FAMILIA PROFESIONAL:	**Informática y Comunicaciones**
CICLOS FORMATIVOS:	**Desarrollo de Aplicaciones Multiplataforma, Desarrollo de Aplicaciones Web**
MÓDULO:	**Programación**

UNIDAD 3: DISEÑO DE CLASES Y ENCAPSULACIÓN

CONTENIDOS

AUTORES: **Fernando Rodríguez Alonso**
Sonia Pasamar Franco

ÍNDICE DE CONTENIDOS

1. ORIENTACIÓN A OBJETOS

La **programación orientada a objetos** es un paradigma de programación que organiza el diseño de programas en torno a objetos, en lugar de datos y procesos. Los objetos se utilizan como metáfora para representar entidades reales o abstractas del mundo real a modelar en un programa.

1.1. OBJETOS

Un **objeto** es una instancia, ocurrencia o ejemplar de una entidad de interés del mundo real.

1.1.1. Características de un Objeto

Todo objeto tiene tres **características** fundamentales:

1) **Identidad.** Permite diferenciar o distinguir un objeto de otros mediante identificadores específicos. Cada objeto tiene un identificador único, de forma que no puede haber dos o más objetos con el mismo identificador. Se utiliza para realizar comparaciones de igualdad entre dos objetos.

   En un programa, por defecto, la identidad de un objeto viene determinada por la referencia o dirección de memoria donde se encuentra el objeto. Sin embargo, el programador puede utilizar otra propiedad (como un número entero o una cadena de caracteres) para redefinir la identidad del objeto a conveniencia.

2) **Estado.** Se define mediante los valores que toma el conjunto de propiedades de un objeto en un instante de tiempo dado. Cada propiedad de un objeto puede ser:

   - **Atributo.** Puede ser una variable o constante de un tipo primitivo del lenguaje de programación o un objeto construido con un tipo de objetos. Toma un literal como valor. Normalmente, cada atributo se define privado.
   - **Relación.** Se define entre dos tipos de objetos, no entre instancias. Una relación puede ser uno a uno (1:1), uno a muchos (1:N) o muchos a muchos (N:M).

3) **Comportamiento.** Está compuesto por el conjunto de acciones u operaciones que un objeto puede realizar o a las que puede responder ante mensajes enviados por otros objetos. Representa las funcionalidades o requisitos de un objeto. Hay dos tipos de operaciones:

   - **Operación de Acceso a Atributo.** Se utiliza para consultar o modificar de forma separada el valor de un atributo de un objeto. Si se modifica un atributo, el estado del objeto cambia.
   - **Operación de Comportamiento.** Realiza una acción u operación que está relacionada con una funcionalidad requerida para un objeto. Representa la transición de un estado a otro mediante la consulta o modificación de los atributos de un objeto de forma conjunta.

   En programación orientada a objetos, cada operación se denomina método y se define pública.

1.1.2. Tipos de Objetos

Los objetos se dividen, clasifican y jerarquizan según diferentes **tipos de objetos**:

- **Tipo de Objetos Simple o Escalar.** Está asociado a un tipo de datos primitivo del lenguaje de programación y añade operaciones relacionadas.

- **Tipo de Objetos Estructurado.** Está asociado a un tipo de datos estructurado (compuesto por varios datos), que está incluido en el lenguaje de programación o está definido explícitamente por el programador.
- **Tipo de Objetos Colección.** Es un tipo particular de tipo de objetos estructurado y representa una colección, contenedor o grupo de objetos del mismo tipo de objetos. Varios ejemplos de este tipo son la lista, el conjunto y el mapa. Estos tipos de objetos son genéricos, y por tanto, se pueden usar para definir colecciones de objetos más específicas.

El estado y el comportamiento de un tipo de objetos se especifica mediante una **clase** o un **prototipo**. A partir de este mecanismo, se pueden construir objetos individuales. Los objetos creados a partir de un mismo tipo de objetos tienen el mismo comportamiento y muestran un rango de estados común.

Un tipo de objetos tiene una interfaz y una o más implementaciones:

- La **interfaz** define los atributos visibles externamente y las operaciones soportadas para todas las instancias del tipo de objetos.
- La **implementación** define la representación física del estado (atributos) de las instancias del tipo de objetos y los métodos que implementan las operaciones definidas en la interfaz.

1.1.3. Interacción entre Objetos

La interacción entre objetos de un programa se realiza mediante mensajes. Un **mensaje** describe la manera de acceder a los atributos y métodos de un objeto y sigue la siguiente sintaxis en la mayoría de lenguajes de programación orientados a objetos:

```
objeto.método(parámetros)
```

Tras definir un tipo de objetos e instanciar un objeto con ese tipo de objetos, para interactuar con este objeto y modificar su estado (modificar uno o varios de sus atributos), se debe provocar la ejecución de los métodos públicos que dicho objeto ofrece:

- Desde el punto de vista del **objeto emisor**, el hecho de provocar la ejecución de un método de otro objeto se llama “realizar una petición”, “solicitar un servicio” o “enviar un mensaje” al objeto.
- Desde el punto de vista del **objeto receptor**, el hecho de recibir una solicitud para que ejecute uno de sus métodos se llama “recibir una petición”, “recibir una solicitud de servicio” o “recibir un mensaje” desde otro objeto.

1.2. TIPOS DE ORIENTACIÓN A OBJETOS

Existen dos tipos de orientación a objetos dependiendo del mecanismo de construcción de objetos:

- **Orientación a Objetos Basada en Clases.** Utiliza la instanciación de objetos a partir de clases como mecanismo de construcción de objetos.

  Una **clase** es una plantilla de código fuente donde se especifican los atributos y métodos que tienen todos los objetos de un tipo específico. La creación de un nuevo objeto se realiza tomando como referencia una clase y se denomina **instanciación**. El objeto creado de esta manera se considera una instancia de dicha clase y hereda los atributos y métodos definidos en la clase.

  Es el tipo de orientación a objetos más utilizado. Varios ejemplos de lenguajes de programación basados en clases son C++, Java y C#.

- **Orientación a Objetos Basada en Prototipos.** Utiliza la clonación de objetos a partir de prototipos como mecanismo de construcción de objetos.

  Un **prototipo** es un objeto genérico ya creado que contiene atributos y métodos. La creación de un nuevo objeto se realiza tomando como referencia un prototipo y se denomina **clonación**. El objeto creado de esta manera se considera un clon de dicho prototipo, hereda los atributos y métodos del prototipo y además se le puede añadir otros nuevos. Se puede utilizar cualquier objeto como prototipo para crear nuevos objetos.

  Este tipo de orientación a objetos sólo permite implementar herencia simple. Varios ejemplos de lenguajes de programación basados en prototipos son Python, JavaScript y Ruby.

1.3. CARACTERÍSTICAS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

Las características más importantes de la programación orientada a objetos son:

- **Abstracción**. Es un aspecto clave en el proceso de análisis y diseño orientado a objetos, permite componer un conjunto de clases para modelar el problema que se quiere resolver. Implica el uso de un diseño descendente que abarca varios niveles de detalle.
- **Modularidad**. Consiste en subdividir un programa en partes más pequeñas y tan independientes como sea posible.
- **Encapsulación**. Se utiliza junto con el principio de ocultación. Consiste en juntar los elementos pertenecientes a una misma entidad en un mismo nivel de abstracción.
- **Herencia**. Es la relación de jerarquía entre clases que favorece la reutilización de código, así como la encapsulación y el polimorfismo.
- **Polimorfismo**. Es un mecanismo por el cual objetos de diferentes clases realizan distintas acciones ante una misma invocación.
- **Recolección de Basura**. Es una técnica automática de liberación de memoria que el programa ya no usa.

2. CLASES Y OBJETOS

Una **clase** es una plantilla a partir de la cual se crean objetos. Dicho de otra forma, una clase es una abstracción de objetos con el mismo conjunto de propiedades y el mismo comportamiento. Representa un tipo de datos que viene dado por el propio lenguaje de programación o que puede ser definido por el programador.

Un **objeto** es una instancia, ocurrencia o ejemplar de una clase dada. Cuando se declara un objeto, la clase indica su tipo de datos. Cuando se instancia un objeto de la clase, se crea el objeto en memoria para que pueda ser utilizado en un programa.

En Java (y en muchos otros lenguajes de programación), el código fuente de una clase se escribe en un único fichero cuyo nombre es el mismo que el de la clase.

2.1. OPERACIONES ESPECIALES

Toda clase dispone de varias operaciones especiales que se utilizan para crear objetos, para obtener los estados de los objetos como cadenas de caracteres y para destruir objetos.

2.1.1. Construcción de un Objeto

Un **constructor** es un método especial de una clase que se invoca de forma automática al crear un objeto para situarlo en memoria e inicializar sus atributos con unos valores concretos.

En la mayoría de lenguajes, el constructor tiene las siguientes características:

- En C++, C# y Java, el nombre del constructor es el mismo que el nombre de la clase. En Python, se denomina `__init__`.
- Puede recibir parámetros con el fin de inicializar los atributos de la clase para el objeto que se está creando en ese momento.
- No tiene tipo de datos de retorno, ni siquiera `void`.
- En general suele ser público, pero algunos lenguajes permiten definirlo privado.

Se denomina **sobrecarga de constructores** cuando una clase incluye más de un constructor. En este caso, se utilizan el número de parámetros y los tipos de datos de los parámetros de los constructores para diferenciarlos. Al constructor sin parámetros se le suele llamar constructor por defecto o predeterminado. A los constructores que tienen argumentos se les llama constructores con parámetros.

Los lenguajes C++, C# y Java permiten definir más de un constructor en una clase. Además, si no se define ningún constructor en la clase, el propio compilador incluirá un **constructor por defecto** (sin parámetros) que se encargará de ubicar el objeto en memoria y asignar valores por defecto a los atributos. Pero si se define un constructor cualquiera (aunque sea con parámetros) en la clase, el compilador no añadirá automáticamente el constructor por defecto.

Si un constructor no establece un valor inicial para un atributo del objeto, el compilador inicializará dicho atributo con el valor por defecto o predeterminado asociado a su tipo de datos:

- El tipo de datos `boolean` tiene el valor por defecto `false`.
- Los tipos de datos numéricos enteros (`byte`, `short`, `int`, `long`) tienen el valor por defecto `0`.
- Los tipos de datos numéricos reales (`float`, `double`) tienen el valor por defecto `0.0`.

- El tipo de datos carácter (`char`) tiene el valor por defecto `'\0'`.
- La referencia a objeto en memoria tiene el valor por defecto `null`.

El constructor siempre se invoca con el operador `new`. Este operador se utiliza para crear un objeto mediante un constructor concreto de la clase y devolver una referencia al objeto creado en memoria.

2.1.2. Representación del Estado de un Objeto en una Cadena de Caracteres

Normalmente, el programador de una clase suele incluir en ella un método especial que se utiliza para **obtener el estado** de un objeto en un instante de tiempo dado.

Este método de obtención del estado devuelve una representación textual del objeto, es decir, un mensaje informativo y conciso que es fácil de leer por cualquier persona.

En general, el estado de un objeto en un momento concreto será una cadena de caracteres con:

- El nombre de la clase con la que se ha instanciado el objeto.
- Una representación textual del conjunto de atributos del objeto, indicando para cada atributo, su nombre y su valor en ese momento específico.

La siguiente tabla muestra varios ejemplos de representación del estado de varios objetos en Java:

`Libro [Título = El Señor de los Anillos, Escritor = J.R.R. Tolkien, Año = 1955, NúmeroPáginas = 1385, Precio = 53.95 euros]`
`Película [Título = Ben-Hur, Director = William Wyler, Año = 1959, Duración = 212 minutos, Nacionalidad = estadounidense]`
`Alumno [Nombre = María González, FechaNacimiento = 12/03/1997, Localidad = Zaragoza, Teléfono = 976112233, Correo = mgonz@gmail.com]`
`Empleado [Nombre = Jorge Martínez, FechaAlta = 30/09/2015, Departamento = Ventas, Salario = 1688.12 euros]`

2.1.3. Destrucción de un Objeto

Un **destructor** es un método especial de una clase que sirve para eliminar definitivamente un objeto concreto de la memoria, liberando el espacio ocupado por los atributos del objeto y la referencia al objeto.

En la mayoría de lenguajes, el destructor tiene las siguientes características:

- El nombre del destructor varía entre lenguajes. En C++ y C#, se nombra con el símbolo ~ y el nombre de la clase. En Python, se denomina `__del__`.
- No recibe parámetros.
- En algunos lenguajes, no tiene tipo de datos de retorno, ni siquiera `void`. En otros, generalmente, tiene `void` como tipo de datos de retorno.
- En general suele ser público.

No todos los lenguajes necesitan implementar un destructor. C++ delega la tarea de gestionar la memoria al programador, y por eso, permite definir un destructor en la clase. Java tiene un recolector de basura que libera espacio de memoria automáticamente, y por tanto, no necesita definir un destructor en la clase.

Otros lenguajes, como C# y Python, permiten definir un destructor en una clase y también incluyen un recolector de basura.

Por norma general, en lenguajes que lo permiten (C++, C#, Python), una clase tiene únicamente un destructor.

El **recolector de basura** (Garbage Collector) de Java es un proceso de baja prioridad que se ejecuta en la máquina virtual de Java (JVM). Su baja prioridad indica que se ejecutará cuando el procesador no tenga en ejecución una tarea con mayor prioridad. Cuando se ejecuta, revisa la memoria y comprueba que todos los espacios o zonas de memoria que han sido reservados con el operador `new` tengan alguna referencia que apunte a ellos. Si no hay ninguna referencia que apunte a un espacio de memoria, lo libera para que vuelva a estar disponible en el programa.

Java realiza esta gestión de la memoria mediante el recolector de basura de forma automática. No obstante, se permite al programador ordenar que se ejecute en un momento determinado con la instrucción: `System.gc();`

2.2. MIEMBROS

Se denominan **miembros de una clase** al conjunto formado por los atributos y los métodos que están definidos dentro de una clase.

Los constructores y el destructor de una clase no se consideran miembros de la clase, ya que ni los constructores ni el destructor se heredan.

2.2.1. Atributos

Un **atributo** (también llamado campo o propiedad) de una clase representa una característica de la clase. Se define mediante una variable o constante de un tipo de datos primitivo del lenguaje de programación o un objeto de otra clase ya existente, la cual puede pertenecer a una biblioteca del lenguaje o estar codificada por el programador.

Los atributos de un objeto indican el espacio de estados disponible para el objeto y los valores de los atributos del objeto en un instante de tiempo dado defiene el **estado** del objeto en ese momento.

2.2.2. Métodos

Un **método** (también llamado subprograma o subrutina) de una clase representa una acción u operación que los objetos de una clase pueden realizar como respuesta a mensajes recibidos de otros objetos. Se define mediante un módulo o subprograma (función o procedimiento).

Los métodos proporcionan una capa de abstracción que facilita la encapsulación y la modularidad.

Los métodos de un objeto representan el **comportamiento** del objeto, es decir, las funcionalidades que el objeto es capar de realizar, a través de la manipulación de los atributos del objeto. Por tanto, puede haber métodos que modifican el estado del objeto y otros métodos que no lo modifican.

Hay varios tipos de métodos:

- **Método Especial de Gestión de Memoria.** Se utiliza para crear objetos en memoria e inicializar los valores de sus atributos o para liberar de memoria los datos de los objetos y sus referencias. Son los constructores y los destructores de una clase.

- **Método de Acceso a Atributo.** Se utiliza para acceder de forma separada al valor de un atributo de un objeto, y a su vez, existen dos tipos según el modo de acceso:
  - *Setter*. Permite acceder en modo escritura a un atributo para modificar su valor. Cuando se usa este método, el estado del objeto cambia.
  - *Getter*. Permite acceder en modo lectura a un atributo para consultar su valor.
- **Método de Comportamiento.** Realiza una acción u operación que está relacionada con una funcionalidad requerida para un objeto. Representa la transición de un estado a otro mediante la consulta o modificación de los atributos de un objeto de forma conjunta.

Se produce **sobrecarga de métodos** cuando una clase incluye dos o más métodos tienen el mismo nombre, pero diferentes números de parámetros y/o diferentes tipos de parámetros de entrada. En este caso, el compilador decidirá qué método invocar comparando los tipos de los parámetros actuales de la llamada con los tipos de los parámetros formales de las cabeceras de los métodos.

2.2.3. Tipos de Miembros

Un objeto es una instancia creada a partir de una clase que puede contener variables, constantes y otros objetos denominados atributos y también puede incluir funciones y procedimientos denominados métodos. El comportamiento del objeto dependerá del estado de la instancia.

Cada objeto tiene su propio **espacio de memoria** reservado para sus atributos y donde se almacenan los valores de estos atributos. Pero para todos los objetos de una clase, solo existe una copia de los métodos con su comportamiento descrito en instrucciones.

Los miembros de una clase se dividen en dos tipos de miembros:

1) **Miembro de Objeto o Instancia.** Cada objeto instanciado en memoria contiene su propia copia del miembro, la cual es exclusivamente suya. Para acceder a un miembro de objeto, es necesario haber instanciado antes un objeto de la clase.
   - Un **atributo de objeto** representa parte del estado del objeto.
   - Un **método de objeto** representa parte del comportamiento o una funcionalidad del objeto. Tiene acceso a los miembros de objeto, los miembros de clase y a sus parámetros. Su ejecución puede provocar un cambio en el estado del objeto.
2) **Miembro de Clase (estático).** La clase contiene una única copia del miembro, la cual comparte con todos los objetos instanciados de la clase. Para acceder a un miembro de clase, no es necesario tener ningún objeto instanciado previamente.
   - Un **atributo de clase** no forma parte del estado de un objeto.
   - Un **método de clase** no forma parte del comportamiento o una funcionalidad de un objeto. Solo tiene acceso a los miembros de clase y a sus parámetros. Su ejecución no modifica el estado de un objeto.

2.2.4. Ámbito de Miembros

Se denomina **ámbito de un miembro** a la parte, porción o bloque de código fuente en la que dicho miembro es visible y se puede utilizar. Como regla general, el ámbito de un miembro corresponde a la clase en la que dicho miembro se declara o define.

Para facilitar la encapsulación entre objetos de un programa, los **atributos** de una clase se definen **privados** y los **métodos** de una clase se definen **públicos**. De esta manera:

- Dentro de la clase, se puede acceder directamente a todos los miembros de la clase (atributos y métodos).
- Para acceder a un **atributo de objeto privado** fuera de la clase, se debe utilizar su método de acceso público correspondiente (*setter* o *getter*) con la siguiente sintaxis:

```
objeto.setAtributo(nuevoValor)      para modificarlo con un nuevo valor
valor = objeto.getAtributo()        para consultarlo
```

- Para acceder a un **atributo de clase privado** fuera de la clase, se debe utilizar su método de acceso público correspondiente (*setter* o *getter*) con la siguiente sintaxis:

```
Clase.setAtributo(nuevoValor)       para modificarlo con un nuevo valor
valor = Clase.getAtributo()         para consultarlo
```

- Fuera de la clase, se puede acceder a cualquier **método de objeto público** con la siguiente sintaxis:

```
objeto.método(parámetros)                   si no devuelve ningún resultado
resultado = objeto.método(parámetros)       si devuelve un resultado
```

- Fuera de la clase, se puede acceder a cualquier **método de clase público** con la siguiente sintaxis:

```
Clase.método(parámetros)                    si no devuelve ningún resultado
resultado = Clase.método(parámetros)        si devuelve un resultado
```

3. ENCAPSULACIÓN

En el paradigma de programación orientada a objetos, la **encapsulación** se refiere a la posibilidad de utilizar una clase sabiendo lo básico para poderla usar y prescindir de sus detalles de implementación. Consiste en agrupar, siguiendo un proceso de abstracción, todos las características (es decir, atributos) y operaciones o acciones (es decir, métodos) relacionadas en una misma clase, de manera que sea sencillo reutilizar la clase y ocultar los detalles (es decir, atributos y métodos).

La encapsulación lleva consigo otras características de la orientación a objetos, como son la **reutilización de código fuente** y la **ocultación de atributos y métodos**.

3.1. REUTILIZACIÓN DE CLASES

Dado que todos los atributos y métodos relacionados entre sí se encuentran juntos en una clase, la encapsulación **facilita el aprovechamiento del código fuente** de la clase en diferentes programas simplemente instanciando un objeto de dicha clase.

No importa cómo está implementada la clase por dentro. Solo interesa saber qué permite hacer y cómo se utiliza. Se supone que la clase está bien construida y sus métodos funcionan correctamente. Cuando el programador diseña una clase, deberá ser preciso en la codificación de los métodos para asegurar su correcto funcionamiento.

En la API de Java existen multitud de clases que se pueden utilizar con la certeza de que han sido probadas exhaustivamente y de que funcionan correctamente.

Al diseñar una clase, se debe pensar muy bien qué atributos y métodos incluir en la clase para cubrir las necesidades actuales y dejar la puerta abierta para futuros usos o mejoras. Por ejemplo, si está previsto que la clase pueda ser usada más allá del problema que se está resolviendo en ese momento, es decir, si es posible que la clase se reutilice en el futuro, se deberá tener una mayor capacidad de abstracción.

3.2. OCULTACIÓN DE MIEMBROS DE UNA CLASE

Cuanta menos información se proporcione de una clase (es decir, cuanto más pequeña sea la parte visible de una clase para las otras clases que pueden interactuar con ella), menos probable será que se produzcan problemas y más sencillo será asegurar ciertos factores de calidad como la reusabilidad, la portabilidad y la facilidad de uso de las clases.

La encapsulación **garantiza la integridad de los datos** que contiene el objeto mediante el mecanismo de ocultación. La ocultación de información es un mecanismo muy útil y una consecuencia directa de la encapsulación.

De manera informal, la ocultación consiste en esconder, tapar o encubrir todo aquello de una clase que no interesa que se sepa o se tenga acceso. La ocultación también se puede denominar **visibilidad** de los miembros de una clase o **restricción de acceso** a miembros de una clase.

El programador que utilice una clase deberá conocer y poder acceder directamente a los atributos y métodos estrictamente necesarios, nada más. Así se evita el acceso a los miembros de la clase por cualquier otro medio distinto a los especificados.

3.2.1. Niveles de Acceso

La **visibilidad u ocultación de un miembro de una clase** se define en relación con el resto de clases. Una clase siempre puede acceder a todos los atributos y métodos que se han definido en ella.

En general, los lenguajes de programación orientados a objetos definen tres **niveles de acceso**, graduados en diferente medida, para los miembros definidos en una clase:

- **Público (`public`).** Cuando un atributo o método de una clase se define público, la clase y cualquier otra clase pueden acceder a dicho miembro para utilizarlo. No conviene que los atributos sean públicos.
- **Protegido (`protected`).** Cuando un atributo o método de una clase se define protegido, la clase y cualquier otra clase descendiente (a través de herencia) de esta clase pueden acceder a dicho miembro para utilizarlo.
- **Privado (`private`).** Cuando un atributo o método de una clase se define privado, sólo la clase puede acceder a dicho miembro para utilizarlo. Para acceder a un atributo privado desde fuera de la clase, se deberá utilizar un método público de acceso a dicho atributo (*setter* o *getter*).

Hay lenguajes de programación que definen más niveles de acceso. C# define el nivel *internal* y Java define el nivel *package*.

Si no se indica la visibilidad u ocultación de un miembro de una clase, el compilador le asignará al miembro un nivel de acceso por defecto. En Java, el nivel de acceso por defecto es **`package`**: cuando un atributo o método de una clase se define a nivel de paquete, la clase y cualquier otra clase contenida en el mismo paquete pueden acceder a dicho miembro para utilizarlo.

Los **niveles de acceso de Java** para miembros de una clase son:

	LA PROPIA CLASE	CLASE DEL PAQUETE	CLASE DESCENDIENTE	CUALQUIER OTRA CLASE
`public`	X	X	X	X
`protected`	X	X	X	
`package`	X	X		
`private`	X			

4. CLASES ENVOLTORIO DE TIPOS PRIMITIVOS

Algunas veces puede resultar interesante tratar datos de tipos primitivos como objetos. Para dar solución a esta circunstancia, el API de Java incorpora las llamadas **clases envoltorio** (*wrappers*), que proporcionan características de objetos a tipos primitivos de datos.

Existe una clase para cada uno de los tipos primitivos.

Tipo primitivo	Clase envoltorio
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

Para utilizar un objeto de una de estas clases, basta con declararlo y asignarle un valor.

```
Integer entero = 23;
```

Anteriormente, se declaraba y se instanciaba, pero el uso de constructores de clases envoltorio ha quedado obsoleto.

```
Integer entero = new Integer(23);
```

En cualquier caso, se crea una referencia nueva y se asigna un valor, de manera que, si se asigna el mismo valor a objetos distintos, se considera que son iguales. Anteriormente, para comprobar si dos objetos de una clase envoltorio eran iguales, se debía utilizar el método equals, pero actualmente ya es posible utilizar el operador == como si de un tipo primitivo se tratara.

Dados los objetos:

```
Integer numero1 = 33;

Integer numero2 = 33;
```

El siguiente bloque:

```
if (numero1 == numero2) {

    System.out.println("Son iguales");

}
```

Es equivalente a:

```
if (numero1.equals(numero2)) {

    System.out.println("Son iguales");

}
```

Otro aspecto destacable sobre las clases envoltorio es que proporcionan métodos de clase que permiten realizar la conversión desde cadenas de texto a tipos primitivos o a objetos de clases envoltorio, siempre y cuando el contenido de dichas cadenas se corresponda con un dato del tipo a transformar.

```
String numeroEnCadena = "127";

int numero = Integer.parseInt(numeroEnCadena);

Integer numeroObjeto = Integer.valueOf(numeroEnCadena);
```

En resumen, los aspectos a tener en cuenta al trabajar con clases envoltorio son los siguientes:

- Los nombres de las clases comienzan por mayúscula, siguiendo la convención de nombres de Java, y en contraposición a los tipos primitivos de datos, que comienzan por minúscula.
- Con el tiempo, las clases envoltorio han incorporado los operadores aritméticos existentes entre sus tipos primitivos correspondientes, así como operadores comparadores.
- El uso de tipos primitivos es mucho más eficiente que el de objetos, por lo que siempre que sea posible, se utilizarán datos de tipos primitivos.
- Estas clases poseen métodos para realizar conversiones desde cadenas de texto a tipos primitivos y viceversa.

5. EJEMPLOS DE USO DE CLASES DE JAVA

El API de Java proporciona algunas clases que resultan de gran utilidad, como por ejemplo **`Math`**, que incorpora un gran conjunto de funciones matemáticas, o **`Random`**, que permite obtener valores aleatorios en diferentes intervalos.

5.1. MATH

Esta clase está compuesta por una serie de atributos y métodos de clase. Para utilizarla no es necesario crear una instancia, sino que basta con poner su nombre seguido de un punto y el nombre del atributo o método al que se quiere acceder.

En el siguiente ejemplo, se muestra el uso de algunos miembros de la clase **`Math`**.

```
public class EjemploMath {

    public static void main(String[] args) {

        double raiz = Math.sqrt(25.5);
        double doblePI = 2 * Math.PI;
        double redondeoAlza = Math.ceil(4.3);
        double redondeoBaja = Math.floor(4.3);
        long redondeo1 = Math.round(4.3);
        long redondeo2 = Math.round(4.7);

        System.out.println("RESUMEN DATOS FUNCIONES DE MATH:");
        System.out.println("Raíz 22.5 = " + raiz);
        System.out.println("Doble de PI = " + doblePI);
        System.out.println("Redondeo al alza de 4.3 = " + redondeoAlza);
        System.out.println("Redondeo a la baja de 4.3 = " + redondeoBaja);
        System.out.println("Redondeo de 4.3 = " + redondeo1);
        System.out.println("Redondeo de 4.7 = " + redondeo2);
    }

}
```

5.2. RANDOM

Esta clase está compuesta por una serie de métodos de objeto y para utilizarla habrá que crear un objeto de la misma.

Se encuentra en el paquete **java.util** y habrá que importar la clase para utilizarla.

En el siguiente ejemplo, se muestra el uso de varios de los métodos de la clase **`Random`**.

Se necesita crear un objeto mediante el cual se realizarán las llamadas para obtener diferentes valores aleatorios:

El método **`nextBoolean`** devuelve un valor aleatorio en el rango de los booleanos, es decir, devolverá true o false.

El método **`nextInt`** devuelve un valor aleatorio en el rango de los enteros, es decir, uno de los 2^{32} posibles números.

El método **`nextInt`** con un parámetro de entrada, devuelve un entero que va desde el cero hasta el número que se indique como parámetro menos uno. Este método permite delimitar el rango de valores

a obtener mediante la siguiente fórmula: **`nextInt (maximo - minimo + 1) + minimo`**, o lo que es lo mismo, se le pasa como parámetro la cantidad de números posibles a obtener y se le suma el valor mínimo.

El método **`nextDouble`** devuelve un valor aleatorio que va desde 0.0 hasta 1.0.

```
import java.util.Random;

public class EjemploRandom {

    public static void main(String[] args) {
        Random aleatorio = new Random();

        boolean booleano = aleatorio.nextBoolean();
        int entero = aleatorio.nextInt();
        int nota = aleatorio.nextInt(10) + 1;       //entre 1 y 10
        int entero5y10 = aleatorio.nextInt(6) + 5;  //entre 5 y 10
        double real = aleatorio.nextDouble();       //entre 0.0 y 1.0

        System.out.println("RESUMEN DATOS ALEATORIOS:");
        System.out.println("Booleano: " + booleano);
        System.out.println("Entero: " + entero);
        System.out.println("Nota: " + nota);
        System.out.println("entero5y10: " + entero5y10);
        System.out.println("Real: " + real);
    }

}
```

6. CASOS PRÁCTICOS

6.1. ACCESO A ATRIBUTOS

La siguiente clase **`Película`** contiene una serie de atributos y sus correspondientes métodos de acceso. Los atributos se han definido como private y los métodos como public, siguiendo el principio de ocultación. No obstante, no siempre deberán crearse métodos de acceso para todos los atributos, sino solo para aquellos que la resolución del problema lo requiera.

```
public class Pelicula {

    private String titulo;
    private int minutosDuracion;
    private boolean dobladaCastellano;
    private double puntuacion;

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public int getMinutosDuracion() {
        return minutosDuracion;
    }

    public void setMinutosDuracion(int minutosDuracion) {
        this.minutosDuracion = minutosDuracion;
    }

    public boolean isDobladaCastellano() {
        return dobladaCastellano;
    }

    public void setDobladaCastellano(boolean dobladaCastellano) {
        this.dobladaCastellano = dobladaCastellano;
    }

    public double getPuntuacion() {
        return puntuacion;
    }

    public void setPuntuacion(double puntuacion) {
        this.puntuacion = puntuacion;
    }

}
```

La siguiente clase **`Principal`** instancia un objeto de la clase `Película` y hace uso, en primer lugar, de sus setters para poner valor a sus atributos, y a continuación, utiliza sus getters para obtener el valor de sus atributos y mostrarlos por pantalla.

En este ejemplo se puede observar que al realizar la instancia de `Película` se llama al constructor de la clase sin argumentos. Éste es el constructor por defecto, que, si no se declara ningún otro, Java permite utilizarlo sin necesidad de declararlo.

```java
public class Principal {

    public static void main(String[] args) {
        Pelicula peli;

        peli = new Pelicula();

        peli.setTitulo("Titanic");
        peli.setMinutosDuracion(194);
        peli.setDobladaCastellano(true);
        peli.setPuntuacion(7.8);

        System.out.println("DATOS DE LA PELÍCULA: ");
        System.out.println("Título: " + peli.getTitulo());
        System.out.println("duración: " + peli.getMinutosDuracion() + " min.");
        System.out.println("Doblada al castellano: " + peli.isDobladaCastellano());
        System.out.println("Puntuación: " + peli.getPuntuacion());
    }

}
```

6.2. CONSTRUCTORES Y ESTADO DE LOS OBJETOS

La siguiente clase **`Película`** contiene una serie de atributos, dos constructores y un método para obtener el estado del objeto.

Al haber dos constructores con distintos parámetros se dice que existe sobrecarga de constructores, así podrá crearse un objeto de la clase `Película` indicando el título, la duración, si está doblada y la puntuación, y también se podrá crear un objeto de la clase `Película` indicando el título, la duración y si está doblada. Dependiendo de los parámetros que se pasen a la hora de instanciar el objeto, se ejecutará uno u otro constructor.

Por otro lado, existe un método que devuelve una cadena de texto con el contenido de los atributos en el momento de la llamada, lo que se conoce como estado del objeto.

```java
public class Pelicula {

    private String titulo;
    private int minutosDuracion;
    private boolean dobladaCastellano;
    private double puntuacion;

    public Pelicula(String titulo, int minutosDuracion,
                                            boolean dobladaCastellano) {
        this.titulo = titulo;
        this.minutosDuracion = minutosDuracion;
        this.dobladaCastellano = dobladaCastellano;
        this.puntuacion = 0.0;
    }

    public Pelicula(String titulo, int minutosDuracion,
                        boolean dobladaCastellano, double puntuacion) {
        this.titulo = titulo;
        this.minutosDuracion = minutosDuracion;
        this.dobladaCastellano = dobladaCastellano;
        this.puntuacion = puntuacion;
    }

    public String obtenerEstado() {
        return "Pelicula [titulo=" + titulo + ", minutosDuracion="
                    + minutosDuracion + ", dobladaCastellano="
                    + dobladaCastellano + ", puntuacion=" + puntuacion + "]";
    }

}
```

La siguiente clase **`Principal`** muestra la instanciación de dos películas, una con cada uno de los constructores. Una vez creados los objetos se llama a su método `obtenerEstado` y se muestra por pantalla la cadena de texto obtenida.

En este caso no se podría utilizar el constructor sin argumentos para instanciar una película porque ya se han creado otros constructores. Si se quiera poder utilizar un constructor sin parámetros, habría que declararlo en la clase.

```java
public class Principal {

    public static void main(String[] args) {
        Pelicula pelicula01, pelicula02;

        pelicula01 = new Pelicula("Titanic", 194, true, 7.8);
        pelicula02 = new Pelicula("Seven", 127, true);

        System.out.println(pelicula01.obtenerEstado());
        System.out.println(pelicula02.obtenerEstado());

    }

}
```

6.3. MIEMBROS DE CLASE Y MIEMBROS DE OBJETO

La siguiente clase **`Película`** contiene una serie de atributos de objeto y el atributo de clase `totalPeliculas`. Cada objeto que se instancie tendrá diferentes valores para sus atributos de objeto, pero existirá un solo valor para el atributo de clase.

Además, esta clase tiene un constructor y dos métodos de clase, uno para obtener el valor del atributo de clase y otro para validar una puntuación. Los métodos de clase son independientes de los objetos que se instancien de la misma y serán invocados partiendo del nombre de la clase. En el constructor de la clase, además de poner valores a los atributos de la clase, se incrementa en uno el valor del atributo de clase `totalPeliculas`, de manera que este atributo almacenará la cantidad de películas que se han instanciado.

```
public class Pelicula {

    private static int totalPeliculas = 0;

    private int numero;
    private String titulo;
    private int minutosDuracion;
    private boolean dobladaCastellano;
    private double puntuacion;

    public static int getTotalPeliculas() {
        return Pelicula.totalPeliculas;
    }

    public Pelicula(String titulo, int minutosDuracion,
            boolean dobladaCastellano, double puntuacion) {

        Pelicula.totalPeliculas++;
        this.numero = Pelicula.totalPeliculas;
        this.titulo = titulo;
        this.minutosDuracion = minutosDuracion;
        this.dobladaCastellano = dobladaCastellano;
        this.puntuacion = puntuacion;
    }

    //Método que verifica si una puntuación dada está en el intervalo correcto
    //Los valores aceptables van de cero a diez
    public static boolean validarPuntuacion(double puntos) {
        if(puntos >= 0.0 && puntos <= 10.0) {
            return true;
        }
        else {
            return false;
        }
    }

}
```

La siguiente clase **`Principal`** crea una serie de objetos de la clase `Película` hasta que el usuario diga que ya no quiere añadir más. Los datos con los que se crea cada objeto se solicitan al usuario y para verificar que el valor de la puntuación es válido se hace uso del método de clase `validarPuntuación`, si la puntuación no es válida, se muestra un mensaje informativo y se pone el valor a cero. Tras haber creado la cantidad de películas indicada por el usuario, se muestra un mensaje con el total de objetos instanciados obteniendo el valor del método de clase que acceder al atributo de clase que contiene este dato.

```
import entrada.Teclado;

public class Principal {

    public static void main(String[] args) {
        Pelicula peli;

        String titulo;
        int duracion;
        double puntuacion;
        boolean seguir;

        do {
            titulo = Teclado.leerCadena("¿Título? ");
            duracion = Teclado.leerNatural("¿Duración? ");
            puntuacion = Teclado.leerReal("¿Puntuación? ");
            if(! Pelicula.validarPuntuacion(puntuacion)) {
                puntuacion = 0.0;
                System.out.println("La puntuación no es válida.");
            }
            peli = new Pelicula(titulo, duracion, true, puntuacion);
            seguir = Teclado.leerBooleano("¿Añadir otra?");
        }while(seguir);

        System.out.println("Se han creado un total de " +
                           Pelicula.getTotalPeliculas() + " películas");
    }

}
```