ILERNA Online

Módulo 3
Programación

UF4: Programación Orientada a Objetos (POO)

Los elementos que componen un programa son siempre similares. En la mayoría de los programas podemos encontrar 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.

1. Lenguaje de programación Java

- **Es un lenguaje interpretado.** El código que diseña se denomina *bytecode* y se puede interpretar a través de una máquina virtual. 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 van a permitir 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), *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, y 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 clases.** Java cuenta con una gran colección de clases agrupadas en los diferentes directorios. Estas clases sirven al usuario para realizar alguna tarea determinada sin necesidad de tenerla que implementar.

1.1. Características

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 con 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 en el siguiente enlace:

http://www.jcp.org/en/home/index

- **Se utiliza a nivel mundial.** Java es una plataforma libre que dispone de un gran número de desarrolladores que 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.

1.2. Descarga e instalación

Descarga e instalación en Windows y Linux

Se puede realizar a través del siguiente enlace:

http://java.sun.com/javase/downloads/index.jsp

En él podemos ver las diferentes versiones del lenguaje Java.

En el siguiente enlace podemos encontrar las descargas del programa Eclipse, el cual utilizaremos para desarrollar código.

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

1.3. Estructura de un programa en Java

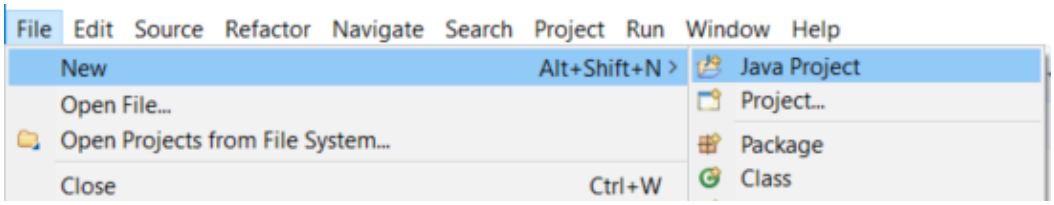
Es una estructura arbórea, muy jerarquica.

Cuando creamos un nuevo proyecto, **este va a incluir un fichero ejecutable que se debe llamar de la misma forma que el proyecto en cuestión**.

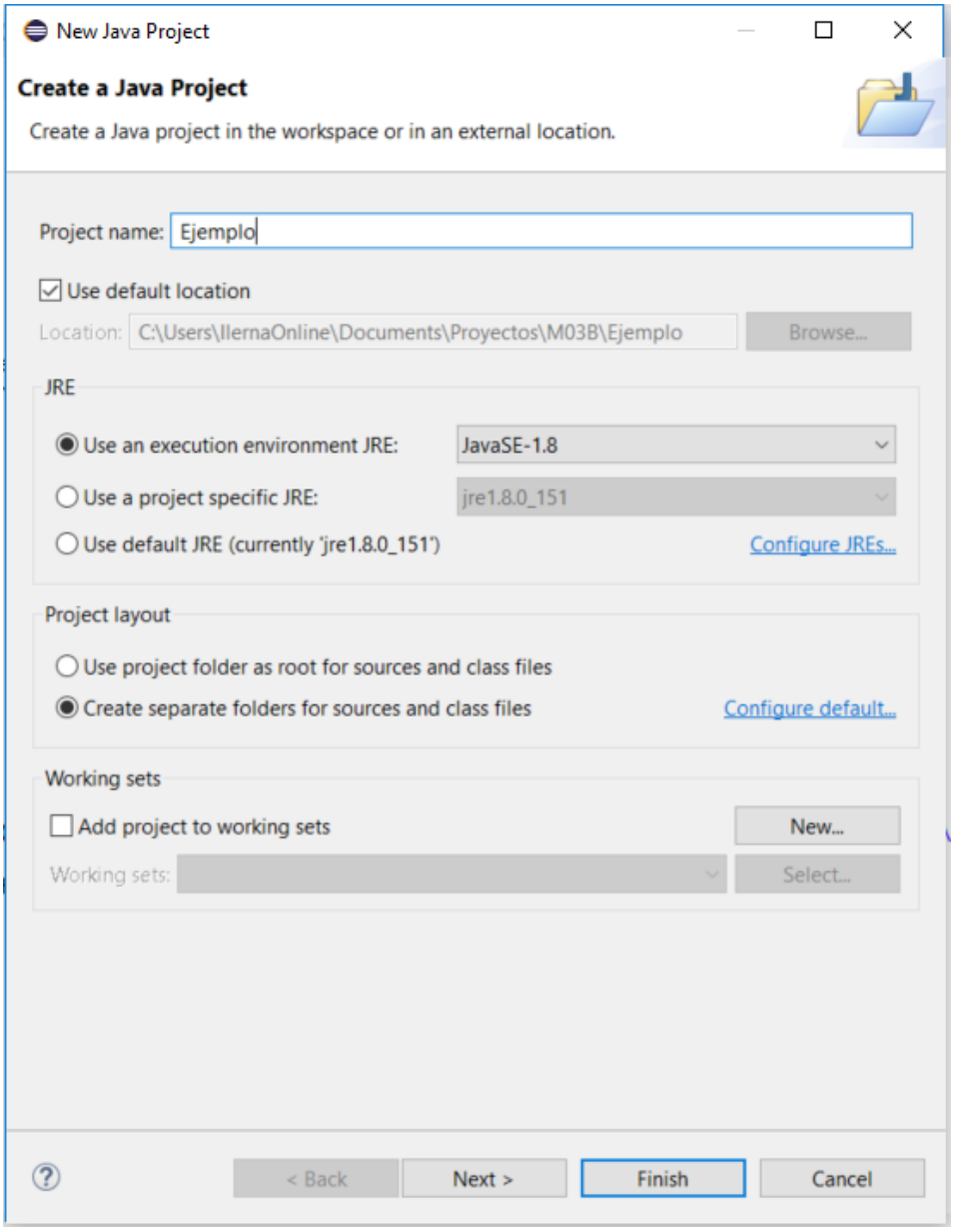
Cuando realicemos una aplicación Java, el fichero en el que se encuentra el código fuente tiene la **extensión ".java"** y, una vez compilado, el **fichero que se genera** de código intermedio (*bytecode*) va a tener la **extensión ".class"**.

Este fichero ".class" es el que utilizaremos para ejecutar nuestro programa en cualquier sistema.

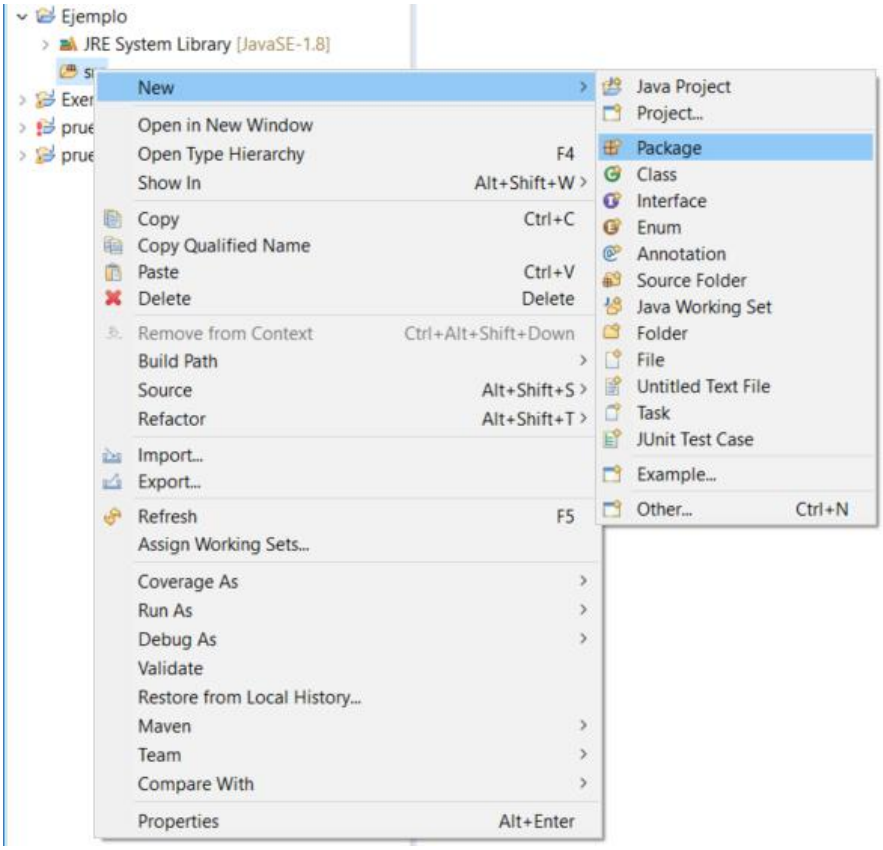
Una vez instalado el IDE Eclipse, para realizar un nuevo proyecto tendremos que, desde el menú, buscar la opción *New > Java Project* (como vemos en la siguiente imagen):

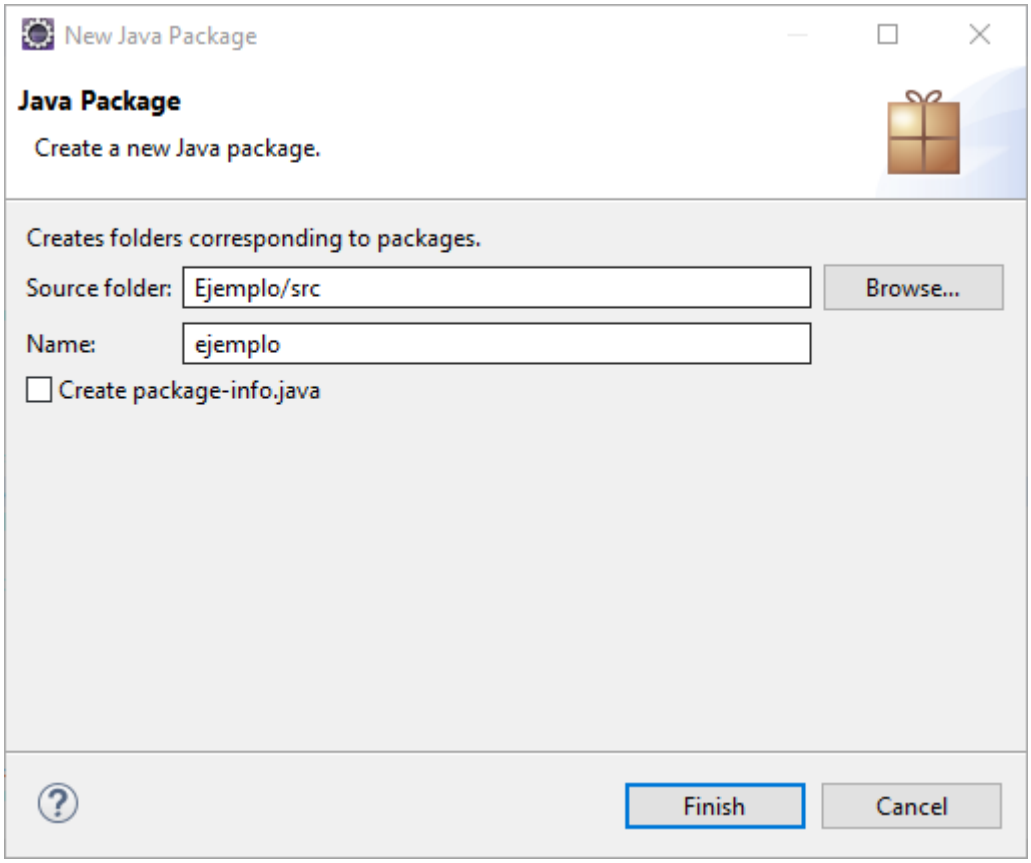


Introducimos un nombre para el proyecto:

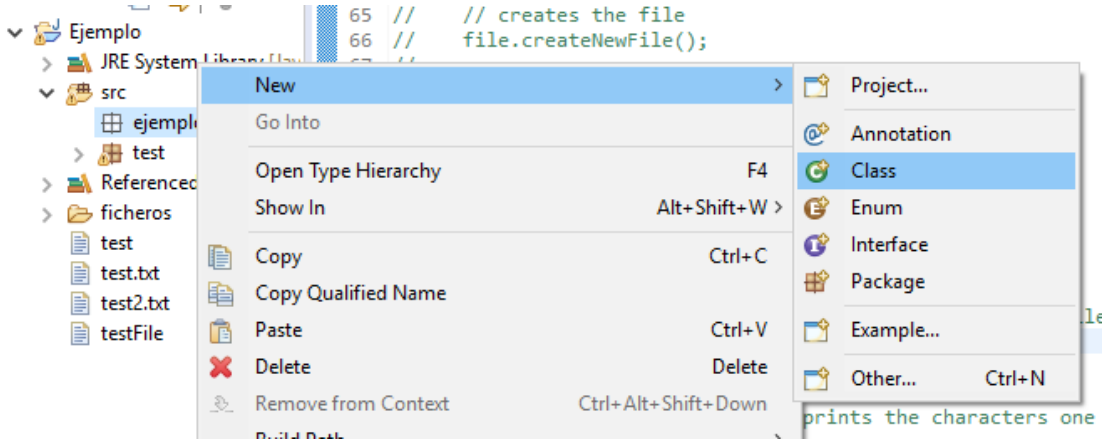


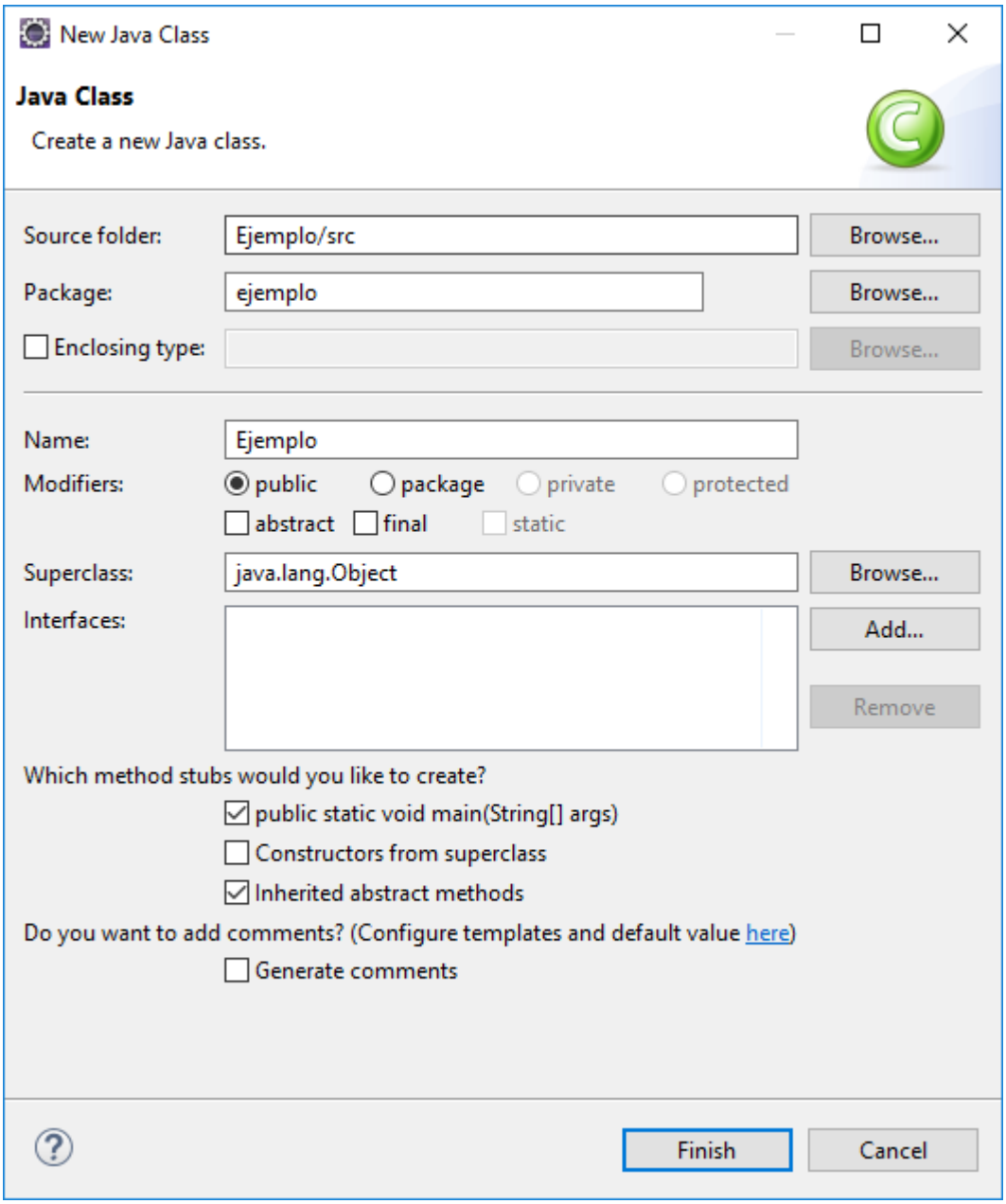
Al crear un proyecto lo que estamos creando es una carpeta que representa la carpeta "padre" que contiene todos los archivos propios del proyecto. Una vez tenemos creada la carpeta que va a contener el proyecto, pasaremos a la creación de un paquete (*package*), que nos servirá para agrupar funcionalidades. Este lo creamos dentro de la carpeta "src" que se ha generado dentro del proyecto:





Finalmente, crearemos las clases dentro del paquete que acabamos de crear. Estas clases son archivos que van a contener el código Java que generemos:




New Java Class
Java Class
Create a new Java class.
Source folder:
Ejemplo/src
Browse...
Package:
ejemplo
Browse...
Enclosing type:
Browse...
Name:
Ejemplo
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

Veamos un ejemplo del código inicial de una clase:

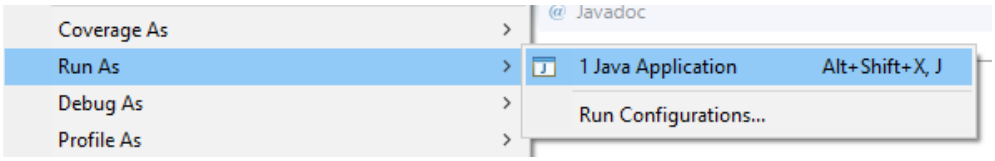
CÓDIGO:

```
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 que vamos a utilizar para es estática y pública, para que la podamos ejecutar. No devuelve ningún valor (void) y va a utilizar como parámetros un *array* de cadenas de caracteres que se necesitan para que el usuario pueda introducir valores a la hora de ejecutar 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 ayudan a llevar 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. Para poner comentarios, escribiremos los caracteres "//" para comentarios de una única línea, y "/*" o "*/" para los que contengan más de una.

2. Introducción a la programación orientada a objetos.

Según **Bruce Eckel**: "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 programación orientada a objetos 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 (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 manera que, del objeto solo se conoce 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 pueden 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.

2.1. Tipos de datos. 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 que se realicen 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 usar los tipos de datos que se usan en Java: *int*, *float*, *char*, *boolean*, etc. Que veremos a continuación.

En el siguiente ejemplo vemos cómo declarar un mensaje en una variable llamada "txt" y mostrarlo por 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, debemos 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 final 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 de datos 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 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 una array) 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 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 poder 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

Si se utilizan 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:

```
public class reglas {
    public static void 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 variables pueden ir tomando valores diferentes.

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

CÓDIGO:

```
public class reglas2 {
    public static void 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 caracteres que siguen el formato Unicode. En Java, existen varias clases para el manejo de estas cadenas de caracteres, y cuentan con bastantes 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.

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*). Además, cuenta con un algoritmo que, si se cumple una determinada condición será verdadero (***true***) y, si no, será falso (***false***).

CÓDIGO:

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

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

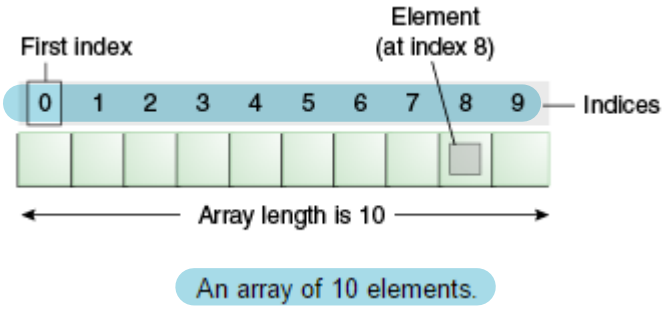
CÓDIGO:

```
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 con la siguiente imagen:

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



An array of 10 elements.

Debe cumplir con la siguiente 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 -1.
- Java hace una comprobación de índices, de forma que, si se quiere acceder a uno que no está, lanza una excepción.
- Sólo 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 a la 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 tamaño en bits.

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

2.2. 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, pueden ser, entre otras: acelerar, frenar y cambiar velocidad.

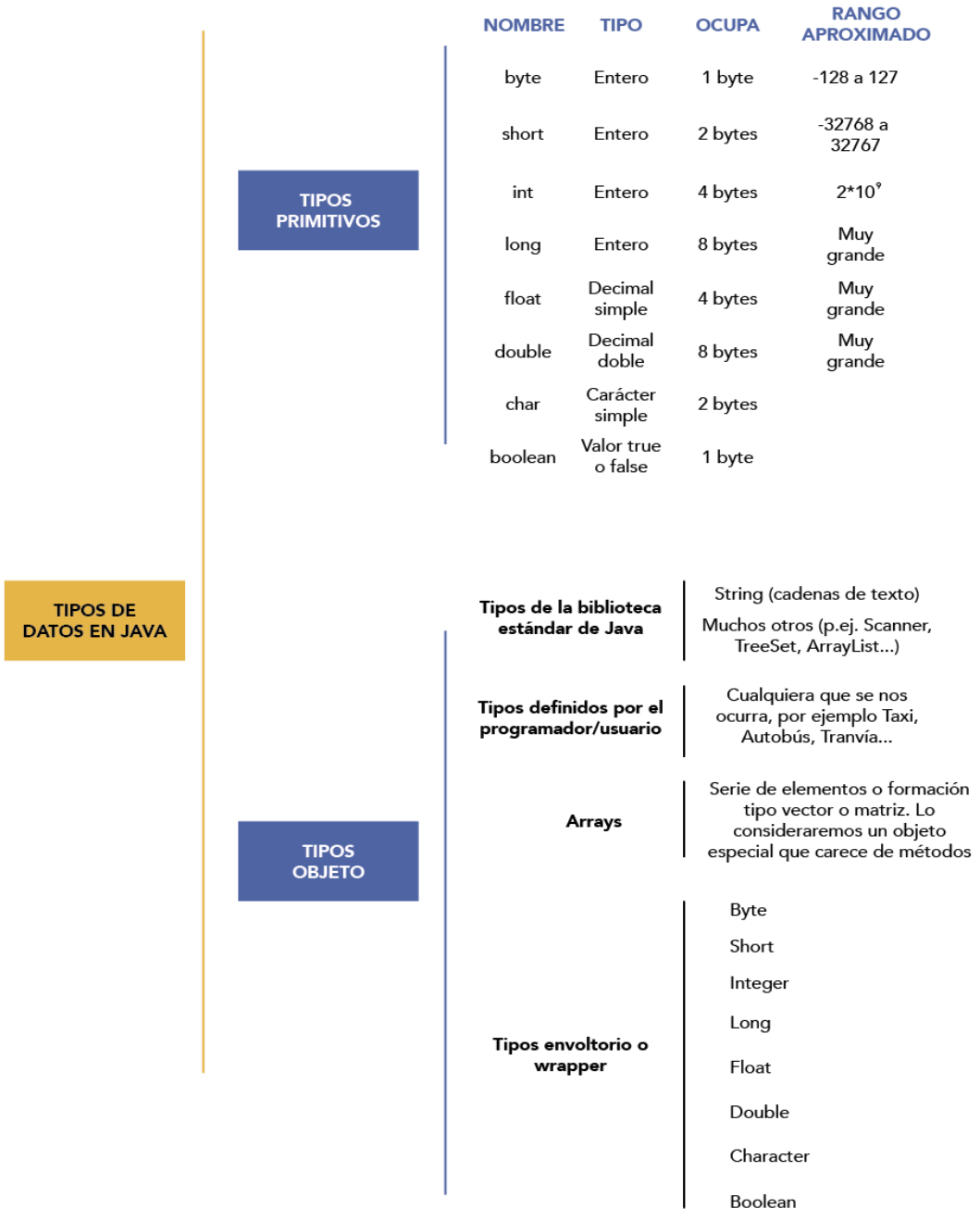
Recordemos que un objeto va a utilizar estos atributos en forma de variables, y los métodos van a ser funciones que se van a encargar de realizar las diferentes acciones.

En nuestro ejemplo tendríamos variables en las que almacenar el color, la marca y el modelo, junto con un conjunto de funciones que van a desarrollar las acciones de acelerar, frenar y cambiar de velocidad.

CÓDIGO:

```
//Clase coche que va a servir para crear objetos coche
public class coche {
    String color;
    String marca;
    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();
    }
}
```

2.3. Tablas de tipos primitivos ante tablas de objetos



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, que son los tipos creados por el programador de la aplicación mediante la definición de una clase previamente, podemos definir los *arrays* o estructuras de tablas.

2.4. Utilización de métodos. Métodos estáticos. Constructores.

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 escribir dentro de la clase en cuestión, excepto cuando los métodos son abstractos, que en cuyo caso se debe indicar mediante la palabra *abstract*. Los métodos se deben definir en las clases derivadas haciendo uso de la palabra *extends*.

- **Constructores**

Los constructores deben tener el mismo nombre que el de la clase a la que pertenezcan, y se ejecutan de forma automática una vez que se crea un ejemplar. Se pueden crear tantos constructores como se desee, siempre y cuando los argumentos que reciba sean distintos.

El compilador va a seleccionar el constructor correcto de forma automática, en función de los parámetros que tenga cada uno de los constructores.

- **Métodos de acceso**

La función principal de estos métodos es habilitar las tareas de lectura y de modificación de los atributos de la clase. 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 se necesiten conocer detalles más específicos de la implementación. Cualquier cambio que se realice en la implementación de los métodos no afectan a las clases clientes.

- **Static**

Es una palabra reservada que distingue entre atributos y métodos.

Los **atributos cualificados con la palabra *static*** son atributos de la clase. Es decir, nos referimos a que este atributo se va a almacenar en una zona de memoria propia de la clase, y va a compartir su valor con todos los ejemplares de la clase en cuestión. Además, ya que es parte de la clase, debemos crear un ejemplar de esta, que es la que vamos a utilizar para poder acceder al atributo al que nos estamos refiriendo.

Los **métodos cualificados con *static*** son propios de la clase y se va a reservar un espacio para ellos cuando arrancamos el programa. Tienen acceso a los atributos *static* de la clase, pero no a aquellos atributos de ejemplar. Para emplearnos no se necesita crear un ejemplar determinado de la clase.

2.5. 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. A continuación, en cualquier parte de la función, definimos un objeto de tipo clase creada anteriormente.

Este tipo en cuestión tiene la posibilidad de poder controlar tanto los atributos como los métodos que el ámbito le permita.

Para poder utilizar tanto los atributos como los métodos definidos en la clase, debemos utilizar el carácter punto.

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 alumnos que nos servirá para crear objetos tipo alumno
public class Alumnos {
    //Atributos
    //Constructor
    //Métodos
    public void evaluar() {//código del método}
}
-----------------------------------------------------------------
//Clase Aula donde utilizaremos los objetos alumnos
public class Aula {
    public static void main(String[] args) {
        //Creación de objetos Alumno
        Alumnos alumno_1 = new Alumnos("Antonio", "DAW");
        //Sintaxis de utilización de los métodos del objeto
        alumno_1.evaluar();
    }
}
```

2.6. Gestión de memoria. 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 se van a ejecutar de forma automática cuando se deba eliminar un objeto. Se trata de una función que no devuelve ningún tipo de dato (ni siquiera *void*), ni recibe ningún tipo de parámetros de entrada a la función. Normalmente, los objetos dejan de existir cuando finaliza su ámbito, antes de terminar su ciclo viral.

También existe la posibilidad del conocido recolector de basura (*garbage collector*) que, cuando existen 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 este 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 explica en la UF5, en el punto 2.

3. Desarrollo de programas organizados en clases.

3.1. 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 un *struct,* 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.

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 la programación orientada a objetos podemos decir que *coche* es una instancia de la clase de objetos conocida como *coches*. 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 coches, es decir, podemos tener un coche negro y otro azul, ya que ambos tienen en común la variable *color*. De tal forma que podremos 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 otros factores, por:

- Tener ruedas características.
- Tener matrícula.
- Tener 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. 4 más una de repuesto.
- Matrícula FNR 9774.
- 3 puertas.
- Negro.
- Seat.
- Ibiza.

De esta forma, tenemos la clase *coche* y el objeto *Seat Ibiza*.

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 que se ha visto en el punto 2.5, durante este punto 3 vamos a detallar la creación de todos los puntos de este 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.

Si no se especifica que un miembro es público, nos estaremos refiriendo a que este solo va a ser accesible (o visible) por otros miembros de la clase.

Mientras que, si lo definimos como público, estamos señalando que otros objetos puedan realizar llamadas a estos miembros.

- **Privados (*private)***

Aquellos componentes de carácter privado solamente pueden ser utilizados por otros miembros de la clase, pero nunca por otras donde se instancien.

Por defecto, cuando definimos un método, si no especificamos nada se considera privado.

También existen 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.*

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, 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

3.2. 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
[Modifficador_de_acceso] Tipo_dato nombre_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 existan, como *int*, *doublé*, *char* o algunos más complejos, como estructuras, tablas o incluso objetos.
- **Nombre_atributo:** identificador que vamos a utilizar para esa variable.

A continuación, vamos a ver un ejemplo que se ha visto ya en el punto 2.5 donde detallaremos la creación de los atributos.

CÓDIGO:

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

3.3. 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 utilizar directamente desde la propia clase en vez de instanciar esta. De la misma forma, podemos también crear atributos estáticos. Cuando utilizamos un método tipo *Static*, utilizamos las variables estáticas definidas en la clase.
- **abstract**: será más sencillo de comprender después de ver el significado de la herencia. De todas formas, señalaremos que los métodos abstractos no se van a declarar en la clase principal, pero sí en las demás que hereden de esta.
- **final**: estos métodos no ofrecen la posibilidad de sobrescribirlos.

- **native**: métodos implementados en otros lenguajes pero que deseamos añadir a nuestro programa. Podremos hacerlo incorporando la cabecera de la función en cuestión, y sustituyendo el cuerpo del programa por ";" (punto y coma).
- **Synchronized**: utilizado en aplicaciones multi-hilo.

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

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

    //Métodos creados por el programador
    public double 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 devolverán los valores de los atributos o nos permitirán 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 va a realizar 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.

3.4. Sobrecarga de métodos

La **sobrecarga de métodos** consiste en crear métodos en la misma clase y con el mismo nombre pero que estos tengan 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 pero que, 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.

3.5. Creación de constructores

Una de las formas que tenemos de identificar a un constructor de una clase es que debe llamarse igual que esta. El constructor se va a ejecutar 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
    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 devolverá un valor por defecto del objeto; en el caso de llamar al constructor con parámetros, le podremos indicar los valores que deseemos a este objeto.

3.6. 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.

Como se comentó con anterioridad, 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.

3.7. 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.

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 reservada ***new*** para poder reservar un espacio en memoria, de tal forma que, si solo declarásemos el objeto, no podríamos utilizarlo. Esta instrucción comienza una expresión para instanciar 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 a 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 que, 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
public class Aula {
    public static void main(String[] args) {
        double nota[] = new double[2];
        //Creación de objetos Alumno
        Alumnos Alumno_1 = new Alumnos("Antonio", "DAW");
        Alumnos Alumno_2 = new Alumnos("Laura", "DAM");
        //Sintaxis de utilización de los métodos del objeto
        nota[0] = Alumno_1.evaluar(6.5);
        nota[1] = Alumno_2.evaluar(8);
    }
}
```

3.8. 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 seguirán la siguiente 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 bastantes **métodos** que detallamos a continuación:

Método	Descripción
println (boolean)	Sobrecarga del método *println* 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 (*cadena para formato*, variables)	Escribe una cadena con los valores de las variables.

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

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 u hora
%x	Entero hexadecimal

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

Caracteres especiales	Significa	Resultado
%tA	Día de la semana	jueves
%ta	Día de la semana abreviado	jue
%tB	Mes	febrero

%tb	Mes abreviado	feb
%tC	Parte del año que señala al siglo	20
%tc	Fecha y hora con formato “%ta %tb %td %tT %tZ %tY”	jue feb 01 13:31:41 CET 2018
%tD	Fecha con formato “%tm/%td/%ty“	02/01/18
%td	Día del mes con formato “01-31”	01
%te	Día del mes con formato “1-31”	1
%tF	Fecha con formato ISO 8601	2018-02-01
%tH	Hora del día con formato “00-23”	13
%th	El mismo formato que %tb	feb
%tI	Hora del día con formato “01-12”	01
%tj	Día del año con formato “001-365”	032
%tk	Hora del día con formato “00-23”	13
%tl	Hora del día con formato “01-12”	1
%tM	Minuto de la hora con formato “00-59”	31
%tm	Mes actual con formato “01-12”	02
%tN	Nanosegundos con formato de 9 dígitos “000000000-999999999”	844000000
%tp	Marcador especifico “am-pm”	pm
%tQ	Milisegundos desde la época del 1 de enero de 1970 a las 00:00:00 UTC	1517488301844
%tR	Hora actual con formato 24 horas	13:31
%tr	Hora actual con formato 12 horas	01:31:41 PM
%tS	Segundos de la hora actual “00-60”	41

%ts	Segundos desde la época del 1 de enero de 1970 a las 00:00:00 UTC	1517488301
%tT	Hora actual con formato 24 horas	13:31:41
%tY	Año actual con formato "YYYY"	2018
%ty	Año actual con formato "YY"	18
%tZ	Abreviación de zona horaria	CET
%tz	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, que equivale al código ASCII del carácter pulsado en teclado.

Veamos un ejemplo para aclarar estos conceptos:

CÓDIGO:

```
int numero = System.in.read ();              //(1)
System.out.printf (%c, (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 por 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 manejo de errores respectivamente.

CÓDIGO:

```
//Variable estática out son 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 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();
```

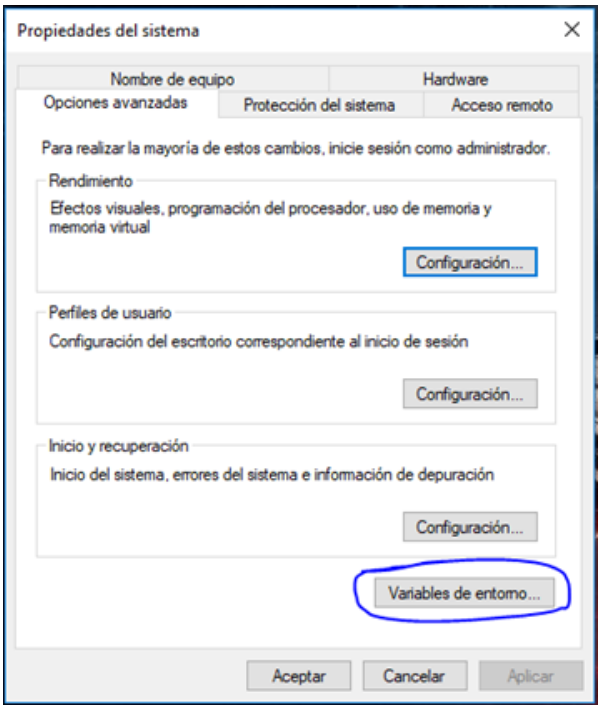
Cuando obtenemos la instancia principal de consola podemos hacer uso de las funciones de input de datos por teclado como son:

- **readLine()**
- **readPassword()**

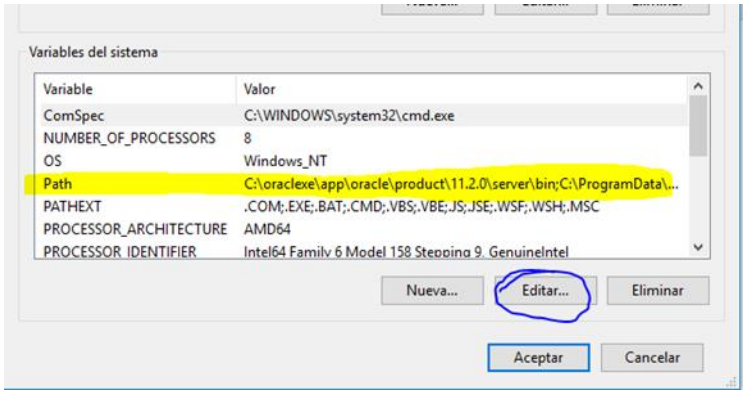
La clase *console* funciona en base a la consola de usuario, vamos a ver como configurar la consola de Windows para poder ejecutar un programa con esta función:

> Las siguientes capturas de pantalla se han realizado con Windows 10, si tenemos 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.

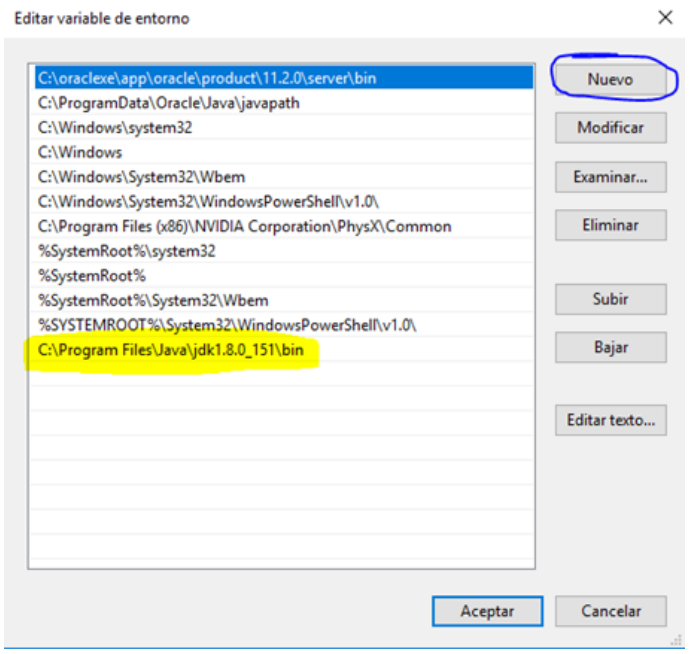
- **Modificar las variables de entorno de Windows:**


Propiedades del sistema
Nombre de equipo
Hardware
Opciones avanzadas
Protección del sistema
Acceso remoto
Para realizar la mayoría de estos cambios, inicie sesión como administrador.
Rendimiento
Efectos visuales, programación del procesador, uso de memoria y memoria virtual
Configuración...
Perfiles de usuario
Configuración del escritorio correspondiente al inicio de sesión
Configuración...
Inicio y recuperación
Inicio del sistema, errores del sistema e información de depuración
Configuración...
Variables de entorno...
Aceptar
Cancelar
Aplicar

- **Modificamos 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 como 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");
        }
    }
}
```

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

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

- **Clase *Scanner*:**

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

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, tendremos que importar la librería "**java.util**" para que este funcione correctamente.

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

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
public class Alumno {
    //Atributos
    private String nombre;
    private String curso;
    //Constructor vacío
    public Alumno() {
        this.nombre = "Ilerna";
        this.curso = "Online";
    }
    //Constructor con parámetros
    public Alumno(String nombre, String curso) {
        this.nombre = nombre;
        this.curso = curso;
    }
    //Métodos GET y SET
    public String getNombre() {return nombre;}
    public void setNombre(String nombre) {this.nombre = nombre;}
    public String getCurso() {return curso;}
    public void setCurso(String nombre) {this.curso = curso;}
    //Métodos creados por el programador
    public double evaluar(double nota) {
        nota = nota * 0.7;
        return nota;
    }
}
```

CÓDIGO:

```
//Clase Aula donde utilizaremos los objetos alumnos
public class Aula {
    public static void main(String[] args) {
        double nota[] = new double[2];
        //Creación de objetos Alumno
        Alumnos Alumno_1 = new Alumno("Antonio", "DAW");
        Alumnos Alumno_2 = new Alumno("Laura", "DAM");
        //Sintaxis de utilización de los métodos del objeto
        nota[0] = Alumno_1.evaluar(6.5);
        nota[1] = Alumno_2.evaluar(8);
        //Impresión de los Atributos del alumno
        System.out.println("Nombre: " + Alumno_1.getNombre() +
                           " Curso "  + Alumno_1.getCurso() +
                           " Nota "   + nota[0]);
        System.out.println("Nombre: " + Alumno_2.getNombre() +
                           " Curso "  + Alumno_2.getCurso() +
                           " Nota "   + nota[1]);
    }
}
```

4. Utilización avanzada de clases en el diseño de aplicaciones

4.1. 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 llevar a cabo este concepto de composición de clases vamos a utilizar un ejemplo y así podremos verlo de una forma 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
    String dni;
    String nombre;
    double sueldo;
    Date fecha_nac;
    //Constructores
    public Empleado() {
        this.dni = "00000000I";
        this.nombre = "Ilerna Online";
        this.sueldo = 2000;
        this.fecha_nac = new Date();
    }
    public Empleado(String d, String n, double s, Date fn) {
        this.dni = d;
        this.nombre = n;
        this.sueldo = s;
        this.fecha_nac = 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 getFecha_nac() {return fecha_nac;}
    public void setFecha_nac(Date f_nac) {this.fecha_nac = f_nac;}
    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 {
    Empleados emp;
    int telefono;
    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;
    }
}
```

4.2. Herencia

La herencia es uno de los términos más importantes en la programación orientada a objetos. 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 su color de ojos, los gestos, la constitución, etc.

Por esto, 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 de sus características y funcionalidades. Se le denomina también **clase *padre*** o **superclase.**
- **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# y 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 una serie de pasos importantes como que tiene una **clase derivada:**

Solo tiene 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 podremos 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 directamente al método privado no podremos.

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 hayan definido en la clase base.

A continuación, veamos un ejemplo de estas definiciones:

CÓDIGO:

```
class Empleado {
    String nombre, apellidos;
    Double sueldo;
    String DNI;
    //código de clase empleado
}
class Cualificados extends Empleado {
    String titulacion;
    Double extra;
    String departamento;
    //código de clase cualificados
}
class Obreros extends Empleado {
    String destino;
    int horas_extra;
    double precio_hora;
    //código de clase obreros
}
class Jefe extends Cualificados {
    int total_trabajadores;
    String [] proyectos;
    double extra;
    //código de clase jefe
}
```

4.3. 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 como es la jerarquía de clases.

Por jerarquía entendemos como la estructura por niveles de algunos elementos, donde los elementos situados en un nivel superior tienen algunos privilegios sobre los situados en el nivel inferior, o que contiene una relación entre los elementos de varios niveles entre los que también exista una relación.

Por tanto, si esa definición la extrapolamos a la programación orientada a objetos, nos referimos a la jerarquía de clases que se definen 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 nuestra 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.

4.4. Clases y métodos abstractos

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 nos 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_acceso abstract class NombreClase {}
```

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

CÓDIGO:

```
public abstract class Profesor {
    //Atributos
    String nombre;
    String dni;
    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) {
        double nota_final;
        nota_final = (pac1 + pac2 + pac3)/3;
        return nota_final;
    }
    public double pacs(double p1,double p2,double p3,double p4) {
        double nota_final;
        nota_final = (pac1 + pac2 + pac3 + pac4)/4;
        return nota_final;
    }
}
```

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 cumplan que una clase es abstracta:

No tiene cuerpo, solamente tiene signatura con paréntesis.

- Finaliza con punto y coma.
- Sólo existe dentro de una clase abstracta.
- Aquellos métodos definidos como abstractos deben estar sobrescritos también en las subclases.

Veamos cómo quedaría su sintaxis:

CÓDIGO:

```
abstract modificador_de_acceso Tipo de retorno (parámetros);
Ejemplo: abstract public void (String a);
```

Notas:

- Cuando declaramos un método abstracto, el compilador de Java nos obliga a que declaramos esa clase abstracta, ya que, si no lo hacemos, tendríamos 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 y, otras, sí que 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 esta operación a cabo 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();
        p.NotaMedia(6.5,7,8);
    }
    //Constructor
    public ProfesorInterino() {
        super.dni = "45633254L";
        super.nombre = "Adrián García";
        super.edad = 29;
    }
    //Método
```

```
    public double NotaMedia(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.NotaMedia(7.5,7,9.3,8.7);
    }
    //Constructor
    public ProfesorOficial() {
        super.dni = "48566221F";
        super.nombre = "Ana Gómez";
        super.edad = 35;
    }
    //Método
    public double NotaMedia(double pac1, double pac2,
                            double pac3, double pac4) {
        double notaMedia;
        notaMedia = super.PACS(pac1,pac2,pac3,pac4);
        System.out.println(notaMedia);
        return notaMedia;
    }
}
```

4.5. 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.

Vemos el siguiente 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.
- Debe tener, no solo el mismo nombre, sino también el mismo número de argumentos y valor de retorno.

Siguiendo con el ejemplo del profesor, vemos 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.NotaMedia(7.5,7,9.3,8.7);
    }
    //Constructor
    public ProfesorOficial() {
        super.dni = "48566221F";
        super.nombre = "Ana Gomez";
        super.edad = 35;
    }
    //Método
    public double NotaMedia(double pac1,double pac2,
                            double pac3,double pac4) {
        double notaMedia;
        notaMedia = super.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;
    }
}
```

4.6. 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 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.

4.7. Interfaces

Las **interfaces** están formadas por un **conjunto de métodos que no necesitan ser implementados**; es decir, son del estilo de las clases abstractas, que están compuestas por un conjunto de métodos abstractos.

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);
}
```

Donde:

- **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, a diferencia de que va a utilizar la palabra **interface**.

Su sintaxis es:

CÓDIGO:

```
Modificador_de_acceso interface 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
    }
}
```

UF5: POO. Librerías de clases fundamentales

1. Aplicación de estructuras de almacenamiento en la programación orientada a objetos

1.1 Creación de arrays

Un array consiste en 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.

La sintaxis de un array en Java la podemos llevar a cabo 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];
```

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, podríamos hacerlo según indicamos a continuación:

CÓDIGO:

```
array[4];
```

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

```
array[0];
```

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 que podemos llevar a cabo 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 programa Eclipse nos da error si escribimos algún valor entre corchetes cuando creamos el array:

CÓDIGO:

```
int [] array = new int [4] {2, 4, 6, 8};     //ERROR!!!
int [] array = new int [] {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 si necesitamos utilizarlos.

En la siguiente tabla podemos ver varios de estos métodos disponibles:

Método	Descripción
length	Obtiene o establece 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 *notify* () o el método *notifyAll* () 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 valores 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);
```

1.2 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:

```
public class Ejemplo {
    public static void main(String[] args) {
        int [] [] matriz = new int [2] [];
        matriz[0] = new int [] {1, 2, 3, 4, 5};
        matriz[1] = new int [] {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, seria de la siguiente forma:

Segunda dimensión

Primera dimensión

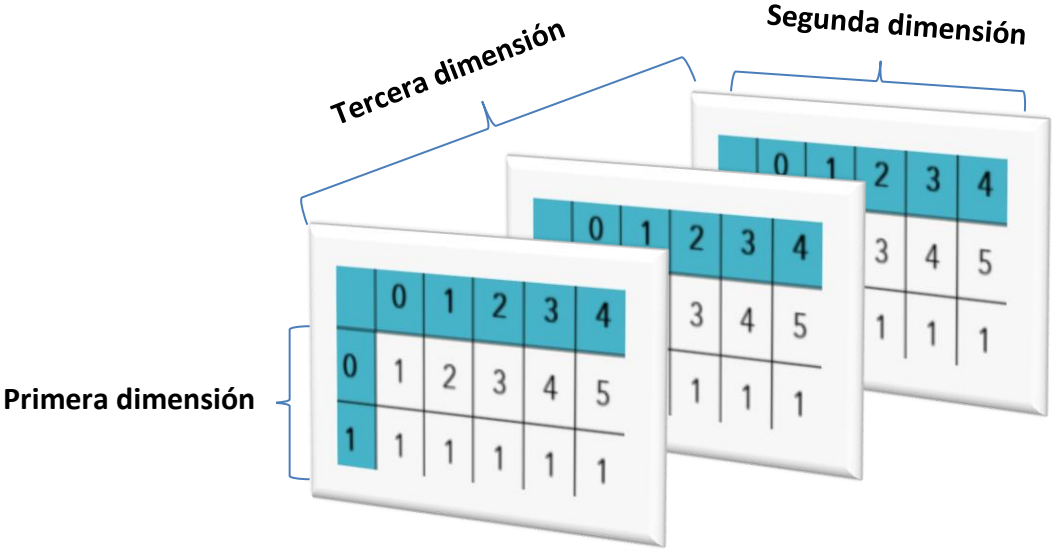
	0	**1**	**2**	**3**	**4**
0	1	2	3	4	5
1	1	1	1	1	1

Ahora vamos a ver un segundo ejemplo de matriz, en este caso vamos a representar tres dimensiones.

CÓDIGO:

```
public class Ejemplo {
    public static void main(String[] args) {
        int [][][] matriz = new int [3][3][3];
        matriz[0][0] = new int [] {1,2,3};
        matriz[1][0] = new int [] {1,2,3};
        matriz[2][0] = new int [] {1,2,3};
        matriz[0][1] = new int [] {4,5,6};
        matriz[1][1] = new int [] {4,5,6};
        matriz[2][1] = new int [] {4,5,6};
        matriz[0][2] = new int [] {7,8,9};
        matriz[1][2] = new int [] {7,8,9};
        matriz[2][2] = new int [] {7,8,9};
        for(int i = 0; i < 3; i++) {
            System.out.println();
            for(int j = 0; j < 3; j++) {
                System.out.println();
                for(int s = 0; s < 3; s++) {
                    System.out.print(matriz[i][j][s] + " ");
                }
            }
        }
    }
}
```

Si representamos la **matriz** visualmente, seria de la siguiente forma:



1.3 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);
}
```

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 *carácterAntigüo* 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.

1.4 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 él, todas las entidades conceptuales están representadas por interfaces, y las clases se usan para proveer implementaciones de esas interfaces. Una introducción conceptual debe entonces 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, son 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 poder 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 porque 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:

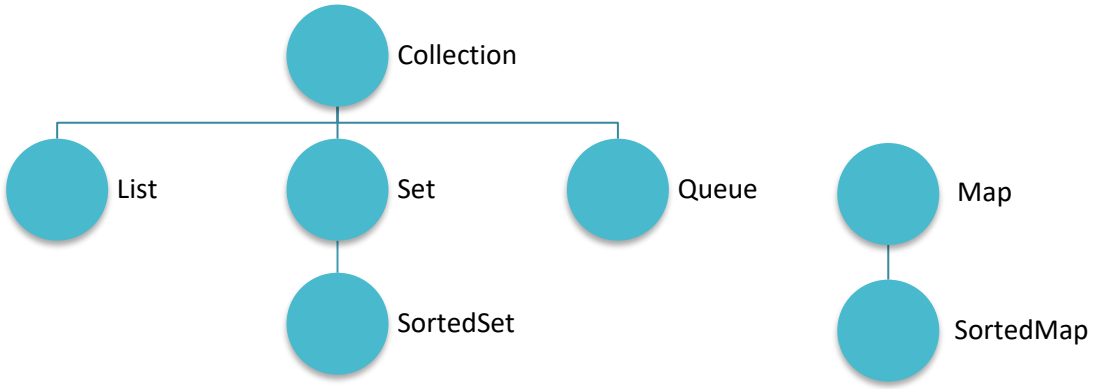
Podemos imaginar que una cola es similar a las colas que se hacen en el supermercado o cuando esperas a que te atiendan en un banco. El primero que llega es el primero en ser atendido.

1.5 Colecciones e Iteradores

Una **colección representa un grupo de objetos (elementos) que se pueden recorrer (o iterar)** y de lo 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 usar una interfaz u otra.

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



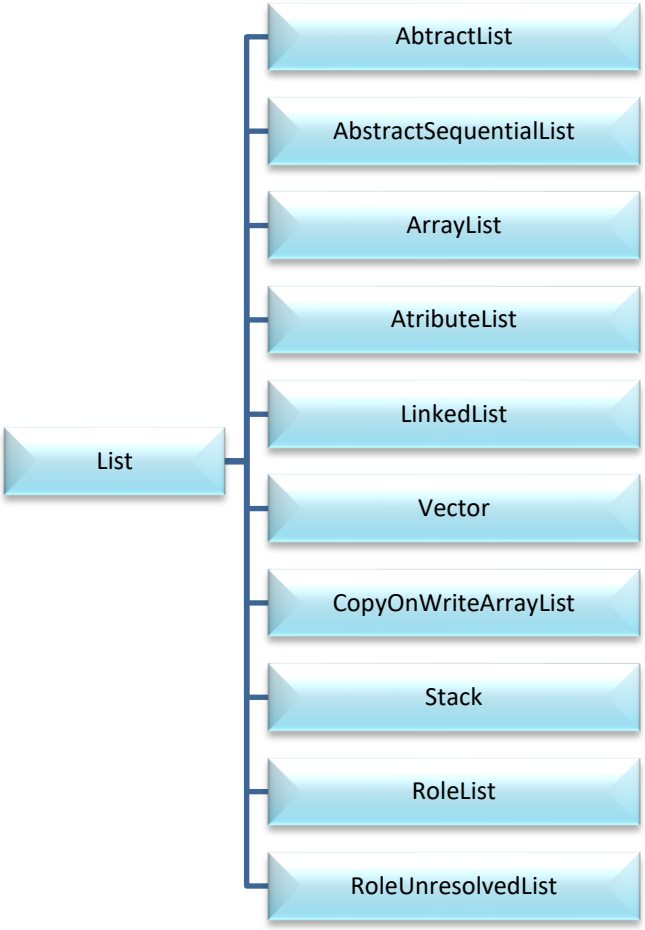
- **List**: Pueden estar repetidos, están indexados con valores numéricos.
- **Set**: Permite almacenar una colección de elementos no repetidos y sin ordenar.
- **Queue**: No permite el acceso aleatorio y solo permiten acceder a los objetos del principio o del final.
- **Map:** No es un tipo de *Collection* 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 <Tipo_de_dato> ();
```

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

- **get (int)**: obtiene el objeto de la posición indicada.
- **indexOf (Object)**: obtiene la posición del objeto indicado.
- **isEmpty ()**: devuelve un *booleano* que indica si el array está vacío o no.
- **add(Object):** inserta Object en la última posición.
- **add(int, Object):** inserta Object en la posición indicada.
- **set (int, Objeto)**: inserta Object en la posición indicada, reemplaza el anterior valor.
- **toArray()**: convierte el ArrayList en un array del tipo especificado.

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 una 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.

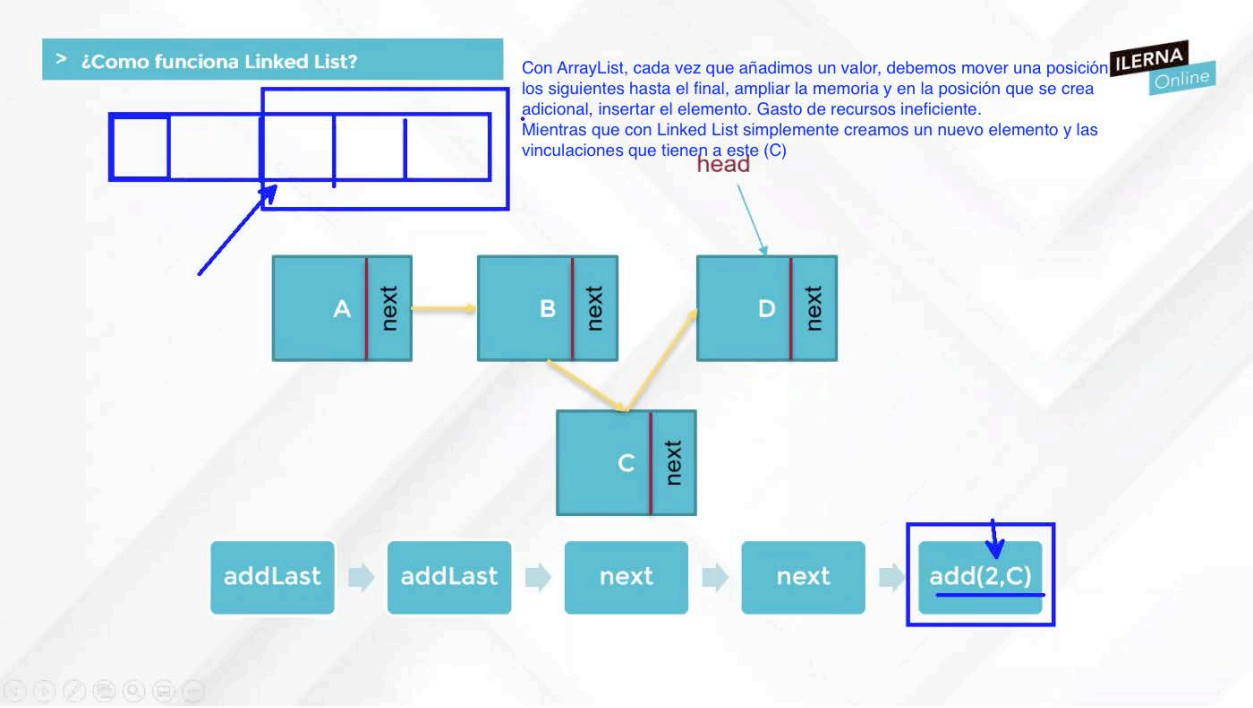
La declaración de la colección tiene la siguiente sintaxis:

CÓDIGO:

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

LinkedList nombre = new LinkedList ();

//Instancia de colección con tipo específico:

LinkedList <Tipo_de_dato> nombre = new LinkedList <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 interface List.

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):
Vector nombre = new Vector ();
//Instancia de colección con tipo específico:
Vector <Tipo_de_dato> nombre = new Vector <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 practico 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<String>();
        vectorCadenas.add("Ilerna");
        vectorCadenas.add("Online");
        //Mostramos el contenido con firstElement y lastElement
        System.out.print(vectorCadenas.firstElement() + " ");
        System.out.print(vectorCadenas.lastElement());
    }
}
```

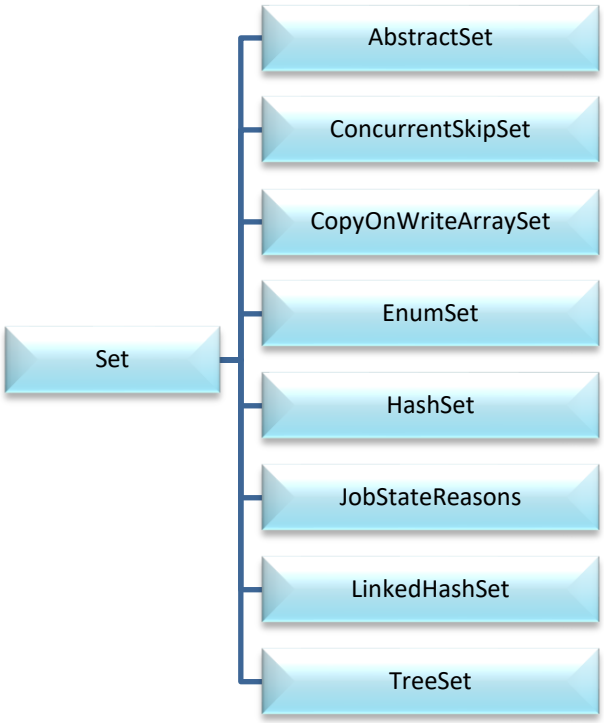
En este apartado hemos comentado las colecciones y métodos más utilizados. Para una mayor información de todas las colecciones adjuntamos el enlace web a la plataforma de Java, donde explica la colección *List* y contiene los enlaces a todas las clases que hemos visto en el esquema anterior:

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, pero 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(),* 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), 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):
HashSet nombre = new HashSet ();
//Instancia de colección con tipo específico:
HashSet <Tipo_de_dato> nombre = new HashSet <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:

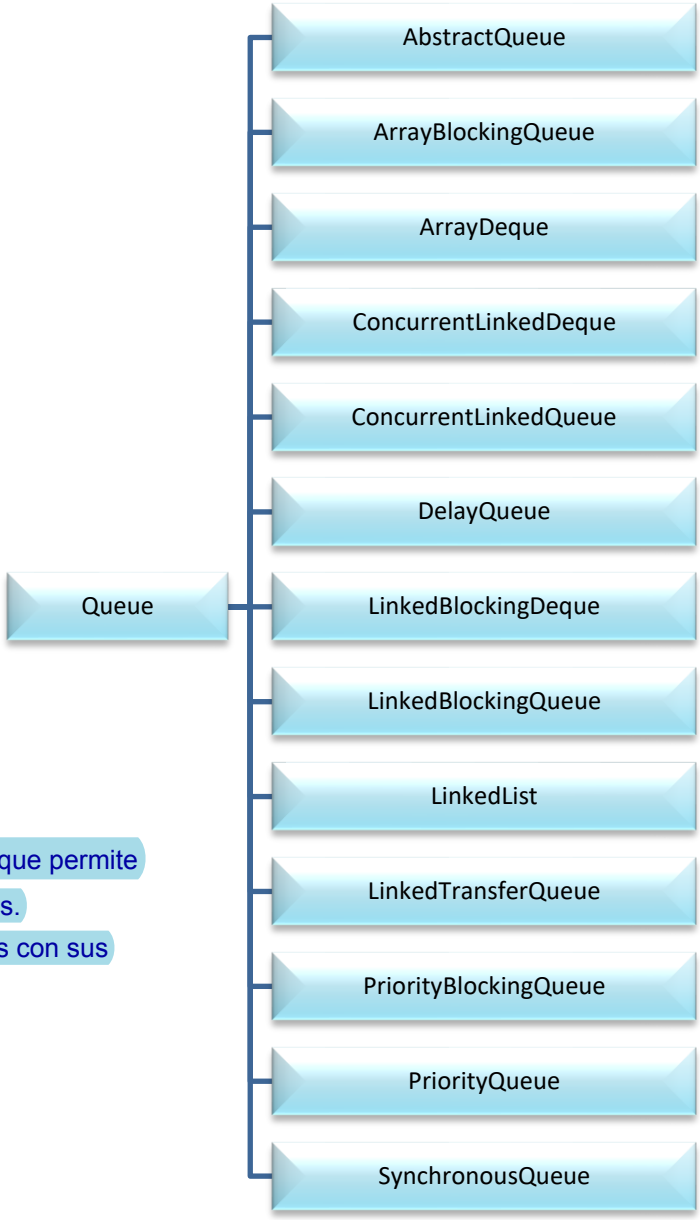
```
import java.util.TreeSet;
public class Ejemplo {
    public static void main(String[] args) {
        //Instancia de tipo Genérico
        TreeSet arbolPersonas = new TreeSet();
        arbolPersonas.add(3);
        arbolPersonas.add(45);
        arbolPersonas.add(72);
        //Vemos el funcionamiento 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 una mayor 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 a 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. Tambien 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:** Es una estructura de datos lineal que permite insertar y eliminar elementos por ambos extremos.
Junta en una única estructura las colas y las pilas con sus metodología.
  - Estrucuturas LIFO (pilas) Last in First Out
  - Estructuras FIFO (colas) First in First Out

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.
- **Peek():** selecciona el último elemento.
- **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 una mayor 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:

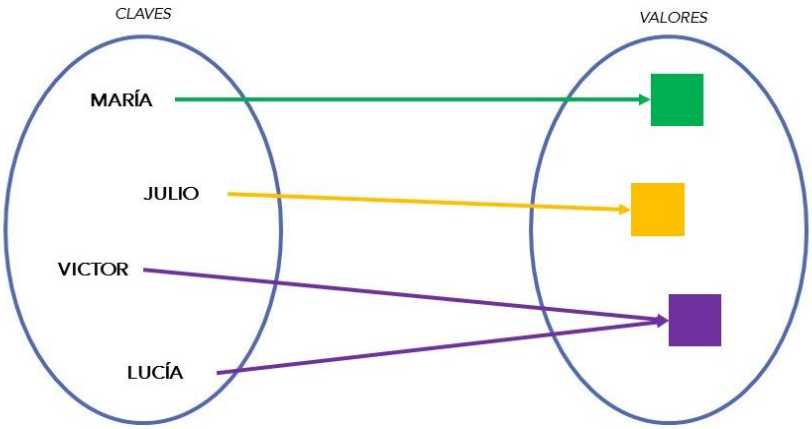
https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

- **Map**

Un map es un conjunto de valores, donde cada uno de los valores tiene asociado una clave mediante el cual se puede realizar la búsqueda. Suelen ser llamados Map<K,V>

A los primeros se les llama **claves o *keys*,** porque nos permiten acceder a los segundos. Los valores clave no aceptan valores duplicados.

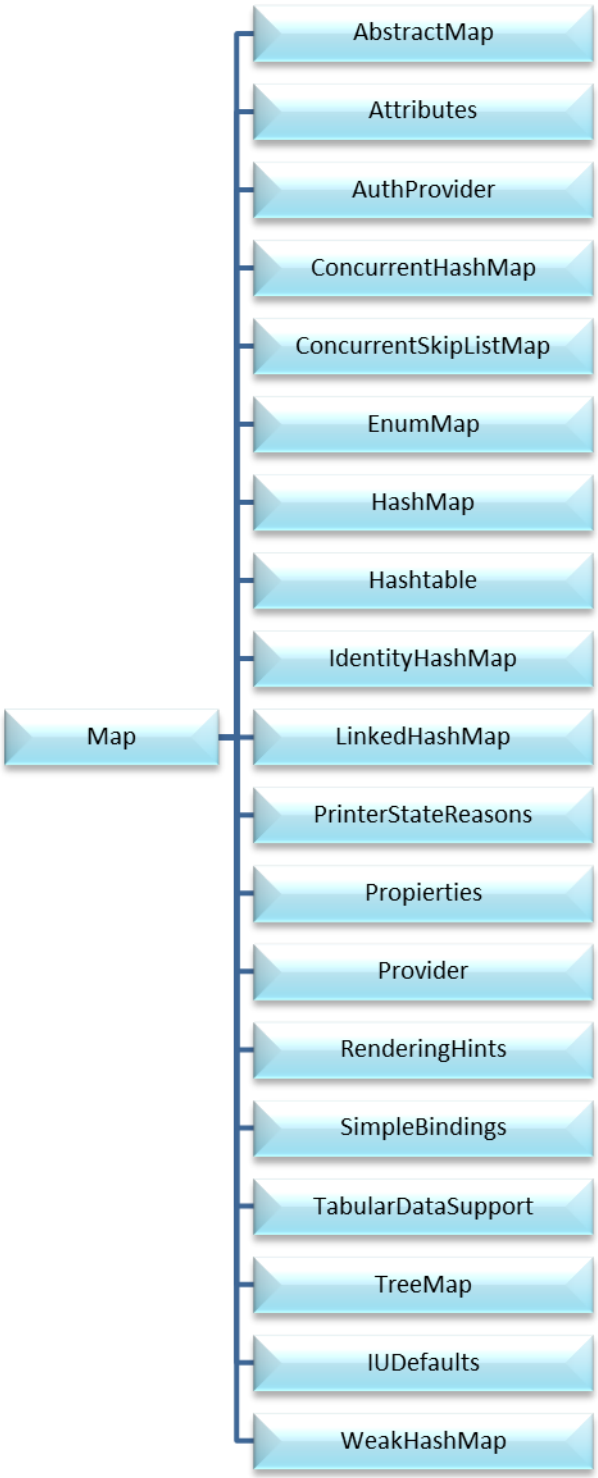
Map no es una interfaz que hereda de *collection*, ya que podríamos decir que collection es una interfaz unidimensional y map es una interfaz bidimensional.



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 remplaza.
- **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)*. Es 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 una mayor 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**

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 la *colleccion*.

Para recorrer las colecciones se puede utilizar un Iterator(), este nos permite recorrer cualquier estructura de datos de exactamente la misma forma, sin importar la implementación interna de la misma.

Este iterador nos proporcionará unos métodos propios de este, 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 donde 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<String>();
        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

1.6 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 genético 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, suponiendo en su historia una de las mayores modificaciones.

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 otro) 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 fuerte en tiempo de compilación.
- Eliminación de *casts* aumentando la legibilidad del código.
- Posibilidad de implementar algoritmos genéricos, con tipado seguro.

1.6.1 Clases genéricas

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

CÓDIGO:

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

Donde "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:

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;}
}
```

Donde *K* representa la clave y *V* representa el valor.

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.

1.6.2 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:

```
public static <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.

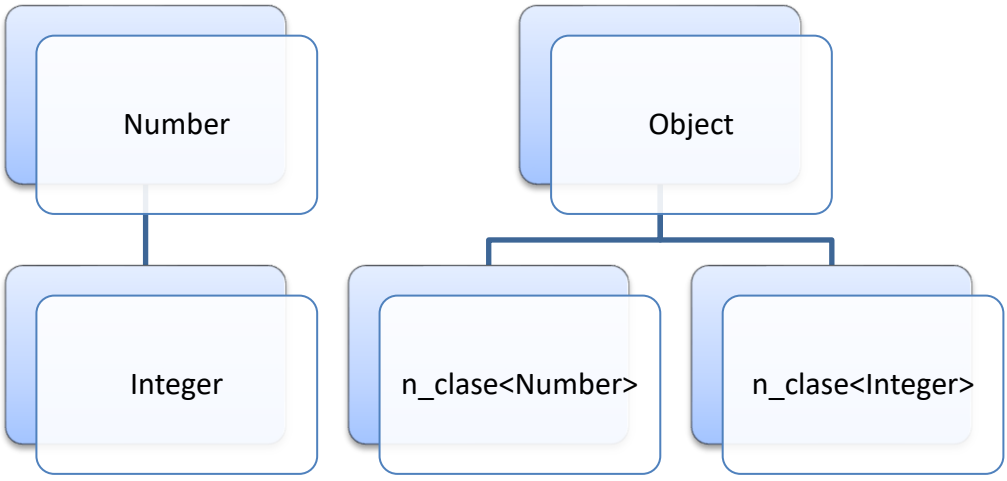
En Java, un tipo puede ser asignado a otro siempre que el primero sea compatible con el segundo; es decir, que tengan una relación uno es a uno. Una referencia de Object puede referenciar una instancia de *Integer* (un *Integer* es un *Object*).

CÓDIGO:

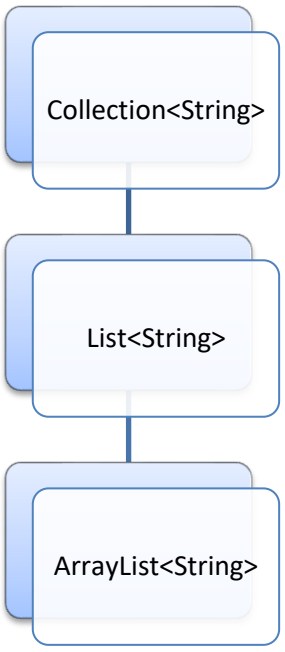
```
Object objeto = new Object();
Integer entero = new Integer(10);
objeto = entero;
```

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>*, aún 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.

1.7 Manipulación de documentos XML. 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 disponible en la API.

Las expresiones regulares se rigen por una serie de caracteres para la construcción del patrón a seguir. Las expresiones regulares solamente 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, o el hecho de validar un correo electrónico también se puede implementar mediante expresiones regulares.

Para hacer uso de las expresiones debemos de conocer la clase más importante del paquete *regex*. Es la clase ***Matcher*** y la clase ***Pattern*** con su excepción *ParrernSyntaxException*. La clase *Matcher* representa a la expresión regular que debe de 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*:

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[]){
        // compilamos el patrón
        Pattern patron = Pattern.compile("aabb");
        // creamos el Matcher a partir del patron, la cadena como parámetro
        Matcher encaja = patron.matcher("aabmaabbnoloaabbbmanoloabmolob");
        // invocamos el metodo replaceAll
        String resultado = encaja.replaceAll("..");
        System.out.println(resultado);
  }
}
```

La salida este ejemplo será: "aabm..nolo..bmanoloabmolob".

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

2. Control de excepciones

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

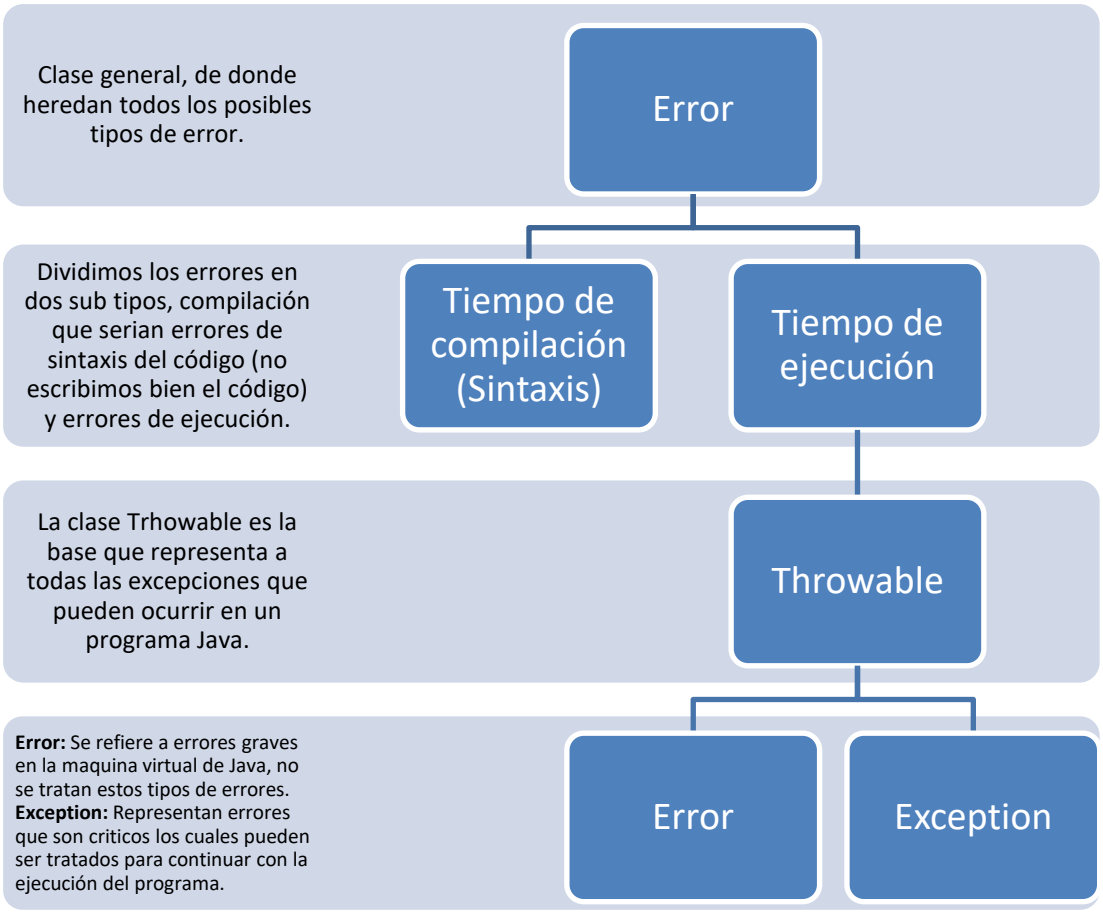
Cuando estamos desarrollando un programa vamos escribiendo código que iremos compilando para ver si existe algún error. En caso de que existan errores, debemos corregirlos para poder seguir adelante. **Estos errores se denominan errores de compilación, mientras que los errores que se muestran durante 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, resolverlos 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 error que tenemos en Java:



A nivel de excepciones tenemos las Checked y las Unchecked.
Las checked descenderían de Exception.
Las unchecked descenderian de RuntimeException, y estas a su vez estarían por debajo de Exception.

2.1 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*.

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
}
```

2.2 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;
}
```

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:

Debemos **capturar** el error cuando:

- 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, cuando tenemos que realizar algún tratamiento con el propio error. Sin embargo, deberíamos **delegar** la excepción cuando:

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

2.3 Lanzamiento de excepciones

Ya hemos visto cómo se pueden capturar o delegar las diferentes excepciones. Ahora vamos a aprender la forma en la que 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***, como podemos ver en el siguiente ejemplo:

CÓDIGO:

```
try {
    //código que se va ejecutar
    throw new Clase_de_error(mensaje); //lanzamiento de error
} catch (clase_de_error) {
    //código que se va a ejecutar si hay error
}
```

```
Capturaremos la Exception cuando:
- 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).
```

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

CÓDIGO:

```
static boolean login (string user, string pass) {
    //Código del método loguin
}
public static void main (string [] args) {
    try{
        //Bloque de código que en caso de error lanzara una excepción
        //que capturara el bloque catch
        if (!loguin (nombre, passwd)) {
            throw new ErrorLoginException();
        }
    }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.

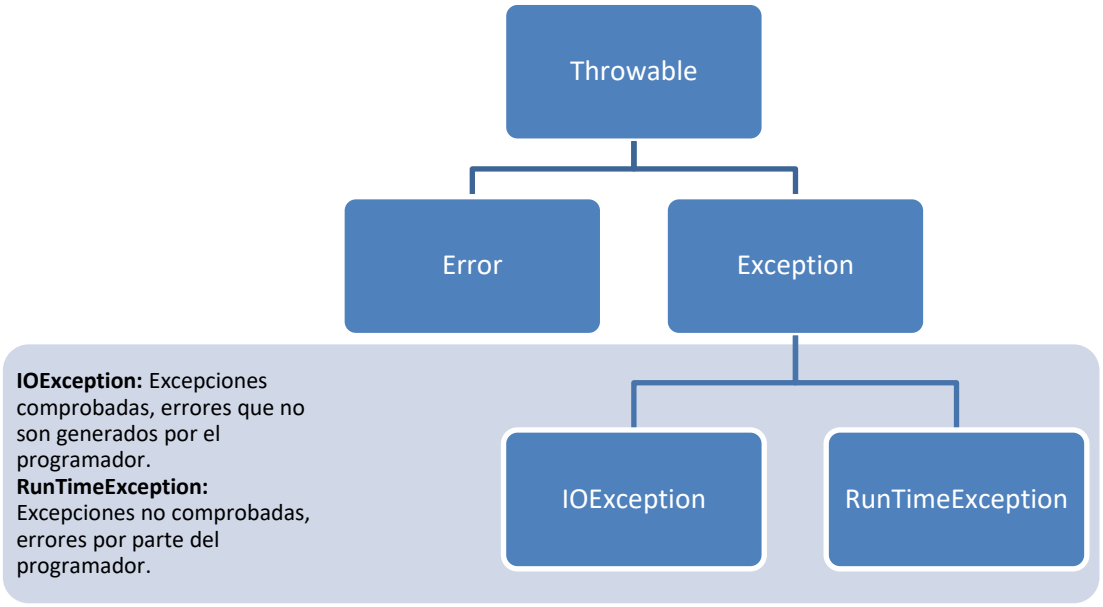
En el siguiente ejemplo se puede ver, como ya vimos anteriormente, una delegación de una excepción. Cuando se delega la excepción directamente ya la estamos lanzando al método anterior del cual ha sido llamado:

CÓDIGO:

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

2.4 Excepciones y herencia

El esquema que vemos a continuación es complementario al que hemos visto anteriormente (en 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 porqué 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 lanzarla en determinadas ocasiones.

2.4.1 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 deberemos seguir los siguientes pasos:

- Añadir una nueva clase al proyecto y ponerle el nombre que queramos que tenga nuestra excepción.
- Hacer que la nueva clase extienda de la clase *Exception*.
- 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 loguin
            System.out.println ("Introduzca el usuario:");
            nombre = sc.nextLine();
            System.out.println ("Introduzca la contraseña:");
            passwd = sc.nextLine();
            //Lanzar el método loguin
            if (!loguin(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;
    }
}
```

3. 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 mejoramos 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. Es este apartado veremos todos los elementos de los que se compone el entorno gráfico, los **distintos paneles, etiquetas, y cajas de información**.

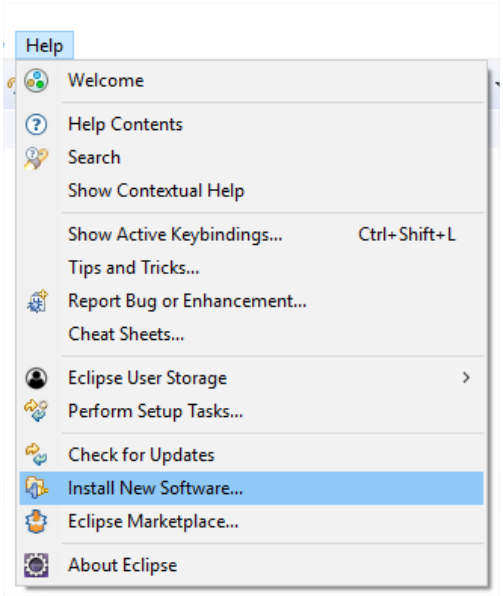
3.1 Creación y uso de interfaces gráficas de usuarios simples

Cuando tengamos que crear un proyecto gráfico, deberemos elegir con qué entorno realizarlo. Eclipse, Visual Studio o Netbeans son algunos de los más utilizados. En el presente apartado continuaremos usando el IDE Eclipse.

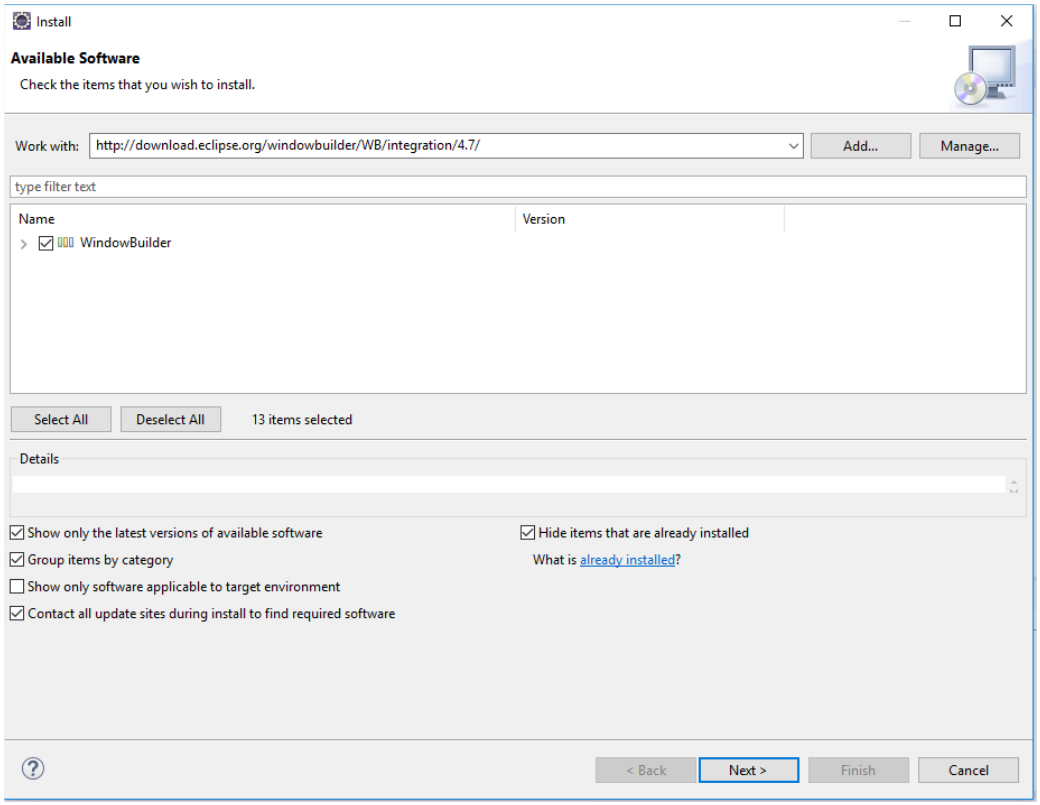
Entorno de trabajo

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 permitirá este desarrollo.

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

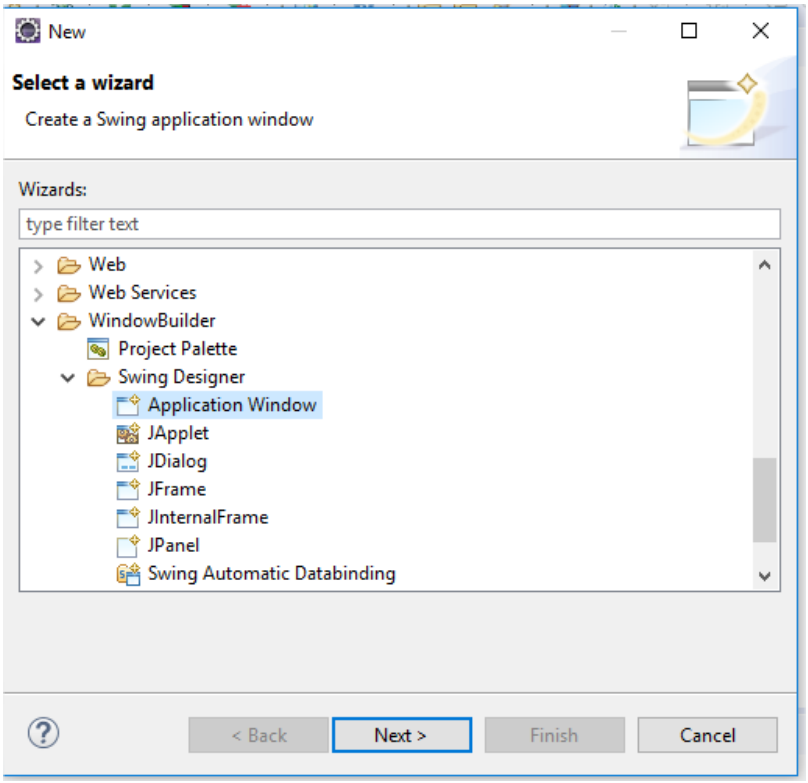


Para instalarlo necesitaremos una dirección URL que podremos obtener (según la versión que utilicemos de Eclipse) en la siguiente dirección: http://www.eclipse.org/windowbuilder/download.php.



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

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

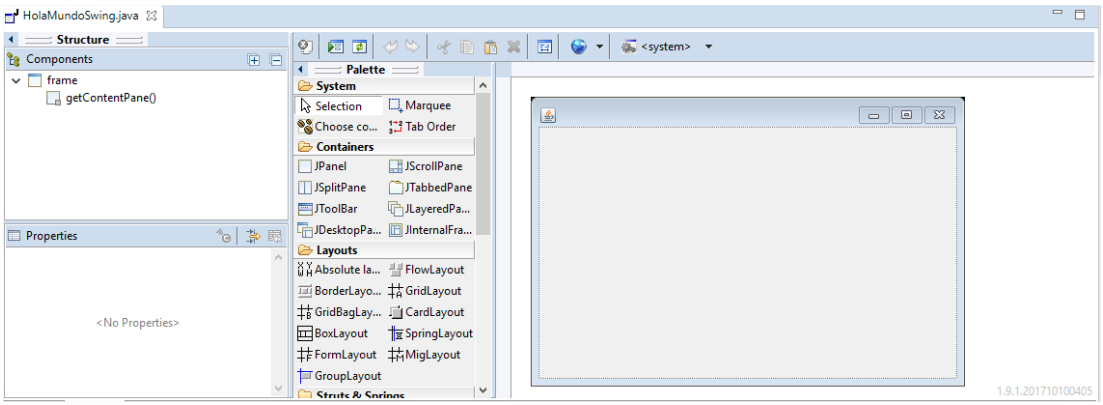

New
Select a wizard
Create a Swing application window
Wizards:
type filter text
Web
Web Services
WindowBuilder
Project Palette
Swing Designer
Application Window
JApplet
JDialog
JFrame
JInternalFrame
JPanel
Swing Automatic Databinding
< Back
Next >
Finish
Cancel

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

CÓDIGO:

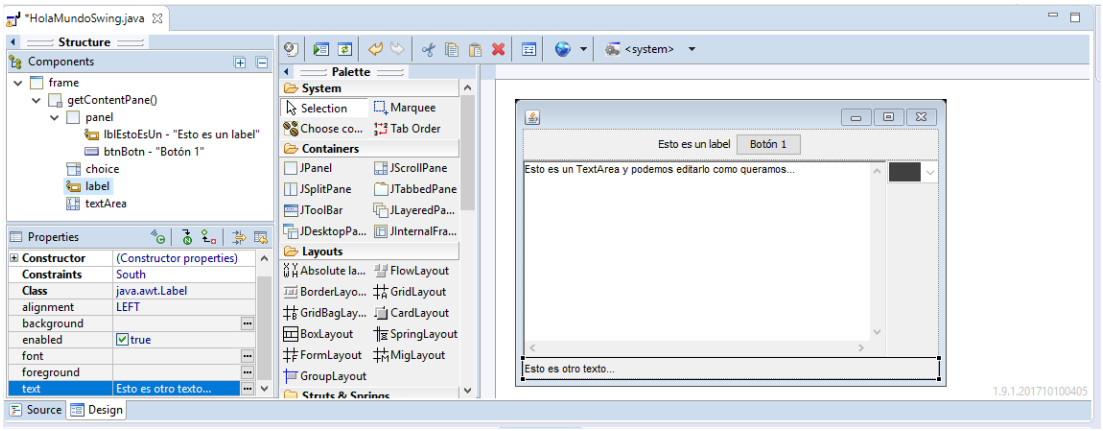
```
public class HolaMundoSwing {

	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 encontraremos la parte de interfaz gráfica que podremos usar para generar nuestras interfaces de una forma más dinámica y usando el ratón. A continuación si 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 parte 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 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 confifurar las distintas características de los elementos (controles): tamaño, color, texto, etc.

3.2 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 Tollkit*): 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	Label
TextField	Text Field	JTextField	Text Field
TextArea	Text Area	JScrollPane	Text Area
Button	Button	JButton	Button
CheckBox	Checkbox	JCheckBox	CheckBox
Choise	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 hara de ventana
        JFrame ventana = new JFrame("Titulo_ventana");
        //Ponemos la ventana visible
        ventana.setVisible(true);
        //Le damos untamaño 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 posicion 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 *JAppelet*. En el editor de código se localizan en el cuadro de herramientas bajo la categoría ***Swing Windows***.

  Los contenedores intermedios 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, tenemos que 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 va a 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.

3.3 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 con 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.
- **ForeCore:** *foreground.*
- **Text:** la propiedad de los cuadros de texto se denomina ***text****.*
- **Enabled**: muestra una casilla de verificación en la que podemos activar propiedades 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 por 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 al 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 hara de ventana
        JFrame ventana = new JFrame("Titulo_ventana");
        //Ponemos la ventana visible
        ventana.setVisible(true);
        //Le damos untamaño 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**.

4. Lectura y escritura de información

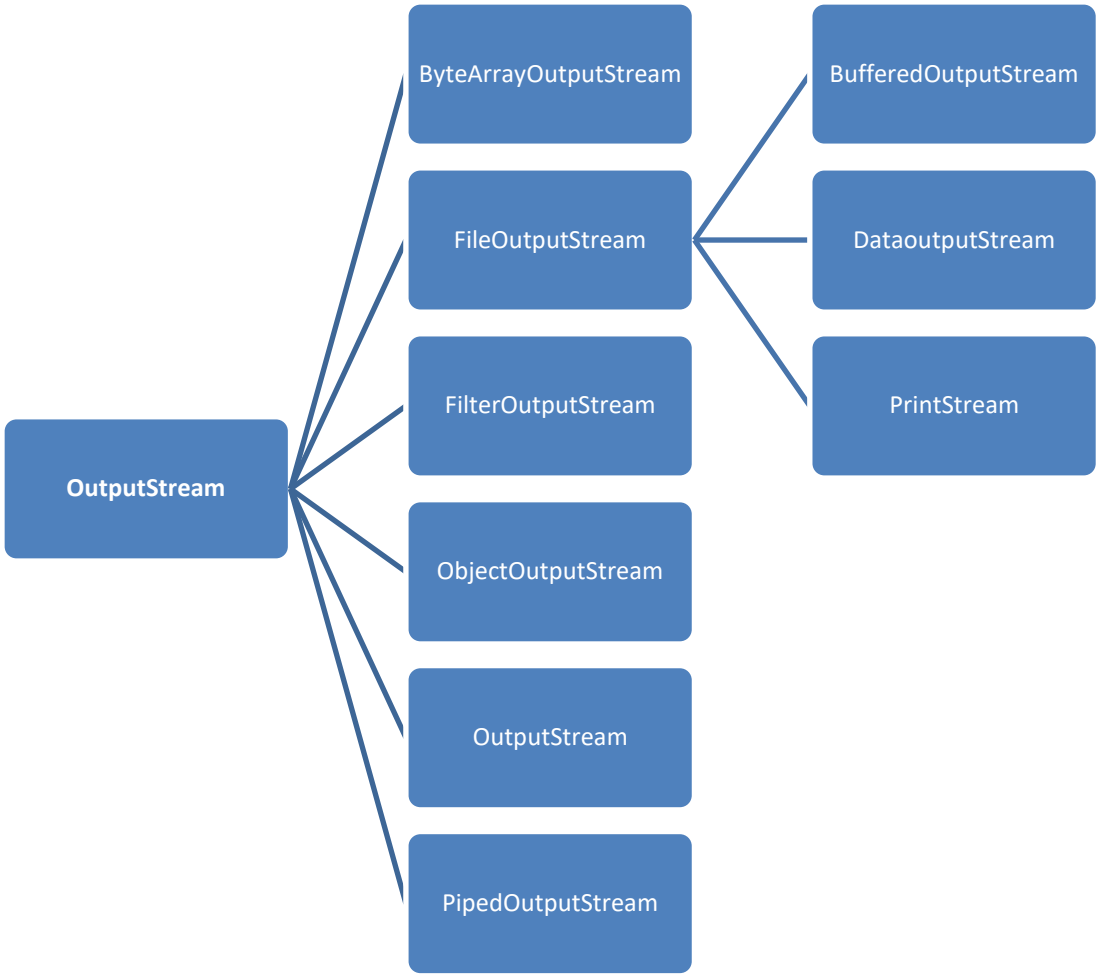
El lenguaje Java dispone de una gran cantidad de clases destinadas a la lectura y/o escritura de datos (información) en distintas fuentes y destinos. Destacan, entre otros, los discos y los dispositivos. Estas clases a las que nos referimos se denominan flujos (*streams*), y se utilizan para leer información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

4.1 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();
        }
    }
}
class Escribir{
    public void escribir() {
        String txt = "\nEscribimos a un fichero con Ilerna Online.";
        try {
            FileWriter doc = new FileWriter("Prueba.txt", true);
            for(int i = 0; i < txt.length(); i++) {
                doc.write(txt.charAt(i));
            }
            doc.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4.2 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.

  .1. 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.

  .1.1. Scanner (File origen)

  .1.2. Scanner (File origen, String nombreCharSet)

  .1.3. Scanner (InputStream origen)

  .1.4. Scanner (InputStream origen, String nombreCharSet)

  .1.5. Scanner (Readable origen)

  .1.6. Scanner (ReadableByteChannel origen)

  .1.7. Scanner (ReadableByteChannel origen, String nombreCharSet)

  .1.8. Scanner (String origen)

  .2. Métodos

  .2.1. 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 (*IlegalStateException*).

  .2.2. hastNext: 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.

  .2.3. 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.

  .1. Constructores

  .1.1. *BufferedReader* (Reader in).

  .1.2. *BufferedReader* (Reader in, int tamaño).

  .2. Métodos

Método	Descripción
void close **()**	Cuando finalice la operación de lectura, debemos cerrar el lector para que otros puedan hacer uso de él.
void mark	(limite)
boolean marksupported **()**	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.
int read **()** ***int read (char [] bufer, int desp, int long*)** *String readLine* ()	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.

  .1. 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)

  .2. Métodos

Método	Descripción
boolean canExecute () **boolean canRead ()** **boolean canWrite ()**	El administrador de seguridad determina si la ruta especificada es apta para leer o escribir información.
boolean exists ()	Este método nos devuelve si existe o no un archivo en el sistema de archivos.
int compareTo (File ruta)	Devuelve el orden entre dos rutas.
boolean equals (Object obj)	Determina si dos rutas son iguales.
boolean createNewFile () **static File createTempFile (String prefijo, String sufijo)** **static File createTempFile (String prefijo, String sufijo, File directorio)**	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.

boolean delete () **void deleteOnExit ()**	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.
File getAbsoluteFile () **File getCanonicaFile ()** **String getAbsolutePath ()** **String getCanonicalPath ()**	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.
String getName ()	Devuelve el nombre del archivo.
String getParent ()	Devuelve la ruta del directorio que lo contiene.
FilegetParentFile ()	Devuelve la ruta abstracta del directorio que contiene la ruta.
String getPath ()	Devuelve la cadena que equivale a la ruta abstracta.
long getFreeSpace () **long getTotalSpace ()** **long getUsableSpace ()** **long length ()**	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.
boolean isAbsolute () **boolean isDirectory ()** **boolean isFile ()** **boolean isHidden ()** **boolean lastModified ()**	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.

String [] list () **String list (FilenameFilter filtro)** **File [] listFiles ()** **File [] listFiles (FileFilter filtro)** **File [] listFiles (FilenameFilter filtro)** **StaticFile [] listRoots ()**	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.
boolean mkdir () **boolean mkdirs ()**	Se utilizan para crear el directorio que especifique la ruta. Además, crea los directorios intermedios que sean necesarios.
boolean renameTo (File destino)	Modifica el nombre de un fichero. Si se realizan los cambios con éxito, los métodos van a devolver el valor de True.
boolean setExecutable (boolean ejecutable) **boolean setExecutable (boolean ejecutable, boolean soloPropietario)** **boolean setLastModified (long hora)** **boolean setReadable (boolean legible)** **boolean setReadable (boolean legible, boolean soloPropietario)** **boolean setReadOnly ()** **boolean setWritable (boolean admiteEscritura)** **boolean setWritable (boolean admiteEscritura, Boolean soloPropietario)**	Utilizados para fijar la ruta a la que se aplican. Si la operación se realiza con éxito devuelve *True* y *False* en caso contrario.
String toString () **URI toURI ()**	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 *Data Output*, 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
boolean readBoolean ()	Lee 1 byte y si es 0, devuelve true, en caso contrario, devuelve 0.
byte readByte ()	Lee y devuelve 1 byte.
char readChar ()	Lee 2 bytes y devuelve un *char*.
double readDouble ()	Lee 8 bytes y devuelve un *double*.

float readFloat ()	Lee 4 bytes y devuelve un *float*.
void readFully (byte [] t)	Lee todos los bytes y los almacena en "t".
void readFully (byte [] t, int off, int leng)	Lee un máximo de *leng bytes* y los almacena en t a partir de la posición *leng*.
int readInt ()	Lee 4 bytes y devuelve un entero.
String readLine ()	Lee una línea y devuelve un *String*.
long readLong ()	Lee 8 *bytes* y devuelve un *long*.
short readShort ()	Lee 2 bytes y devuelve un *short*.
int readUnsignedByte ()	Lee 1 *byte*, añade un *vlor* nulo y devuelve un entero (de 1 … 255).
int readUnsignedShort ()	Lee 2 *bytes*, añade un *vlor* nulo y devuelve un entero (de 1 … 65535).
String readUTF ()	Lee una cadena codificada previamente en UTF- 8.
int skipBytes (int n)	Pretende descartar "n" *bytes*.

- **La interfaz DataOutput**

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

DataOutputStream	MemoryCacheImageOutputStream
ImageOutputStreamImpl	FileImageOutputStream
RandomAccessFile	ObjectOutputStream
FileCacheImageOutputStream	

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

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

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
}
```

4.3 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 a reunir 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 de 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.

4.4 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 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.

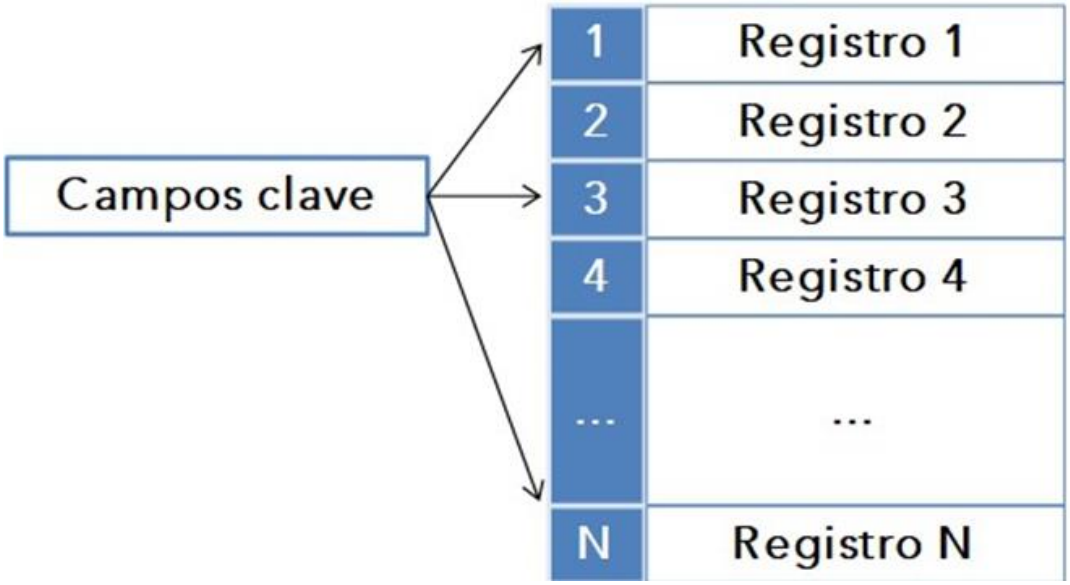
Ficheros Secuenciales Indexados

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 hacer el acceso 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 poder 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 fijarnos 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.

- **Operaciones lectura/ escritura de ficheros**

Pasos para leer o escribir en un fichero secuencial:

.4.1. Lectura secuencial

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
```

.4.2. Escritura secuencial

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
```

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

.4.3. Lectura aleatoria

CÓDIGO:

```
Fichero f1;                              //variable tipo fichero
f1.Abrir (lectura);                      // Abrimos el fichero
Mientras condición según el programa  //Tratamos el fichero
    // mientras cumpla una condición
    //situamos puntero en la posición deseada
    f1.Leer (registro);
    Operaciones con el registro leído;
Fin Mientras;
f1.Cerrar();                  //cerramos fichero
```

.4.4. Escritura aleatoria

CÓDIGO:

```
Fichero f1;                           //variable tipo fichero
f1. Abrir (escritura);                //Abrimos el fichero
Mientras se necesiten escribir datos hacer //Mientras cumpla una condición
    //situamos puntero en la posición deseada
    f1.Escribir (registro);
    Operaciones con el registro escrito;
Fin Mientras;
f1.Cerrar();                    //cerramos fichero
```

Cierre de ficheros

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

Como hemos visto en el ejemplo anterior:

CÓDIGO:

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

UF6: POO. Introducción a la persistencia en las Bases de Datos

En este apartado vamos a estudiar la forma de utilizar un **gestor de bases de datos combinado con el lenguaje de programación Java**.

1. 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 relacional** es 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 llegar 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.

1.1 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.

- Podemos comenzar introduciendo en el código fuente es 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:

.1. Registramos el driver mediante la función *forName*.

.2. 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 su contraseña.

CÓDIGO:

```
public Connection conexionBBDD(){
    //Declaramos una variable para la cadena de conexión
    Connection conec = null;    //Controlamos las excepciones que aparecen al interactuar con la BBDD
    try{
        //Registrar el driver
        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
        msjErr.mensajeError("Se ha producido un error al conectar con la Base
                                de Datos.\n" + e.toString());
                }
       return conec;
}
```

Estos son los pasos que tenemos que llevar a cabo 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 conexion){
    try{
        //Cierre de conexión
        conexion.close();
    }catch(SQLException e){
 //Controlamos excepción que se pueda producir al cierre de la conexión
        msj.Err.mensajeError("Se ha producido un error al conectar
                             con la Base de Datos.\n"
                             + e.toString());
    }
}
```

1.2 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:

```
//Declaramos las variables para los datos y la query
String datos = "";
String consultaInsercion = "INSERT INTO sorteo (fecha, n1, n2, n3, n4, n5, complementario) VALUES (";
//Preparamos los datos a insertar en la tabla mediante
//la query de inserción
datos = "\'" + formateaFecha(this.txtFecha.getText()) + "\', ";
datos = datos + this.textN1.getText() + ", ";
datos = datos + this.textN2.getText() + ", ";
datos = datos + this.textN3.getText() + ", ";
datos = datos + this.textN4.getText() + ", ";
datos = datos + this.textN5.getText() + ", ";
datos = datos + this.textComplementario.getText() + ")";
//Montamos la consulta completa
 consultaInsercion = consultaInsercion + datos;
//Es necesario controlar las excepciones al interactuar con la BBDD
try{
    //Creamos la sentencia
    Statment consulta = con.createStatement();
    consulta.executeUpdate(consultaInsercion);
    conectado.cerrarConexion(consulta);
}
```

2. Diseño de programas con lenguajes de POO para gestionar bases de datos objeto-relacionales

Una **base de datos objeto-relacional** 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, pueden usar colecciones o conjuntos (por ejemplo: *arrays*), pueden representar de forma directa los atributos compuestos, y pueden 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 necesitarán 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.

2.1 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.

2.2 Recuperación y manipulación de la información

En el ejemplo anterior vimos 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 los siguientes pasos:

1) Primero creamos la sentencia de conexión (***Statement***).
2) Después, obtenemos en ***resulSet*** 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{
    //Variable para mostrar el resultado en la label. Aprovechamos
    //que el JLabel puede interpretar HTML para sacar los
    //datos correctamente
    String SalidaAmostrar = "<html><body>";
    //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("n1")
                         + ", " + salida.getInt("n2") + ", "
                         + salida.getInt("n3") + ", "
                         + salida.getInt("n4") + ", "
                         + salida.getInt("n5") + ", "
                         + salida.getInt("complementario") + "<br>";
    }
    //Cerramos las conexiones
    conectado.cerrarConexion(consulta);
    conectado.cerrarConexion(salida);
```

3. Diseño de programas con lenguajes de POO para gestionar las bases de datos orientada a objetos

3.1 Introducción a las bases de datos orientada a objetos

Para las bases de datos orientadas a objetos vamos a utilizar un gestor combinado con el lenguaje de programación Java. No es muy diferente a los SGBDOO utilizados en los gestores de las bases de datos relacionales.

3.2 Características de las bases de datos orientadas a objetos

A continuación vamos a indicar las diferentes **características** que presentan las **bases de datos orientadas a objetos**.

- Se diseñan de la misma forma que los programas orientados a objetos, es decir, debemos pensar como si se tratara de un programa real.
- Cada tabla que definíamos 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 SGBDOO (Sistemas de Gestión de Base de Datos Orientada a Objetos) 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.

3.3 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 programación orientada a objetos**.

Para trabajar con las bases de datos vamos a utilizar un tipo de objetos 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.

3.3.1 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 BBDDOO 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 que 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 orientado a objetos permite:

- Atributos multi-avaluados.
- 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 bases de datos orientadas a objetos deben soportar dos tipos de herencia: la relación "es un" y la relación "extiende".
    - "**es un**" (generalización- especialización): crea jerarquías donde las diferentes subclases que existan son tipos específicos de la superclase.
    - "**extiende**": una clase expande su superclase en vez de hacerla más pequeña en un tipo más específico.

3.3.2 Integridad de las relaciones

Los identificadores de los objetos se deben corresponder en ambos extremos de una relación para que una base de datos orientada a objetos funcione de forma correcta. La integridad en una base de datos es una de las características más importante a llevar a cabo. Los datos almacenados en el campo de "clave ajena" guardados en una tabla tienen que estar contenidos en el campo clave de la tabla a la que se relaciona.

3.3.3 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 puede 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.
  - **.1.1.** Diagrama de clases.
  - **.1.2.** Diagrama de estructuras compuestas.
  - **.1.3.** Diagrama de objetos.
  - **.1.4.** Diagrama de componentes.
  - **.1.5.** Diagrama de implementación (despliegue).
  - **.1.6.** Diagrama de paquetes.
- **Diagramas de comportamiento:** Basados en lo que debe suceder en el sistema.
  - **.1.7.** Diagrama de interacción:
    - Diagrama de secuencia.
    - Diagrama resumen de interacción.
    - Diagrama de comunicación.

- Diagrama de tiempo.

.1.8. Diagrama de actividad.

.1.9. Diagrama de casos de uso.

.1.10. Diagrama de máquina de estado.

En otros modulos se profundizará en muchos de estos tipos de diagramas.

3.4 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 pueda ser reutilizable y soportado por otros sistemas que soporten este modelo.

Este modelo de objetos **permite realizar el diseño de una BDDOO** implementada desde lenguajes POO, especificando los elementos que se definirán para la persistencia en una BDDOO.

Por tanto, su función es definir los elementos y la persistencia entre ellos en las BDDOO, permitiendo que sean portables entre los sistemas de POO que los soportan. Definen así un estándar en los SGBDOO.

ODMG se compone por:

.2. Modelo de Objeto

.3. Lenguaje de definición de objetos ODL.

.4. Lenguaje de consulta de objetos OQL.

A continuación, se detallan estos dos puntos.

3.4.1 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.

3.4.2 Lenguaje de consulta de objetos OQL

Es un lenguaje declarativo del tipo de SQL que permite realizar consultas de modo eficiente sobre bases de datos orientadas a objetos, 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.

3.5 Prototipos y productos comerciales de 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 sistema gestor de bases de datos orientadas a objetos 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.
    - Para el **SGBD**, las características 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