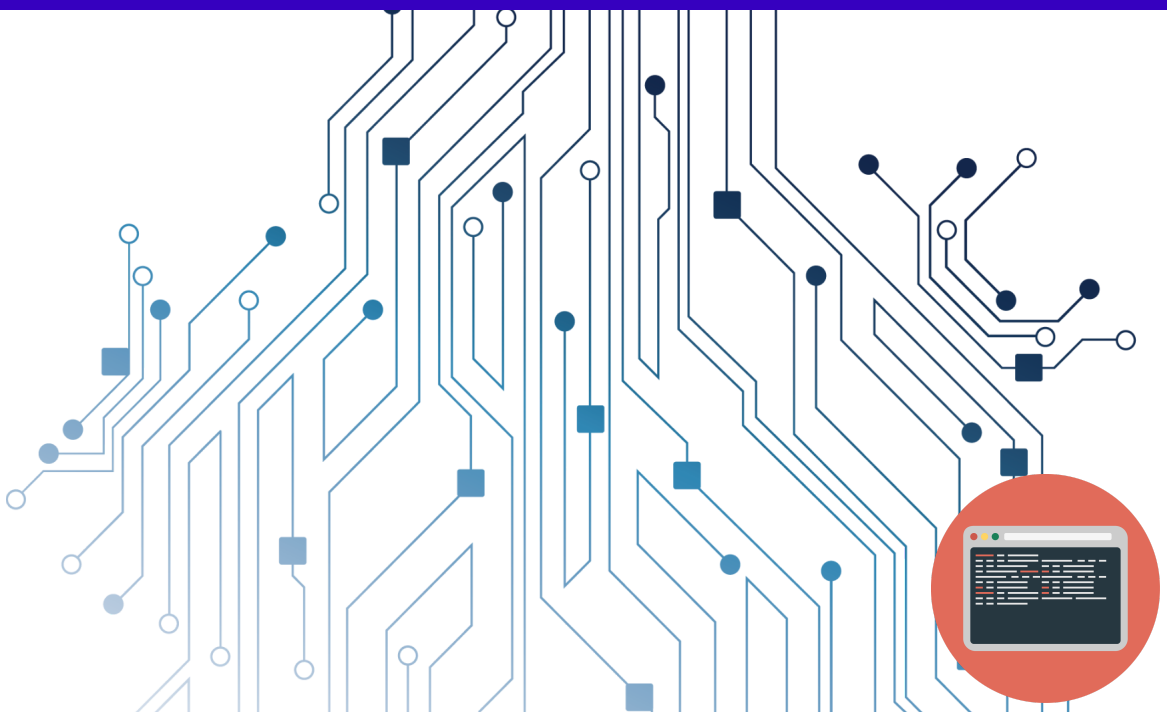
CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN

ANEXO

IMPLEMENTACIÓN Y USO DE MÉTODOS Y CLASES RECOMENDADAS



DIEGO VALERO ARÉVALO
BASADO EN APUNTES Y EJERCICIOS PROPORCIONADOS POR
Mª CARMEN DÍAZ GONZÁLEZ - IES VIRGEN DE LA PALOMA

ANEXO - **IMPLEMENTACIÓN Y USO DE MÉTODOS Y CLASES RECOMENDADAS**
ÍNDICE

Este anexo contiene **temario adicional** al resto de unidades, por lo que, a menos que se puntualice en alguna unidad, no se recomienda consultarlo hasta haber completado la **unidad 7, *Uso avanzado de clases***, ya que contiene terminología y sintaxis que probablemente aún no se hayan estudiado.

IMPLEMENTACIÓN Y USO DE MÉTODOS Y CLASES RECOMENDADAS

UNIDAD 1: INTRODUCCIÓN A LA PROGRAMACIÓN

>> CLASE `MATH`

Como hemos visto, hay muchos y diferentes **operadores** para hacer **diferentes cálculos en JAVA**, pero hay muchos que no tenemos a mano, como **potencias**, **raíces cuadradas**, **valores absolutos**, **seno**, **coseno**,...

Para solucionarlo, JAVA trae consigo una clase con métodos de este tipo y más: la clase **`Math`**. Estos son algunos de los métodos más empleados:

`.abs(n)`	Devuelve el valor absoluto de **`n`**.
`.max(a, b)`	Devuelve el valor más alto entre **`a`** y **`b`**.
`.min(a, b)`	Devuelve el valor más bajo entre **`a`** y **`b`**.
`.pow(n, p)`	Devuelve **`n`** elevado a la potencia **`p`**.
`.random()`	Devuelve un valor de **tipo double** aleatorio con signo positivo, mayor o igual a `0.0` y menor a `1.0`.
`.round(n)`	Devuelve **`n`** redondeado hacia arriba o hacia abajo según la parte decimal.
`.ceil(n)`	Devuelve **`n`** redondeado hacia arriba, sin contar la parte decimal.
`.floor(n)`	Devuelve **`n`** redondeado hacia abajo, sin contar la parte decimal.
`.sqrt(n)`	Devuelve la raíz cuadrada de **`n`**.

Para más información sobre los métodos disponibles en **`Math`**, puedes consultar la API:

MATH (JAVA PLATFORM SE 8) >

· NÚMEROS ALEATORIOS SEGÚN UN RANGO (USO DE `MATH.RANDOM()`)

Para crear un número aleatorio necesitamos utilizar el método **`Math.random()`** que proporciona un **número pseudoaleatorio de tipo double** entre `0.0` y `1.0`. Añadiendo algunas **operaciones matemáticas** podemos obtener un número entero aleatorio en el rango que nos interese:

`(Math.random() * (n + 1))`	Devuelve un número aleatorio entre `0` y **`n`**.
`(Math.random() * n + 1)`	Devuelve un número aleatorio entre `1` y **`n`**.
`(min + Math.random() * (max - min + 1))`	Devuelve un número aleatorio entre **`min`** y **`max`**.

>> CONVERSIÓN DE TIPOS: CASTING IMPLÍCITO Y EXPLÍCITO

Como hemos visto, Java es un lenguaje **tipado**, lo que significa que es bastante estricto a la hora de **asignar valores a las variables**. A priori, el compilador **sólo admite asignar a una variable un dato del tipo declarado en la variable**, no obstante, en ciertas circunstancias, es posible **realizar conversiones de tipo** que permitan almacenar en una variable un dato de tipo diferente al declarado. A estas conversiones se les llama **casting**.

Es posible realizar castings entre **todos los tipos básicos**, excepto `boolean`, que es **incompatible** con el resto de tipos. Los castings pueden ser de **dos tipos**:

IMPLÍCITOS	EXPLÍCITOS

· CASTING IMPLÍCITO

Este se realiza de forma **automática**, es decir, el valor o expresión que va a asignar a la variable es **convertido automáticamente** al tipo de ésta por el **compilador**, antes de almacenarlo en la variable.

```
int i;
byte b = 10;
i = b;
```

En este ejemplo, el dato de tipo `byte` almacenado en la variable `b` es convertido a `int` antes de asignarse a la variable `b`.

Para que una conversión pueda realizarse de **forma automática o implícitamente**, el tipo de la variable destino debe ser **de tamaño igual o superior al tipo de origen**, aunque hay **dos excepciones**:

- Cuando la variable destino es **entera** y el origen es **decimal** (`float` o `double`), el casting **no podrá ser automático**.
- Cuando la variable destino es `char` y el origen es **numérico**, el casting **no podrá ser automático**.

En estas excepciones, el casting deberá ser **explícito**.

```
int variableInt1 = 5;
int variableInt2;
short variableShort = 10;
char variableChar = 'ñ';
float variableFloat;

variableInt2 = variableChar; //CI correcta de char a int
variableFloat = variableInt1; //CI correcta de int a float
variableInt1 = variableShort; //CI correcta de short a int
```

```
int variableInt;
long variableLong = 20;
float variableFloat = 2.4f;
char variableChar;
byte variableByte = 4;

variableInt = variableLong; //CI errónea de long a int
variableChar = variableByte; //CI errónea de byte a char
variableInt = variableFloat; //CI errónea de float a int
```

· CASTING EXPLÍCITO

Cuando **no se cumplan las condiciones para una conversión implícita**, esta **podrá realizarse** a través de la siguiente expresión:

```
varDestino = (tipoDestino) datoOrigen;
```

`varDestino`	La **variable** donde **guardaremos el casting explícito**.
`(tipoDestino)`	El **tipo** al que **vamos a convertir el dato de origen**, que será el mismo tipo que el de `varDestino`.
`datoOrigen`	El **dato** que **queremos convertir**.

En resumen, le decimos al compilador que convierta `datoOrigen` al `tipoDestino` para después de convertirlo poder almacenarlo en la `varDestino`.

TEN EN CUENTA

Al convertir un dato de un tipo en otro **de tamaño inferior** se realiza un **estrechamiento** que, en algunos casos, puede provocar **la pérdida de datos**, aunque ello **no provocará errores de ejecución**.

```
double variableDouble = 34.6;
int variableInt = 400;
char variableChar;
byte variableByte;

variableChar = (char) variableDouble; //Se trunca la parte decimal
variableByte = (byte) variableInt; //Se pierden datos pero se castea
```


>> CONVERSIÓN DE TIPOS: `PARSE` Y `VALUEOF()`

Otra manera de **convertir**, o mejor dicho, **extraer unos datos** y **convertirlos a otros** es con el uso de los métodos `parse()` y `valueOf()`. Estos se usan sobre todo cuando queremos transformar datos de tipo `String` a **tipos de datos simples**. Veámoslos:

· PARSE

Para usar los métodos `parse` debemos seguir la siguiente expresión:

```
varDestino = wrapperDestino.parseTipoDestino(String);
```

`varDestino`	La **variable** donde **guardaremos los datos del parsing**.
`wrapperDestino`	Para poder usar un **tipo de parser**, debemos llamar al **wrapper del tipo simple que queremos usar** (de `int` es `Integer`, de `double` es `Double`, ...). Debe ser del **tipo de la variable** donde estamos asignando la conversión.
`parseTipoDestino`	Dependiendo del **tipo de dato a guardar y del wrapper**, el **nombre del parser cambiará**, aunque **suele coincidir con el del tipo**.
`(String)`	El `String` que queremos **parsear**.

En este curso, al usar métodos **`parse`** lo soleremos usar con **`Strings`** que queremos pasar a **tipos de datos numéricos**.

EJEMPLO

Parseador.java

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

      String miString = "12";
      int variableInt;
      double variableDouble;

      variableInt = Integer.parseInt(miString);
      variableDouble = Double.parseDouble(miString);

      System.out.println(variableInt / 2);
      System.out.println(variableDouble);
   }
}
```

Console

```
6
12.0
```

Los métodos **`parse`** se pueden usar para convertir **`Strings`** a **prácticamente todos los datos usados en JAVA**. Sólo hay que saber cuál usar en cada momento.

· VALUEOF()

Parecido al **casting explícito**, podemos usar el método **`valueOf()`** para convertir unos tipos de datos a otros. Su sintaxis es muy parecida a los **`parse`**.

```java
varDestino = wrapperDestino.valueOf(dato);
```

`varDestino`	La **variable** donde **guardaremos los datos extraídos**.
`wrapperDestino`	Para poder extraer los datos, debemos llamar al **wrapper del tipo que queremos usar** (de **`int`** es **`Integer`**, de **`double`** es **`Double`**, e incluso podemos usar **`String`**). Debe ser del **tipo de la variable** donde estamos asignando la conversión.
`valueOf`	El método **`valueOf`** se encarga de coger el dato pasado por parámetro y convertirlo automáticamente.
`(dato)`	El **dato** que queremos **convertir**.

EJEMPLO

UsoValueOf.java

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

      String miString = "12";
      String miStringConvertido;
      int variableInt;

      variableInt = Integer.valueOf(miString);
```

```java
        miStringConvertido = String.valueOf(variableInt);

        System.out.println(variableInt / 2);
        System.out.println(miStringConvertido);
    }
}
```

Console

```
6
12
```


>> MOSTRAR EL CONTENIDO DE UN ARRAY: `DEEPTOSTRING()`

¡Recuerda que JAVA tiene **métodos** para casi todo! Si no quieres molestarte recorriendo un array para mostrar sus elementos, existe un método **que lo muestra directamente**:

`Arrays.deepToString(array);`	Devuelve un **`String`** en el que los elementos están dentro de corchetes (`[]`) y separados por comas (`,`).

EJEMPLO

Nombres.java

```java
import java.util.Arrays;

public class Nombres{

    private String[] listaNombres;

    public Listado(String[] listaNombres){
        this.listaNombres = listaNombres;
    }

    @Override
    public String toString(){
        return Arrays.deepToString(listaNombres);
    }
}
```

Main.java

```java
public class Main{

    public static void main(String[] args) {

        String[] array = new String[3];

        array[0] = "Juan";
        array[1] = "Ana";
        array[2] = "Eva";

        Nombres listaNombres = new Nombres(array);

        System.out.println(listaNombres.toString());
    }
}
```

Console

```
[Juan, Ana, Eva]
```

>> TOSTRING()

Al declarar nuestra **propia construcción de un método que ya está implementado** en otra clase, este se declara como `Override` (sobreescritura) automáticamente.

El método `toString()` es un método que devuelve un `String`. Este método es útil para devolver en forma de **línea por consola la información de un objeto**. Sustituye al método `print()` para poder trabajar directamente con un `String`.

Este es el contenido por defecto cuando sobreescribimos el método `toString()`:

```java
@Override
public String toString() {
	// TODO Auto-generated method stub
	return super.toString();
}
```

EJEMPLO

Persona.java

```java
@Override
public String toString() {
	return "DNI: "+getNombre()
		+", Nombre: "+getNombre();
}
```

PersonaMain.java

```java
System.out.println(persona.toString());
```

Console

```
DNI: 12345678X, Nombre: Diego
```

Podemos imprimir directamente lo que queramos de cada clase implementando el método `toString()`.

>> EQUALS()

Para poder hacer uso de muchos métodos de cualquier **colección** en general, es necesario que el objeto del que creemos el array **tenga implementado el método** `equals()`, que pertenece a la clase `Object`. Este es el contenido por defecto cuando lo sobreescribimos:

```java
@Override
public boolean equals(Object obj) {
	// TODO Auto-generated method stub
	return super.equals(obj);
}
```

Dentro del método podemos borrar esas líneas y establecer **nuestras propias instrucciones** para **comparar lo que queramos del objeto que contiene el método con otro que le pasemos** por parámetro, para ver si son iguales o no. Este método devuelve siempre un tipo `boolean`.

>> INTERFAZ COMPARABLE Y MÉTODO COMPARETO()

El método `compareTo()` se utiliza para **comparar** el propio objeto con otro del mismo tipo. Se suele implementar, por ejemplo, para poder usar los **métodos de ordenación** en un `array` de **objetos**. Este método **se implementa** desde la interfaz `Comparable<T>` y devuelve un tipo `int` que define si el primer elemento es, según el alfabeto ASCII:

+	-	0
Mayor	Menor	Igual

```java
public class nombreClase implements Comparable<tipo>{
    @Override
    public int compareTo(Object otroObjeto) {
        ...
    }
}
```

Dentro del método **somos nosotros los que especificamos las reglas que se deben cumplir** para devolver cada uno de los valores y luego trabajar con ellos, por ejemplo **comparar dos de los atributos de los objetos (el del propio objeto y el de otro que le pasemos)**.

```java
public class nombreClase implements Comparable<tipo>{
    @Override
    public int compareTo(Object obj2) {
        return getAtributo().compareTo(obj2.getAtributo());
    }
}
```

Vamos a ver un ejemplo. Veamos qué pasa si hacemos un listado con objetos y luego los intentamos ordenar sin implementar el `compareTo()`.

EJEMPLO

Cliente.java

```java
public class Cliente{

    private String nombre;

    public Cliente(String nombre){
        this.nombre = nombre;
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(){
        this.nombre = nombre;
    }

    @Override
    public void toString(){
        return nombre;
    }
}
```

Listado.java

```java
public class Listado{

    private Cliente[] listaClientes;

    public Listado(Cliente[] lista){
        listaClientes = lista;
    }

    public void ordenar(){
        Arrays.sort(listaClientes);
    }

```

```java
    @Override
    public String toString(){
        String imprimir = "";
            for (int i = 0; i < listaClientes.length; i++) {
                imprimir += listaClientes[i].getNombre() + ", ";
            }
        return imprimir;
    }
}
```

Main.java

```java
public class Main{

    public static void main(String[] args) {

        Cliente[] array = new Cliente[3];
        array[0] = new Cliente("Juan");
        array[1] = new Cliente("Ana");
        array[2] = new Cliente("Eva");

        Listado cont = new Listado(array);

        cont.ordenar();

        System.out.println(cont.toString());
    }
}
```

Console

```
Exception in thread "main" java.lang.ClassCastException: class
myLibrary.ejemploImplementacionCompare.Cliente cannot be cast to class java.lang.Comparable
(myLibrary.ejemploImplementacionCompare.Cliente is in unnamed module of loader 'app';
java.lang.Comparable is in module java.base of loader 'bootstrap')
        at
java.base/java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort.java:320)
        at java.base/java.util.ComparableTimSort.sort(ComparableTimSort.java:188)
        at java.base/java.util.Arrays.sort(Arrays.java:1041)
        at myLibrary.ejemploImplementacionCompare.Listado.ordenar(Listado.java:13)
        at myLibrary.ejemploImplementacionCompare.Main.main(Main.java:16)
```

Como ves, nos saltan errores que tienen que ver con la clase **`Comparable`**, ya que es la que usa el método **`.sort()`** para poder ordenar los elementos.

Volvamos a hacer este ejemplo, pero implementando el método **`compareTo()`**. Vamos a hacer que compare los atributos **`nombre`** de cada objeto de tipo **`Cliente`** para que pueda ordenarlos alfabéticamente.

EJEMPLO

Listado.java

```java
public class Cliente implements Comparable<Cliente>{

    private String nombre;

    public Cliente(String nombre){
        this.nombre = nombre;
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(){
        this.nombre = nombre;
    }

    @Override
```

```
18      public void toString(){
19          return nombre;
20      }
21
22      @Override
23      public int compareTo(Cliente otroCliente){
24          return getNombre().compareTo(otroCliente.getNombre());
25      }
26  }
```

Console

```
Ana, Eva, Juan,
```

Al implementar **`Comparable`** y su método **`compareTo()`**, y ejecutar de nuevo el **`main`**, ahora vemos que sabe **qué debe comparar** para poder ordenar los objetos.

>> INTERFAZ COMPARATOR Y MÉTODO COMPARE ()

Un objeto que implementa la **interfaz** **`Comparator<T>`** tiene dentro de sí mismo el **método** **`compare()`**, el cual tiene las instrucciones definidas por el usuario para saber cuándo uno es:

+	-	0
Mayor	Menor	Igual

Estos objetos se crean para pasarlos como parámetro en un método de ordenación para que el programa sepa cómo ordenar el contenido sin importar su tipo de dato.

COMPARATOR (objeto que se crea para comparar dos objetos, el cual podemos usar en cualquier parte)
Compara dos objetos distintos

EJEMPLO

MiComparador.java

```
1   public class MiComparador implements Comparator<String>{
2
3       @Override
4       public int compare(String o1, String o2){
5           int index;
6           if(o1.compareTo(o2)>0) {
7               index= -1;
8           }
9           else if(o1.compareTo(o2)<0) {
10              index= 1;
11          }
12          else {
13              index= 0;
14          }
15          return index;
16      }
17  }
```

Listado.java

```
1   public class Listado{
2
3       private Cliente[] listaClientes;
4
5       public Listado(Cliente[] lista){
6           listaClientes = lista;
7       }
```

```java
    public void ordenar(){
        Arrays.sort(listaClientes);
    }

    @Override
    public String toString(){
        String imprimir = "";
            for (int i = 0; i < listaClientes.length; i++) {
                imprimir += listaClientes[i].getNombre() + ", ";
            }
        return imprimir;
    }
}
```

Main.java

```java
public class Main{

    public static void main(String[] args) {

        Cliente[] array = new Cliente[3];
        array[0] = new Cliente("Juan");
        array[1] = new Cliente("Ana");
        array[2] = new Cliente("Eva");

        Listado cont = new Listado(array);

        cont.ordenar();

        System.out.println(cont.toString());
    }
}
```

Console

```
Exception in thread "main" java.lang.ClassCastException: class
myLibrary.ejemploImplementacionCompare.Cliente cannot be cast to class java.lang.Comparable
(myLibrary.ejemploImplementacionCompare.Cliente is in unnamed module of loader 'app';
java.lang.Comparable is in module java.base of loader 'bootstrap')
        at
java.base/java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort.java:320)
        at java.base/java.util.ComparableTimSort.sort(ComparableTimSort.java:188)
        at java.base/java.util.Arrays.sort(Arrays.java:1041)
        at myLibrary.ejemploImplementacionCompare.Listado.ordenar(Listado.java:13)
        at myLibrary.ejemploImplementacionCompare.Main.main(Main.java:16)
```

Como ves, nos saltan errores que tienen que ver con la clase **`Comparable`**, ya que es la que usa el método **`.sort()`** para poder ordenar los elementos.

compare() -> Compara dos objetos pasados por parámetro. Dentro del método nosotros decidimos qué es lo que va a determinar el +, - ó 0.

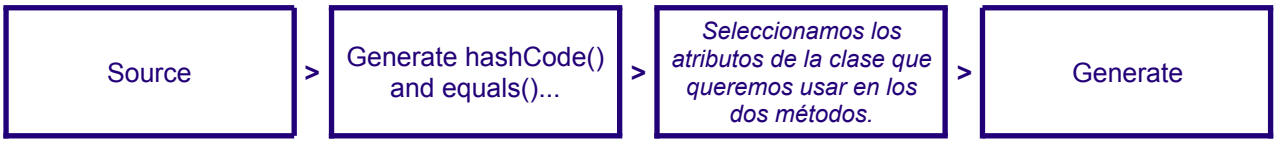
Es útil para **`Arrays.sort(miArray, miComparador)`**, ya que pasamos al método **`.sort`** la manera que debe utilizar para ordenar el array, guardada en el comparador que hemos creado.

>> COMPARAR OBJETOS DE UNA COLECCIÓN: HASHCODE()

Ahora, ¿qué ocurre si queremos **comparar los objetos que forman parte de una colección para saber si son iguales**? Podríamos usar **el método `equals()` y comparar dentro uno o varios atributos de cada objeto**, pero imagina un objeto con más de 10 atributos… ¿Vamos a dar condiciones para todos? No, es tedioso y poco eficiente. En este caso usaremos el método **`hashCode()`**, el cual lleva de la mano al método **`equals()`**.

Este método devuelve un **código único de tipo `int` de 32 bits** que tiene cada objeto. Se usa para las **estructuras de tipo `Hash`: `HashMap`**, **`HashSet`**,... y se debe **implementar** antes de usar el método **`equals()`** especificando **los atributos que vamos a comparar**. Si dos de ellos **comparten el mismo hashcode**, el programa **sabrá que son iguales**.

Una manera de **generar estas condiciones** es con la ayuda de Eclipse a través de:



EJEMPLO

Usuario.java

```java
import java.util.Objects;

public class Usuario{

    private int id;
    private String nombre;
    private String email;

    public Usuario(int id,  String nombre, String email){
        this.id = id;
        this.nombre = nombre;
        this.email = email;
    }

    @Override
    public int hashCode(){
        return Objects.hash(id, nombre, email);
    }

    @Override
    public String equals(Object obj){
        if(this == obj)
            return true;

        if(obj == null)
            return false;

        if(getClass() != obj.getClass())
            return false;

        Usuario other = (Usuario) obj;

        return Objects.equals(email, other.email) && id == other.id &&
               Objects.equals(nombre, other.nombre);
    }
}
```

Main.java

```java
public class Main{

    public static void main(String[] args) {

        Usuario usuario1 = new Usuario(123, "Juan", "miemail");
        Usuario usuario2 = new Usuario(123, "Juan", "miemail");
        Usuario usuario3 = new Usuario(234, "Alberto", "miemail");

        System.out.println(usuario1.equals(usuario2));
        System.out.println(usuario1.equals(usuario3));
    }
}
```

Console

```
true
false
```


>> SOBREESCRIBIR MÉTODOS DE UNA CLASE EN EL `MAIN`

Existe una manera con la cual podemos **implementar métodos fuera de su clase directamente al trabajar con ellos** en el `main`. Se suele usar con métodos abstractos pero la manera es válida con cualquier método, ya que emplea la **sobreescritura**.

Para ello simplemente debemos **abrir unas llaves** (`{}`) justo después de **declarar el objeto** y **hacer una sobreescritura del método** que queramos cambiar como haríamos en cualquier clase. Incluso podemos **implementar el método directamente con la clase abstracta**.

EJEMPLO

Animal.java

```java
public abstract class Animal{
    public abstract void emiteSonido();
}
```

Gato.java

```java
public class Gato extends Animal{
    @Override
    public void emiteSonido(){
        System.out.println("Miau");
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Gato gato = new Gato(){
            @Override
            public void emiteSonido(){
                System.out.println("Marramiau");
            }
        };

        Animal animal = new Animal(){
            @Override
            public void emiteSonido(){
                System.out.println("Roar");
            }
        };

        gato.emiteSonido();
        animal.emiteSonido();

    }
}
```

Console

```
Marramiau
Roar
```

Al haber **sobreescrito el método en la instanciación** del objeto, llamarlo en este **ámbito** implica que **usará la nueva implementación**. También hemos implementado directamente con la clase abstracta `Animal` su método abstracto en el `main`.

Este método **no es recomendado** y se usa solo en **casos puntuales**, pero es interesante saber que contamos con esta opción.

TEN EN CUENTA

El sobreescribir un método en un ámbito fuera de su clase hace que sólo afecte a **ese ámbito en concreto**. Usar el mismo método en otro sitio no traerá consigo la sobreescritura.