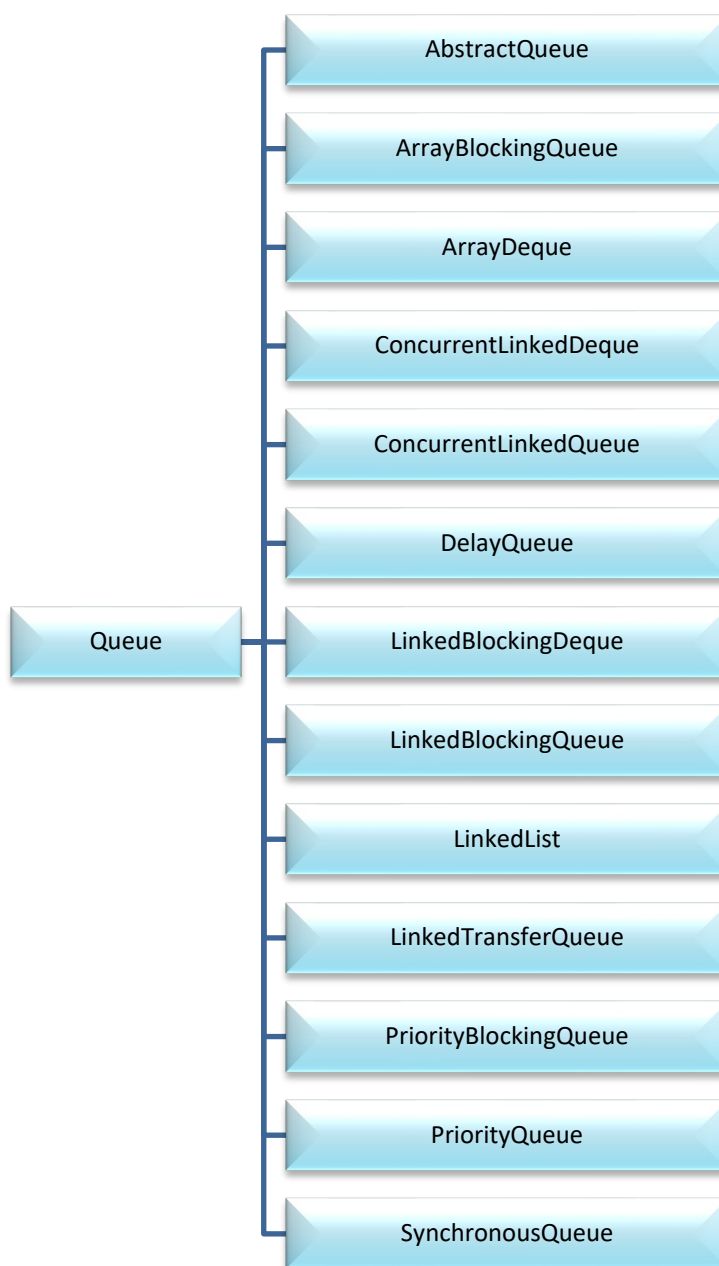
}
```

En este apartado hemos comentado las colecciones y métodos más utilizados. Para obtener más información de todas las colecciones, adjuntamos el enlace web a la plataforma de Java donde explica la colección List y contiene los enlaces a todas las clases que hemos visto en el esquema anterior:

<https://docs.oracle.com/javase/8/docs/api/java/util/Set.html>

- **Queue**

Se conoce como cola una colección especialmente diseñada para ser usada como almacenamiento temporal de objetos a procesar. Las colas siguen un patrón que en computación es conocido como FIFO (*First in – First out*), lo que entra primero, sale primero. También se crearon clases *Deque*, que representan una *double-ended-queue*, es decir, una cola en la que los elementos pueden añadirse no solo al final, sino también empujarse al principio.



- **ArrayDeque**

Esta clase son una pila y una cola optimizadas. Se usa en analizadores y cachés. Su rendimiento es excelente y versátil. Los cambios de estado se basan en la última etiqueta encontrada, primero tratamos con los elementos más profundos de esta pila. Tiene optimizaciones respecto a colecciones más antiguas como Stack.

La declaración de la colección tiene la siguiente sintaxis:

#### CÓDIGO

```
//Instancia de tipo Genérico (Cuando no sabemos qué tipo de datos
vamos a introducir en la colección):
ArrayDeque<nombre>=new ArrayDeque();

//Instancia de colección con tipo específico:
ArrayDeque<Tipo_de_dato>nombre=new ArrayDeque<Tipo_de_dato>();
```

Algunos de sus métodos más importantes son:

- **push():** añade un elemento al principio de la cola
- **pop():** elimina el elemento de la cola que se ha insertado primero
- **pollFirst():** elimina el primer elemento de la cola
- **pollLast():** elimina el ultimo elemento de la cola

A continuación, vamos a ver un ejemplo muy sencillo de la clase *ArrayDeque*, donde aplicaremos algunos de sus métodos propios:

## CÓDIGO

```
import java.util.ArrayDeque;

public class Ejemplo {
    public static void main(String[] args) {
        //Instancia de tipo Genérico
        ArrayDeque cola = new ArrayDeque();
        cola.add("primer elemento");
        cola.add(2);
        cola.add("tercer elemento");
        cola.add(4);

        System.out.println(cola);
        cola.pollFirst();
        System.out.println(cola);
        cola.pollLast();
        System.out.println(cola);
        //Instancia de tipo específico
        ArrayDeque<Integer> pila = new ArrayDeque<Integer>();
        pila.add(1);
        pila.add(2);
        pila.add(3);
        pila.add(4);

        System.out.println(pila);
        pila.push(0);
        System.out.println(pila);
        pila.pop();
        System.out.println(pila);
    }
}
```

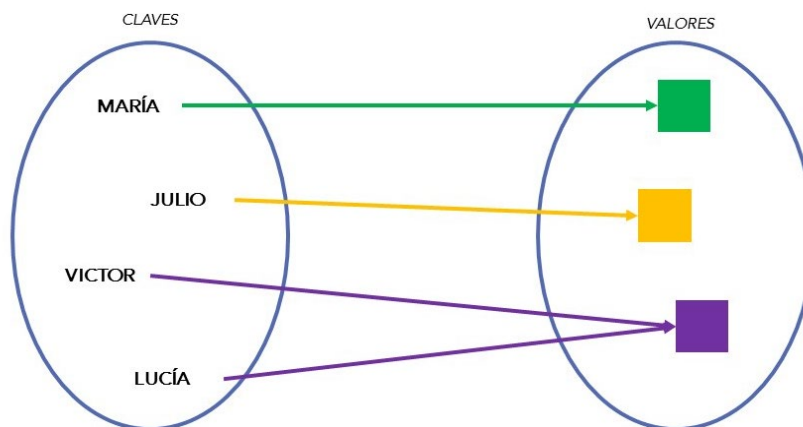
En este apartado hemos comentado las colecciones y los métodos más utilizados, para más información de todas las colecciones, adjuntamos el enlace web a la plataforma de Java donde explica la colección Queue y contiene los enlaces a todas las clases que hemos visto en el esquema anterior:

<https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html>

- **Map**

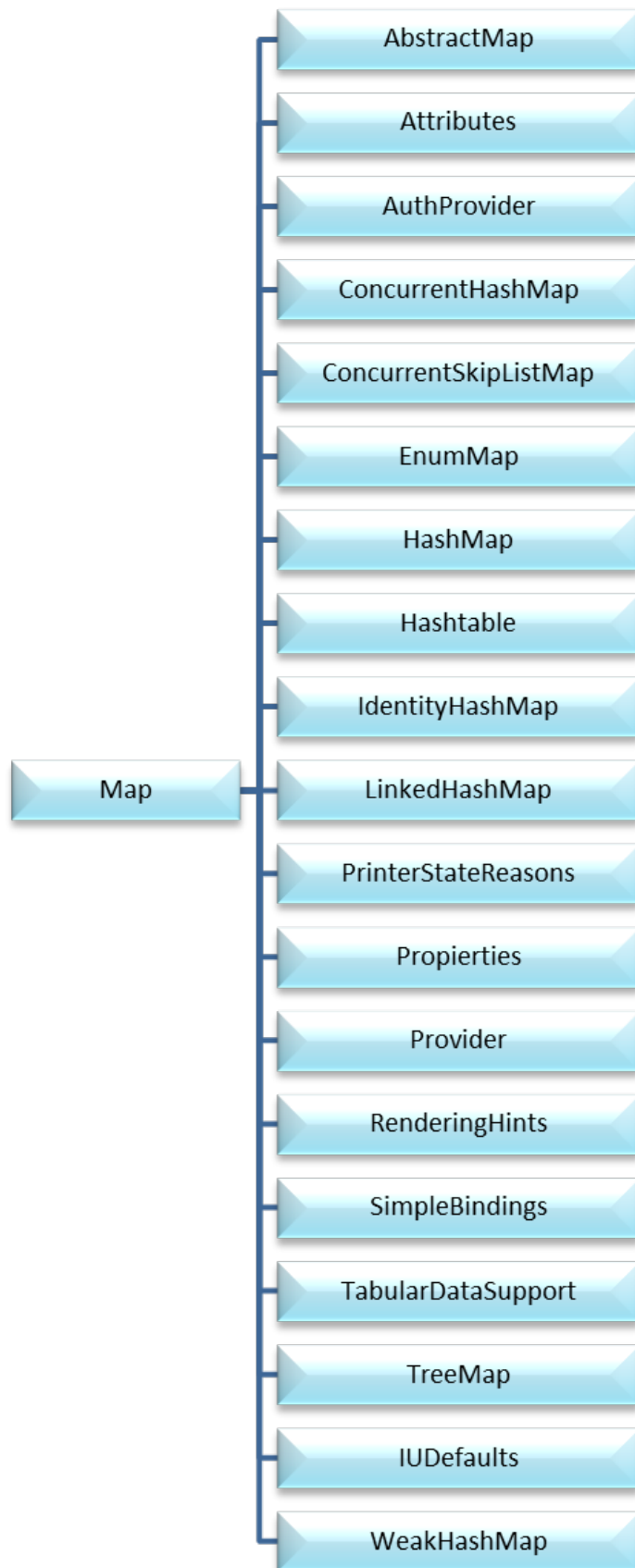
Un *Map* es un conjunto de valores con un objeto extra asociado para cada uno de ellos. A los primeros se los llama **claves o keys**, porque nos permiten acceder a los segundos. Los valores clave no aceptan valores duplicados.

Map no es una interfaz que hereda de Collection, ya que podríamos decir que Collection es una interfaz unidimensional y Map es una interfaz bidimensional.



Algunos de sus métodos más importantes son:

- **get (Object):** obtiene el valor correspondiente a una clave. Devuelve *null* si la clave no existe en el Map
- **put (clave, valor):** añade un par clave-valor al Map. Si ya había un valor para esa clave, lo reemplaza
- **keySet():** devuelve todas las claves (devuelve un Set, es decir, sin duplicados)
- **values():** todos los valores (en este caso sí pueden estar duplicados)
- **entrySet():** todos los pares clave-valor (devuelve un conjunto de objetos Map.Entry, cada uno de los cuales devuelve la clave y el valor con los métodos getKey() y getValue() respectivamente)



Para añadir una asociación (clave, valor):

CÓDIGO

```
coloresPreferidos.put(ILERNA, Color.AZUL);
```

Para cambiar el valor de una asociación (clave, valor):

CÓDIGO

```
coloresPreferidos.put(ILERNA, Color.AZUL);
```

Si queremos devolver un valor asociado a una determinada clave:

CÓDIGO

```
Color colorPreferido = coloresPreferidos.get(ILERNA);
```

Si lo que deseamos es borrar una clave y su valor:

CÓDIGO

```
ColoresPreferidos.remove(ILERNA);
```



- **HashMap**

Permite almacenar los datos en una tabla de dispersión (*hash*). ¿Pero qué es una tabla hash? Es una tabla que, mediante una función matemática, permite establecer la posición donde se guardará el elemento. Dicha función, dependiendo del objeto que se desea guardar, nos indicará su posición exacta. Si dos elementos coinciden en el mismo índice, se deberá de buscar una forma óptima, una posición libre. Para ello, existen diferentes opciones como, por ejemplo, mirar la siguiente posición hasta encontrar una libre.

Un *HashMap* es una colección rápida en cuanto a operaciones básicas (inserción, borrado y búsqueda), no admite duplicados, la iteración a través de sus elementos es más costosa, ya que necesitará recorrer todas las entradas de la tabla y la ordenación puede diferir del orden en el que se han insertado los elementos.

La declaración de la colección tiene la siguiente sintaxis:

#### CÓDIGO

```
//Instancia de tipo Genérico (cuando no sabemos qué tipo de
datos vamos a introducir en la colección):

HashMap nombre=new HashMap();

//Instancia de colección con tipo específico:
HashMap<Tipo_de_dato_clave, Tipo_de_dato_valor> nombre;
nombre=new HashMap< Tipo_de_dato_clave, Tipo_de_dato_valor>();
```

A continuación, encontramos un ejemplo de la clase HashMap, donde aplicaremos algunos de sus métodos:

#### CÓDIGO

```
import java.util.Map;
import java.util.HashMap;

public class Ejemplo {
    public static void main (String[] args) {
        //Instancia de tipo genérico
        HashMap alumno = new HashMap();
        alumno.put(1, 15.55);
        alumno.put(2.2, 19);
        alumno.put("3", "María");
        System.out.println(alumno);
        //Eliminar un elemento
        alumno.remove("3");
        System.out.println(alumno);
        //Sustituir un elemento
        alumno.put("1", "Marc");
        System.out.println(alumno);
        //Instancia de tipo específico
        HashMap<Integer, String> alum;
        alum = new HashMap<Integer, String>();
        alum.put(1, "Joan");
        alum.put(2, "Sara");
        alum.put(3, "Lola");
        //Recorremos la colección con entrySet()
        for (Map.Entry<Integer, String> ent : alum.entrySet()) {
            System.out.print("Clave: " + ent.getKey() + " ");
            System.out.println("Valor: " + ent.getValue());
        }
    }
}
```

En este apartado, comentamos las colecciones y algunos de los métodos más utilizados. Para más información de todas las colecciones adjuntamos el enlace web a la plataforma de Java donde explica Map y contiene los enlaces a todas las clases que hemos visto en el esquema de Map:

<https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>

Cuando trabajamos con cualquiera de las colecciones comentadas, tendremos que importar su librería correspondiente en el archivo .java, tal y como vemos en los ejemplos. `import java.util.*;`

- **Iterator**

Si necesitamos recorrer todos los elementos para acceder a uno de ellos, podemos hacerlo mediante la utilización del iterador (*Iterator*). Un iterador es un componente existente en todas las colecciones que, independientemente del tipo que sean, nos permite recorrerlas utilizando un mismo mecanismo. Así, facilita en gran medida la programación y nos permite tener una gran simplificación de nuestro código, evitando tener que adaptar el recorrido para cada una de las colecciones y tipos utilizados.

Las colecciones tienen el método *Iterator()* que, tal y como hemos visto en el apartado de estructuras avanzadas, lo heredan de la interfaz padre *Collection*, método que va a permitir crear un iterador con los datos de *Collection*.

Este iterador nos proporcionará unos métodos propios, que facilitarán recorrer este objeto *Collection*. A continuación, vemos los métodos más utilizados:

- **next ()**: devuelve el siguiente elemento en la iteración
- **hasNext ()**: devuelve verdadero, si la iteración tiene más elementos. En caso contrario, devuelve falso.
- **remove()**: elimina de la colección subyacente el último elemento devuelto por este iterador.

## CÓDIGO

```

Iterator <String> iterador =nombre.iterator();

while (iterador.hasNext()){

    String nombre =iterador.next();

    //código bucle while

}

//Con las listas podemos utilizar un bucle for mejorado

for(String.nombre : nombres){

    //código bucle for each

}

```

Ahora vamos a ver un ejemplo muy sencillo, en el que aplicamos un iterador a una colección tratada en los puntos anteriores. El ejemplo será el mismo visto anteriormente, pero vamos a mostrar los resultados recorriendo la colección con un iterador:

## CÓDIGO

```

import java.util.LinkedList;

public class Ejemplo {

    public static void main(String[] args){

        //Instancia de tipo específico

        LinkedList<String> listaEn =new LinkedList<>();

        listaEn.add("Maria");

        listaEn.add("Carlos");

        listaEn.add("Marc");

        listaEn.add("Lucia");

        Iterator<String> it = listaEn.iterator();

        while(it.hasNext()){

            System.out.println(it.next().toString());

        }

    }

}

```

Para profundizar más en la interfaz de iteradores, adjuntamos el link del Javadoc que explica más detalladamente su funcionamiento y todos sus métodos:

<https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html>

## 9.6. Clases y métodos genéricos

Cuando trabajamos en Java es habitual conocer el tipo de dato con el que estamos trabajando: *String*, *Alumno*, *Coche*, *Integer*, etc. Sin embargo, es posible que queramos crear alguna clase o método genérico, de forma que no sepamos previamente el tipo de dato con el que vamos a trabajar. A esto es a lo que llamamos clase y métodos genéricos.

Los métodos genéricos fueron introducidos en la versión 5 de Java en 2004, lo que supuso una de las mayores modificaciones de su historia.

Los métodos genéricos son importantes, ya que permiten al compilador informar de muchos errores de compilación que hasta el momento solo se descubrirían en tiempo de ejecución. Al mismo tiempo permiten eliminar los *cast* (casting o conversiones) simplificando, reduciendo la repetición y aumentando la legibilidad del código. Los errores por realizar castings inválidos (tipos que no pueden ser convertidos a otros) son especialmente problemáticos de *debuggear*, ya que el error se suele producir en un sitio alejado del de la causa.

Los beneficios son:

- Comprobación de tipos más fuertes en tiempo de compilación
- Eliminación de *casts* aumentando la legibilidad del código
- Posibilidad de implementar algoritmos genéricos, con tipado seguro

## 9.7. Clases genéricas

Las clases genéricas pueden ser utilizadas por cualquier programador que desee utilizar este mecanismo. Su sintaxis debe ser la siguiente:

### CÓDIGO

```
modificador_de_acceso class nombre_clase<T>{
    T variable;
}
```

En esta, “T” representa un tipo de referencia válido en el lenguaje Java: *String*, *Integer*, *Alumno*, *Libro*, *Coche* o cualquier otro tipo. Los parámetros de clases por tipos no se limitan a un único parámetro (T). Por ejemplo, la clase *HashMap* que indicamos a continuación, permite dos parámetros:

### CÓDIGO

```
class Hash <A, B>{
}
```

En este caso, tenemos dos parámetros A y B que hacen referencia al tipo de clave y el valor. También podemos aplicar esta sintaxis en interfaces:

### CÓDIGO

```
public interface nombre_interface<K,V>{
    public K getKey();
    public V getValue();
}
```

Y esta interface podría ser implementada por una clase, donde *K* representa la clave y *V* representa el valor:

## CÓDIGO

```
public class n_clase<K,V> implements n_interface<K,V>{
    private K key;
    private V value;
    public n_clase(K key, V value){
        this.key = key;
        this.value = value;
    }
    public K getKey(){return key;}
    public K getValue(){return value;}
}
```

En el momento de la instanciación de un tipo genérico indicaremos el argumento para el tipo.

A partir de Java 7, aparece el operador *diamond*(<>), en el que el compilador inferirá el tipo (sin necesidad de indicarlo nosotros) según su definición para mayor claridad en el código. Podemos usar cualquiera de estas dos maneras, aunque con preferencia de usar el operador *diamond* por tener mayor claridad.

## CÓDIGO

```
nombre_clase<Integer> intClase = new nombre_clase<Integer>();
nombre_clase<Integer> intClase1 = new nombre_clase<>(); //diamond
n_clase<String, Integer> p1 = new n_clase<>("Evento", 17);
```

Como puede verse en la clase anterior, en la segunda línea de código, cuando creamos el tipo con *new*, no indicamos el tipo. Además de las clases, los métodos también pueden tener su propia definición de tipos genéricos.

## 9.8. Métodos genéricos

Podemos crear métodos para que puedan utilizar los tipos parametrizados, bien sea en las clases genéricas o en las normales. La sintaxis para crear un tipo genérico es:

CÓDIGO

```
publicstatic<T> T metodogenerico (T parametroFormal) {  
    //código método  
}
```

Y, para invocar este método:

CÓDIGO

```
claseDelMetodoGenerico.<TipoConcreto> método (ParametroReal) ;
```

Aunque, en algunos casos, puede que el compilador deduzca qué tipo de parámetro se va a utilizar y, en este caso, se puede obviar:

CÓDIGO

```
ClasedelMetodoGenerico.metodo (parametroReal) ;
```

### Tipos de comodín

Cuando utilizamos un tipo concreto como parámetro (tanto en clases genéricas como en métodos genéricos) se produce mucha restricción. Por eso es conveniente indicar el tipo que se va a utilizar como parámetro para implementar la interfaz.

También es conveniente considerar las restricciones del tipo de la superclase, es decir, este tipo debe ser predecesor en la jerarquía de herencia de un cierto tipo dado.

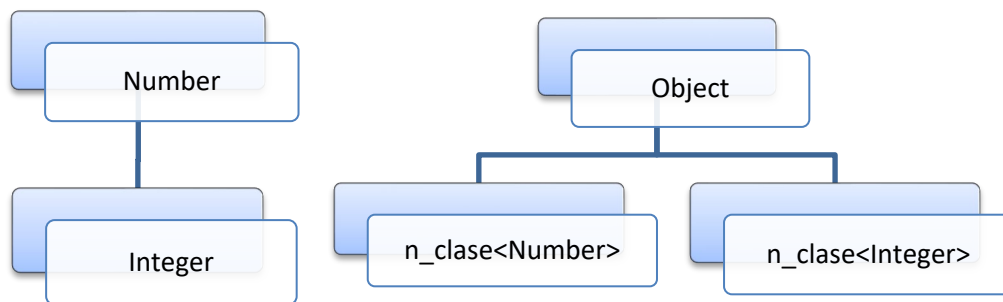


En Java, un tipo puede ser asignado a otro siempre que el primero sea compatible con el segundo; es decir, que tengan una relación uno es a uno. Una referencia de Object puede referenciar una instancia de Integer (un Integer es un Object).

**CÓDIGO**

```
Object objeto=new Object ();
Integer entero=new Integer(10);
objeto=entero;
```

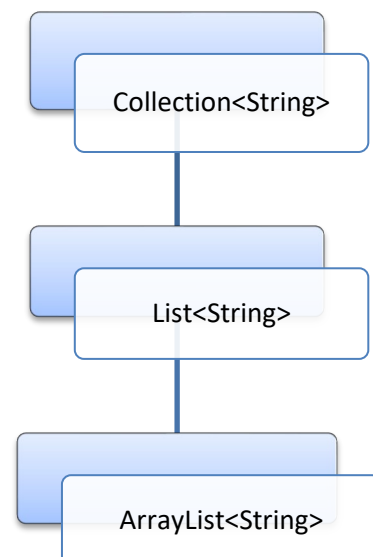
Sin embargo, en el caso de los genéricos, una referencia de *n\_clase<Number>* no puede aceptar una instancia *n\_clase<Integer>* o *n\_clase<Double>*, aun siendo Integer y Double subtipos de Number, porque en Java no son subtipos de *n\_clase<Number>*.



La **jerarquía de tipos** es la siguiente:

Los **tipos genéricos** pueden extenderse o implementarse mientras no se cambie el tipo del argumento. De modo que *ArrayList<String>* es un subtipo de *List<String>*, que a su vez es un subtipo de *Collection<String>*.

Los **tipos comodín** son usados para reducir las restricciones de un tipo, de modo que un método pueda funcionar con una lista de *List<Integer>*, *List<Double>* y *List<Number>*.



El término `List<Number>` es más restrictivo que `List<? extends Number>`, porque el primero solo acepta una lista de `Number` y el segundo una lista de `Number` o de sus subtipos.

#### CÓDIGO

```
public static void process (List<? extends Number> list) { /* ... */ }
```

Se puede definir una lista de un tipo desconocido, `List<?>`, en los casos en los que:

- La funcionalidad se puede implementar usando un tipo `Object`
- Cuando el código usa métodos que no dependen del tipo de parámetro. Por ejemplo, `List.size` o `List.clear`

## 9.9. Expresiones regulares de búsqueda

En la versión 1.4 del compilador de Java aparecieron las **expresiones regulares** para facilitar mediante patrones el tratamiento de las cadenas de caracteres. El JDK de Java incluye un paquete **`java.util.regex`** donde se puede trabajar con los métodos disponibles en la API.

Las expresiones regulares se rigen por una serie de caracteres para la construcción del patrón a seguir. Las expresiones regulares solo pueden contener letras, números o los siguientes caracteres:

```
. < $ ^ , . * , + , ? , [ , ] , . >
```

Los ejemplos de tratamientos de cadenas de caracteres con patrones pueden ser, por ejemplo, la búsqueda de una cadena de caracteres que empiecen por una determinada letra. El hecho de validar un correo electrónico también se puede implementar mediante expresiones regulares, lo cual nos ahorrará muchas líneas de código y comprobaciones redundantes que se pueden simplificar a una comprobación mediante una expresión regular.

Para hacer uso de las expresiones debemos conocer la clase más importante del paquete *regex*. Es la clase **Matcher** y la clase **Pattern**, con su excepción *PatternSyntaxException*. La clase *Matcher* representa la expresión regular que debe estar compilada, es decir, el patrón.

**Para crear el patrón se debe crear un objeto *Matcher* e invocar al método *Pattern*.** Una vez realizado este paso ya podemos hacer uso de las operaciones disponibles.

Podemos ver un ejemplo del método *compile*. Este método se utiliza para crear el patrón, en este caso mediante la palabra “camion”:

CÓDIGO

```
Pattern patron = Pattern.compile("camion");
```

Una vez creado el patrón, mediante la clase *Matcher*, podemos comprobar distintas cadenas contra el patrón anterior:

CÓDIGO

```
Matcher encaja = patron.matcher();
```

Podemos ver un ejemplo explicativo en el que vamos a crear un método *replaceAll* para sustituir todas las apariciones que concuerden con la cadena “aabb” por la cadena “..”:

## CÓDIGO

```
// se importa el paquete java.util.regex
import java.util.regex.*;
public class EjemploReplaceAll {
    public static void main (String[] args) {
        // Creamos el patrón
        Pattern patron = Pattern.compile("aabb");
        // creamos el Matcher a partir del patrón, la cadena como parámetro
        Matcher encaja = patron.matcher("aabmaabbnoloaabbmbmanoloabmolob");
        // invocamos el metodo replaceAll
        String resultado = encaja.replaceAll("..");
        System.out.println(resultado);
    }
}
```

La salida de este ejemplo será: “aabm..nolo..bmanoloabmolob”. Aunque se pueden realizar operaciones más complejas, como por ejemplo, la comprobación de un DNI.

## CÓDIGO

```
public boolean comprobarDNI (String dni) {
    String regex = "^[0-9]{8}-?[a-zA-Z]{1}$";
    Pattern pattern = Pattern.compile(regex);
    return pattern.matcher(dni).matches();
}
```

**Para ampliar**

Podemos ver todos los métodos disponibles en la siguiente API de JAVA:

<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>

## 10. Control de excepciones

Las **excepciones** son fragmentos de código que se diseñan para tener en cuenta los posibles errores que pudieran surgir durante la ejecución de un programa.

Cuando estamos desarrollando un programa vamos escribiendo código que iremos compilando para ver si existe algún error. En caso de que existan errores, debemos corregirlos para poder seguir adelante. **Estos errores se denominan errores de compilación. Por otro lado, los errores que se muestran durante el tiempo de ejecución se denominan errores de excepción.**

Cuando se producen errores en tiempo de ejecución y no se controlan, el programa finaliza de una manera brusca.

En este apartado vamos a profundizar la forma de asegurarnos, pese a estos errores, de que el programa funciona de una forma correcta y, en el caso de que presente errores, controlarlos de alguna forma para que todo pueda seguir funcionando.

A partir de ahora, se van a diseñar aplicaciones de tal manera que, si el código presenta una excepción, esta se va a tratar en otra zona aparte del código fuente, siempre que la excepción se haya nombrado de alguna forma.

Vamos a ver un esquema con los posibles tipos de errores que tenemos en Java:

Clase general, de donde heredan todos los posibles tipos de error.

Error

Dividimos los errores en dos subtipos, compilación que serían errores de sintaxis del código (no escribimos bien el código) y errores de ejecución.

Tiempo de compilación  
(Sintaxis)

Tiempo de ejecución

La clase Throwable es la base que representa a todas las excepciones que pueden ocurrir en un programa Java.

Throwable

**Error:** Se refiere a errores graves en la máquina virtual de Java. Estos tipos no se tratan.

**Exception:** Representan errores que son críticos, los cuales pueden ser tratados para continuar con la ejecución del programa.

Error

Exception

## 10.1. Captura de excepciones

La captura de excepciones se lleva a cabo en el lenguaje Java mediante los bloques `try ... catch`. Cuando tiene lugar una excepción, la ejecución del bloque `try` termina.

La palabra `catch` recibe como argumento un objeto `Throwable`. El objeto correspondiente a la `Exception` que ha causado el error mientras se ejecuta el código.

Veamos un ejemplo de este tipo de bloques:

```
CÓDIGO

try{
    //Código que puede lanzar una excepción
}catch(Exception error){
    //Código que se ejecutara en caso de error
}
```

Pero este bloque no termina aquí, también tiene una tercera parte llamada *finally*. Aunque esta es totalmente opcional y no afecta a nuestro programa, dicho bloque se ejecutará siempre, sin importar si se ha producido o no la excepción.

Por lo que finalmente, el código quedaría estructurado de la siguiente manera:

- **Bloque try:** fragmento de código que se va a intentar ejecutar, esperando que no se produzca ningún error. En caso de que se produzca un error no continúa con su ejecución
- **Bloque catch:** fragmento de código que se va a ejecutar siempre que se produzca un error en el bloque try
- **Bloque finally:** fragmento de código que siempre se ejecuta tanto si se produce como si no se produce la excepción

```
CÓDIGO

try{
    //Código que puede lanzar una excepción
}catch(Exception error){
    //Código que se ejecutara en caso de error
}

finally{
    //Código que se ejecutara siempre
}
```

## 10.2. Captura frente a delegación

En ocasiones, en vez de capturar una excepción, podemos delegarla. Esta delegación consiste en enviar la excepción al método anterior el cual ha llamado al método que actualmente estamos implementando. Para ello, se declara en cabecera de nuestro método la excepción, que se puede producir de la siguiente forma:

### CÓDIGO

```
public String leerFichero(BufferedReader fichero) throws IOException {
    String linea = fichero.readLine();
    return linea;
}
```

Para indicarle al método que debe delegar la excepción y no controlarla, deberá hacerse mediante la palabra *throws*, junto a la firma del método.

En este método de ejemplo, si se produce algún error al leer el archivo, en lugar de capturar la excepción, esta se delega a quien ha llamado al método *leerFichero*, por lo que la excepción será capturada en otra parte del código que no es nuestro propio método.

Para saber cuándo capturar o lanzar una excepción debemos tener en cuenta lo siguiente:

- Podemos recuperarnos del error que se ha producido y seguir con la ejecución
- Queremos registrar un error o mostrarlo (por ejemplo, en un log por consola)
- Queremos lanzar el error, pero con un tipo de excepción distinta (por ejemplo, *MiErrorException*)

Es decir, debe hacerse cuando tenemos que realizar algún tratamiento con el propio error. Sin embargo, deberemos **delegar** la excepción cuando:

- No tenemos por qué hacer nada con el error o no es competencia nuestra
- Sabemos que en la llamada anterior se realiza un tratamiento de este error



### 10.3. Lanzamiento de excepciones

Ya hemos visto cómo se pueden capturar o delegar las diferentes excepciones. Ahora vamos a aprender cómo podemos lanzarlas.

Para lanzar toda excepción (indicar que se ha producido) hay que crear una instancia de esta excepción en la zona correspondiente del código. Esto se hace incluyendo un bloque *try ... catch* y anteponiendo la palabra reservada **throw**. Lo vemos en el siguiente ejemplo:

#### CÓDIGO

```
try{
    //código que se va a ejecutar

    throw new Clase_de_error(mensaje); //lanzamiento de error

} catch (clase_de_error) {

    //código que se va a ejecutar si hay error
}
```

A continuación, vemos un ejemplo en el que simulamos la introducción de un login de usuario (este ejemplo está detallado al final del apartado):

## CÓDIGO

```
staticboolean login (string user, string pass)

//Código del método login
}

publicstaticvoid main (string [] args){
try{

//Bloque de código que en caso de error lanzara una excepción
//que capturara el bloque catch
if(!login(nombre, passwd)){
thrownewErrorLoginException();
}
}catch(ErrorLoginException error){

//Código que se ejecutará si el código del bloque try lanza
//una excepción
System.out.print(error.getMessage());
}finally{

//Código que se va a ejecutar siempre (aunque se entre en
//catch)
}
}
```

Podemos lanzar cualquier tipo de excepción siempre que sigamos la sintaxis anterior, bien sea creada por el usuario o no.

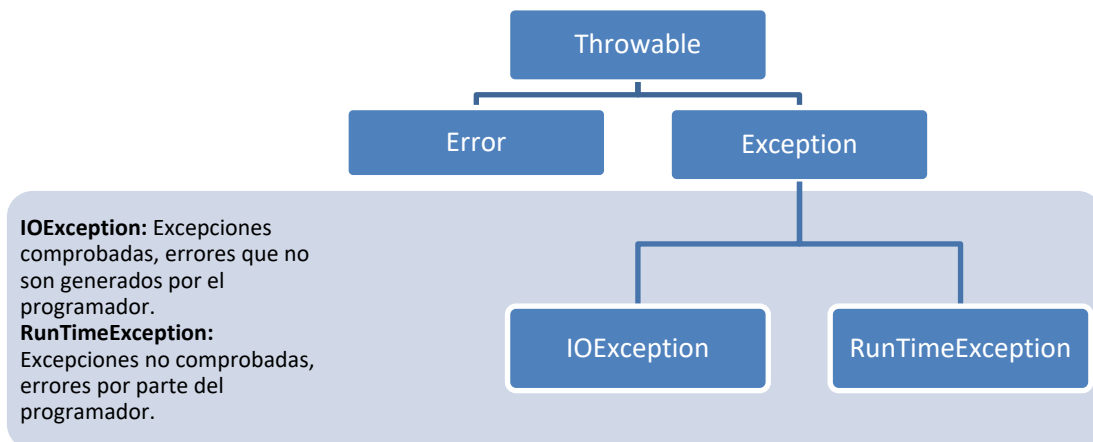
En el siguiente ejemplo se puede ver, como ya vimos anteriormente, una delegación de una excepción. Cuando se delega la excepción, directamente ya la estamos lanzando al método anterior del cual ha sido llamado:

CÓDIGO

```
public static void main (string [] args) throws ErrorLogin,
IOException, NumberFormatException {
    //Código
}
```

## 10.4. Excepciones y herencia

El esquema que presentamos a continuación es complementario al que hemos visto anteriormente (en el punto principal de este apartado). Explica las dos subclases que heredan de Exception.



Hay bastantes tipos de objetos que derivan de la clase Exception, por lo que existen muchos programas y pueden darse muchísimas causas de por qué se ha producido un determinado error. Aunque existan clases específicas para determinar los errores, algunas veces no llegamos a dar con la causa que ha producido el error.

A continuación, vamos a ver las clases propias de error y la forma de lanzarlas en determinadas ocasiones.

### 10.4.1. Creación clases error en Java

Además de utilizar las excepciones que nos proporciona Java, también podemos crear nosotros mismos distintos tipos de excepciones que se adecuen a nuestras necesidades.

Para crear nuestra excepción debemos seguir los siguientes pasos:

- Añadir una nueva clase al proyecto y ponerle el nombre que queramos que tenga nuestra excepción. Generalmente el nombre hace referencia al error que puede controlar, seguido de la palabra Exception
- Hacer que la nueva clase extienda de la clase Exception o RuntimeException. La gran diferencia entre la una y la otra es que Exception nos obliga a establecer un bloque try..catch y RuntimeException, no
- Diseñar en la nueva clase una variable en la que podemos almacenar el código específico del error
- Configurar dos tipos de constructores de la clase, uno vacío y otro para inicializar una variable con el posible mensaje de error
- Sobrescribir un método getMessage para devolver el error producido en la clase

Su estructura la planteamos de la siguiente forma:

#### CÓDIGO

```
class ErrorLoginException extends Exception {
    String sms;

    public ErrorLoginException () {
        this.sms = "El usuario o contraseña no son válidos";
    }

    public ErrorLoginException (String sms) {
        this.sms = sms;
    }

    @Override
    public String getMessage () {
        return sms;
    }
}
```

Ahora vamos a ver un ejemplo de un programa Java que va a contemplar todos los puntos vistos en este apartado de excepciones:

## CÓDIGO

```
import java.util.Scanner;

public class Ejemplo {

    static boolean login (String user, String pass){
        if (user.equals("Ilerna") && pass.equals("Online")){
            return true;
        } else {
            return false;
        }
    }

    public static void main (String [] args){
        //Inicializar las variables

        Scanner sc = new Scanner(System.in);

        String nombre = "";
        String passwd = "";

        boolean valido = true;

        try{
            //Pedir los datos para el login

            System.out.println ("Introduzca el usuario:");
            nombre = sc.nextLine();

            System.out.println ("Introduzca la contraseña:");
            passwd = sc.nextLine();

            //Lanzar el método login
            if (!login(nombre, passwd)){
                valido = false;

                throw new ErrorLoginException ();
            }
        } catch (ErrorLoginException error){
            //Código que se ejecutará si el código del bloque try lanza una
            //excepción
        }
    }
}
```

```

System.out.println(error.getMessage());
}finally{
//Código que se va ejecutar siempre (aunque se entre en el catch)
if(valido){

        System.out.println("Bienvenido "+ nombre);
    }else{

        System.out.println("Vuelva a reiniciar el programa
para registrarse como usuario");
    }
}
}
}

class ErrorLoginException extends Exception {
    String sms;
public ErrorLoginException (){
this.sms ="El usuario o contraseña no son válidos";
}
public ErrorLoginException (String sms){
this.sms = sms;
}

    @Override
public String getMessage (){
return sms;
}
}

```

## 11. Lectura y escritura de información

El lenguaje Java dispone de una gran cantidad de clases destinadas a la lectura y/o escritura de datos (información) en distintas fuentes y destinos. Destacan, entre otros, los discos y los dispositivos. Estas clases a las que nos referimos se denominan flujos (streams), y se utilizan para leer información (de entrada) o para escribir información (de salida).

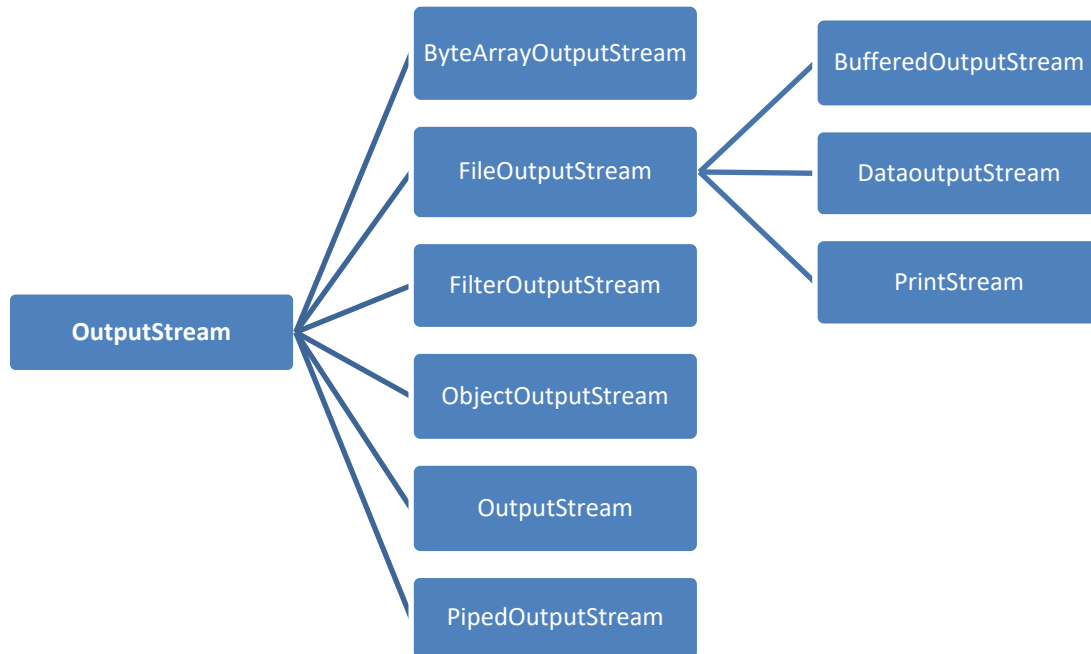
Tanto la lectura como la escritura se efectúan en términos de bytes, y las clases básicas de lectura escritura son las *InputStream* y *OutputStream*. Ambas dan lugar a una serie de clases adecuadas para lectura y escritura en distintos tipos de datos.

### 11.1. Clases relativas de flujos. Entrada/salida

Este esquema representa las clases relativas de flujos de entrada:



A continuación, encontramos esquematizadas las clases relativas de flujos de salida:



A continuación, vemos un ejemplo práctico donde aplicamos las clases de entrada y de salida más utilizadas en Java:

**CÓDIGO**

```

public class Ejemplo {
    public static void main(String[] args) {
        Leer leer = new Leer();
        Escribir escribir = new Escribir();
        escribir.escribir();
        leer.lee();
    }
}

class Leer {
    public void lee() {
        try {
            FileReader doc = new FileReader("Prueba.txt");
            BufferedReader b_doc = new BufferedReader(doc);
            String txt = "";
        }
    }
}
  
```



```
while(txt !=null){
    txt = b_doc.readLine();
    if(txt !=null)
        System.out.println(txt);
}
doc.close();
}catch(IOException e){
System.out.println("No se ha encontrado el fichero");
e.printStackTrace();
}
}
}
classEscribir{
publicvoidescribir(){
String txt ="\\nEscribimos a un fichero con Ilerna Online.";
try{
FileWriter doc =newFileWriter("Prueba.txt",true);
for(int i =0; i < txt.length(); i++){
doc.write(txt.charAt(i));
}
doc.close();
}catch(IOException e){
e.printStackTrace();
}
}
}
```

## 11.2. Tipos de flujos. Flujos de byte y de caracteres

### Flujos de bajo nivel

Cuando se realizan operaciones de E/S se traslada información entre la memoria del ordenador y el sistema de almacenamiento seleccionado. El lenguaje de programación Java cuenta con una implementación de InputStream y OutputStream, que se utiliza para este movimiento de información.

## FileInputStream y FileOutputStream

Son clases que pueden realizar operaciones de lectura y escritura de bajo nivel (*byte* a *byte*) mediante el uso de los métodos `read` y `write`.

Son métodos sobrecargados que nos permiten utilizar bytes de forma individual o, incluso, un búfer completo.

- **FileInputStream**

Devuelve un valor entero (*int*) entre 0 y 255:

CÓDIGO

```
int read ();
```

Devuelve el número de bytes leídos:

CÓDIGO

```
int read (byte[] b);
```

Ambas funciones devuelven -1 si llegan al final del fichero.

Cierre del archivo:

CÓDIGO

```
void close ();
```

- **FileOutputStream**

Escribe un byte:

CÓDIGO

```
void write (int x);
```

Escribe el número de bytes del rango:

CÓDIGO

```
void write (byte[] x);
```

Cierre del archivo:

CÓDIGO

```
void close ();
```

### Flujos de texto

Tenemos la posibilidad de leer y escribir siempre respetando, entre otros factores, la codificación, los diferentes signos diacríticos o el separador de líneas.

### BufferedReader y PrintWriter

Permiten el paso al formato de caracteres más utilizado. Admiten el concepto de línea como un conjunto de caracteres que se encuentran entre dos separadores de línea, aunque cada sistema operativo utiliza un separador de línea diferente.

Sistema Operativo	Separador	Caracteres
Unix	LF	'\n'
Windows	CRLF	'\n'\n'

La clase `BufferedReader` supone que llega al final de una línea cuando se encuentra con alguno de estos marcadores. Con el “%n” podemos escribir un marcador correcto en aquellos métodos de la clase `PrintWriter`.

Otras clases como: `InputStreamReader`, `FileReader`, `OutputFile-Writer` y `FileWriter` también ofrecen la posibilidad de escribir caracteres, aunque en estos casos, cuentan con unos constructores bastante más flexibles.

### Clase Scanner

Esta clase se va a utilizar cada vez que tengamos que acceder a valores de variables individuales, sobre todo, cuando se produzca interacción con los usuarios desde el teclado.

- **Constructores:** ofrecen la posibilidad de leer la información que proceda de un objeto que esté identificado mediante un ejemplar *File* o, en algunos casos, información procedente de flujos.
  - `Scanner (File origen)`
  - `Scanner (File origen, String nombreCharSet)`
  - `Scanner (InputStream origen)`
  - `Scanner (InputStream origen, String nombreCharSet)`
  - `Scanner (Readable origen)`
  - `Scanner (ReadableByteChannel origen)`
  - `Scanner (ReadableByteChannel origen, String nombreCharSet)`
  - `Scanner (String origen)`
- **Métodos**
  - **`void close ()`:** es la función principal que desarrolla la clase *Scanner* es la lectura. Si empleamos la función `close` de un *Scanner* ya cerrado se produce una excepción (*IllegalStateException*).
  - **`hasNext`:** es un método que devuelve *True* si el scanner cuenta con otro símbolo que se defina como una cadena de caracteres situada entre dos ejemplares del delimitador.
  - **`next`:** proporciona el símbolo siguiente disponible. Si no existe ninguno, lanza una excepción (*NosuchElementException*).

## Clase `BufferedReader`

Realiza diferentes operaciones de lectura por bloques, por lo que resulta bastante eficiente.

- **Constructores**
  - `BufferedReader (Reader in)`
  - `BufferedReader (Reader in, int tamaño)`
- **Métodos**

Método	Descripción
<b><code>void close ()</code></b>	Cuando finalice la operación de lectura, debemos cerrar el lector para que otros puedan hacer uso de él
<b><code>void mark</code></b>	(límite)
<b><code>boolean marksupported ()</code></b>	Pueden situar un puntero en la dirección de memoria seleccionada para que, cuando realicemos un reset, el puntero vuelva a esa posición en vez de al principio del fichero
<b><code>int read ()</code></b> <b><code>int read (char [] bufer, int desp, int long)</code></b> <b><code>String readLine ()</code></b>	Funciones que se van a utilizar para leer caracteres de forma individual, un conjunto de caracteres o, incluso, líneas de caracteres completas. Cuando lleguemos al final del fichero, la función devuelve -1, en caso contrario, devuelve el valor del carácter

### Clase `BufferedWriter`

Realiza la operación de escritura de la forma más eficiente, es decir, haciendo uso de bloques mayores. Esta almacenará el texto en una memoria intermedia para que después se pueda volcar su contenido al disco.

### Clase `PrintWriter`

Cuenta con una serie de métodos que se van a utilizar para escribir información en un fichero. El método `PrintWriter` es bastante más específico que el `println`.

### Clase `File`

Se refiere a un tipo de clase que cuenta con una serie de métodos relacionados con distintas rutas de ficheros o directorios, incluidos los métodos para crear y eliminar estos ficheros. Estas rutas, una vez que aparecen en los métodos, ya no se pueden modificar.

- **Constructores**

Pueden aportar una cadena cuyo contenido sea la ruta (absoluta/relativa), o una ruta del directorio padre y el nombre de un objeto hijo. También existe la opción de aportar un identificador uniforme de recursos (URI).

Aunque lo más importante de estos métodos es no olvidar nunca que lo que devuelven es una ruta, no un valor determinado:

- `File (File padre, String hijo)`
- `File (String ruta)`
- `File (URI uri)`

- Métodos

Método	Descripción
<b>boolean canExecute ()</b> <b>boolean canRead ()</b> <b>boolean canWrite ()</b>	El administrador de seguridad determina si la ruta especificada es apta para leer o escribir información.
<b>boolean exists ()</b>	Este método nos devuelve si existe o no un archivo en el sistema de archivos.
<b>int compareTo (File ruta)</b>	Devuelve el orden entre dos rutas.
<b>boolean equals (Object obj)</b>	Determina si dos rutas son iguales.
<b>boolean createNewFile ()</b> <b>static File createTempFile (String prefijo, String sufijo)</b> <b>static File createTempFile (String prefijo, String sufijo, File directorio)</b>	Pretenden crear un nuevo método según los parámetros que se le pasen por parámetros. El primero devuelve un booleano, que será True si todo va bien, mientras que los dos siguientes crean un archivo temporal en el directorio que se indica.
<b>boolean delete ()</b> <b>void deleteOnExit ()</b>	Se utilizan cuando deseemos borrar un archivo. En el primer caso, devuelve un booleano que indique si es posible borrar el archivo o no, mientras que el segundo, va a borrar el archivo seleccionado cuando el ordenador concluya.
<b>File getAbsolutePath ()</b> <b>File getCanonicalFile ()</b> <b>String getAbsolutePath ()</b> <b>String getCanonicalPath ()</b>	Devuelven la ruta absoluta (parte del directorio raíz) o canónica (parte de la raíz, pero elimina las abreviaturas) en una cadena o un ejemplar file.
<b>String getName ()</b>	Devuelve el nombre del archivo.
<b>String getParent ()</b>	Devuelve la ruta del directorio que lo

	contiene.
<b>FilegetParentFile ()</b>	Devuelve la ruta abstracta del directorio que contiene la ruta.
<b>String getPath ()</b>	Devuelve la cadena que equivale a la ruta abstracta.
<b>long getFreeSpace ()</b> <b>long getTotalSpace ()</b> <b>long getUsableSpace ()</b> <b>long length ()</b>	Métodos que devuelven la cantidad de espacio libre total o reutilizable del que disponemos en la parte que se ejecuta la aplicación. El método length devuelve la cantidad de bytes del archivo indicado.
<b>boolean isAbsolute ()</b> <b>boolean isDirectory ()</b> <b>boolean isFile ()</b> <b>boolean isHidden ()</b> <b>boolean lastModified ()</b>	Indican si la ruta especificada es absoluta, un directorio, un archive, si corresponde a un archive oculto. El último método indica el último momento en el que se ha realizado algún cambio en el fichero.
<b>String [] list ()</b> <b>String list (FilenameFilter filtro)</b> <b>File [] listFiles ()</b> <b>File [] listFiles (FileFilter filtro)</b> <b>File [] listFiles (FilenameFilter filtro)</b> <b>StaticFile [] listRoots ()</b>	Estos métodos devuelven listas de archivos tipo String o File, relacionados con la ruta especificada. Pueden ser todos o algunos, si se hace uso de algún filtro que lo controle. El último método devuelve una lista de directorios raíz.
<b>boolean mkdir ()</b> <b>boolean mkdirs ()</b>	Se utilizan para crear el directorio que especifique la ruta. Además, crea los directorios intermedios que sean necesarios.
<b>boolean renameTo (File destino)</b>	Modifica el nombre de un fichero. Si se realizan los cambios con éxito, los métodos van a devolver el valor de True.



<b>boolean setExecutable (boolean ejecutable)</b> <b>boolean setExecutable (boolean ejecutable, boolean soloPropietario)</b> <b>boolean setLastModified (long hora)</b> <b>boolean setReadable (boolean legible)</b> <b>boolean setReadable (boolean legible, boolean soloPropietario)</b> <b>boolean setReadOnly ()</b> <b>boolean setWritable (boolean admiteEscritura)</b> <b>boolean setWritable (boolean admiteEscritura, Boolean soloPropietario)</b>	Utilizados para fijar la ruta a la que se aplican. Si la operación se realiza con éxito devuelve True y False, en caso contrario.
<b>String toString ()</b> <b>URI toURI ()</b>	Traducen el fichero a String o URI, según corresponda.

### Flujos Binarios

Cuando utilizamos ficheros en formato binario, trabajamos de manera diferente según si utilizamos:

### Tipos primitivos y String

Estos tipos se tratan mediante el uso de dos interfaces: `DataInput` y `DataOutput`, ya que son la base principal de las jerarquías de clases.

- **La interfaz DataInput**

Se puede implementar en las clases que detallamos a continuación:

DataInputStream	MemoryCacheImageInputStream
FileImageInputStream	RandomAccessFile
ImageOutputStreamImpl	FileCacheImageOutputStream
ObjectInputStream	ImageInputStreamImpl
FileCacheImageInputStream	MemoryCacheImageOutputStream
FileImageOutputStream	

Y, a continuación, vamos a detallar los métodos con los que cuenta esta interfaz:

MÉTODO ()	DESCRIPCIÓN
<b>boolean readBoolean ()</b>	Lee 1 byte y si es 0, devuelve true, en caso contrario, devuelve 0
<b>byte readByte ()</b>	Lee y devuelve 1 byte
<b>char readChar ()</b>	Lee 2 bytes y devuelve un char
<b>double readDouble ()</b>	Lee 8 bytes y devuelve un double
<b>float readFloat ()</b>	Lee 4 bytes y devuelve un float
<b>void readFully (byte [] t)</b>	Lee todos los bytes y los almacena en "t"
<b>void readFully (byte [] t, int off, int leng)</b>	Lee un máximo de leng bytes y los almacena en t a partir de la posición leng
<b>int readInt ()</b>	Lee 4 bytes y devuelve un entero

<b>String readLine ()</b>	Lee una línea y devuelve un String
<b>long readLong ()</b>	Lee 8 bytes y devuelve un long
<b>short readShort ()</b>	Lee 2 bytes y devuelve un short
<b>int readUnsignedByte ()</b>	Lee 1 byte, añade un valor nulo y devuelve un entero (de 1 ... 255)
<b>int readUnsignedShort ()</b>	Lee 2 bytes, añade un valor nulo y devuelve un entero (de 1 ... 65535).
<b>String readUTF ()</b>	Lee una cadena codificada previamente en UTF- 8
<b>int skipBytes (int n)</b>	Pretende descartar “n” bytes

- **La interfaz DataOutput**

En esta interfaz se implementan también un gran número de clases, entre las que señalamos:

DataOutputStream	MemoryCachedImageOutputStream
ImageOutputStreamImpl	FileImageOutputStream
RandomAccessFile	ObjectOutputStream
FileCachedImageOutputStream	

Vamos a detallar los métodos con los que cuenta esta interfaz:

MÉTODO ()	DESCRIPCIÓN
<b>void write (byte [] t)</b>	<i>Escribe todos los bytes de "t"</i>
<b>void write (byte [] t, int pos, int leng)</b>	Escribe, como mucho, leng bytes de "t" a partir de la posición pos
<b>void write (int x)</b>	Escribe el byte inferior al entero "x"
<b>void writeBoolean (boolean b)</b>	Escribe un byte de valor 0 si "b" es falso, y true en caso contrario
<b>void writeByte (int x)</b>	Escribe el byte inferior al entero "x"
<b>void writeBytes (String s)</b>	Escribe un byte por cada carácter de "s"
<b>void writeChar (int x)</b>	Escribe los 2 bytes de los que consta un char
<b>void writeChars (String s)</b>	Escribe todos los chars (a 2 bytes por char)
<b>void writeDouble (double v)</b>	Escribe 8 bytes
<b>void writeFloat (float f)</b>	Escribe 4 bytes
<b>void writeInt (int x)</b>	Escribe 4 bytes
<b>void writeLong (long l)</b>	Escribe 8 bytes
<b>void writeShort (short s)</b>	Escribe 2 bytes
<b>Void writeUTF (string s)</b>	Escribe el contenido (codificado en UTF) de la cadena

## Ficheros de colecciones u objetos

Existen mecanismos que se van a utilizar para **garantizar la persistencia de los objetos**. De hecho, esta es una de las principales características de la metodología orientada a objetos.

El lenguaje Java dispone de un mecanismo automatizado de recuperación y almacenamiento para los diferentes tipos de herramientas de la interfaz *Serializable* y, a continuación, vamos a desglosar uno de sus métodos principales.

### CÓDIGO

```
public interface Serializable {
    private void writeObject (java.io.ObjectOutputStream out) throws
        IOException
    private void readObject (java.io.ObjectInputStream in) throws
        IOException, ClassNotFoundException;
    //...
}
```

El **mecanismo de serialización o pasivación** almacena en disco el contenido de un objeto y, de esta forma, permite que posteriormente se pueda reconstruir, volviendo así al estado inicial que tenía antes de la serialización.

El estado de un objeto es bastante peculiar porque sus atributos pueden ser, a su vez, tipos de referencia. El mecanismo de serialización intenta almacenar el cierre transitivo del objeto, es decir, debe guardar el estado de todos los objetos que tengan como referencia parte del estado original. Para ello, el mecanismo de introspección de Java permite que se conozcan todos los métodos y atributos de la clase.

Cuando tengamos que implementar un fichero donde sus elementos sean tipos objetos, debemos indicar que la clase que trata a este fichero de objetos tiene que implementar la interfaz serializable.

Por ejemplo:

#### CÓDIGO

```
class Persona {
    String nombre;
    int edad;
    Persona();
};

class Fichero_Persona implements Serializable {
    //Operaciones para tratar el fichero de Personas
}
```

### 11.3. Ficheros de datos. Registros

En este apartado vamos a estudiar las diferentes clases que utiliza el lenguaje Java para llevar a cabo la gestión de ficheros (directorios), el control de errores que se realiza en el proceso de lectura/escritura y, además, los distintos tipos de flujos de entrada/salida de información.

Podemos definir los ficheros como una **secuencia de bits que está organizada de una forma determinada** y que se reúne en un dispositivo de almacenamiento secundario (disco duro, CD/ DVD, USB, etc.).

El programa encargado de generar el fichero es el que puede traducirlo interpretando su secuencia de bits, es decir, cuando diseñamos un programa, **el programador va a ser el encargado de almacenar información en un fichero y dictar las normas a seguir en este proceso**. De esta forma, el mismo programador que realice estas tareas va a ser el encargado de descifrarlo.

Cuando vamos a trabajar con ficheros **debemos tener en cuenta** que:

- La información está compuesta por un conjunto de 1 y 0
- Los bits se agrupan para formar bytes o palabras
- Se utilizan, entre otros, tipos de datos como int y double. Van a estar formados por un conjunto de bytes
- Al agrupar los distintos campos, se van formando los registros de información
- Los archivos están constituidos por diferentes conjuntos de registros, de tal forma que todos van a tener la misma estructura

Las **características principales** de los ficheros son:

- Se sitúan en dispositivos de almacenamiento secundarios para que la información siga existiendo ahí a pesar de que la aplicación se cierre
- La información se puede utilizar en diferentes dispositivos o equipos
- Ofrece independencia a la información, ya que no necesita otras aplicaciones ejecutándose para que exista dicha información

#### **11.4. Gestión de ficheros: modos de acceso, lectura/escritura, uso, creación y eliminación**

Los ficheros se pueden clasificar según el **tipo de organización de la información** que realicen. De esta forma, los ficheros pueden ser:

- **Secuenciales**
- **Aleatorios o directos**
- **Secuenciales indexados**

- **Ficheros Secuenciales**

Los ficheros **secuenciales** son aquellos que almacenan los registros en posiciones consecutivas. Para acceder a ellos debemos hacer un recorrido completo, es decir, comenzamos por el principio y terminamos por el final, recorriendo los registros de uno en uno.

En este tipo de ficheros solamente podemos realizar una operación de lectura/escritura a la vez, ya que, cuando un fichero se está leyendo, no se puede escribir. De la misma forma, cuando un fichero está siendo escrito no se puede llevar a cabo ninguna operación de lectura.



- **Ficheros Aleatorios**

Estos ficheros se denominan **aleatorios o directos** y, como bien su nombre indica, pueden acceder de forma directa a un registro concreto, indicando en qué posición se encuentra.

Estos ficheros pueden ser leídos o escritos en cualquier orden, porque como sabemos en qué posición se encuentran, solamente debemos colocar el manejador en esa posición para que lleve a cabo la operación indicada.



- **Ficheros Secuenciales Indexados**

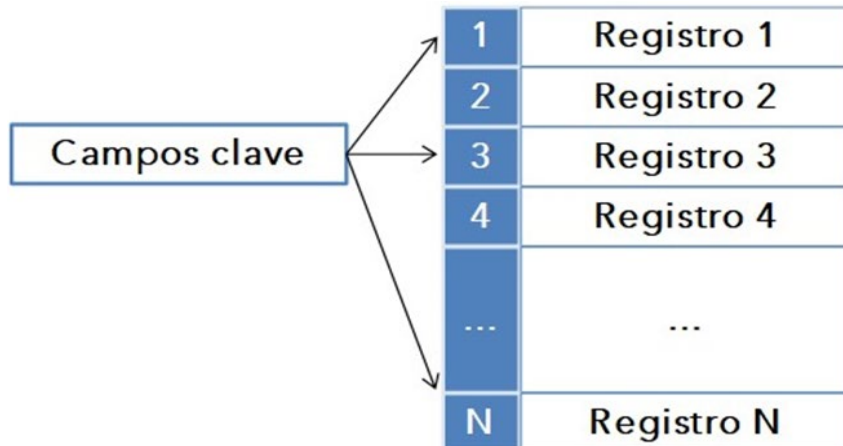
Los ficheros **secuenciales indexados** cuentan con un campo denominado "clave", que se utiliza para que cada registro se pueda identificar de forma única.

Este tipo de ficheros permiten el acceso secuencial y aleatorio (directo), de forma que:

- Primero buscamos de forma secuencial el campo clave del registro
- Una vez que lo conocemos, ya podemos acceder a este de forma directa, porque tenemos la posición de su campo clave



Los índices están ordenados con la intención de que se pueda realizar un acceso de forma más rápida:



### Operaciones

Cuando utilizamos ficheros existen una serie de operaciones que son las que nos van a permitir trabajar con ellos:

#### - Apertura

Tenemos que indicar al fichero el modo en el que lo queremos abrir, según las operaciones que deseemos realizar. Cuando abrimos un fichero, estamos relacionando un objeto de nuestro programa con un archivo que se encuentra almacenado en el disco mediante su nombre. Por eso, tenemos que indicar de qué modo vamos a trabajar con él.

#### - Lectura/ escritura

Debemos leer la información del fichero y fijar la posición en la que se encuentra en el manejador de archivos. Se debe indicar el punto desde el cual se comienza a leer. Si estamos al final del fichero no es posible realizar esta operación.

#### - Cierre

Cuando ya terminamos el proceso de almacenar información, es decir, cuando terminamos de escribir la información, debemos cerrar el fichero. La información queda almacenada en el buffer.

## - Apertura de ficheros

Cuando abrimos un fichero debemos indicar el modo en el que deseamos abrirlo, según la operación que deseemos realizar sobre él.

Estos son diferentes modos de los que disponemos a la hora de abrir un fichero:

1. **Lectura:** solo permite realizar operaciones de lectura sobre el fichero
2. **Escritura:** permite realizar operaciones de escritura sobre el fichero. Si el fichero en el que se quiere escribir ya existe, será borrado
3. **Añadir:** actúa de forma similar al de escritura, aunque en este caso, si el fichero en el que deseamos escribir ya existe, no se eliminará
4. **Lectura/ Escritura:** permite realizar operaciones de lectura/escritura sobre un fichero

## Pasos para leer o escribir en un fichero secuencial

### Lectura secuencial

En pseudocódigo quedaría algo similar a:

#### CÓDIGO

```
Fichero f1;//variable tipo fichero
f1.Abrir(lectura);// Abrimos el fichero
Mientras no final de fichero hacer    //Tratamos el fichero
// mientras cumpla una condición
    f1. Leer (registro);
    Operaciones con el registro leído;
Fin Mientras;
f1.Cerrar();//cerramos fichero
```

Veamos cómo realizar esto mismo en lenguaje Java. Para leer un fichero en Java necesitaremos como mínimo dos clases, File y FileReader.

La clase File nos proporciona los mecanismos necesarios para abrir y ubicar en el disco un fichero. La clase FileReader nos proporciona los mecanismos para leer un fichero.

Si unimos las clase File con un fichero y la clase FileReader con el fichero anterior, podremos leerlo de una forma muy sencilla. Hay que tener en cuenta que al intentar leer un fichero podemos hallar la excepción IOException. La causa más común es que no ha encontrado el fichero indicado en la clase File.

#### CÓDIGO

```
// Creamos la variable de tipo File, la cual corresponde a un
// fichero
File alumnosFile = new File("ficheros/alumnos.txt");
// Creamos la variable de tipo FileReader, la cual nos dará la
// capacidad para leer de un fichero.
FileReader reader;
try {
    //Inicializamos la variable reader, pasandole como parámetros
    // el fichero anterior.
    reader = new FileReader(alumnosFile);
    // La forma de determinar si existen datos en el fichero es
    // comprobando qué integer hemos recibido al leer. Si es un -1
    // querrá decir que el fichero no contiene datos.
    int finish = -1;
    int character;
    // Leemos carácter a carácter.
    while ((character = reader.read()) != finish)
        // Mostramos el carácter convirtiéndolo a char.
        System.out.print((char) character);
} catch (IOException ex) {
    System.out.println("No se ha podido crear el FileReader sobre el
    fichero " + alumnosFile.getName());
    System.out.println("Error:\n" + ex.getMessage());
    System.exit(-3);
}
```

## Escritura secuencial

En pseudocódigo quedaría algo similar a:

### CÓDIGO

```
Fichero f1                                //variable tipo fichero
f1.Abrir(escritura); // Abrimos el fichero
Mientras no final de dichero hacer      //Tratamos el fichero
// mientras cumpla una condición
Configuramos registros según los datos;
f1.Escribir(registro);
Fin Mientras;
f1.Cerrar(); //cerramos fichero
```

Veamos como realizar esto mismo en lenguaje Java:

### CÓDIGO

```
// Creamos la variable de tipo File, la cual corresponde a un
// fichero
File alumnosFile = new File("ficheros/alumnos.txt");
// Creamos la variable de tipo FileWriter, la cual nos dará la
// capacidad para escribir en un fichero.
FileWriter writer;
try {
    // Inicializamos la variable writer, pasandole como parámetro
    // el fichero abierto..
    writer = new FileWriter(alumnosFile);
    // Mediante la función write escribimos el texto deseado que
    // será insertado en el fichero.
    writer.write("Esto es un mensaje de texto en caracteres\n");
    writer.close();
} catch (IOException ex) {
    System.out.println("No se ha podido crear el FileWriter sobre el
    fichero " + alumnosFile.getName());
    System.out.println("Error:\n" + ex.getMessage());
    System.exit(-2);
}
```

## Pasos para leer o escribir en un fichero aleatorio (directo)

### Lectura y escritura aleatoria

La lectura y escritura aleatoria se realiza utilizando la misma clase, `RandomAccessFile`. Esta nos proveerá de los mecanismos para leer y escribir en una posición determinada. Para realizar esta acción utilizará un apuntador, el cual mediante la función `seek` podrá ser colocado en la posición deseada.

Una cuestión muy importante en los ficheros aleatorios es que al leer y escribir, el apuntador cambia de posición al último carácter leído/escrito, por lo que siempre hay que saber dónde se encuentra. Veamos un ejemplo:

#### CÓDIGO

```
try {
    RandomAccessFile randomAccess = new RandomAccessFile(random,
        "rw");

    long pos = random.length();
    randomAccess.seek(pos);
    randomAccess.writeBytes("Línea desde randomAccessFile");
    randomAccess.seek(pos);
    String next;
    while ((next = randomAccess.readLine()) != null) {
        System.out.println(next + ": Tamaño = " + next.length());
    }
} catch (IOException e) {
    e.printStackTrace();
}

//cerramos fichero
```

### Cierre de ficheros

Siempre que trabajemos con ficheros, debemos abrirlos para poder trabajar con ellos y, cuando terminemos con la operación deseada, no podemos olvidar cerrarlos.

Como hemos visto en el ejemplo anterior:

#### CÓDIGO

```
f1.Cerrar(); //cerramos fichero
```

## 12. Interfaces gráficas de usuario

Hasta ahora hemos implementado código fuente desde un entorno en modo texto, es decir, sin ningún tipo de gráfico, introduciendo toda la información a través del teclado.

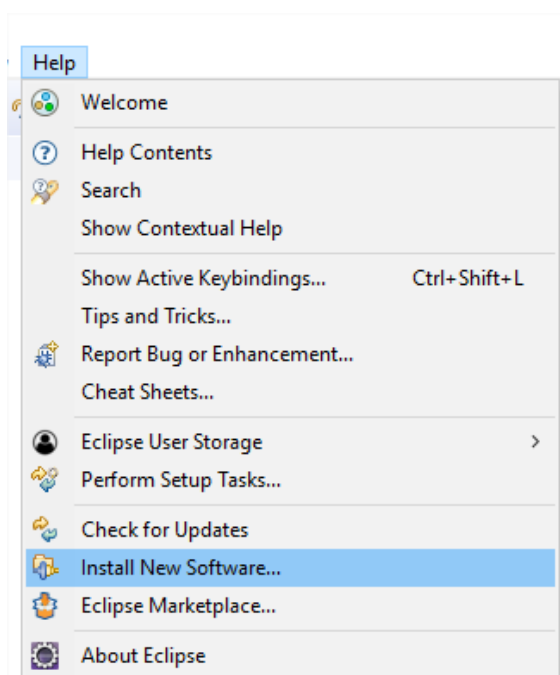
A partir de este punto, mejoraremos el diseño de nuestro programa y lo crearemos más vistoso, con un **entorno gráfico** donde podemos utilizar tanto el teclado como el ratón, y también los periféricos de entrada de información. En este apartado veremos todos los elementos de los que se compone el entorno gráfico, los **distintos paneles, etiquetas, y cajas de información**.

### 12.1. Creación y uso de interfaces gráficas de usuarios simples

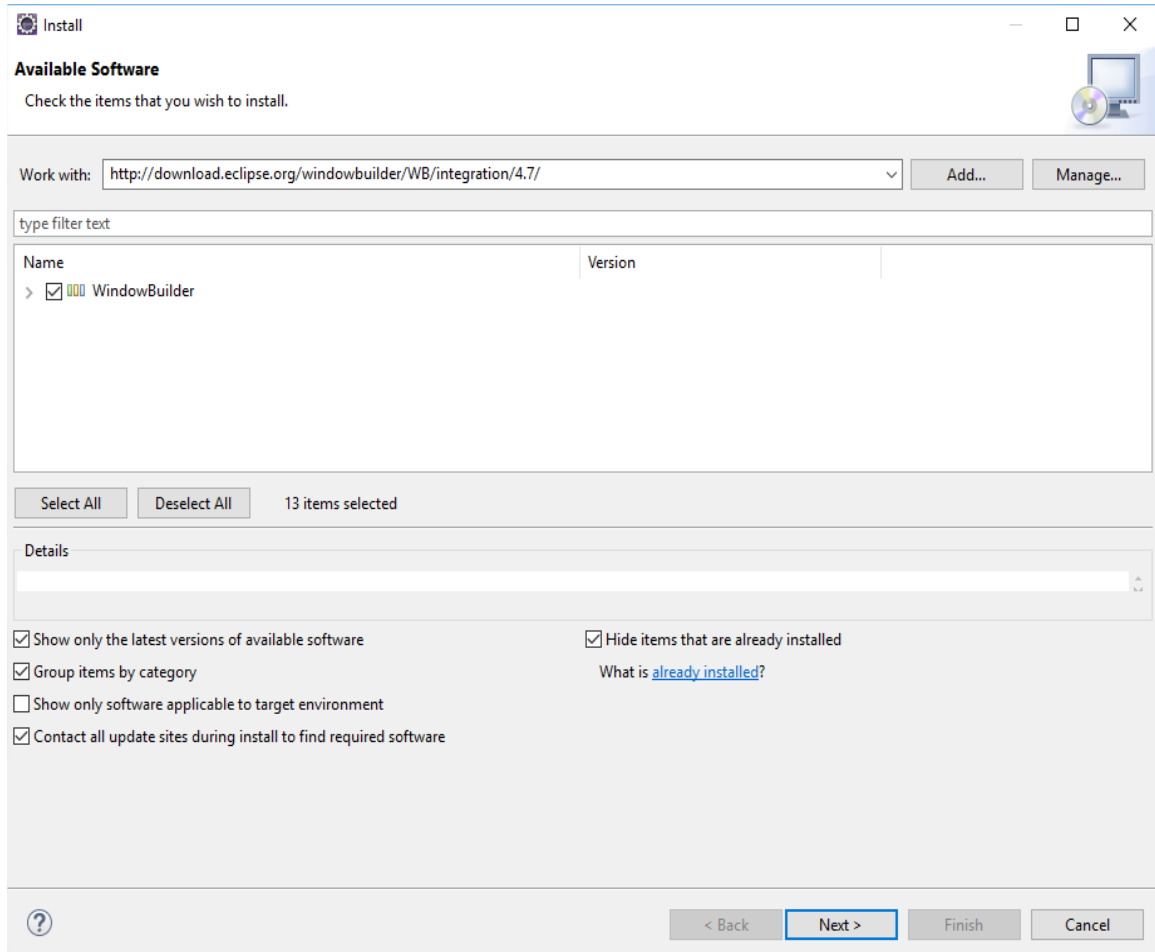
#### Entorno de trabajo

Para comenzar a desarrollar interfaces gráficas necesitaremos tener preparado el entorno de trabajo. En este caso, utilizaremos Eclipse y deberemos instalar un plugin que nos permita este desarrollo.

Vamos a instalar WindowBuilder. Esto lo podremos hacer desde el propio Eclipse, seleccionando *Help > Install New Software*:

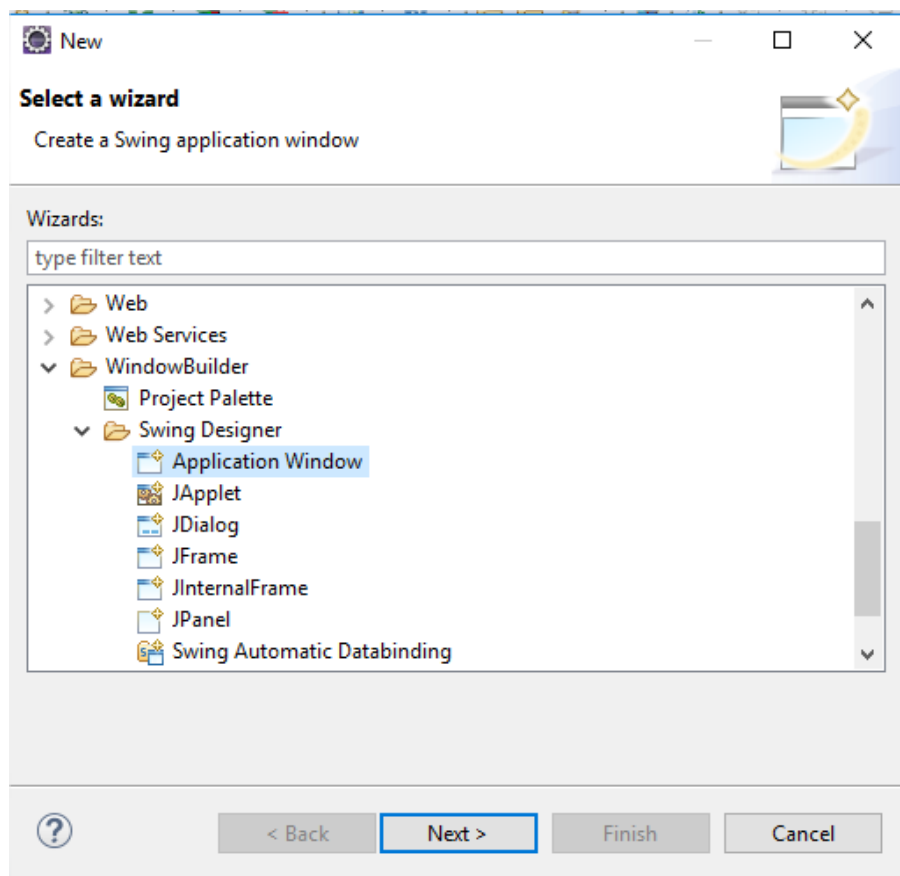


Para instalarlo, necesitaremos una dirección URL que podremos obtener (según la versión que utilicemos de Eclipse) en la siguiente dirección: <http://www.eclipse.org/windowbuilder/download.php>.



Este plugin incorpora los componentes de Swing que necesitaremos para desarrollar nuestra interfaz.

Una vez instalado, Eclipse pedirá reiniciar y, posteriormente, ya podremos crear nuestro proyecto. Debemos ir a *Archivo > Nuevo > Other > WindowBuilder > Swing Designer > Application Window*, y poner el nombre de la clase, por ejemplo: *HolaMundoSwing*.



Esta acción generará una clase que ya tiene parte de código implementado. La clase generada es la siguiente:



## CÓDIGO

```
public class HolaMundoSwing {

    private JFrame frame;

    /**
     * Launch the application.
     */
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    HolaMundoSwing window =
new HolaMundoSwing();
                    window.frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

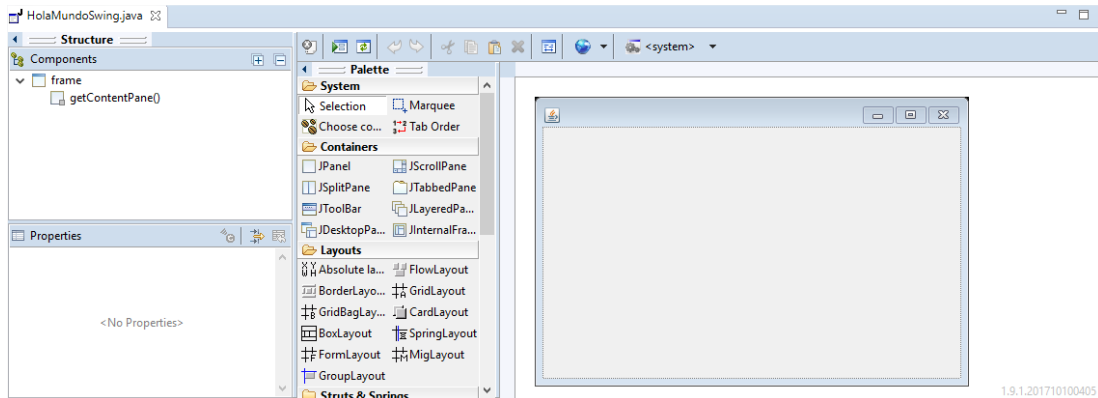
    /**
     * Create the application.
     */
    public HolaMundoSwing() {
        initialize();
    }

    /**
     * Initialize the contents of the frame.
     */
    private void initialize() {
        frame = new JFrame();
        frame.setBounds(100, 100, 450, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

}
```

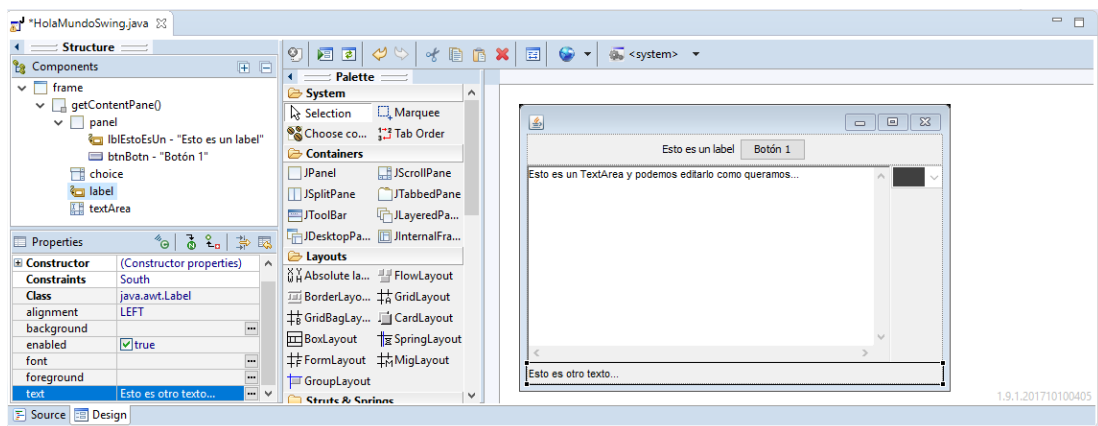
En esta clase, en la parte inferior, podremos visualizar dos pestañas: *Source* y *Design*. En *Source* encontraremos todo el código Java y en *Design*, la parte de interfaz gráfica que podremos usar para generar nuestras interfaces de una forma más dinámica y usando el ratón.

A continuación, se visualiza en la siguiente imagen la vista de Design que se crea por defecto al crear la clase que hemos visto anteriormente:



A partir de esta base, podremos añadir nuestros componentes, diseñando así nuestra ventana. En la sección intermedia tenemos la parte de *Palette*, que contiene todas las opciones a añadir: paneles, *labels*, cajas de texto, botones, etc.

Además, cada elemento tiene una configuración propia que puede ser modificada de forma dinámica, ayudándonos del ratón y sin tener que escribir el código.



Después de realizar las modificaciones necesarias, si volvemos a la pestaña de Source, encontraremos que se ha generado el código necesario correspondiente a los cambios que hemos realizado.

CÓDIGO GENERADO CON LAS MODIFICACIONES REALIZADAS EN DESIGN

```

private JFrame frame;

/**
 * Launch the application.
 */
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                HolaMundoSwing window =
new HolaMundoSwing();

                window.frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

/**
 * Create the application.
 */
public HolaMundoSwing() {
    initialize();
}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
    frame = new JFrame();
    frame.setBounds(100, 100, 450, 300);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JPanel panel = new JPanel();
    frame.getContentPane().add(panel, BorderLayout.NORTH);

    JLabel lblEstoEsUn = new JLabel("Esto es un label");
    panel.add(lblEstoEsUn);

    JButton btnBotn = new JButton("Bot\u00F3n 1");
    panel.add(btnBotn);

    Choice choice = new Choice();
    choice.setBackground(Color.DARK_GRAY);
    choice.setEnabled(false);
    choice.setFont(new Font("Bauhaus 93", Font.PLAIN, 12));
    choice.setForeground(Color.PINK);
    frame.getContentPane().add(choice, BorderLayout.CENTER);
}

```

```

        Label label = new Label("Esto es otro texto...");
        frame.getContentPane().add(label, BorderLayout.SOUTH);

        TextArea textArea = new TextArea();
        textArea.setText("Esto es un TextArea y podemos editarlo
como queramos...");
        frame.getContentPane().add(textArea, BorderLayout.WEST);
        frame.getContentPane().setFocusTraversalPolicy(new
FocusTraversalOnArray(new Component[]{panel, lblEstoEsUn, btnBotn,
choice, label}));
    }
}

```

También podremos realizar las modificaciones en el código y veremos los cambios si vamos a la pestaña de Design.

Para crear las distintas aplicaciones con formularios deberemos añadir un nuevo formulario y, posteriormente, los controles que necesitemos. Una vez añadidos los controles, si es necesario, podemos modificar sus características.

### Componentes de un entorno gráfico

Tanto en Eclipse (con el uso del plugin que hemos instalado) como en otros entornos gráficos, disponemos de una serie de componentes que nos ayudan a conseguir una interfaz gráfica acorde a nuestras necesidades. Las partes más importantes son:

- **Vista de diseño:** se trata de la vista predeterminada de los formularios en la que vamos a poder insertar diferentes controles de una manera bastante sencilla y rápida.
- **Paleta:** en paleta vamos a tener las distintas herramientas necesarias para las diferentes acciones que se pueden realizar con los elementos. Podremos elegir qué elementos queremos añadir a nuestra interfaz.
- **Propiedades:** mediante el cuadro de propiedades podemos hacer todas las modificaciones y configurar las distintas características de los elementos (controles): tamaño, color, texto, etc.
















## 12.2. Paquetes de clases para el diseño de interfaces

Las librerías que vamos a usar para desarrollar las Interfaces de Usuario Gráficas (GUI) son Swing y AWT.

- **AWT** (*Abstract Windowing Toolkit*): es una librería que permite hacer interfaces gráficas con texto, botones, menús, barras de desplazamiento, etc.
- **SWING**: Es la evolución de AWT y mejora el aspecto de los diferentes controles.

### Controles

Los controles son cada uno de los elementos o componentes que podemos añadir a nuestra interfaz. Los diferentes controles que tenemos disponible en Java para hacer un proyecto gráfico son:

AWT		SWING	
Label	 Label	JLabel	 JLabel
TextField	 Text Field	JTextField	 Text Field
TextArea	 Text Area	JScrollPane	 Text Area
Button	 Button	JButton	 Button
CheckBox	 Checkbox	JCheckBox	 - CheckBox
Choice	 Choice	JComboBox	 ComboBox
List	 List	JScrollPane	 List
		JRadioButton	 - Radio Button

Vamos a ver un ejemplo de clase que instancia componentes gráficos:

#### CÓDIGO

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class Ejemplo {
    public static void main (String [] args) {
        //Creamos un JFrame que nos hará de ventana
        JFrame ventana = new JFrame("Titulo_ventana");
        //Ponemos la ventana visible
        ventana.setVisible(true);
        //Le damos un tamaño a nuestra ventana
        ventana.setSize(600,400);
        //Creamos un panel
        JPanel panel = new JPanel();
        panel.setLayout(null);
        ventana.add(panel);
        //Creamos un Label para mostrar un texto
        JLabel etiqueta = new JLabel();
        etiqueta.setText("Primera prueba con JFrames de Java");
        //Le asignamos una posición y un tamaño
        etiqueta.setBounds(100,100,400,35);
        //Añadimos la etiqueta al panel
        panel.add(etiqueta);
    }
}
```

## Contenedores en Java

En el lenguaje Java podemos diferenciar entre dos tipos diferentes de contenedores:

- **Superiores:** *JFrame*, *JDialog* y *JApplet*. En el editor de código se localizan en el cuadro de herramientas bajo la categoría **Swing Windows**.

Los contenedores superiores incluyen a los intermedios y su diseño les permite almacenar menús o diferentes barras de herramientas (barra de título, botones para maximizar o minimizar entre otros).

- **Intermedios:** *JPanel*, *JSplitPane*, *JScrollPane*, *JToolBar* o *JInternalFrame*. Al igual que los anteriores, podemos encontrarlos en el cuadro de herramientas, pero en la categoría **Swing Containers**.

Cuando diseñamos una aplicación gráfica en Java, debemos tener en cuenta:

1. Creamos y configuramos un contenedor superior, estableciendo el tamaño del contenedor **setSize()**. Después la hacemos visible **setVisible()** cuando arrancamos el programa.
2. Debemos incluir un contenedor mediante el método **add()**.
3. Por último, añadimos los controles que vamos a necesitar para nuestra aplicación.

Cuando necesitamos crear alguna aplicación con varios formularios, podemos crearnos un único *JFrame* e ir añadiendo los distintos contenedores que iremos haciendo visibles según nuestras necesidades.

- **JApplet:** este contenedor permite ser visualizado en Internet.
- **JPanel:** agrupa los controles que están relacionados con la aplicación.
- **JSplitPane:** podemos utilizarlo para dividir dos componentes.
- **JScrollPane:** permite utilizar una barra de desplazamiento que nos dejará mover (de forma horizontal y vertical) por las diferentes zonas de control.
- **JToolBar:** en esta barra de herramientas podemos controlar, mediante el uso de botones, el acceso rápido a las diferentes zonas.
- **JInternalFrame:** las diferentes barras internas.

### 12.3. Acontecimientos (eventos). Creación y propiedades

Existen una serie de propiedades que tiene cada componente (lista desplegable, cuadros de texto, botones, etc.) y que pueden ser de bastante utilidad.

#### Propiedades Java

- **name:** nos permite identificar el objeto. Si queremos cambiar el nombre de un objeto solo tenemos que hacer clic en el botón derecho y seleccionar la opción ***"Change Variable Name ..."***.
- **Location:** desde el cuadro de propiedades no podemos sustituir la localización. Mediante el modificador ***setLocation (int x, int y)***, podemos establecer una nueva posición.
- **Visible:** visible.
- **BackColor:** background.
- **Font:** podemos modificar tipo, tamaño y estilo de letra.
- **ForeColor:** foreground.
- **Text:** la propiedad de los cuadros de texto se denomina **text**.
- **Enabled:** muestra una casilla de verificación en la que podemos activar propiedades de tipo booleanas.
- **Size:** utiliza las propiedades para el tamaño horizontal y vertical.
- **Icon:** iconImage.

#### Propiedades relacionadas con cuadros de texto

- **editable:** va a establecer si podemos escribir o no en el cuadro de texto.
- **border:** establece el tipo de borde del control.
- **disabledTextColor:** hace referencia al color del texto cuando se encuentra deshabilitado.
- **margin:** permite establecer distancia entre bordes y texto.

#### Propiedades relacionadas con casillas de verificación

- **selected:** ofrece la posibilidad de elegir si queremos que la casilla de verificación aparezca en pantalla o no.



### Propiedades relacionadas con botones de opción

- **selected:** ofrece la posibilidad de elegir si queremos que el botón de opción esté seleccionado o no al comenzar la aplicación.

### Propiedades relacionadas con cuadros de lista desplegable

- **selectedIndex:** devuelve la posición del elemento seleccionado.
- **selectedItem:** similar al anterior y enlazada con esta, ya que, si modificamos un valor, los demás también se van a alterar.
- **Model:** selecciona los distintos elementos (separados por comas) que van a formar la lista.

### Propiedades relacionadas con cuadros de lista (List)

- **model:** selecciona los distintos elementos (separados por comas) que van a formar la lista.
- **selectionMode:** determina el modo en que se pueden seleccionar los distintos componentes de la lista. Podemos seleccionar más de un dato, entre los siguientes valores:
  1. **SINGLE:** permite seleccionar solamente un elemento.
  2. **MULTIPLE\_INTERVAL:** permite seleccionar más de un elemento (no tienen por qué estar juntos).
  3. **SINGLE\_INTERVAL:** permite seleccionar más de un elemento (deben estar juntos).
- **selectedIndex:** ofrece la posibilidad de modificar el índice de la lista que se encuentre seleccionado.
- **selectedValue:** parecida al anterior pero, en este caso, indicamos el valor que se va a seleccionar.
- **selectionBackground:** determina el color de fondo.

## Propiedades relacionadas con botones

- **icon:** determina un icono a un botón para que podamos verlo en un determinado lugar sustituyendo al texto
- **buttonGroup:** asocia un botón a un *Button Group* y así permite modificar su comportamiento
- **iconTextGap:** espacio determinado entre el botón y un texto
- **pressedIcon:** este icono se va a mostrar al presionar el botón

Desde la ventana de propiedades podemos ver los diferentes eventos que se pueden aplicar a los distintos controles que incorpora la aplicación. Solo es necesario hacer clic sobre **Events**.

Eventos Java
actionPerformed
componentMoved
componentResized
focusGained
focusLost
propertyChange
mousePressed
mouseReleased
mouseEntered
mouseExited
mouseMoved
keyPressed
KeyReleased

Siguiendo con el ejemplo del apartado anterior, vamos a ver cómo aplicar botones y algunos de sus métodos:

#### CÓDIGO

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class Ejemplo {
    public static void main (String [] args) {

        //Creamos un JFrame que nos hará de ventana
        JFrame ventana = new JFrame("Titulo_ventana");

        //Ponemos la ventana visible
        ventana.setVisible(true);

        //Le damos un tamaño a nuestra ventana
        ventana.setSize(600,400);

        //Creamos un panel
        JPanel panel = new JPanel();
        panel.setLayout(null);
        ventana.add(panel);

        //Creamos un Label para mostrar un texto
        JLabel etiqueta = new JLabel();
        etiqueta.setText("Primera prueba con JFrames de Java");
    }
}
```

```
//Aplicamos una fuente al texto del Label
etiqueta.setFont(new Font("Courier New", Font.BOLD,18));

//Le asignamos una posición y un tamaño
etiqueta.setBounds(100,100,400,35);

//Añadimos la etiqueta al panel
    panel.add(etiqueta);

//Creamos un boton para cerrar la ventana
JButton boton =new JButton("Close");

//Le asignamos una posición y un tamaño
boton.setBounds(250,250,120,35);

//Le damos una acción al botón
boton.addActionListener(new CerrarVentana());

//Añadimos el botón al panel
panel.add(boton);
}
}
class CerrarVentana implements ActionListener{
    @Override
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```

## Menús en Java

En Java podemos crear menús para ver interfaces más sencillas y, para ello, haremos clic en la categoría ***Swing menus***.

Cuando insertamos un menú:

- Insertar barra de menú.
- Esta barra de menú dispone de dos elementos. Si deseamos añadir más menús lo haremos a través del control menú.
- Para añadir submenús, podemos añadir nuevos *Menús* o hacer uso de *Menú Ítem*, *Menú Ítem/ CheckBox* o *Menú Ítem/ RadioButton*.
- Si deseamos añadir separadores conceptuales que diferencien elementos, podemos hacerlo mediante *Separator*.

Por tanto, los elementos de los que disponemos a la hora de crear los diferentes menús son los siguientes:

- **Menú Bar (JMenuBar)**: objeto contenedor de menús.
- **Menú (JMenu)**: se utiliza para representar los elementos del menú principal o secundario.
- **Menú Ítem (JMenuItem)**: genera una acción al pulsar una opción concreta.
- **Separator (JSeparator)**: permite dividir las diferentes opciones.

Aparte de todas estas opciones, también podemos añadir menús conceptuales o **Popup Menú**.

## 13. Diseño de programas con lenguajes de POO para gestionar bases de datos relacionales

El **SGBD (Sistema de Gestión de Base de Datos)** es el programa que ofrece la posibilidad de almacenar, modificar y extraer información de una base de datos determinada. También proporciona una serie de herramientas que nos permiten realizar otra serie de operaciones sobre los datos.

En resumen, el objetivo de estas bases de datos es crear diferentes aplicaciones en Java que actúen como gestor y permitan actualizar, modificar o eliminar los distintos datos. Partiremos de una base de datos ya creada para, a continuación, conectarnos a ella y poder realizar cualquiera de estas operaciones.

Una **base de datos relacionales** una base de datos que almacena la información del mundo real a través de tablas que se relacionan entre sí para organizar mejor la información y poder pasar de un dato a otro sin ningún tipo de problema. Dicha base de datos se basa en el **modelo relacional** que define la base de datos en función de la lógica de predicados y la teoría de conjuntos.

Las bases de datos relacionales suelen utilizarse cuando tenemos que trabajar con una gran cantidad de información. Estas bases de datos tienen la información organizada en tablas que, a su vez, se encuentran relacionadas entre ellas.

Todos los datos son almacenados en la base de datos en forma de relaciones que se visualizarán como una tabla, que a su vez tiene filas y columnas. Las columnas de la tabla son las características o atributos de la tabla. Las filas serán los registros o tuplas donde daremos valores a los campos: la información propiamente dicha.

La característica principal de una base de datos relacional es que puede componerse de varias relaciones o tablas. Las tablas no podrán tener el mismo nombre y se compondrán de filas (registros) y columnas (campos). Lo más característico de las bases de datos relacionales es que las tablas que la componen deben contener un campo clave, es decir, un campo donde su valor en todos los registros sea único y no se repita. Dos tablas pueden relacionarse a través de claves primarias y claves foráneas. La clave primaria se situará en la tabla padre y la clave ajena o *foreign key*, en la tabla hija, que se relaciona con la tabla padre.

Sus principales desventajas son: los problemas a la hora de manejar bloques de texto como tipo de dato y los problemas para visualizar información gráfica, multimedia o geográfica.

### 13.1. Establecimiento de conexiones

Para poder llevar a cabo la conexión desde un código fuente Java hasta una base de datos debemos utilizar una serie de colecciones dentro de la API de SQL de Java, incluida en la versión 7 de dicho compilador.

También necesitaremos los datos de la base de datos a conectar: driver JDBC, dirección de la base de datos, usuario y contraseña.

- **Driver:** es el encargado de adaptar los comandos de la BD a la librería JDBC. Cada distribuidor tendrá su propio driver.
- **JDBC:** Java DataBase Connectivity es la librería que nos da la funcionalidad para conectarnos a diferentes BDs.
- **Dirección de la BD:** dirección de la URL para la BD.
- **Usuario:** usuario que realiza la conexión.
- **Contraseña:** contraseña del usuario que realiza la conexión.

Podemos comenzar introduciendo en el código fuente la importación de dichas librerías, especialmente las funciones **Connection** y **DriverManager**.

Estas funciones cuentan, en su implementación, con diferentes procedimientos que ayudan al enlace entre el programa Java y la base de datos.

#### CÓDIGO

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

**Nota:** cuando tengamos que importar varias librerías de una misma raíz, lo podemos hacer en una sola línea con el símbolo de asterisco, con el cual importaremos todas las librerías de esa raíz.

Ejemplo: `import java.sql.*;`

Una vez importadas las distintas librerías (aunque también se pueden importar con posterioridad), podemos comenzar nuestro programa declarando cuatro variables que detallaremos a continuación:

- En la primera, vamos a almacenar el driver *JDBC*.
- En la segunda, la dirección de la base de datos MySQL que, normalmente en los proyectos que estemos desarrollando, se encuentra en nuestro propio servidor: *localhost*.
- En las dos últimas variables almacenaremos el valor del usuario y la contraseña que, previamente, ha tenido que ser configurada en la base de datos.

#### CÓDIGO

```
private static final String DRIVER = "com.mysql.jdbc.Driver";
private static final String BBDD = "jdbc:mysql://localhost/sorteo";
private static final String USUARIO = "root";
private static final String PASSWORD = "root";
```

Una vez declaradas las variables ya solo nos falta diseñar la función para conectarnos a la base de datos. En dicha función realizaremos los siguientes pasos:

- Registramos el driver mediante la función *forName*.
- Creamos la conexión a la base de datos haciendo uso de la operación *getConnection*, que nos facilita la conexión *DriverManager*, a la que se le pasa por parámetro la dirección de la base de datos, su usuario y contraseña.



## CÓDIGO

```
public Connection conexionBBDD(){
    Connection conec = null;    //Controlamos las excepciones que aparecen
    // al interactuar con la BBDD
    try {
        Class.forName(DRIVER);
        //Crear una conexión a La Base de Datos
        conec = DriverManager.getConnection(BBDD, USUARIO, PASSWORD);
    } catch (Exception errores) {
        //Control de errores de La conexión La BBDD
        System.err.println("Se ha producido un error al conectar con la Base de
        Datos.\n" + errores);
    }
    return conec;
}
```

Estos son los pasos que tenemos que seguir para poder establecer la conexión a una base de datos. Ahora nos falta indicar la finalización de dicha conexión cuando ya no sea necesaria (cuando finalicemos nuestro programa).

Necesitamos crear una función en la que vamos a utilizar la función *Close* de la función *Connection* y, de esta forma, ya podemos cerrar la conexión.

## CÓDIGO

```
public void cerrarConexion(Connection conection){
    try{
        //Cierre de conexión
        conection.close();
    }catch(SQLException e){
        //Controlamos excepción que se pueda producir al cierre de la conexión
        System.err.println("Se ha producido un error al conectar con la Base de
        Datos." + e);
    }
}
```

## 13.2. Recuperación y manipulación de información

Para poder recuperar y manipular información debemos diseñar funciones que combinen el lenguaje SQL de recuperación o manipulación de datos, con las diferentes instrucciones o consultas de SQL.

Las diferentes cláusulas SQL que podremos utilizar son:

- **SELECT:** consultas de bases de datos
- **INSERT:** para introducir nuevos datos
- **UPDATE:** modifica o actualiza los datos almacenados
- **DELETE:** para eliminar datos

A continuación, detallaremos un ejemplo para ver el uso de estas instrucciones en un programa Java:

### CÓDIGO

```
public void insertData() {
    Connection conec = conexionBBDD();
    if (conec != null)
    try {
        //Datos a insertar
        String consultaInsercion = "INSERT INTO sorteo (fecha, num1, num2, num3,
num4, num5, complementario) " +
"VALUES ('25/11/2020', '23', '34', '45', '65', '34','9');"
        System.out.println(consultaInsercion);
        // Creación del Statement para poder realizar las consultas.
        Statement consulta = conec.createStatement();
        // Ejecución de la consulta
        consulta.executeUpdate(consultaInsercion);
        System.out.println("Datos insertados correctamente");
        // Cierre del Statement
        consulta.close();
    } catch (SQLException e) {
        System.err.println("Se ha producido un error al insertar en la
Base de Datos.\n" + e);
    } finally {
        //Cierre de la conexión.
        cerrarConexion(conec);
    }
}
```

En el ejemplo anterior, utilizamos el método `executeUpdate` para realizar la inserción de unos datos.

Veamos los métodos más utilizados:

Método	Descripción
<code>executeUpdate</code>	Utilizado para realizar inserciones, actualizaciones y eliminaciones.
<code>executeQuery</code>	Se utiliza para operaciones de tipo <code>select</code> . Nos devolverá un objeto del tipo <code>ResultSet</code> , el cual contendrá el resultado de nuestras consultas.
<code>execute</code>	Método utilizado para cualquier tipo de consulta SQL. El resultado devuelve un booleano. Si el resultado es verdadero querrá decir que ha recuperado resultados y los podremos obtener mediante el método <code>getResultSetm</code> . En caso contrario, devolverá un valor falso.

Veamos un ejemplo del método `execute`:

## CÓDIGO

```

public void getData() {
    Connection conec = conexionBBDD();
    if (conec != null)
    try {
        //Datos a consultar
        String consultaSeleccion = "SELECT * FROM sorteo;";
        System.out.println(consultaSeleccion);
        // Creación del Statement para poder realizar las consultas.
        Statement consulta = conec.createStatement();
        // Ejecución de la consulta
        if(consulta.execute(consultaSeleccion)){
            ResultSet resultSet = consulta.getResultSet();
            while(resultSet.next()){
                Sorteo sorteo = new Sorteo
                    (resultSet.getInt("id"),
                     resultSet.getString("num1"),
                     resultSet.getString("fecha"),
                     resultSet.getString("num2"),
                     resultSet.getString("num3"),
                     resultSet.getString("num4"),
                     resultSet.getString("num5"),
                     resultSet.getString("complementario")
                    );
                System.out.println(sorteo.toString());
            }
        }

        System.out.println("Datos recuperados correctamente");
        // Cierre del Statement
        consulta.close();
    } catch (SQLException e) {
        System.err.println("Se ha producido un error al insertar en la
Base de Datos.\n" + e);
    } finally {
        //Cierre de la conexión.
        cerrarConexion(conec);
    }
}

```

## 14. Diseño de programas con lenguajes de POO para gestionar bases de datos objeto-relacionales

Una **base de datos objeto-relacionales** es una base de datos relacional a la cual se le añade una extensión para poder programar sus tablas o relaciones, de forma que se pueda orientar a objetos. Gracias a esta extensión se puede guardar un objeto en una tabla que incluso puede tener una referencia con respecto a una relación de otra tabla. Se podría decir que es una base de datos híbrida que alberga dos modelos: el **modelo relacional** y el **modelo orientado a objetos**.

Las principales características de este tipo de bases de datos son que pueden definir tipos de datos más complejos, usar colecciones o conjuntos (por ejemplo: arrays), representar de forma directa los atributos compuestos y almacenar objetos de gran tamaño.

Este tipo de bases de datos permitirá aplicar: **herencia, abstracción y encapsulación**, típicas de la programación orientada a objetos. Puede haber herencia a nivel de tipos, en la que el tipo derivado hereda de la superclase o clase padre atributos o métodos. O puede haber herencia a nivel de tabla, en la que la clave primaria de la tabla padre es heredada por la tabla hija y los atributos heredados no necesitan ser guardados.

Un ejemplo de este tipo de bases de datos son las de Oracle, que implementan el modelo tradicional relacional, pero también implementan un modelo orientado a objetos en su sistema de gestión de la base de datos.

### 14.1. Establecimiento de conexiones

Las conexiones y la recuperación y/o manipulación de la información se van a llevar a cabo de la misma forma en el caso de las bases de datos objeto-relacionales. Esto se debe a que el código en Java no cambia, sino que lo hace la base de datos desde su SGBD correspondiente.

Por tanto, seguiremos los mismos pasos que hemos detallado en el apartado anterior y veremos más tipos de ejemplos.

## 14.2. Recuperación y manipulación de la información

En el ejemplo anterior hemos visto la introducción de información en la base de datos. En este apartado vamos a ver la consulta de la información almacenada en tablas. Para poder realizar consultas y manipular la información deberemos seguir estos pasos:

1. Primero creamos la sentencia de conexión (**Statement**)
2. Después, obtenemos en **resultSet** los datos de la consulta correspondiente (que la lanzamos mediante el método **executeQuery**)
3. El siguiente paso debe ser **tratar la consulta** que hemos realizado según las indicaciones del enunciado
4. Por último, **cerraremos la conexión** (que previamente habíamos abierto)

Veamos estos pasos en el siguiente ejemplo:

### CÓDIGO

```
//Controlamos las excepciones del sistema
try{
    String SalidaAmostrar;
    //Creamos la sentencia
    Statment consulta =con.createStatement();
    //Obtenemos el ResultSet con los datos de la consulta
    ResultSet Salida =consulta.executeQuery("SELECT * FROM sorteo;");
    //Iteramos mientras tengamos registros en el ResultSet
    while(salida.next()){
        //Preparamos y formateamos los datos que vamos a mostrar
        //en la label
        salidaAmostrar = salidaAmostrar
            +"Jornada n: "+salida.getInt("jornada")
            +"; Fecha: "
            + desFormateaFecha(salida.getDate("fecha"))
    }
}
```

```
+"; Combinación"+salida.getInt("num1")
+", "+salida.getInt("num2")+", "
        +salida.getInt("num3")+", "
+salida.getInt("num4")+", "
        +salida.getInt("num5")+", "
+salida.getInt("complementario");
}

//Cerramos las conexiones
conectado.cerrarConexion(consulta);
conectado.cerrarConexion(salida);
```

## 15. Diseño de programas con lenguajes de POO para gestionar las Bases de Datos Orientadas a Objetos (BBDDOO)

### 15.1. Introducción a Bases de Datos Orientadas a Objetos (BBDDOO)

Para las Bases de Datos Orientadas a Objetos (BBDDOO) vamos a utilizar un gestor combinado con el lenguaje de programación Java. No es muy diferente a los SGBDDOO (Sistemas de Gestión de Base de Datos Orientada a Objetos) utilizados en los gestores de las bases de datos relacionales.

### 15.2. Características de las Bases de Datos Orientadas a Objetos (BBDDOO)

A continuación, vamos a indicar las diferentes **características** que presentan las **BBDDOO**:

- Se diseñan de la misma forma que los POO, es decir, debemos pensar como si se tratara de un programa real
- Cada tabla que definamos en las bases de datos relacionales va a convertirse, a partir de ahora, en objetos de nuestra base de datos
- Cada objeto que definamos debe tener un identificador único que los diferencie del resto
- Ofrecen la posibilidad de almacenar datos complejos sin que necesitemos darle un trato más complejo de lo normal
- Los objetos que se utilicen en la base de datos pueden heredar los unos de los otros
- Es el usuario el que se va a encargar de decidir los elementos que van a formar parte de la base de datos con la que se esté trabajando



- Los SGBDDOO son los que se van a encargar de generar los métodos de acceso a los diferentes objetos
- Añaden más características propias de la POO como, por ejemplo, la sobrecarga de métodos y el polimorfismo

### 15.3. Creación de Bases de Datos Orientadas a Objetos (BBDDOO)

En el caso de las BBDDOO deberemos de buscar un gestor de BD que admita este formato. Existen varios y en esta unidad nos centraremos en Oracle, con su lenguaje PL/SQL. Este nos permitirá crear objetos que posteriormente podremos recuperar desde nuestro programa en Java y trabajar de forma muy sencilla con la base de datos, dado que no tendremos que convertir objetos.

Para trabajar con Oracle, debemos tener instalada en nuestro ordenador la base de datos de Oracle. Podemos descargarla desde este enlace. Hay que tener en cuenta que la versión express no permite la creación de nuevas BD y la propia instalación nos creará una.

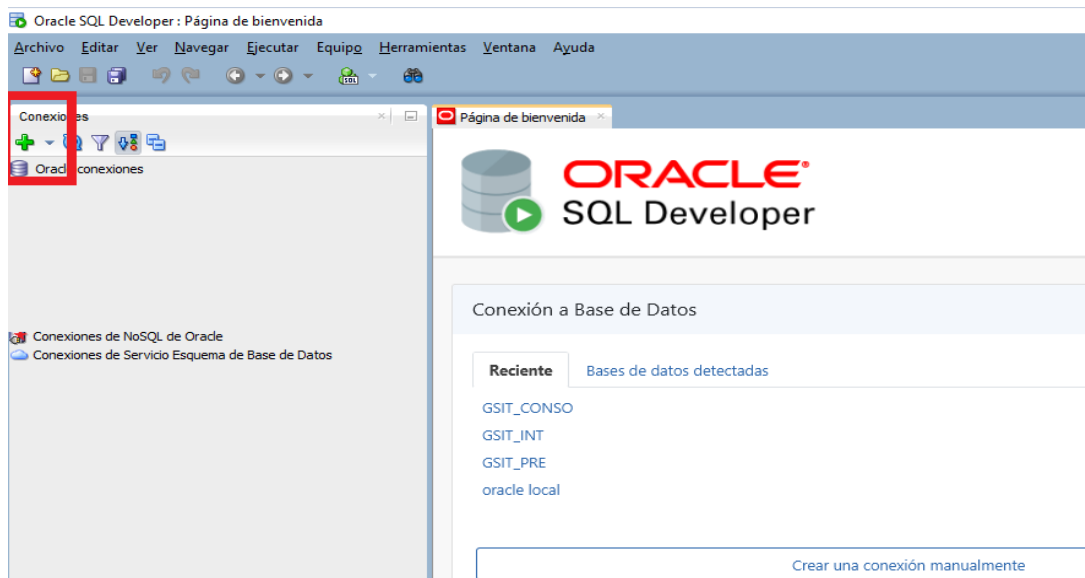
<https://www.oracle.com/database/technologies/xe-downloads.html>

Durante la instalación, se nos solicitará una nueva contraseña para la base de datos. Es muy importante guardarla, ya que la necesitaremos posteriormente. En la última ventana de la instalación nos informará de la URL de conexión a la base de datos. Generalmente es “localhost:1521/xe”, donde **localhost** es la dirección donde se encuentra **1521**, el puerto de conexión por defecto y **xe** el nombre de la base de datos.

Una vez instalada la base de datos Oracle, tenemos que instalarnos el Sql developer, que es la aplicación que nos va a permitir gestionar las conexiones con nuestra base de datos Oracle. Podemos realizar la instalación accediendo a este enlace:

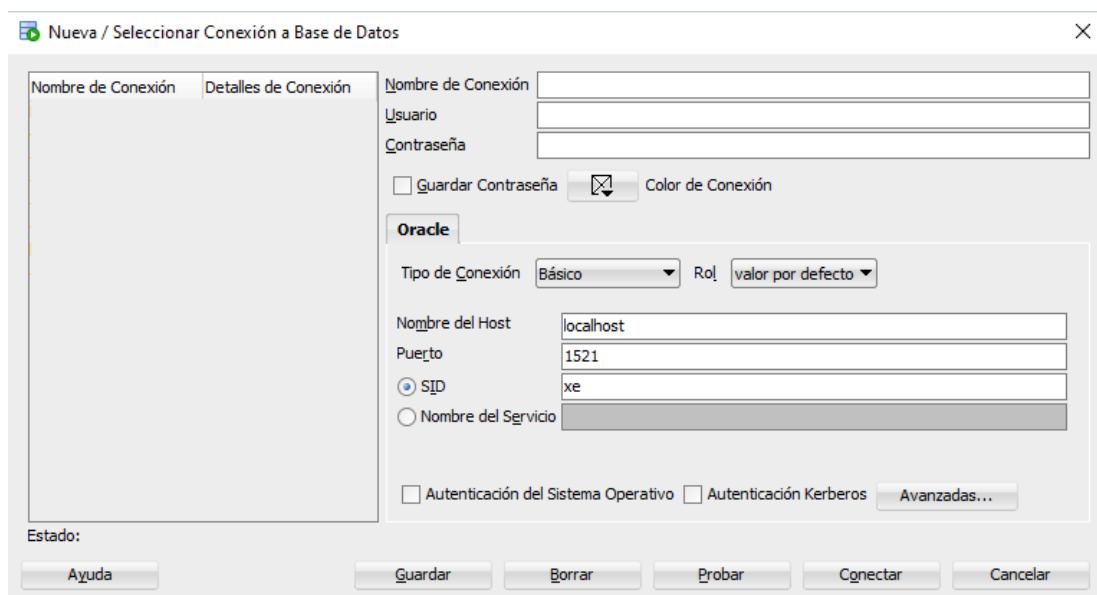
<https://www.oracle.com/database/technologies/appdev/sql-developer.html>

Una vez instalado, tenemos que abrir este programa y veremos que en la parte superior izquierda tenemos la opción de crear una nueva conexión a la base de datos:



*Ilustración 1. Captura para mostrar cómo se realiza una nueva conexión.*

Se nos abrirá una ventana como esta:



*Ilustración 1. Ventana para añadir datos de la conexión a la base de datos de Oracle.*

Nueva / Seleccionar Conexión a Base de Datos

Nombre de Conexión	Detalles de Conexión
Nombre de Conexión	oracle local
Usuario	SYSTEM
Contraseña	.....
<input checked="" type="checkbox"/> Guardar Contraseña	<input type="checkbox"/> Color de Conexión
<b>Orade</b>	
Tipo de Conexión	Básico
Rol	valor por defecto
Nombre del Host	localhost
Puerto	1521
<input checked="" type="radio"/> SID	xe
<input type="radio"/> Nombre del Servicio	
<input type="checkbox"/> Autenticación del Sistema Operativo	<input type="checkbox"/> Autenticación Kerberos
Avanzadas...	

Estado:

Ayuda Guardar Borrar Probar Conectar Cancelar

Ilustración 3. Datos que usaremos para la conexión.

En esta pantalla tenemos que configurar los datos de conexión que hemos rellenado durante la instalación. Con este paso, lo que vamos a hacer es crear una conexión con el usuario del sistema de esta base de datos, que nos permitirá gestionar todo lo que realicemos en Oracle. Es recomendable tener un usuario admin, como el que hemos creado durante la instalación para poder crear los usuarios y las distintas bases de datos de nuestras aplicaciones.

En este caso, en la sección *Nombre de conexión* añadiremos un nombre para identificar la conexión. En *Usuario*, añadiremos el que hemos creado durante la instalación y su contraseña. Es importante guardar esta información, ya que resulta esencial para poder administrar las conexiones.

Como *nombre de host*, añadiremos *localhost*, ya que trabajaremos en un entorno local. El puerto es el 1521 por defecto y el *sid* es el *xe*, que es el por defecto durante la instalación. Antes de guardar la conexión, es recomendable darle al botón de *Probar* para comprobar que la conexión se realiza correctamente. Si no da un resultado correcto, tendremos que verificar los datos introducidos. Al final, quedará algo así:

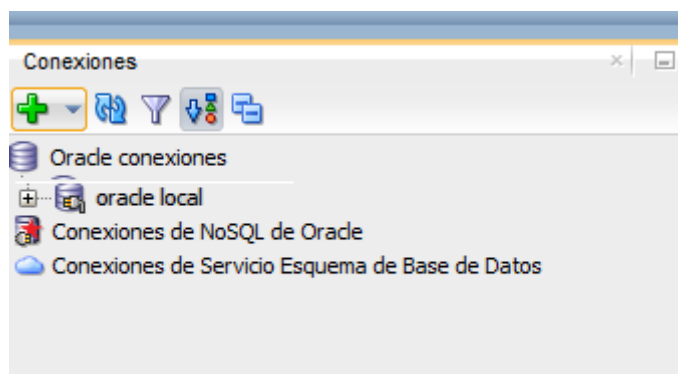


Ilustración 4. Conexión a la base de datos de Oracle realizada correctamente.

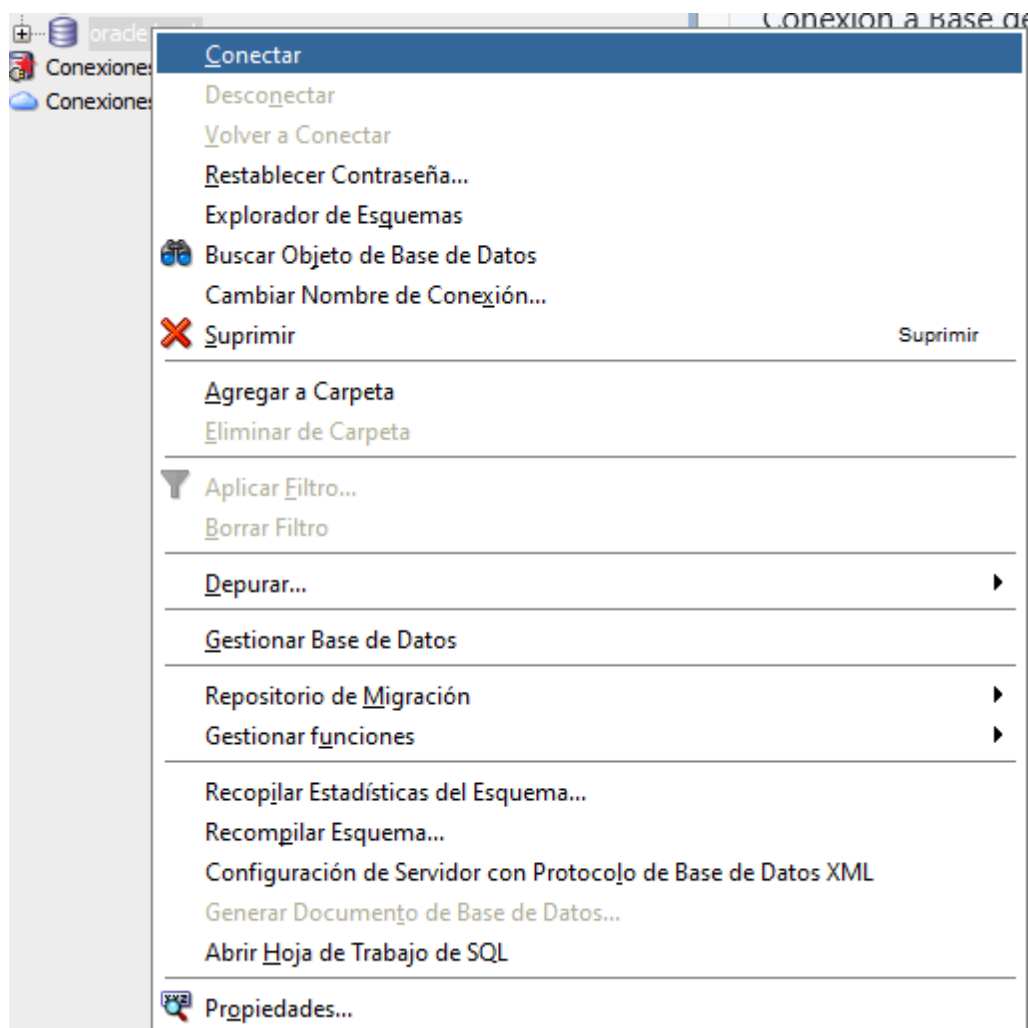


Ilustración 5. Proceso para conectarse a la base de datos.

Para conectarnos a la base de datos, bastará con dar con el botón derecho del ratón encima de la conexión y conectar. A continuación, debemos crear un usuario para nuestra aplicación usando esta sentencia:

```
create user ilerna identified by 'contrasena';
```

Podemos usar cualquier nombre y cualquier contraseña. Hay que guardarla bien, porque nos será de utilidad para establecer más adelante la conexión entre la base de datos y nuestra aplicación. Tenemos que ejecutarlo en la conexión que hemos abierto a nuestra conexión *admin* de la base de datos.

Con esto ya tenemos la base de datos configurada. Lo más recomendable es crear una nueva conexión, como hemos hecho en los pasos anteriores, con los datos de esta conexión nueva. Nos va a permitir ver las tablas de nuestra base de datos.

Una vez instalada la base de datos, tenemos que añadir el driver JDBC de Oracle (OJDBC) a proyecto Java. Como nosotros usamos la versión de Java 8, necesitamos descargar la versión 8 del driver de Oracle. Si usáramos una versión Java 10 o superior, necesitaríamos la versión de OJDBC 10.

Podemos descargarlo desde este enlace:

[https://download.oracle.com/otn-pub/otn\\_software/jdbc/ojdbc8.jar](https://download.oracle.com/otn-pub/otn_software/jdbc/ojdbc8.jar)

Una vez descargado, tenemos que añadir la librería a nuestro proyecto Java. Tenemos dos métodos para realizar esta tarea según qué tipo de proyecto tengamos. Si se trata de un proyecto Java debemos dirigirnos con el botón derecho a nuestro proyecto e ir a *Properties>Java Built Path*. Cuando se nos abra esta pestaña, deberemos darle al botón *Add External JAR* y añadir el jar descargado.

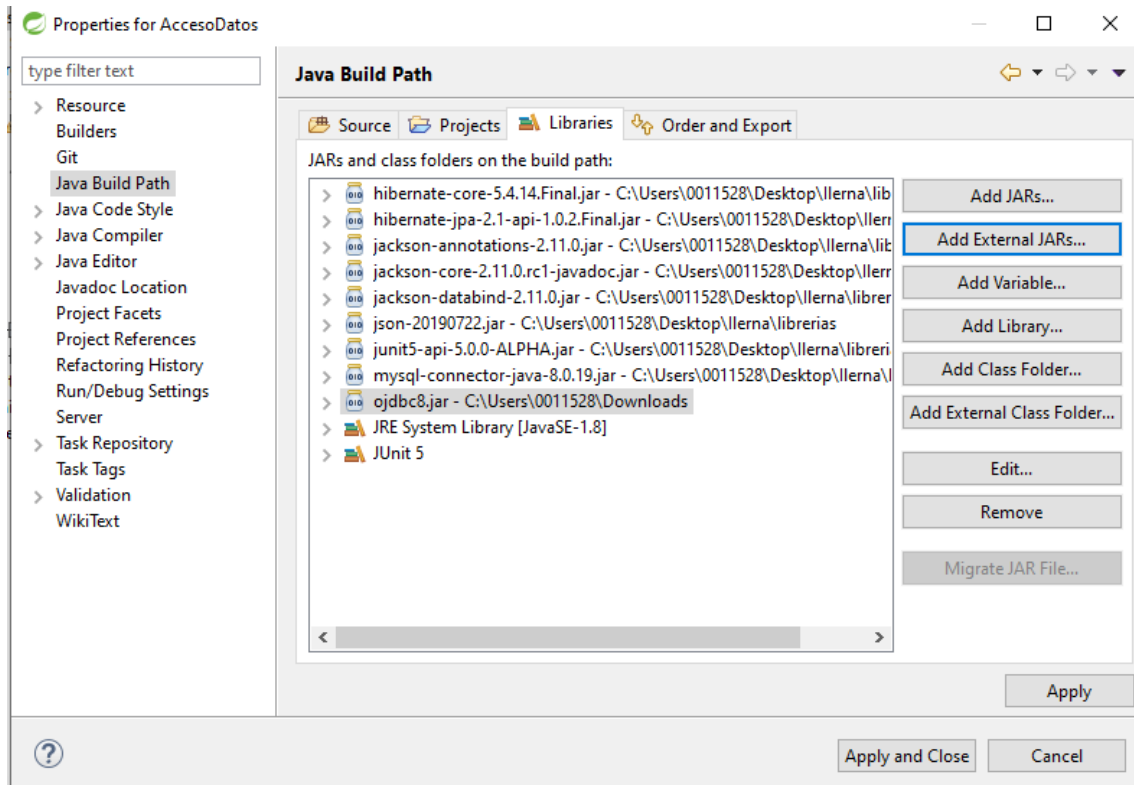


Ilustración 6. Ventana que nos permitirá añadir el driver de Oracle a nuestra aplicación.

Tipo	Descripción	Tipo driver
<b>Driver thin</b>	Para aplicaciones enfocadas a cliente sin una instalación Oracle	<i>thin</i>
<b>Driver OCI</b>	Para aplicaciones enfocadas a cliente con una instalación Oracle	<i>oci</i>
<b>Driver thin servidor</b>	Es igual que el driver thin, pero se ejecuta dentro de un servidor de oracle con acceso remoto	<i>thin</i>
<b>Driver interno servidor</b>	Se ejecuta dentro del servidor elegido	<i>kprb</i>

### Preparación para la conexión a la base de datos

A continuación, tenemos que preparar la URL que utilizaremos para realizar la conexión a la base de datos.

La sintaxis de la URL para la base de datos de Oracle sigue esta estructura:

```
jdbc:oracle:<tipodriver>:@<basededatos>  
o  
jdbc:oracle:<tipodriver>:<nombreusuario>/<contraseña>@<basededatos>
```

Para el tipo de driver puede ser: *thin*, *oci* o *kprb*. Para nuestra aplicación Java, utilizaremos el tipo de driver *thin*, ya que tenemos una instalación Oracle y para que funcione la aplicación debemos arrancarla en un servidor.

Para el *nombreusuario*, debemos utilizar el usuario y la contraseña de acceso para la base de datos que hemos creado en pasos anteriores.

Con los datos de la base de datos creada y el usuario y la contraseña podemos terminar de construir la URL que necesitaremos para realizar la conexión a la base de datos. Tendremos que indicar el nombre del host, en este caso *localhost*, el puerto 1521 y el nombre de la base de datos. Todo separado por dos puntos. Tiene que quedar algo parecido a esto:

```
jdbc:oracle:thin:ilerna/contrasena@localhost:1521:xe
```

## Registrar el driver de Oracle JDBC

El proceso para registrar el driver de Oracle es muy parecido al que ya vimos durante el tema 1, pero en esta ocasión el nombre de la clase que vamos a utilizar será este:

```
oracle.jdbc.OracleDriver
```

Para realizar el registro del driver se utiliza:

```
Class.forName("oracle.jdbc.OracleDriver");
```

Desde la versión 6 de Java, registrar el driver como hemos explicado se convierte en un paso opcional. Siempre que añadamos la librería de *OJDBC* a nuestro proyecto, el registro del driver se hará de manera automática. No está de más, saber cómo se realiza de manera manual, ya que en según el tipo de base de datos que gestionemos, puede sernos útil.

## Establecer la conexión con Oracle

Con *OJDBC* el proceso de conexión con la base de datos es igual que en *JDBC*. Vamos a recordar cómo se realizaba la conexión. Para establecer la conexión con la base de datos, necesitamos llamar al método `getConnection()` de la clase `DriverManager`. Tenemos estos tres métodos para realizar la conexión:



- **getConnection (String url):** utilizaremos este método cuando la url contenga toda la información que necesitamos para conectarnos a la base de datos. Utilizaremos el driver *thin* para la conexión, con el nombre de usuario *ilerna*, la contraseña y los datos de la base de datos.

```
String url =
"jdbc:oracle:thin:ilerna/contrasena@localhost:1521:xe";
Connection conn = DriverManager.getConnection(url);
if (conn != null) {
    System.out.println("Conexión realizada.");
}
```

La clase DriverManager llamará al método getConnection, pasándole el string con la URL que hemos construido.

- **getConnection(String url, String usuario, String contraseña):** en este método se pasan por parámetros los datos del usuario y la contraseña, además de la URL. La implementación sería así:

```
String url = "jdbc:oracle:thin:@xe";
String usuario = "ilerna";
String pass = "contrasena";
```

Para demostrar cómo realizar todos los pasos, vamos a crear una clase auxiliar que se encargue de realizar el registro del driver y la conexión con la base de datos.

```

publicclass ConexionOracle {
    publicstaticvoidmain(String[] args) {
        Connection conn1 = null;
        Connection conn2 = null;
        try {
            // Se encarga de registrar el driver de Oracle, es opcional
            Class.forName("oracle.jdbc.OracleDriver");
            // Método 1
            String url = "jdbc:oracle:thin:ilerna/contrasena@localhost:1521:tema4";
            conn1 = DriverManager.getConnection(url);
            if (conn1 != null) {
                System.out.println("Conectado usando el metodo 1");
            }
            // Método 2
            String url2 = "jdbc:oracle:thin:@tema4";
            String usuario = "ilerna";
            String pass = "contrasena";
            conn2 = DriverManager.getConnection(url2, usuario, pass);
            if (conn2 != null) {
                System.out.println("Conectado usando el metodo 2");
            }
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        } catch (SQLException ex) {
            ex.printStackTrace();
        } finally {
            try {
                conn1.close();
                conn2.close();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }
    }
}

```

## 15.4. Mecanismos de consulta

Para la **creación, modificación, eliminación e inserción** de objetos PL/SQL, podemos usar la interfaz Statement, tal y como introducimos en el Tema 2. Este tipo de objeto se utiliza para conexiones de carácter general. Es bastante útil cuando queremos usar consultas estáticas SQL. Este tipo de objeto no acepta parámetros. Las sentencias más utilizadas son todas aquellas que no necesitan datos de los objetos Java. En nuestro caso, solo queremos ejecutar una sentencia. Como nuestra base de datos es apta para PL/SQL, con la interfaz Statement podemos realizar esta tarea.

Vamos a mostrar un ejemplo de implementación:

```
String url = "jdbc:oracle:thin:ilerna/ilerna" +
"@localhost:1521:xe";
Connection conn = DriverManager.getConnection(url);
if (conn != null) {
    System.out.println("Conexión realizada.");
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
    }
    String sql;
    sql = "CREATE Or Replace TYPE ProductoType AS OBJECT" +
    "(id NUMBER," +
    " nombre VARCHAR2(15)," +
    " descripcion VARCHAR2(22)," +
    " precio NUMBER(5, 2)," +
    " dias_validez NUMBER" +
    ");";
    System.out.println(sql);
    stmt.execute(sql);

    sql = "CREATE TABLE productos (producto ProductoType, contador NUMBER);";
    System.out.println(sql);
    stmt.execute(sql);
} catch (SQLException e) {
    e.printStackTrace();
} try {
    stmt.close();
    conn.close();
} catch (SQLException e) {
    e.printStackTrace();
}
}
```

En primer lugar, tenemos la conexión que nos servirá para conectarnos a nuestra nueva base de datos. En segundo, el Statement que servirá para ejecutar las sentencias.

Después de declarar los objetos que utilizaremos, vamos a inicializar la instancia de la interfaz Statement con ayuda de la instancia Connection y la llamada al método `createStatement()`.

Una vez realizados todos estos pasos, ya podemos crear en un String las sentencias PL/SQL. Con esta estructura, podemos realizar todo tipo de acciones: desde crear tablas y objetos a eliminar, actualizar o introducir datos. El proceso para estas acciones será siempre con esta estructura. Para ejecutar la operación, solo tenemos que utilizar `stmt.execute()` y le pasaremos la consulta por parámetro. Si no se produce ningún error, la operación habrá terminado y solo nos quedará cerrar los objetos abiertos.

Debemos de tener en cuenta que cada vez que creemos este tipo de objetos, tendremos que cerrarlos. Por eso, es necesario utilizar el método `close()` para cerrar la conexión.

## 15.5. Tipos de datos objeto. Atributos y métodos

Los tipos de datos se subdividen en:

- **Number:** engloba todos los tipos de datos. Podemos encontrar subtipos de NUMBER. Los más importantes que podemos utilizar son:

Subtipo	Descripción
<b>PLS_INTEGER</b>	Valor numérico de tipo integer que puede estar en el rango -2,147,483,648 hasta 2,147,483,648 representado en 32 bits
<b>BINARY_INTEGER</b>	Integer binario que puede estar en el rango -2,147,483,648 hasta 2,147,483,648 representado en 32 bits
<b>FLOAT</b>	Tipo de dato float con un máximo de 126 dígitos binarios
<b>INT</b>	Integer no permite decimales con un máximo de 38 dígitos
<b>NUMBER</b>	Número de punto fijo o de punto flotante con un valor absoluto en el rango 1E-130 a (pero sin incluir) 1.0E126. También puede representar el 0.
<b>REAL</b>	Spota decimales float de un máximo 63 bits

*Tabla 1. Tipos de datos de un objeto.*

- **Boolean:** este tipo de dato soporta el valor true, false o null. Se utilizan normalmente en estructuras de control de flujo como un if, case o loop. Este tipo de datos no los podemos encontrar en SQL. Solo en tipos de lenguaje orientado a objetos.
- **Character:** engloba todos esos tipos de datos para la gestión de caracteres como char, varchar2, long, raw, long raw, rowid y unrowid.
- **Datetime:** se utiliza para almacenar datos de tipo fecha. En la base de datos se guardan como un tipo numérico, por lo que da la posibilidad de trabajar con este tipo de datos con operaciones aritméticas.

## 15.6. Tipos de datos colección

En Oracle PLSQL existen 3 tipos de colecciones:

- Matriz asociativa: primer tipo de colección disponible en PLSQL
- Tablas anidadas: disponibles desde la versión 8 de la BD de Oracle
- Varray (variable array): colección en la que se debe de saber el tamaño máximo de elementos permitidos dentro de la colección. A causa de este hecho es poco usada, ya que generalmente no conocemos el tamaño máximo de elementos que debemos de introducir.

## 15.7. Modelo de Datos Orientado a Objetos

Cuando hablamos del Modelo de Datos Orientado a Objetos debemos indicar que es **una extensión del paradigma de la POO**.

Para trabajar con las bases de datos vamos a utilizar un tipo de objetos de entidad muy similar al de las bases de datos puras, pero con una gran diferencia: los objetos del programa van a desaparecer al finalizar su ejecución. En cambio, los objetos de la base de datos sí que van a permanecer.

## 15.8. Relaciones

Las relaciones se pueden representar mediante claves ajenas. No existe una estructura de datos en sí que forme parte de las bases de datos para la representación de los enlaces entre las diferentes tablas.

Gracias a las relaciones, podemos realizar concatenaciones (*join*) de las diferentes tablas. Sin embargo, los vínculos de las BDOO deben incorporar en las relaciones de cada objeto a los identificadores de los diferentes objetos con los que se van a relacionar.

Entendemos que un identificador de un objeto es un atributo que poseen los objetos y que es asignado por el SGBD. Por lo que este es el único que los puede utilizar.

Este identificador puede ser un valor aleatorio o, en algunos casos, puede que almacene una información necesaria que permita encontrar el objeto en un fichero determinado de la base de datos.

Cuando tenemos que representar relaciones entre diferentes datos debemos tener en cuenta que:

- El identificador del objeto no debe cambiar mientras forme parte de la base de datos.
- Las relaciones que están permitidas para realizar cualquier tipo de consulta sobre la base de datos son aquellas que tienen almacenados aquellos identificadores de objetos que se pueden utilizar.

El Modelo de Datos Orientado a Objetos permite:

- Atributos multivaluados.
- Agregaciones denominadas conjuntos (sets) o bolsas (bags).
- **Si queremos crear una relación uno a muchos (1 .. N):** definimos un atributo de la clase objeto en la parte del uno con el que se va a relacionar. Este atributo va a tener el identificador de objeto del padre.
- **Las relaciones muchos a muchos (N .. N):** se pueden representar sin crear entidades intermedias. Para representarlas, cada clase que participa en la relación define un atributo que debe tener un conjunto de valores de la otra clase con la que se quiere relacionar.

- Además, las BDOO deben soportar dos tipos de herencia: la relación “es un” y la relación “extiende”.
  - o “**es un**” (generalización- especialización): crea jerarquías donde las diferentes subclases que existan son tipos específicos de la superclase
  - o “**extiende**”: una clase expande su superclase en vez de hacerla más pequeña en un tipo más específico

## 15.9. Integridad de las relaciones

Los identificadores de los objetos se deben corresponder en ambos extremos de una relación para que una BDOO funcione de forma correcta. La integridad en una base de datos es una de las características más importantes a tener en cuenta. Los datos almacenados en el campo de “clave ajena” y guardados en una tabla tienen que estar contenidos en el campo clave de la tabla a la que se relaciona.

## 15.10. UML

**UML (Unified Modeling Language)** es un **lenguaje modelado unificado** que visualiza, especifica y documenta todas las partes necesarias para desarrollar el software. Aunque se puede utilizar para modelar tanto sistemas de software como de hardware.

Para poder desarrollar estas tareas, utiliza una serie de diagramas en los que se pueden representar varios puntos de vista del modelado.

UML es un tipo de lenguaje que se suele utilizar para documentar.

A continuación, vamos a desarrollar las dos principales versiones de UML que suelen utilizarse hoy en día:

- **UML 1. X (Desde 1. 1, ..., 1. 5):** se empezó a utilizar a finales de los 90 y en los años siguientes fueron incorporando una serie de mejoras.
- **UML 2. X (Desde 2. 1, ..., 2. 6, ...):** aparece sobre el año 2005 y va aportando y desarrollando nuevas versiones.

A partir de la versión UML 2.0 se definen 13 tipos de diagramas diferentes que, a su vez, se dividen en 2 categorías:

- **Diagramas de estructuras (parte estática):** Define los elementos que deben existir en un sistema de modelado.
  - Diagrama de clases
  - Diagrama de estructuras compuestas
  - Diagrama de objetos
  - Diagrama de componentes
  - Diagrama de implementación (despliegue).
  - Diagrama de paquetes.
  
- **Diagramas de comportamiento:** Basados en lo que debe suceder en el sistema.
  - Diagrama de interacción:
    - Diagrama de secuencia
    - Diagrama resumen de interacción
    - Diagrama de comunicación
    - Diagrama de tiempo
  - Diagrama de actividad
  - Diagrama de casos de uso
  - Diagrama de máquina de estado

En otros modulos se profundizará en muchos de estos tipos de diagramas.



## 15.11. El modelo estándar ODMG

El **modelo ODMG (Object Data Management Group)** es un estándar que establece la semántica que deben tener los objetos de una base de datos. Este modelo permite que tanto los diseños como la implementación puedan ser reutilizables y soportados por otros sistemas que soporten este modelo.

Este modelo de objetos **permite realizar el diseño de una BDOO** implementada desde lenguajes POO, especificando los elementos que se definirán para la persistencia en una BDOO.

Por tanto, su función es definir los elementos y la persistencia entre ellos en las BBDDOO, permitiendo que sean portables entre los sistemas de POO que los soportan. Definen así un estándar en los SGBDOO.

El modelo ODMG se compone por:

1. Modelo de Objeto
2. Lenguaje de definición de objetos ODL
3. Lenguaje de consulta de objetos OQL

A continuación, se detallan estos dos puntos.

### 15.10.1. Lenguaje de definición de objetos ODL

Es un lenguaje de especificación para definir tipos de objetos para sistemas compatibles con ODMG. Es el equivalente al DDL (Lenguaje de Definición de Datos) de los SGBD tradicionales. **Define los atributos y las relaciones** entre tipos y **especifica la signatura** de las operaciones.

### 15.10.2. Lenguaje de consulta de objetos OQL

Es un lenguaje declarativo del tipo de SQL que permite realizar consultas de modo eficiente sobre BDOO, incluyendo primitivas de alto nivel para conjuntos de objetos y estructuras. OQL no posee primitivas para modificar el estado de los objetos, ya que las modificaciones se pueden realizar mediante los métodos que estos poseen.

OQL es el lenguaje de consulta de objetos con el que vamos a poder consultar los valores de esos objetos creados, la estructura de la base de datos y los datos introducidos en ella. Es similar a los SELECT con el que consultamos las bases de datos.

## 15.12. Prototipos y productos comerciales del Sistema Gestor de Bases de Datos Orientadas a Objetos (SGBDOO)

SGBDOO significa **Sistema Gestor de Bases de Datos Orientadas a Objetos**. Podríamos definirlo como un sistema gestor de bases de datos con la característica de almacenar objetos. Para los usuarios del sistema tradicional de bases de datos esto quiere decir que se puede tratar directamente con objetos y no se tiene que hacer la traducción de registros o tablas. Debe combinar un sistema gestor de bases de datos con un sistema orientado a objetos.

Todo SGBDOO debe tener unas **características** que se pueden agrupar en dos grupos:

- **Características obligatorias:** son las características esenciales que obligatoriamente debe tener. Por un lado, están los criterios que debe tener un SGBD y, por otro lado, los que debe tener un sistema orientado a objetos.
  - Las características del **SGBD** son: persistencia, gestión del almacenamiento secundario, concurrencia, recuperación y facilidad de consultas.
  - Para el sistema **orientado a objetos**: objetos complejos, identidad de objetos, encapsulamiento, tipos y clases, herencia, sobrecarga, extensibilidad y completitud computacional.
- **Características optativas:** son características que debería implementar, pero no está obligado. Estas son: herencia múltiple, chequeo e inferencia de tipos, distribución, transacciones de diseño y versiones.

## Bibliografía

Jiménez, I. M. (2013b). *Programación* (1ª ed.). Madrid, España: Garceta.

Moreno, J. C. (2011). *Programación*. Madrid, España: Ra-Ma.

Sznajdleder, P. A. (2015). *El gran libro de Java a fondo* (3ª ed.). Buenos Aires, Argentina: Marcombo.

## Webgrafía

Oracle Help Center. (s.f.). Java Documentation. Recuperado 24 agosto, 2018, de <https://docs.oracle.com/en/java/>

ILERNA

Online

```
5 function updatePhotoDescription() {  
6     if (descriptions.length > (page * 9) + (currentImage.substring(0) - 1)) {  
7         document.getElementById("bigImageDesc").innerHTML = descriptions[page * 9 +  
8     }  
9 }  
10  
11 function updateAllImages() {  
12     var i = 1;  
13     while (i < 10) {  
14         var elementId = "foto" + i;  
15         var elementIdBig = "bigImage" + i;  
16         if (page * 9 + i - 1 < photos.length) {  
17             document.getElementById(elementId).src = "images/mini" + photos[  
18             document.getElementById(elementIdBig).src = "images/wide" +  
19         } else {  
20             document.getElementById(elementId).src = "images/mini" + photos[0].src;  
21         }  
22         i++;  
23     }  
24 }
```