Implementación de la clase Disco

Esqueleto general de la clase

Regresamos a DrJava para escribir ya la definición de la clase Disco.

Diste ya la interfaz ServiciosDisco que lista los métodos públicos para tu clase, además de haberlos ya comentado. Para la clase **Disco** puedes empezar con cargar el archivo de la interfaz **ServiciosDisco** y guardarlo (*Save as*) en el mismo subdirectorio pero con el nombre Disco.java, ya que, al menos, tienes que implementar estos métodos; como verás en la ventana izquierda, ahora el archivo se llama Disco.java. Procederás a "adaptarlo" para que sea una clase.

La primera modificación que tienes que hacer es que, en este caso se trata de una clase, no de una interfaz y, además, esta clase implementa a la interfaz **ServiciosDisco**, por lo que cambias el encabezado de la interfaz al de una clase que implementa esta interfaz.

Antes:

```
public interface ServiciosDisco {
```

Ahora:

```
public class Disco implements ServiciosDisco  {
```

También reemplazarás en el resto de los encabezados el punto y coma que termina cada encabezado por las llaves que empiezan y terminan la definición de un método. Las llaves "rodean" a la implementación del método, como lo hiciste para el método main.

Tienes ya un esqueleto para trabajar, pero si tratas de compilar, para cada uno de los métodos que regresan algún valor te dirá el compilador que falta el enunciado **return**, que es el que dice exactamente cuál valor regresa, y tiene que ser del mismo tipo que el que declara el método en su encabezado. Pospondrás por un momento la corrección de estos errores, pues se solucionan cuando des la implementación.

Por lo pronto, y para poder verificar la compilación correcta de lo que vayas haciendo, agregarás a los métodos que lo requieren un enunciado **return** antes de que termine el bloque que lo define. La sintaxis de **return** es la siguiente:

```
    return ⟨expresion⟩;
```

La ⟨*expresion*⟩ a que se refiere es algo que de lo que se obtiene un valor que, como ya escuchaste, tiene que ser del mismo tipo que el que está anotado en el encabezado del método. Para los tipos numéricos la ⟨*expresion*⟩ será cero (0) y para los que tienen que regresar una referencia, la ⟨*expresion*⟩ tomará el valor de **null**, que es una referencia vacía (no hay nada en esa variable o expresión).

El enunciado **return** va a tener una ⟨*expresión*⟩, dependiendo del tipo del valor de regreso del método. La ⟨*expresión*⟩ valdrá:

0: Cero si el método regresa un valor numérico, ya sea entero o real.

null: Una referencia nula si el método regresa una referencia (un objeto).

false: El valor falso si el método regresa un valor booleano.

' ': Un carácter blanco si el método regresa un valor de carácter.

Llenas todos los métodos con el enunciado **return**. Al compilar todo parece estar bien ahora.

Atributos (variables propias o campos)

Una clase, además de proporcionar servicios, tiene que tener un conjunto de variables o **atributos** que le sirven para que el objeto pueda tener un *estado* propio, diferente al de otro objeto de la misma clase; esto es, guarde valores que le ayuden a definirse. Cuando hiciste la tarjeta de responsabilidades los identificaste. Colocas su declaración al principio de la clase.

La declaración de un atributo debe contener su acceso, que en la mayoría de los casos es privado (**private**) porque no quieres que lo puedan manipular cualquiera que sepa de la existencia de la clase. Le sigue el tipo del atributo, que puede ser un tipo primitivo o el nombre de una clase (tipo referencia) y le sigue el nombre que hayas elegido para ese atributo. La declaración del atributo puede tener dos modificadores, entre el acceso y el tipo, que son:

final Indica que el valor sólo se asigna una única vez, con la declaración del atributo o en los constructores.

static Como ya escuchaste al implementar el método main, este modificador indica que el atributo queda definido al compilar la clase, por lo que pertenece a la clase. Únicamente existe un ejemplar de ese atributo y aunque cualquier objeto de la clase puede modificarlo (si no es **final**), todos los objetos construidos usan el mismo ejemplar. El valor que tiene (si no es **final**) en un momento dado es el que le haya puesto el último en usarlo.

Una declaración de atributo o método puede tener ambos modificadores, uno de ellos o ninguno. En el caso de los métodos el modificador final se refiere a que el método no puede ser redefinido en alguna clase que herede de ésta. En el caso de clases quiere decir que la clase no puede ser heredada (o extendida). Todo esto tiene que ver con la herencia, que no verás en esta lección.

En la clase **Disco** tienes tres atributos que son **final** pero ninguno de ellos es **static**, pues cada objeto que se construya tendrá valores constantes diferentes para ellos. Se acostumbra en Java nombrar a estos atributos exclusivamente con mayúsculas.

```
private final short  TIPO_DISCO;    // 1:CD, 2:DVD, 3:BR
private final String NOMBRE;        // artista o pelicula
private final int    ANHO;          // fecha de grabacion
```

Para que no tengas que recordar cuál entero corresponde a cada tipo de disco, puedes declarar constantes de la clase, ya que no importa el objeto deberán ser interpretadas igual. Esto lo haces declarando atributos que tengan ambos modificadores:

```
private static final short CD  = 1;
private static final short DVD = 2;
private static final short BR  = 3;
```

En estas declaraciones estás, al mismo tiempo, definiendo el valor que van a tener estas variables, a lo que se llama *inicializar* o *definir* el valor de una variable. Consiste de un operador de asignación (=) y del lado derecho un valor que se puede calcular en el momento de la declaración. Como estás dando enteros constantes, no hay ningún problema de que no los pueda calcular el compilador. También se pueden inicializar atributos que no sean constantes o variables locales a los métodos, como lo hiciste en el método main del usuario.

Además tienes dos atributos más, el número máximo de transmisiones permitidas y el número de transmisiones activas.

```
private int permitidas;      // Maximo transmisiones permitidas
private int activas;         // Transmisiones activas
```

Es conveniente poner un límite superior al número de transmisiones permitidas mediante una constante simbólica que se llame MAX_PERMITIDAS. Como va a ser la misma para todos los objetos de la clase, la constante va a ser estática; como también debe existir desde el momento de compilación, elegirás en este momento el valor máximo, que será 50 para poder hacer pruebas con ella:

```
private static final int MAX_PERMITIDAS = 50;
```

Al igual que usas constantes estáticas para el tipo de disco y el número máximo de transmisiones permitidas, agregamos las constantes también estáticas PRIMER_ANHO para el primer año válido y ULT_ANHO para el último año válido:

```
private static final int PRIMER_ANHO = 1900; // Primer anho valido
private static final int ULT_ANHO    = 2017; // Ultimo anho valido
```

Con estas últimas declaraciones tienes los límites inferiores y superiores para los campos que quieres vigilar.

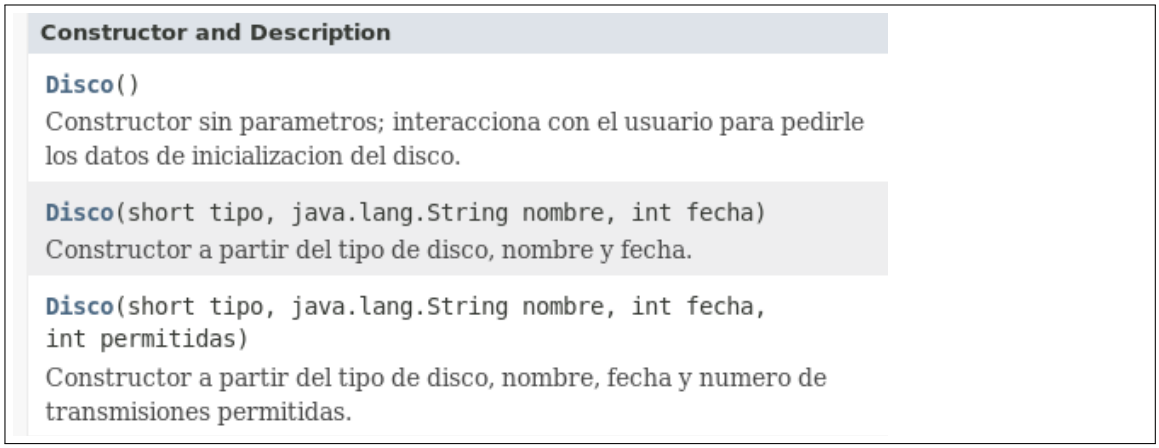
Conforme necesites más información almacenada para que los métodos puedan funcionar, agregarás atributos, ya sea a los objeto o la clase.

Aun así la clase no compila, porque detecta que hay atributos de objeto constantes que deberían ser inicializados en los constructores y no lo son. Pero si haces lo mismo que para las constantes de clase, inicializarlas al declararlas, todos los objetos van a tener el mismo valor. Por ello, debes dejar a los constructores que las inicialicen, lo que harás al implementarlos.

Las interfaces no tienen constructores (métodos que construyen objetos de esa clase) pero las clases sí. Si el programador no escribe ningún constructor, Java, por omisión, proporciona un constructor sin parámetros, que inicializa todos los atributos en cero (falso, en su caso) o **null**, dependiendo de si se trata de valores primitivos o referencias respectivamente. Estos valores no te servirán, pues para los atributos constantes, no los podrás cambiar después. Por ello tendrás que escribir tus propios constructores.

Cuando hiciste a la clase que usa a **Disco**, la clase Usuario, contamos con la documentación de la clase y se marcaban los constructores que aparecen en la siguiente página.

Tienes que agregar a tu esqueleto estos tres constructores. Al hacerlo, el constructor por omisión de Java ya no está disponible. Por lo pronto agregarás únicamente los encabezados, pero en lugar de terminarlos con punto y coma, les pondrás las llaves de inicio y final de la definición de los métodos. En el caso de los constructores, lo único que hay que agregar al encabezado que te muestra la documentación es el acceso que, si aparece en la documentación se trata de acceso **public**. El tipo de valor que regresa es, precisamente, un objeto de la clase; por ello los constructores tienen como nombre del método el nombre de la clase, que también se puede interpretar como el tipo de valor que regresan.



Constructor and Description
`Disco()` Constructor sin parametros; interacciona con el usuario para pedirle los datos de inicializacion del disco.
`Disco(short tipo, java.lang.String nombre, int fecha)` Constructor a partir del tipo de disco, nombre y fecha.
`Disco(short tipo, java.lang.String nombre, int fecha, int permitidas)` Constructor a partir del tipo de disco, nombre, fecha y numero de transmisiones permitidas.

Colocas los constructores poniendo al final al constructor sin parámetros, ya que es el último que llenarás.

Sigue sin compilar tu clase por las mismas razones que antes, que son que no aparece en el código la inicialización de las constantes de objeto. Tienes que escribir la codificación (implementación) de los constructores y para ello requieres del uso de *expresiones* que son las que asignan y manipulan valores, propósito fundamental de todo proyecto. En el próximo vídeo verás cómo codificar estos tres constructores.

¡Espero contar con tu presencia en el siguiente video!