

Módulo 3

Programación

```

function updatePhotoDescription() {
    if (descriptions.length > (page * 9) + (currentimage.substring(0, 1)) {
        document.getElementById('bigImageDesc').innerHTML = descriptions[page * 9 + currentimage.substring(0, 1)];
    }
}

function updateAllImages() {
    var i = 1;
    while (i < 10) {
        var elementId = 'foto' + i;
        var elementIdBig = 'bigImage' + i;
        if (page * 9 + i - 1 < photos.length) {
            document.getElementById( elementId ).src = 'images/min/' + photos[page * 9 + i - 1];
            document.getElementById( elementIdBig ).src = 'images/max/' + photos[page * 9 + i - 1];
        } else {
            document.getElementById( elementId ).src = 'images/min/';
            document.getElementById( elementIdBig ).src = 'images/max/';
        }
        i++;
    }
}

```

UF4: PROGRAMACIÓN ORIENTADA A OBJETOS (POO).....4

1. Java	4
1.1. Características	5
1.2. Descarga e instalación	6
1.3. Estructura de un programa en Java.....	7
2. Introducción a la programación orientada a objetos.	8
2.1. Tipos de datos primitivos.	9
2.2. Definición de objetos. Características	18
2.3. Tablas de tipos primitivos ante tablas de objetos.	20
2.4. Métodos.	20
2.5. Utilización de propietarios.	21
2.6. Destrucción de objetos. Liberación de memoria.....	22
3. Desarrollo de programas organizados en clases.....	22
3.1. Concepto de clase.	23
3.2. Creación de atributos.	26
3.3. Creación de métodos.	26
3.4. Sobrecarga de métodos.	27
3.5. Creación de constructores.	29
3.6. Creación de destructores y/o métodos de finalización.	29
3.7. Uso de clases y objetos. Visibilidad.	29
3.8. Conjuntos y librerías de clases. Clase System mejor de título	31
4. Utilización avanzada de clases en el diseño de aplicaciones.	34
4.1. Composición de clases.....	34
4.2. Herencia.	37
4.3. Jerarquía de clases: superclases y subclases.	40
4.4. Clases y métodos abstractos.	41
4.5. Sobreescritura de métodos (Overriding).....	42
4.6. Herencia y constructores/destructores/métodos de finalización.....	43
4.7. Interfaces.....	43

UF5: POO. LIBRERÍAS DE CLASES FUNDAMENTALES46

1. Aplicación de estructuras de almacenamiento en la programación orientada a objetos.....	46
1.1 Estructuras de datos avanzadas.	46
1.2 Creación de arrays.....	50
1.3 Arrays multidimensionales.....	53
1.4 Cadena de caracteres. String.....	53
1.5 Colecciones e iteradores.	55
1.6 Clases y métodos genéricos.	57
1.7 Manipulación de documentos XML. Expresiones regulares de búsqueda. (buscar en internet expresiones regulares).....	59
2. Control de excepciones.....	61
2.1 Captura de excepciones.	62
2.2 Lanzamiento de excepciones.	62
2.3 Excepciones y herencia.	65
3. Interfaces gráficas de usuario.	66
3.1 Creación de interfaces gráficas de usuarios simples	66
3.2 Eventos y propiedades.	69

3.3	Paquetes de clases para el diseño de interfaces.....	72
4.	Lectura y escritura de información.	74
4.1	Clases relativas de flujos. Entrada/salida	74
4.2	Tipos de flujos. Flujos de byte y de caracteres.....	75
4.3	Ficheros de datos. Registros.	87
4.4	Gestión de ficheros.	88
 UF6: POO. INTRODUCCIÓN A LA PERSISTENCIA EN LAS BASES DE DATOS.....		94
1.	Diseño de programas con lenguajes de POO para gestionar las bases de datos relacionales.	94
1.1	Establecimiento de conexiones.....	95
1.2	Recuperación y manipulación de información.....	96
2.	Diseño de programas con lenguajes de POO para gestionar las bases de datos objeto- relacionales.	97
2.1	Establecimiento de conexiones.....	98
2.2	Recuperación y manipulación de la información.	98
3.	Diseño de programas con lenguajes de POO para gestionar las bases de datos orientada a objetos.	99
3.1	Introducción a las bases de datos orientada a objetos.	99
3.2	Características de las bases de datos orientadas a objetos.	99
3.3	Modelo de datos orientado a objetos.....	99
3.3.1	Relaciones	100
3.3.2	Integridad de las relaciones.....	101
3.3.3	UML.	101
3.4	El modelo estándar ODMG.	102
3.4.1	Lenguaje de definición de objetos ODL.	103
3.4.2	Lenguaje de consulta de objetos OQL.	103
3.5	Prototipos y productos comerciales de SGBDOO	103
 BIBLIOGRAFÍA		105

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, etcétera. La principal diferencia la podemos encontrar en las diferentes palabras reservadas y en cómo se van a definir en un lenguaje de programación específico.

1. 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 cuestión en la que se ejecuta el programa, y se basa en aquellos servicios que ofrece el sistema operativo que van a permitir atender las solicitudes que necesite el 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.

¡Importante!

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.

- **Alto rendimiento**

Todas las aplicaciones que genere van a ser más rápidas y óptimas que las de otros lenguajes.

- **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

Cuando necesitemos instalar Java en un ordenador, necesitamos un entorno de ejecución (*runtime*), que va a ser el encargado de traducir lo que los programas escriban.

Los diferentes sistemas operativos, como *Windows* y *Linux*, no cuentan con una versión propia para Java, es decir, tienen que descargarla.

Sin embargo, el sistema operativo MAC sí que cuenta con una versión de Java que se instala directamente, sin necesidad de descargarla de las páginas destinadas a tal fin.

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, además de *NetBeans* (entorno de desarrollo creado por Java).

Una vez descargado, debemos seguir los pasos para instalarlo.

1.3. Estructura de un programa en Java

Cuando creamos un nuevo proyecto en *Netbeans*, **éste va a incluir un fichero ejecutable que se debe llamar de la misma forma que el proyecto en cuestión**, y va a llevar la extensión `.java`.

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`.

El fichero `.class` es el que utilizaremos para ejecutar nuestro programa en cualquier sistema.

Veamos un ejemplo:

CÓDIGO:

```
package ejemplo;

/**
 * @author ilerna
 */
public class Ejemplo {
    public static void main (String [] args) {
        /*
         * Código de las aplicaciones
        */
    }
}
```

```
        */  
    }  
}
```

Comprobamos que la clase “Ejemplo” comienza a partir de la instrucción:

public class Ejemplo

Entre llaves podemos incluir aquellos atributos y métodos que necesitemos en nuestro código para, posteriormente, realizar las llamadas correspondientes en la función **main ()** cuyo formato de cabecera lo escribiremos de la siguiente forma:

public static void main (String [] args) {...}

Esta función que vamos a utilizar para el **main**, 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.

2. Introducción a la programación orientada a objetos.

Según 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 sólo 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. Sólo se debe conocer la forma en la que debemos contestar a estos mensajes.

2.1. Tipos de datos primitivos.

Cuando hablamos de tipos de datos primitivos, nos referimos a los que denotan magnitudes de tipo numérico, carácter o lógico. Se caracterizan, principalmente, por su eficiencia, ya que consumen menos cantidad de memoria y permite 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 constantes o en variables.

De manera que, si queremos llevar a cabo la suma de dos valores constantes (el número 2 y el 3), lo podríamos hacer de la siguiente forma:

CÓDIGO:

```
public class Suma {  
    public static void main (String [] args) {  
        System.out.println(2+3);  
    }  
}
```

Este programa se compone de varias fases:

- Primero se debe escribir.
- Después, compilar.

- Y, por último, ejecutar.

En Java, las cuatro reglas básicas de operaciones, están representadas por los operadores: +, -, * y /.

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 cada vez que 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 que las siguientes palabras reservadas:

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

```
int num1;
```

Declaramos la variable num1 de tipo entero.

```
Num1=2;
```

Asignamos a la variable Num1 el valor de 2;

- 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 asignarle un valor.

• 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 `“/*” “*/”` para los que contengan más de una.

2.1.1. Tipos numéricos enteros

En los ejemplos anteriores, hemos trabajado con variables de tipo entero (int). Este tipo de datos, es uno de los muchos que tenemos disponible en Java a la hora de trabajar con números. Debemos tener en cuenta que las variables ocupan un espacio en la memoria, y eso es lo que diferencia unas de las otras.

Supongamos que tenemos que escribir 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.

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

Nombre	Tamaño (bits)
short	16
Int	32
Long	64

2.1.2. 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)

Hasta ahora, hemos trabajado con todos estos operadores excepto con el “%” (módulo), que es el resto del resultado obtenido después de realizar la división entre el primer factor y el segundo.

2.1.3. Tipos numéricos reales

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

Nombre	Tamaño (bits)
float	32
double	64

Si se utilizan clases como *BigInteger* y *BigDecimal*, permiten realizar cálculos (enteros o coma flotante) con precisión arbitraria.

2.1.4. Operadores aritméticos para números reales

Estos operadores son los habituales (+, -, *, /), es decir, todos excepto el módulo (%), que no existe cuando se utilizan números reales.

2.1.5. 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';
dato='%n';
```

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

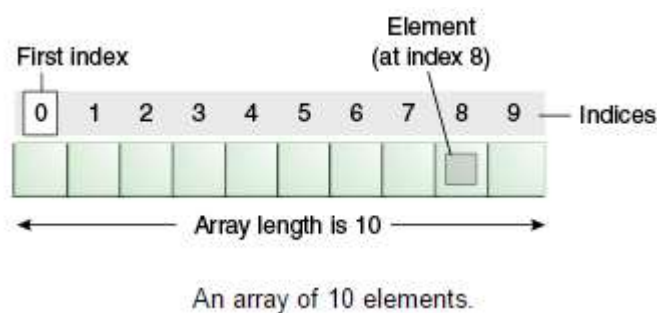
```
char dato = 'A';
```

2.1.6. Tipo boolean

El tipo de dato *boolean* sólo almacena valores lógicos (*true* o *false*). El tipo *boolean* cuenta con un algoritmo que, si se cumple una determinada condición será verdadero (*true*) y, si no, será falso (*false*).

2.1.7. Listas y tablas en Java

En Java se pueden crear tablas, listas y estructuras de una forma bastante simple, y éstas pueden tener cualquier número de índices.



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

Debe cumplir con la siguiente sintaxis:

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 listas y tablas

- En Java, el primer índice de una tabla 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 tipo *ArrayIndexOutOfBoundsException*.
- Sólo se puede dar valor a una lista completa asignando un valor en el momento de su declaración. Una vez que ya esté declarada, sólo se pueden asignar valores a la tabla de forma individual.
- En Java es posible hacer una copia de una lista por asignación, aunque con limitaciones.

2.1.8. Punteros en Java y listas lineales de clases

Los punteros son direcciones variables en memoria. Son muy utilizados en C y C++, ya que ofrece bastantes ventajas, como:

- Es muy práctico a la hora de acceder a los distintos bloques.
- Bastante flexible, ya que permite crear o destruir variables de forma muy sencilla mientras se ejecuta el programa.

Aunque, como inconveniente, **debemos destacar que los punteros provocan una gran cantidad de errores, que son difíciles de localizar y corregir.** Lo que supone que la ejecución se ralentice de forma considerable.

La aritmética de los punteros ha sido eliminada en Java, aunque esto no quiere decir que los punteros hayan desaparecido de Javax, sino que, en el caso de Java, los punteros se denominan criptores y se utiliza para todo lo que no sean variables atómicas de tipos primitivos.

Todas las variables de clase y todas las listas se pueden manejar a través de punteros. Las listas necesitan una reserva de espacio para estos descriptores.

Sintaxis para las listas:

CÓDIGO:

```
TipoBase [ ] nombre_Lista = new TipoBase [DIMENSION];
TipoBase [ ] [ ] nombre_Tabla = new TipoBase [FILAS][COLUMNAS];
TipoBase [ ] [ ] [ ] nombre_Trid = new TipoBase
[CAPAS][FILAS][COLUMNAS];
```

- Si **TipoBase** es un tipo primitivo (int, float, char...), estas sentencias van a crear una lista de valores de ese mismo tipo, y se puede hacer uso de ella de forma inmediata.
- Si **TipoBase** es de tipo referencia, las sentencias van a reservar un espacio de memoria para una lista de descriptores tipo base. Pero esa lista contiene valores nulos (*null*), porque los punteros no tienen el valor de la dirección de elementos del tipo base, por lo que no permite que se utilice de forma directa.

2.1.9. Cuadro de tipos primitivos

Tipo primitivo	Tamaño (bits)	Tipo envoltorio
boolean	-	Boolean
char	16	Carácter
byte	8	Byte
short	16	Short
int	32	Integer
long	64	Long
double	64	Double
Doid	-	Void

2.2. Definición de objetos. 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; datos 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 definirlo como un contenedor de datos, mientras que una clase actúa como un molde en la construcción de objetos.

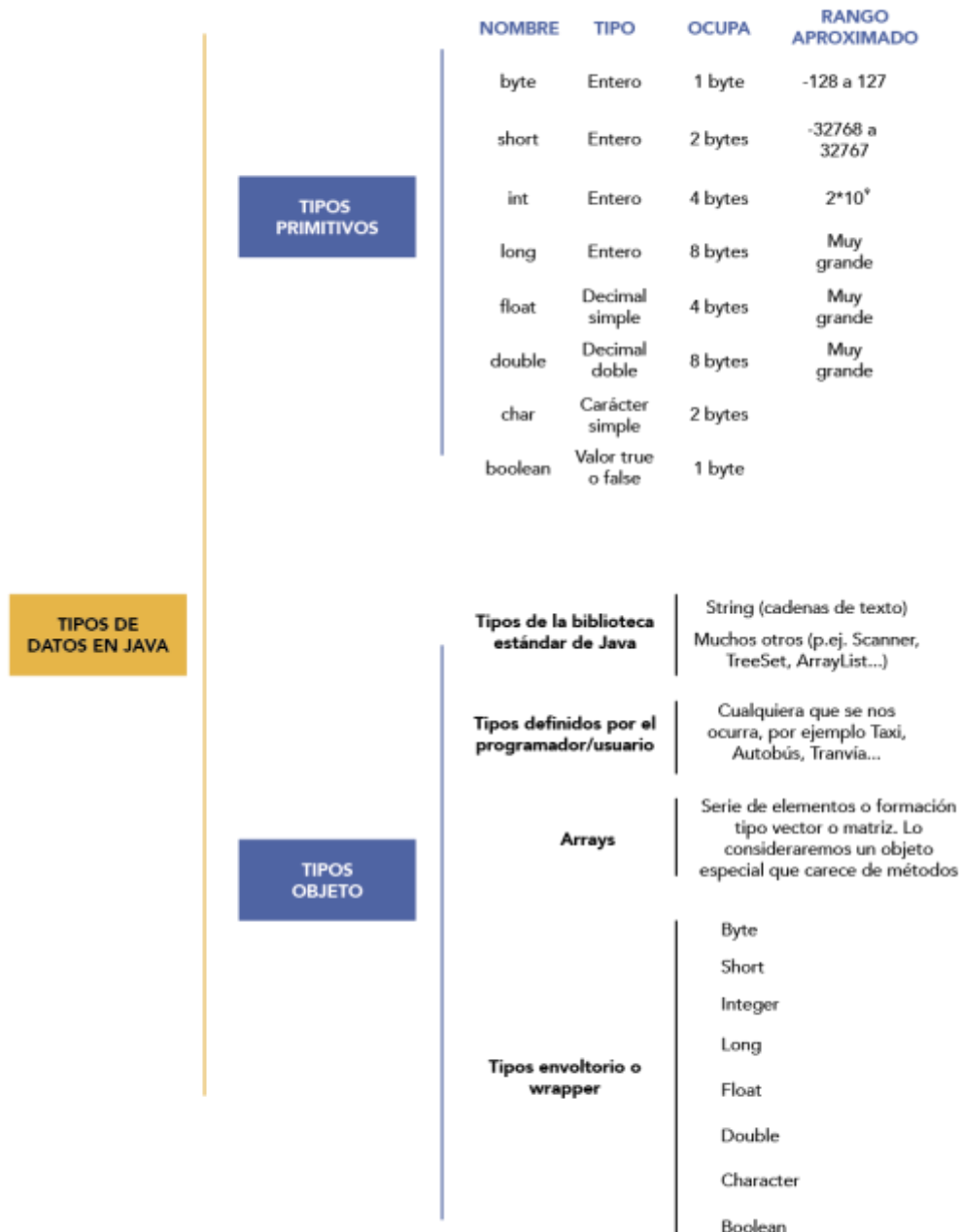
A continuación, vamos a ver un ejemplo relacionado con la vida cotidiana en el que aclararemos todos estos conceptos:

- 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.

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 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. La estructura vector, junto con sus operaciones disponibles se han visto ya previamente.

2.4. Métodos.

Los métodos son las funciones “propias” que tiene una clase, capaz 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 sean diferentes entre ellos, es decir, que el número y el tipo de sus argumentos sea diferente.

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, no ejemplar. 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, nos debemos crear un ejemplar de ésta, que es el 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 van 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, y para ser empleados, no necesitan crear un ejemplar determinado de la clase.

2.5. Utilización de propietarios.

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 de utilizar el carácter punto “.”. En siguientes apartados podemos ver algún ejemplo.

2.6. Destrucción de objetos. 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 el objeto en cuestión.

Es 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 vital.

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 éstos se vayan eliminando.

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

Para ello, crea un método (*finalize*) que debe contener las siguientes características:

CÓDIGO:

```
Protected void finalize () throws throwable {
    //Cuerpo del destructor
}
```

3. Desarrollo de programas organizados en clases.

3.1. Concepto de clase.

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:

```
public class nombre {  
    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 definir los componentes de un objeto: **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 modelo

Aunque si a nuestro taller llega un seat Ibiza de 3 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.

3.1.1. Estructura y miembros

CÓDIGO:

```
public class nombre {  
    atributos;  
    métodos;
```



```
}
```

- **class** → palabra reservada que se utiliza para indicar el inicio de una clase.
- **nombre** → 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** 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 éste sólo 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 sólo 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:

- **Miembros protected**

Lo utilizamos cuando trabajamos con varias clases que heredan unas de otras, de tal forma que, aquellos miembros que queremos que actúen de forma privada, se suelen declarar como protected. Así que, 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, la clase puede actuar de forma similar sin utilizar package.

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:

Tipo_dato nombre_atributo;

Donde:

- **Tipo_dato:** un atributo puede ser de cualquier tipo de datos que existan, como int, double, char o algunos más complejos, como estructuras, tablas o incluso objetos.
- **Nombre_atributo:** identificador que vamos a utilizar para esa variable.

3.3. Creación de métodos.

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

CÓDIGO:

```
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 devuelve 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 ésta.
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 ésta.
- **final.** Estos métodos no ofrecen la posibilidad de sobreescribirlos.
- **native.** Métodos implementados en otros lenguajes pero que deseamos añadir a nuestro programa. Podremos hacerlo añadiendo la cabecera de la función en cuestión, y sustituyendo el cuerpo del programa por; (punto y coma).
- **synchronized.** Utilizado en aplicaciones multi-hilo.

Y, además, como cualquier clase, debemos definir los métodos correspondientes:

- **Constructores.** Crean un objeto y se ejecutan automáticamente.
- **Selectores.** funciones que han sido diseñadas para poder devolver el valor de los atributos miembros de una clase determinada.
- **Modificadores.** Ofrecen la posibilidad de realizar cualquier modificación sobre los valores de los atributos miembro de la clase mediante el uso del programa principal.

3.4. Sobrecarga de métodos.

La sobrecarga de operadores permite asignar más funcionalidad a un operador que ya existe, ofreciendo la posibilidad de que realice diferentes operaciones para conseguir operar con otros objetos.

En Java, podemos definir numerosas funciones que tengan el mismo nombre, pero que con diferentes parámetros.

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

CÓDIGO:

```
public void visualizar () {
    ...
}
public void visualizar (Objeto X) {
    ...
}
public void visualizar (Objeto X, int num1) {
    ...
}
```

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 porque, si escribimos:

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 ésta. Es una función miembro de una clase y se va a ejecutar siempre de forma automática al crearse una instancia de esta.

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

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 antemano de la eliminación o liberación de memoria que puede ocupar un objeto determinado a través de la recolección de basura.

Para ello, crea un método “**finalize ()**”, que podemos utilizarlo siguiendo la estructura que detallamos, a continuación:

CÓDIGO:

```
public void finalize () {  
    Sentencias;  
}
```

Esta función puede ser heredada por todas las clases, y su función principal es hacer una llamada al recolector de basura para que se encargue de eliminar aquellos elementos que no sean necesarios en el menor tiempo posible.

3.7. Uso de clases y objetos. Visibilidad.

Como ya hemos visto en apartados anteriores la definición de las clases y de los objetos, tenemos claro la estructura que deben ir siguiendo a la hora de implementar un programa en Java. Vamos a ver, a modo de ejemplo, 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 y 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 sólo declarásemos el objeto, no podríamos utilizarlo.

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.

Veamos un caso práctico:

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 ();
```

3.8. Conjuntos y librerías de clases. Clase System mejor de título

- **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. Dispositivos estándar para entrada / salida.

- **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. Método ();
//Donde método puede ser cualquier método de impresión de
información en la //salida estándar.
En el siguiente ejemplo podemos verlo de una forma más clara:
System. out. println ("Este es el texto");
//Esta función muestra por pantalla el texto escrito entre
comillas con un salto de //línea.
```

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
%d	Números enteros.
%c	Caracteres.
%s	Cadena de caracteres.
%f	Número real (<i>float</i> o <i>double</i>).
%x. yf	Número <i>float</i> o <i>double</i> mostrado en formato s dígitos de la parte entera e y dígitos en la parte decimal.
%x.ye %n	Número float o double en notación científica.
%o	Número entero mostrado en octal.
%h	Número entero mostrado en hexadecimal.

- **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.

4. Utilización avanzada de clases en el diseño de aplicaciones.

4.1. Composición de clases.

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 nueva aproximación a una clase ya existente *Rectángulo*.

Vamos a definir el origen como un objeto, en vez de como dos coordenadas x e y.

- La clase *Punto*

CÓDIGO:

```
public class Punto {  
    int x;  
    int y;  
}
```

Un constructor de la clase Punto lo podemos escribir de la siguiente forma:

CÓDIGO:

```
public Punto (int x1, int y1) {  
    x= x1;  
    y= y1;  
}
```

Al ser el nombre de los parámetros igual que el de los miembros, lo escribimos:

CÓDIGO:

```
public Punto (int x1, int y1) {
    this. x= x1;
    this. y= y1;
}
```

La función mover se va a encargar de cambiar la posición del punto desde una posición (x, y) hasta otra (x+ dx, y+ dy).

CÓDIGO:

```
public void mover (int dx, int dy) {
    x+=dx;
    y+=dy;
}
```

Ahora nos vamos a crear un objeto de la clase “Punto” con el valor de las coordenadas 10 y 23.

CÓDIGO:

```
Punto p= new Punto (10, 23);
```

A continuación, podemos desplazar el punto “p” 10 unidades hacia la izquierda, y 40 hacia abajo.

p. mover (-10, 40).

Por lo que el código de la clase punto quedaría:

CÓDIGO:

```
public class Punto {
    int x= 0;
    int y= 0;

    public Punto (int x, int y) {
        this. x= x;
        this. y= y;
    }
    public Punto () {
        x= 0;
        y= 0;
    }
    void mover (int dx, int dy) {
        x+= dx;
        y+=dy;
    }
}
```

Y vamos a ver ahora la clase “Rectángulo”:

Va a tener como miembros, el “origen”, que es un objeto de la clase *Punto* y las dimensiones de ancho y alto.

CÓDIGO:

```
public class Rectangulo {
    Punto origen;
    int ancho;
    int alto;
    public Rectangulo () {
```

```
        origen = new Punto (0, 0);
        ancho =0;
        alto =0;
    }
    public Rectangulo (Punto p) {
        this (p, 0, 0);
    }
    public Rectangulo (int w, int h) {
        this (new Punto (0, 0), w, h);
    }
    public Rectangulo (Punto p, int w, int h) {
        origen = p;
        ancho = w;
        alto = h;
    }
    void mover (int dx, int dy) {
        origen. mover (dx, dy);
    }
    int area () {
        return ancho * alto;
    }
}
```

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, etcétera.

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 ella sus características y funcionalidades. Se le denomina también **clase padre** o **superclase**
- **Clase derivada.** La 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 {  
    ...  
}  
class Derivada extends Base {  
    ...  
}
```

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

Debemos tener en cuenta una serie de pasos importantes como:

- **Una clase derivada**

Sólo 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.

- **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;
    ...
}

class cualificados extends empleado {
    String titulacion;
    Double extra;
    String departamento;
    ...
}

class obreros extends empleado {
    String destino;
    int horas_extra;
    double precio_hora;
    ...
}

class jefe extends cualificados {
    int total_trabajadores;
    String [] proyectos;
    double extra;
    ...
}
```

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 que exista 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 como “Superclase”, y la que representamos en el nivel inferior la pasamos a llamar 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, sólo 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.

Podemos declararla de la siguiente forma:

CÓDIGO:

```
public abstract class Nombre_Clase {...}
```

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

CÓDIGO:

```
public abstract class Profesor { }
```

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, como:

- No tiene cuerpo, sólo 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 public/ private/ protected Tipo de Retorno/ void  
(parámetros...);
```

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 en Java.
- Las clases se pueden declarar como abstractas, aunque no tengan métodos abstractos. Algunas veces estas clases realizan operaciones comunes a las subclases sin que necesite 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).

4.5. Sobreescritura de métodos (*Overriding*).

Podemos definir la sobreescritura 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 sobreescritura.

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

- Debemos comprobar que la estructura del método sea la misma a la de la superclase.
- Debe tener, no sólo el mismo nombre, sino también el mismo número de argumentos y valor de retorno.

4.6. Herencia y constructores/destructores/métodos de finalización.

• Constructores en clases derivadas

Cuando utilizamos herencias y tenemos algunas clases que heredan de otras, éstas pueden heredar los atributos de la clase base que deben ser inicializados. Si la clase base posee atributos privados, no son accesible 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 éste 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.

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 interface nombre {
...
tipo nombre1 (parámetros);
tipo nombre2 (parámetros);
...
}
```

Donde:

- **modificador:** puede ser cualquiera de los utilizados en las diferentes definiciones de clases: *public*, *private*, *protected*, etc.
- **Nombre:** 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 interface IDerivada extends IBase {
...
}
```

```
}
```

Y, 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 {
    ...
    public tipo nombre (parámetros) {
        ...
    }
}
```

UF5: POO. Librerías de clases fundamentales

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

1.1 Estructuras de datos avanzadas.

- Conjuntos

El marco de lenguaje de colecciones Java cuenta con una interfaz (*Set*), que podemos utilizar a la hora de representar estructuras de datos del mismo estilo que los conjuntos. **Un conjunto es un grupo de valores (no duplicados), es decir, son un grupo de valores únicos que, dependiendo del caso en cuestión, pueden estar ordenados o no.**

Entre las diferentes implementaciones de la interfaz *Set* para crear conjuntos, tenemos:

- **HashSet**: permite almacenar los datos en una tabla de hash.
- **TreeSet**: permite almacenar los datos en un árbol.

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*.

De esta forma:

- Si queremos utilizar el conjunto *Set* en una aplicación, tenemos dos implementaciones a elegir:

FORMA 1:

```
Set <String> nombre = new HashSet <String> ();
```

FORMA 2:

```
Set <String> nombre = new TreeSet <String> ();
```

- Para añadir elementos a un conjunto:

CÓDIGO:

```
nombre. add ("ILERNA");
```

- Para eliminar elementos de un conjunto:

CÓDIGO:

```
nombre. remove ("ILERNA");
```

- Para comprobar si un elemento pertenece a un conjunto de elementos:

CÓDIGO:

```
if (nombre. contains ("ILERNA")) ...
```

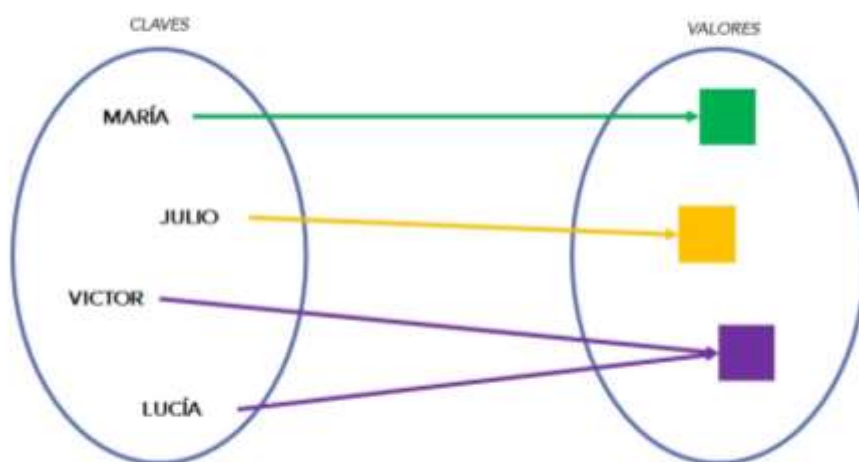
- Si necesitamos recorrer todos los elementos para acceder a uno de ellos, podemos hacerlo mediante la utilización del “**iterador**” (*iterator*), de la misma forma que se utilizan los métodos “next ()” y “hasNext ()” para ir pasando por los diferentes elementos de un conjunto.

CÓDIGO:

```
Iterator <String> iterador = nombre.iterator ();  
While (iterador.hasNext ()) {  
    String nombre = iterador.next ();  
    ...  
}  
Con las listas podemos utilizar un bucle for mejorado  
for (String nombre : nombres) {  
    ...  
}
```

Mapas

Los mapas son colecciones que van a asociar objetos entre una clave y un determinado valor.



Al igual que en los conjuntos, para crear mapas contamos con dos implementaciones distintas:

- **HashMap:** implementa la interfaz *Map*. La interfaz *SortedMap* extiende a *Map*. Los objetos de esta clase se pueden almacenar en tablas de hash.
- **TreeMap:** implementa a *SortedMap*. En este caso, los objetos de esta clase, se almacenan en árboles.
- Si queremos Crear un mapa, tenemos dos implementaciones a elegir:

FORMA 1:

```
Map <String, Color> coloresPreferidos = new HashMap <String, Color>
();
```

FORMA 2:

```
Map <String, Color> coloresPreferidos = new TreeMap <String, Color>
();
```

- 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");
```

A continuación, vamos a detallar un poco algunas estructuras de Java bastante importantes también, como son: las colas(*Queue*), Pilas (*Stack*) y Listas (*List*).

- **Pilas**

Las pilas se definen 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 sólo ver el libro que se encuentra arriba del todo en la cima, por lo que los demás libros, no son accesibles porque la pila se vendría abajo.

- **Colas**

Las colas se definen como una sucesión de varios elementos del mismo tipo, cuya forma de poder acceder a ellos sigue el método de que, los elementos que se inserten los primeros, van a ser también los primeros que van a salir.

Primer elemento en entrar, va a ser también el primero en salir.

Para ir añadiendo elementos, se utiliza el método encolar (*enqueue*), mientras que, para eliminar elementos, utilizaremos la función desencolar (*dequeue*).

- **Listas**

Las listas podemos definir las 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.

1.2 Creación de arrays.

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 de un array, podríamos hacerlo según indicamos a continuación:

array [5];

La **primera casilla** de una tabla siempre es la posición **cero**.

Para inicializar una tabla entera en Java, podemos hacerlo como indicamos en el siguiente ejemplo:

CÓDIGO:

```
//Declaramos el array
Int array = new int [20];
double res = 0.0;

//Inicializamos el primer elemento
System.out.println ("Introduzca el primer número");
array [0] = Integer.parseInt (teclado. readLine());
```

```
//Inicializamos el quinto elemento del array
System.out.println ("Introduzca el quinto número");
array [5] = Integer.parseInt (teclado. readLine ());
//Realiza la suma de sus dos posiciones y ese va a ser el segundo
elemento
array[2] = array [0]+ array[5];
```

Tenemos otras opciones que podemos llevar a cabo a la hora de inicializar los elementos de una tabla.

- Podemos inicializar una tabla cuando la 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 *Netbeans* 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. 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 [100];
array [1] = 5;
array [5] = 2;

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

1.3 Arrays multidimensionales.

El lenguaje Java ofrece la posibilidad de trabajar con *arrays* de más de una dimensión. Presentan una sintaxis muy parecida a los arrays de una dimensión, aunque, en este caso, necesita dos corchetes para indicar las dimensiones de la matriz, tal y como indicamos a continuación:

CÓDIGO:

```
tipo_dato [] [] nombre = new tipo_dato [MAX1] [MAX2];
int [] [] matriz1 = new int [2] [];
matriz1[0] = new int [] {1, 2, 3, 4, 5};
matriz1[1] = new int [] {1, 1, 1, 1, 1, 1, 1, 1, 1};
```

Si representamos la **matriz**, podemos comprobar que es de la siguiente forma:

1	2	3	4	5					
1	1	1	1	1	1	1	1	1	1

1.4 Cadena de caracteres. String

El lenguaje Java no cuenta con ningún tipo de datos 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*).

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

CÓDIGO:

```
String nombre = new String (literal_cadena_caracteres);  
String nombre = literal_cadena_caracteres;  
String nombre = new String (char [] array);  
String nombre = new String (variable_tipo_string);
```

Algunos de los métodos más frecuentes de variables tipo *String*, podemos encontrar algunos como los que detallamos, a continuación:

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

- **String toUpperCase ()**. Devuelve un *array* en el que aparecen los caracteres de la cadena que hace la llamada al método en mayúsculas.
- **String trim ()**. Devuelve una copia de la cadena, pero sin los espacios en blanco.
- **String valueOf (tipo variable)**. Devuelve la cadena de caracteres que resulta al convertir la variable del tipo que se pasa por parámetro.

1.5 Colecciones e iteradores.

Las **colecciones se utilizan para almacenar los datos que están relacionados entre sí**. De esta forma, podemos utilizar un mismo código que sirva para poder procesar todos los elementos que contiene una colección.

Se dispone de una serie de colecciones diferentes, que se generan a partir de otras propias de los datos que conocemos, como pueden ser: las pilas, tablas o colas.

El lenguaje Java, ofrece una serie de clases destinadas a las colecciones de datos.

Estas colecciones pueden realizar operaciones para insertar, eliminar, acceder o buscar diferentes objetos de una determinada colección. Podemos encontrar, entre otras, las diferentes colecciones:

- **ArrayList**
- **HashMap**
- **HashSet**
- **Hashtable**
- **LinkedList**
- **Properties**
- **Stack**
- **TreeMap**
- **TreeSet**
- **Vector**
- **WeakHashMap**

Están en la librería **java. Útil**.

ArrayList

Sigue la siguiente sintaxis:

CÓDIGO:

```
ArrayList array = new ArrayList ();  
ArrayList <Integer> arrayEnteros = new ArrayList <Integer> ();  
//Ejemplo de las declaraciones:  
array.add (3);  
array.add (5.2);  
array.add ("Hola");
```

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

- **add (Object):** permite añadir un elemento a la colección. Como el método está sobrecargado, podemos insertar un elemento en una posición determinada
- **contains (Object):** comprueba si el objeto pasado por parámetros pertenece a la lista.
- **get (int):** obtiene el objeto que se encuentra en la posición pasada por parámetro.
- **indexOf (Object):** devuelve la posición en la que se encuentra el objeto que se pasa por parámetro.
- **isEmpty ():** devuelve un booleano que indica si el array está vacío o no.
- **remove (Objeto):** elimina el primer dato del objeto que se le pasa por parámetro. Como el método está sobrecargado, devuelve el objeto que se encuentra en la posición del objeto que se desea eliminar.
- **set (int, Objeto):** sobrescribe el elemento que se encuentra en la posición *int* por un objeto que se indica por parámetro.
- **size ():** devuelve el tamaño de la lista, es decir, el número de elementos que contiene.
- **toArray():** devuelve una lista de objetos y, en cada casilla de esta lista, inserta un objeto de *ArrayList*.

Vector

El objeto del tipo vector, es muy parecido al de *ArrayList*, aunque dispone de una mayor cantidad de métodos.

Vector dispone de un array de objetos que puede aumentar o disminuir de forma dinámica según las operaciones que se vayan a llevar a cabo.

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.

1.6 Clases y métodos genéricos.

Los **métodos genéricos** son el mecanismo mediante el cual se pretende poder resolver los diferentes problemas que pueden surgir en las clases cuando éstas cuentan con algoritmos susceptibles, que se aplican a varias clases que poseen las mismas características. Las colecciones son un claro ejemplo, ya que las clases pueden pertenecer a clases de cualquier tipo.

Presentan una serie de inconvenientes como:

- Cuando deseamos recuperar los objetos que se han almacenado.
- Cuando el programador almacenaba varios objetos de distintos tipos.

Para solucionar estos inconvenientes, se apuesta por utilizar el mecanismo de genericidad, que permite que sea el programador el que especifique el tipo del que van a ser los objetos que se encuentran almacenados en una colección. Puede que el compilador detecte los intentos de seguir añadiendo objetos incompatibles a una colección. Además, cuando se conoce el tipo de todos los objetos que están almacenados, ya no es necesario volver a dar forma al descriptor cuando se tenga que extraer un objeto.

- **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:

```
[public] class Nombre <T> {  
    T variable;  
    ...  
}
```

Donde “T” representa un tipo de referencia válido en el lenguaje Java.

Los parámetros de clases por tipos no se limitan a un único parámetro, como, por ejemplo, la clase *HashMap* que indicamos a continuación:

CÓDIGO:

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

En este caso, tenemos dos parámetros A y B que hacen referencia al tipo de clave y el valor.

- **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) {
    ...
}
```

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 tipos genéricos) se produce mucha restricción. Por lo que, 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.

1.7 Manipulación de documentos XML. Expresiones regulares de búsqueda. (buscar en internet expresiones regulares)

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 tanto letras, como número o como los siguientes caracteres:

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

Los ejemplos de tratamientos de cadenas de caracteres con patrones pueden ser: Buscar 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 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 patrón = Pattern.compile ("camion");
```

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

CÓDIGO:

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

Podemos ver un ejemplo explicativo en el cual 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 patron
        Pattern patron = Pattern.compile("aabb");
        // creamos el Matcher a partir del patron, la cadena como parametro
        Matcher encaja =
        patron.matcher("aabmaabbnoloaabbmanoloabmolob");
        // 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:

<http://java.sun.com/j2se/1.4/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, ésta se va a tratar en otra zona aparte del código fuente, siempre que la excepción se haya nombrado de alguna forma.

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*.

A continuación, vemos un ejemplo de este tipo de bloques.

CÓDIGO:

```
//Bloque 1
try {
  //Bloque 2
} catch (Exception error) {
  //Bloque 3
}
//Bloque 4
```

2.2 Lanzamiento de excepciones.

En este apartado hemos visto cómo se pueden capturar las diferentes excepciones, pero ahora, vamos a aprender la forma en la que podemos lanzar nuestras propias excepciones.

Para indicar que se ha producido una excepción, creamos una instancia de la misma en una zona crítica del código, incluyendo un bloque *try ... catch* y, anteponiendo la palabra reservada **throw**.

CÓDIGO:

```
try {  
    ...  
    throw new clase_de_error (mensaje);  
    ...  
}  
catch (clase_de_error) {  
    ...  
}
```

A continuación, vamos a ver un ejemplo en el que se debe introducir el login de un usuario.

CÓDIGO:

```
static boolean login (string user, string pass) {
    ...
}
public static void main (string [] args) {
    try
    {
        ...
        if (! login (nombre, passwd)) {
            throw new ErrorLoginException ("Usuario o password
            Incorrecto. Vuelva a intentarlo");
        }
    }
    catch (ErrorLoginException error) {
        System. out. print (error. GetMessage ());
    }
    finally {
        ...
    }
}
```

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

Algunas veces, Java permite omitir una serie de bloques *try ... catch* y añadir una sentencia *throws* en la cabecera del método correspondiente, de tal forma que indiquemos si el método va a lanzar o a generar una excepción.

CÓDIGO:


```
public static void main (String {} args) throws ErrorLogin,  
IOException, NumerFormatException {  
  
    ...  
  
}
```

2.3 Excepciones y herencia.

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.

Creación clases error en Java:

Para crear una clase de error en el lenguaje Java, tenemos:

- Añadir una nueva clase al proyecto.
- Hacer que la nueva clase derive de la clase *Exception*.
- Diseñar en la nueva clase una variable en la que podemos almacenar el código específico del error.
- Configurar el tipo de constructor de la clase para inicializar la variable.
- Sobrescribir un método *getMessage* para devolver el error producido en la clase.

Su estructura la planteamos de la siguiente forma:

CÓDIGO:

```
public class ErrorLoginException extends Exception {  
    String sms;  
    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, y toda información es introducida por el teclado.

El entorno de texto se diferencia por la pantalla negra con las letras en blanco. 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 o como periféricos de entradas 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 de interfaces gráficas de usuarios simples.

Cuando tengamos que crear un proyecto gráfico, es el usuario el que se debe encargar de seleccionar en qué entorno lo va a realizar (*Netbeans*, *Visual Studio* o *Eclipse*, son los más utilizados). Una vez seleccionado el entorno gráfico, se debe elegir el tipo del proyecto, que, en este caso, debe ser gráfico. Una vez creado, ya podemos insertar los diferentes formularios o elementos de un entorno gráfico y pasar a codificar su método principal.

Entorno de trabajo:

En el entorno de trabajo que vamos a utilizar cuenta con una serie de paletas que nos permiten configurar el aspecto gráfico y las diferentes características de los controladores que podremos utilizar.

A continuación, vamos a detallar las más importantes:

- **Navegador**

El navegador nos permite explorar los diferentes objetos que están incluidos en un fichero .java.

- Si el código fuente se realiza en modo consola, sin entorno gráfico, el navegador, muestra aquellas funciones que han sido creadas y sus variables.
- Si el objeto activo que se ha realizado es de tipo formulario, en su vista gráfica, podemos observar aquellos controles que se han ido insertando.

- **Objetos abiertos**

Se van a situar en la zona más alta de la interfaz. Vamos a tener una pestaña por cada archivo de código fuente que tengamos abierto. Iremos accediendo a cualquiera de ellos, haciendo click sobre él.

- **Vista de diseño**

Es la vista predeterminada de los formularios en la que vamos a poder insertar diferentes controles de una manera bastante sencilla y rápida. Podemos cambiar la vista haciendo click en diseño para verlo en otro formato.

- **Paleta**

En paleta vamos a tener las distintas herramientas necesarias para las diferentes acciones que se pueden realizar con los elementos.

- **Propiedades**

Mediante el cuadro de propiedades podemos hacer todas las modificaciones oportunas sobre las distintas características de los controles.

Para crear las distintas aplicaciones con formularios, seguiremos los pasos:

- Añadimos un nuevo formulario.
- Añadimos los controles que necesitamos.
- Seleccionamos los controles que hemos añadido y modificaremos sus características si es necesario.
















Controles

Es fundamental que, antes de profundizar sobre los controles de Java, nos detengamos un poco en las librerías que usa este lenguaje cuando vamos a desarrollar las Interfaces de Usuario Gráficas (GUIs). En el cuadro de herramientas se pueden ver las distintas categorías. En este caso, nos vamos a centrar en **SWING** y **AWT**.

AWT (Abstract Windowing Toolkit). Se refiere a las primeras clases encargadas de los componentes gráficos en Java.

SWING. Se utiliza para mejorar el aspecto de los diferentes controles.

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

3.2 Eventos y propiedades.

En el lenguaje Java existen una serie de propiedades que van a ser bastante útiles y únicas para objetos Java.

Propiedades Java

- **name.** Nos permite identificar el objeto. Si queremos cambiar el nombre de un objeto, sólo tenemos que hacer click 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 ésta, 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 el que se pueden seleccionar los distintos componentes de la lista. Podemos seleccionar más de un dato, entre los siguientes valores:
 - **SINGLE.** Permite seleccionar sólo un elemento.
 - **MULTIPLE_INTERVAL.** Permite seleccionar más de un elemento (no tienen por qué estar juntos).

- **SINGLE_INTERVAL.** Permite seleccionar más de un elemento (deben estar juntos).
- **selectedIndex.** Ofrece la posibilidad de poder 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. Sólo es necesario hacer click sobre **“Events”**.

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

3.3 Paquetes de clases para el diseño de interfaces.

Contenedores en Java:

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

- **Superiores:** *JFrame*, *JDialog* y *JApplet*. En el editor de código se localizan en el cuadro de herramientas bajo la categoría “**Swing Windows**”. Los contenedores 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()**, y 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* y le vamos añadiendo los distintos contenedores que iremos haciendo visibles según las 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 dejar desplazarnos (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.

Menús en Java:

Podemos crearnos menús en Java para ver interfaces más sencillas, y, para ello, hacemos click 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/escritura de datos (información) en distintas fuentes y destinos de información. Destacando, 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

InputStream

- `AudioInputStream`
- `ByteArrayInputStream`
- `FileInputStream`
- `FilterInputStream`
- `InputStream`
- `ObjectInputStream`
- `PipedInputStream`
- `SequenceInputStream`

- `StringBufferInputStream`

OutputStream

- `ByteArrayOutputStream`
- `FileOutputStream`
- `FilterOutputStream`
- `ObjectOutputStream`
- `OutputStream`
- `PipedOutputStream`

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 van a utilizar 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.

Cierra 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, separador de líneas.

- ***BufferedReader y PrintWriter***

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

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

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

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

- **Clase Scanner.** Esta clase se va a utilizar cada vez que tengamos que acceder a valores de variables individuales, sobre todo, cuando se produzca interacción con los usuarios desde el teclado.
 - **Constructores.** Ofrecen la posibilidad de leer la información que proceda de un objeto que esté identificado mediante un ejemplar *File* o, en algunos casos, información procedente de flujos.
 - Scanner (File origen)
 - Scanner (File origen, String nombreCharSet)
 - Scanner (InputStream origen)
 - Scanner (InputStream origen, String nombreCharSet)
 - Scanner (Readable origen)
 - Scanner (ReadableByteChannel origen)
 - Scanner (ReadableByteChannel origen, String nombreCharSet)
 - Scanner (String origen)
 - **Métodos**
 - **void close ().** La función principal que desarrolla la clase *Scanner* es la lectura. Si empleamos la función *close* de un *Scanner* ya cerrado se produce una excepción (*IllegalStateException*).
 - **hasNext** es un método que devuelve *True* si el scanner cuenta con otro símbolo, que se defina como una cadena de caracteres situada entre dos ejemplares del delimitador.
 - **next** proporciona el símbolo siguiente disponible. Si no existe ninguno, lanza una excepción (*NosuchElementException*).

- **Clase `BufferedReader`**. Realiza diferentes operaciones de lectura por bloques, por lo que resulta bastante eficiente.

- **Constructores**

- `BufferedReader (Reader in)`
- `BufferedReader (Reader in, int tamaño)`

- **Métodos**

- **`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 forma más eficiente, es decir, haciendo uso de bloques mayores, va a ir almacenando el texto en una memoria intermedia para que, después, se pueda volcar su contenido al disco.

- **Clase `PrintWriter`**

Cuenta con una serie de métodos que se van a utilizar para escribir información en un fichero.

El método *PrintWriter* es bastante más específico que el *println*.

- **Clase `File`**

Se refiere a un tipo de clase que cuenta con una serie de métodos relacionados con distintas rutas de ficheros o directorios, incluidos los métodos para crear y eliminar estos ficheros.

Estas rutas, una vez que aparecen en los métodos, ya no se pueden modificar.

○ Constructores

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

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

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

○ Métodos

- **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)**.
Esos tres primeros métodos 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 ()**.
Estos dos últimos métodos se van a utilizar 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 getCanonicalFile ()**
- **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
- **File getParentFile ()** → 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 archivo, si corresponde a un archivo 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 ().**
Estos dos métodos se utilizan para crear el directorio que especifique la ruta. Además, crea los directorios intermedios que sean necesarios
- **boolean renameTo (File destino).** Este último método 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).**
Estos métodos los utilizamos para fijar la ruta a la que se aplican. Si la operación se realiza con éxito, devuelve True; False en caso contrario.
- **String toString ()**
- **URI toURI ().** Traducen el fichero a String o URI, según corresponda.

Flujos Binarios:

Cuando trabajamos con ficheros en formato binario, vamos a trabajar de manera diferente según si utilizamos:

1) 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 valor nulo y devuelve un entero (de 1 ... 255)
int readUnsignedShort ()	Lee 2 bytes, añade un valor 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
<code>void write (byte [] t)</code>	Escribe todos los bytes de t
<code>void write (byte [] t, int pos, int leng)</code>	Escribe como mucho leng bytes de t a partir de la posición pos
<code>void write (int x)</code>	Escribe el byte inferior al entero x
<code>void writeBoolean (boolean b)</code>	Escribe un byte de valor 0 si b es falso, y true en caso contrario
<code>void writeByte (int x)</code>	Escribe el byte inferior al entero x
<code>void writeBytes (String s)</code>	Escribe un byte por cada carácter de s
<code>void writeChar (int x)</code>	Escribe los 2 bytes de los que consta un char
<code>void writeChars (String s)</code>	Escribe todos los chars (a 2 bytes por char)
<code>void writeDouble (double v)</code>	Escribe 8 bytes
<code>void writeFloat (float f)</code>	Escribe 4 bytes
<code>void writeInt (int x)</code>	Escribe 4 bytes
<code>void writeLong (long l)</code>	Escribe 8 bytes
<code>void writeShort (short s)</code>	Escribe 2 bytes
<code>Void writeUTF (string s)</code>	Escribe el contenido (codificado en UTF) de la cadena

2) Ficheros de colecciones u objetos

Existen mecanismos que se van a utilizar para garantizar la persistencia de los objetos, y 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.ObjectInputStream in)  
    throws IOException, ClassNotFoundException;  
  
    ...  
}
```

El mecanismo de serialización o pasivación almacena en disco el contenido de un objeto y, de esta forma, permite que posteriormente se pueda reconstruir, volviendo así al estado inicial que tenía antes de la 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, además de 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 almacena 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 que debemos tener en cuenta para poder trabajar con ellos, son:

- Se sitúan en unos 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 ésta no necesita otras aplicaciones ejecutándose para que exista dicha información.

4.4 . Gestión de ficheros.

Podemos hacer una clasificación de los diferentes ficheros según por cómo organizan la información, que se dividen en:

- Secuenciales
- Aleatoria o directa
- Secuencial indexada

4.7.1. Ficheros Secuenciales

Los ficheros secuenciales son aquellos que almacenan los registros en posiciones consecutivas, y 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 sólo podemos realizar una operación de lectura/escritura a la vez, ya que, cuando un fichero se está leyendo, no se puede escribir o de la misma forma, cuando está siendo escrito, no puede llevar a cabo ninguna operación de lectura.



4.7.2. 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 está, a la hora de realizar alguna operación con él, sólo debemos colocar el manejador en esa posición para que lleve a cabo la operación indicada.



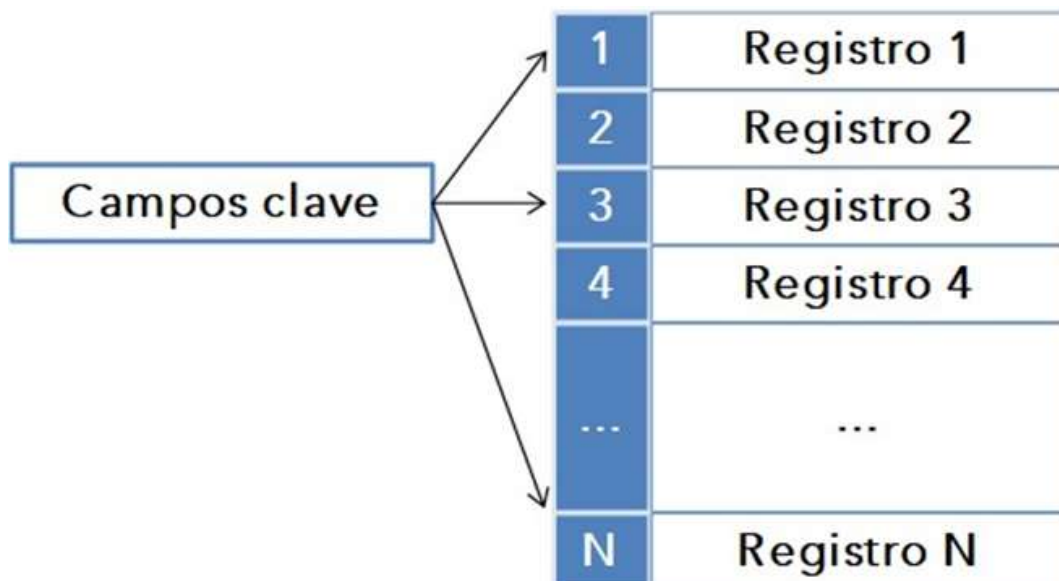
4.7.3. Ficheros Secuenciales Indexados

Estos ficheros, 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 que buscamos.
- Una vez que lo conocemos, ya podemos hacer el acceso a éste 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, y 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.

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

- **Lectura.** Sólo permite realizar operaciones de lectura sobre el fichero.
- **Escritura.** Permite realizar operaciones de escritura sobre el fichero. Si el fichero en el que se quiere escribir ya existe, será borrado.
- **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.
- **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.**

▪ 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
```

▪ Escritura secuencial

CÓDIGO:

```
Fichero f1                                //variable tipo fichero
f1. Abrir (escritura);                     // Abrimos el fichero
Mientras no final de fichero 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)

▪ 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
```

▪ 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 olvidarnos de cerrarlos.

Como hemos visto en el ejemplo anterior;

CÓDIGO:

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

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

Vamos a estudiar la forma de utilizar un gestor de bases de datos orientado a objetos combinado con el lenguaje de programación Java. Debemos conocer que, no existe una gran variedad de estos sistemas (SGBDOO). Algunos de ellos son: db4o, Objectivity/ DB o EyeDB entre otros.

Hay una versión de db4o disponible para la plataforma .NET que podemos descargar desde la web oficial, pero NO es compatible con la versión Visual Studio 2012.

1. Diseño de programas con lenguajes de POO para gestionar las bases de datos relacionales.

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 se encuentran relacionadas entre ellas.

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 de las bases de datos 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 que nos permiten gestionar los datos.

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 una dato a otro sin ningún tipo de problema, ya que sus tablas están relacionada. 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.

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.

Las características de la base de datos relacional es que una misma base de datos 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 debe de 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.

- Lo primero que debemos introducir 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.

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

- Una vez importadas las distintas librerías, 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, 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 declarada en la base de datos.

```
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 sólo nos falta diseñar la función para conectarnos a la base de datos. En dicha función, realizaremos los siguientes pasos:
 - Registramos el driver mediante la función `forName`.
 - Creamos la conexión a la base de datos haciendo uso de la operación `getConnection`, que nos facilita la conexión `DriverManager` a la que se le pasa por parámetro la dirección de la base de datos, su usuario y su contraseña.

```
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 a la BBDD
        msjErr.mensajeError("Se ha producido un error al conectar con la Base de Datos.\n"+
            errores.toString());
    }

    return conec;
}
```

Hemos visto, todos los pasos que tenemos que llevar a cabo para poder establecer la conexión a una base de datos, por tanto, nos queda por 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.

```
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
        msjErr.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.

```
//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 que vamos a insertar en la tabla mediante
//la query de inserción
datos = "\"" + formateaFecha(this.txtFecha.getText()) + "\", ";
datos = datos + this.txtN1.getText() + ", ";
datos = datos + this.txtN2.getText() + ", ";
datos = datos + this.txtN3.getText() + ", ";
datos = datos + this.txtN4.getText() + ", ";
datos = datos + this.txtN5.getText() + ", ";
datos = datos + this.txtComplementario.getText() + ")";

//Montamos la consulta completa
consultaInsercion = consultaInsercion + datos;

//Es necesario controlar las excepciones al interactuar con la BBDD
try
{
    //Preparamos la sentencia, ejecutamos y cerramos
    Statement consulta = con.createStatement();
    consulta.executeUpdate(consultaInsercion);
    conectado.cerrarConexion(consulta);
}
```

2. Diseño de programas con lenguajes de POO para gestionar las 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.

En el caso de las bases de datos objeto-relacionales, las conexiones, la recuperación y la manipulación de la información, las vamos a llevar a cabo de la misma forma, ya que no cambia el código en Java, sino 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, y en este apartado vamos a ver la consulta de la información almacenada en tablas.

Para ello, primero creamos la sentencia de conexión (*Statement*), después obtenemos en *resultSet* los datos de la consulta correspondiente, que la lanzamos mediante el método *executeQuery*. El siguiente paso debe ser tratar la consulta que hemos realizado según las indicaciones del enunciado, y, por último, cerraremos la conexión, ya que, previamente la habíamos abierto.

```
// 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 salidaMostrar="<HTML><Body>";

    //Creamos la sentencia
    Statement consulta = con.createStatement();

    //Obtenemos el resultSet con los datos de la consulta
    ResultSet salida = consulta.executeQuery("SELECT * FROM cursos");

    //Iteramos mientras tengamos registros en el resultSet
    while(salida.next())
    {
        //Preparamos y formateamos los datos que vamos a mostrar en la label.
        salidaMostrar = salidaMostrar + "Cursada N°: " + salida.getInt("cursada") + "; Fecha: " + desFormateaFecha(salida.getDate("fecha"))
        + "; Continuar: " + salida.getInt("c1") + ", " + salida.getInt("c2") + ", " + salida.getInt("c3") + ", "
        + salida.getInt("c4") + ", " + salida.getInt("c5") + ", " + 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, van 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, entre las que se encuentran, entre otras, 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, aunque 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, las relaciones de las bases de datos orientadas a objetos deben incorporar en las relaciones de cada objeto, 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 éste 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-evaluados
- 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.
REPASAR
- **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 empezaron 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 3 categorías:

- **Diagramas de estructuras (parte estática)**

Define los elementos que deben existir en un sistema de modelado.

- Diagrama de clases

- Diagrama de estructuras compuestas
- Diagrama de objetos
- Diagrama de componentes
- Diagrama de implementación (despliegue)
- Diagrama de paquetes

○ Diagramas de comportamiento

Basados en lo que debe suceder en el sistema.

- Diagrama de interacción:
 - Diagrama de secuencia
 - Diagrama resumen de interacción
 - Diagrama de comunicación
 - Diagrama de tiempo
- Diagrama de actividad
- Diagrama de casos de uso
- Diagrama de máquina de estado

3.4 El modelo estándar ODMG.

El **estándar ODMG (Object Data Management Group)**, es utilizado para definir el grupo de empresas que han formado parte del desarrollo de un nuevo modelo de objetos.

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

Su función es definir los elementos y la persistencia entre ellos en las BDOO, permitiendo que sean portables entre los sistemas de POO que los soportan.

Definen así un estándar en los SGBD Orientados a Objetos.

Los componentes básicos de una BDOO, como ya se explicó en el ejercicio 3, son los objetos y los literales, los objetos igualmente vienen de una clase, que define su estado y comportamiento, y que, además, ofrece operaciones para poder acceder a estos datos.

La definición de una base de datos está contenida en un esquema creado mediante el lenguaje ODL, lenguaje empleado para el establecer el manejo de datos.

Se pueden vincular a una base de datos por medio de programas orientados a objetos.

3.4.1 Lenguaje de definición de objetos ODL.

Es un lenguaje de especificación para definir tipos de objetos para sistemas compatibles con ODBMS. 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 éstos poseen.

Como se puede observar en la definición ODL, es un lenguaje de definición, con la que crearías los tipos de objetos, relaciones....

OQL por el contrario es el lenguaje de consulta de objetos, con el que vamos a poder consultar los valores de esos objetos creados y la estructura de la base de datos y 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 de gestor de bases de datos orientadas a objetos. Podríamos definirla 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 ser un sistema gestor de bases de datos y un sistema orientado a objetos, combinándolos.

Todo sistema de gestor de bases de datos orientadas a objetos debe tener unas características que se pueden agrupar en dos grupos.

- **Características obligatorias:** son las esenciales y debe tener por un lado un sistema gestor de bases de datos y por otro un sistema orientado a objetos. Para el SGBD debe tener: Persistencia, Gestión del almacenamiento secundario, Concurrencia, Recuperación y Facilidad de consultas. Para el sistema orientado a objetos debe tener: Objetos complejos, Identidad de objetos, Encapsulamiento, Tipos y Clases, Herencia, Sobrecarga, Extensibilidad y Completitud computacional.

- **Características optativas:** son características que debería implementar pero que no está obligado. Como Herencia múltiple, Chequeo e inferencia de tipos, Distribución, Transacciones de Diseño, Versiones.

Bibliografía

Isabel M^a Jiménez Cumbreñas, “Programación”. Garceta, 2013.

José R. García-Bermejo, “JAVA SE6 & Swing. El lenguaje más versátil”. PEARSON Prentice Hall, 2007.

Webgrafía

<https://docs.oracle.com>

ILERNA

Online

```
5 function updatePhotoDescription() {  
6     if (descriptions.length > (page * 9) + (currentImage.substring(0, 1) - 1)) {  
7         document.getElementById("bigImageDesc").innerHTML = descriptions[page * 9 +  
8     }  
9 }  
10  
11 function updateAllImages() {  
12     var i = 1;  
13     while (i < 10) {  
14         var elementId = "foto" + i;  
15         var elementIdBig = "bigImage" + i;  
16         if (page * 9 + i - 1 < photos.length) {  
17             document.getElementById(elementId).src = "images/" + photos[page * 9 + i - 1] + ".jpg";  
18             document.getElementById(elementIdBig).src = "images/" + photos[page * 9 + i - 1] + ".jpg";  
19         } else {  
20             document.getElementById(elementId).src = "images/default.jpg";  
21         }  
22         i++;  
23     }  
24 }
```