

## Práctica 1. Introducción a NetBeans

Entornos de Usuario

6 de octubre de 2025

### Objetivos

Esta es la primera sesión del módulo práctico de la asignatura “Entornos de Usuario”. Esta práctica es introductoria y proporcionará una visión global de la asignatura. En ella se verán algunos de los elementos básicos que se utilizarán en el resto de las sesiones.

Los objetivos que se pretenden alcanzar con la realización de esta práctica son:

- Aprender el uso de NetBeans, un IDE (*Integrated Development Environment*, Entorno Integrado de Desarrollo) que facilita la implementación de aplicaciones en Java. Se trata de una plataforma que proporciona todos los servicios y ayudas que un programador de Java puede necesitar.
- Usar, a través de ejemplos, el lenguaje de programación Java y algunas de las características avanzadas proporcionadas por el API estándar.

### Índice

1	Introducción	1
2	Ejercicio 1 – Nuestra primera clase en Netbeans	1
2.1	Crear y ejecutar una aplicación Java . . . . .	2
3	Ejercicio 2 – Primera aplicación Java.	5
3.1	La clase <code>Pelicula</code> . . . . .	6
3.2	La clase <code>Coleccion</code> . . . . .	12
3.3	Punto de entrada a la aplicación . . . . .	14
4	Ejercicio 3 – Contenedores, tipos primitivos y clases envoltorio	15
5	Ejercicio 4 – Herencia. Construcción de objetos.	16
6	Uso del depurador	17
7	Ejercicio 5 – Aplicación	18
8	Entrega de la Práctica	19

## 1 Introducción

Esta práctica se compone de una serie de ejercicios dirigidos.

**Nota:** Entendemos por ejercicio dirigido aquel en el que el estudiante dispone del código íntegro de los programas y de las explicaciones detalladas sobre cómo implementarlo y ejecutarlo.

La forma correcta de abordar la realización de estos ejercicios es:

- Lea atentamente en qué consiste el ejercicio, qué se espera de su realización y cualquier otra explicación relevante.
- Siga paso a paso las indicaciones del ejercicio, asegurándose en cada paso de que entiende qué es lo que se está haciendo y por qué.
- No aborde un paso posterior sin haber completado con éxito los pasos anteriores. Si encuentra problemas para entender o llevar a cabo algún punto, recurra a su Profesor.
- Cuando tenga que introducir código, no utilice la función *Cut & Paste*. No va a ahorrar tiempo ni esfuerzo y además perderá la oportunidad de adquirir experiencia en el lenguaje Java y en el uso del editor inteligente que integra NetBeans.
- La complejidad de los ejercicios aumenta a lo largo de la sesión. No cambie el orden de realización de los ejercicios ni inicie un ejercicio sin haber completado satisfactoriamente el ejercicio anterior. Esta norma es importante, ya que en cada ejercicio se da por supuesto que no es necesario explicar con detalle determinados pasos u operaciones que se han realizado en ejercicios anteriores.

## 2 Ejercicio 1 – Nuestra primera clase en Netbeans

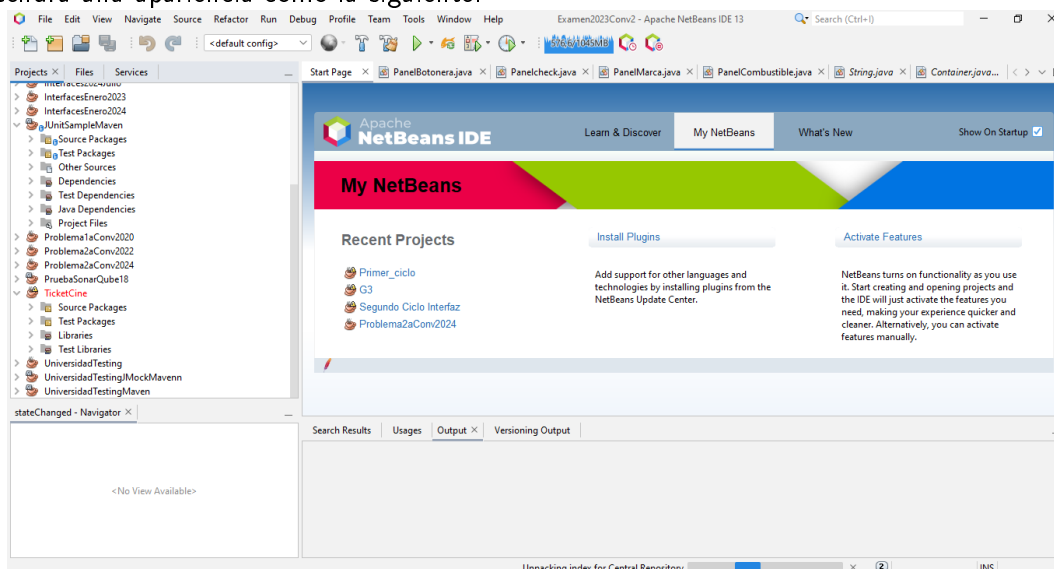
En este apartado realizaremos una serie de actividades dirigidas para ilustrar algunos de los aspectos básicos del IDE NetBeans y del desarrollo en Java. Una vez completado el ejercicio Usted sabrá:

- Cómo se crea un proyecto Java en NetBeans.
- Cómo se selecciona un proyecto principal y cómo se ejecuta el proyecto (básicamente, utilizando la tecla **F6** ).

**Nota:** Desde la pagina de NetBeans <http://ui.netbeans.org> se puede descargar el entorno, y acceder a su documentación, foros y otros recursos. En esta página, también podremos encontrar recursos relacionados con la programación de interfaces gráficas.

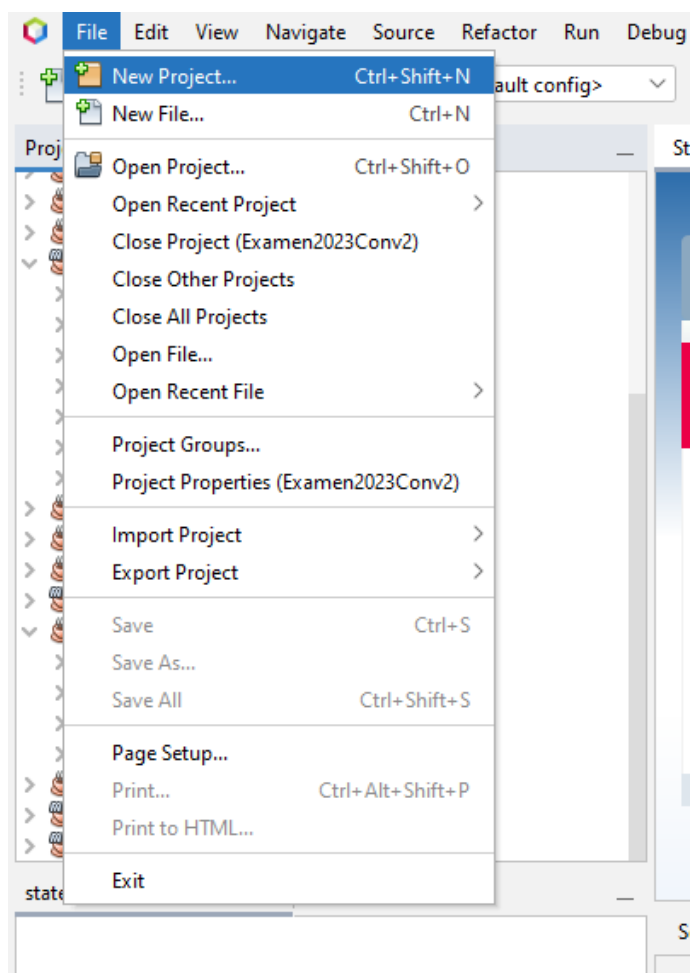
### 2.1 Crear y ejecutar una aplicación Java

Cuando abrimos el entorno de NetBeans (quizá se abra un dialogo de bienvenida que podemos cerrar), el entorno tendrá una apariencia como la siguiente:

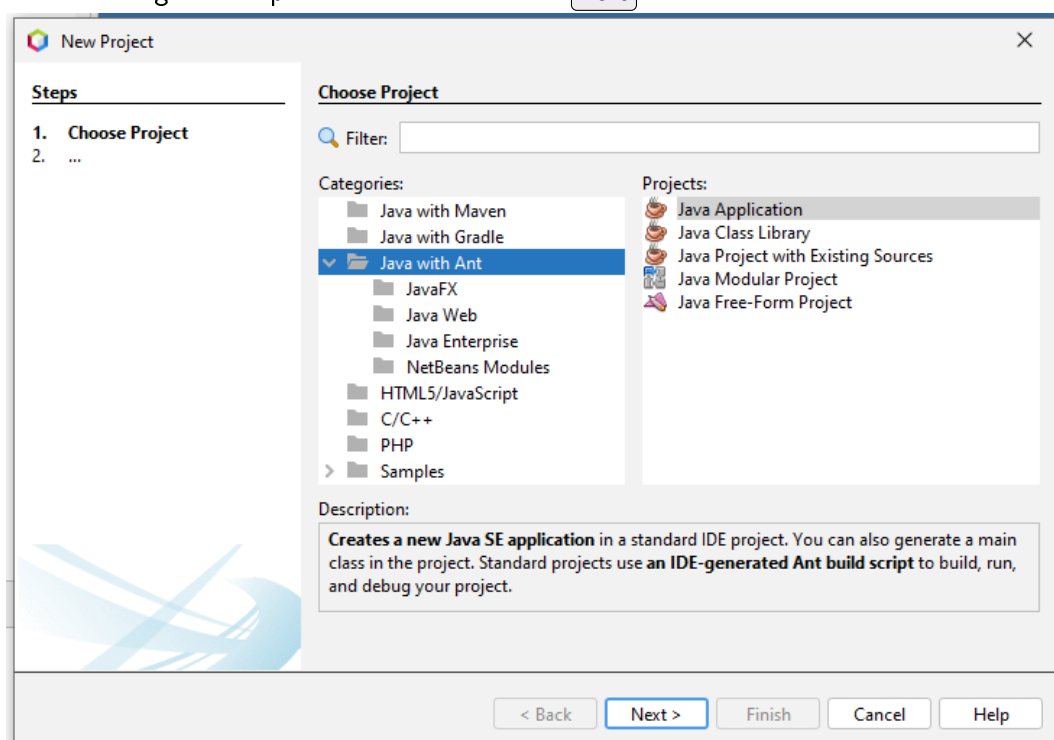


Para crear nuestra primera aplicación, seguiremos los siguientes pasos:

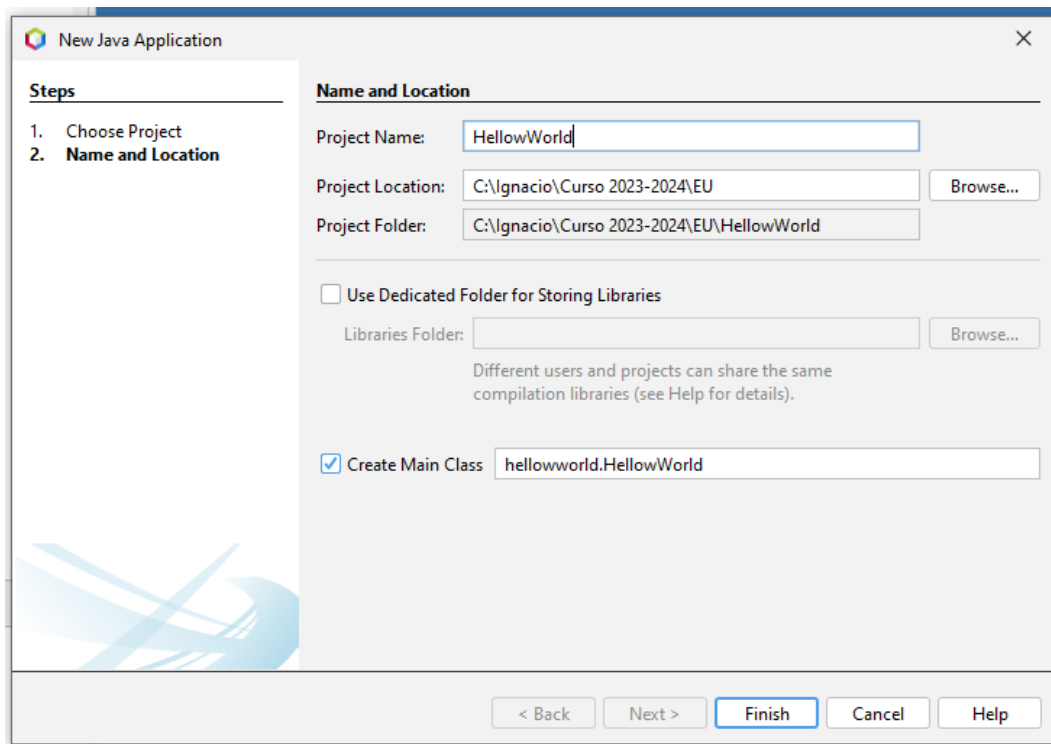
1. Para comenzar a crear una aplicación, la manera más simple es utilizar la opción **File** → **New Project** :



2. Posteriormente expande la categoría Java with Ant y selecciona Java Application como se muestra en la figura. Después selecciona el botón **Next**.



3. A continuación tendremos que indicar el paquete (y su ubicación física en un sistema de ficheros) en el que queremos que se incluya el código que se va a generar automáticamente. Utiliza HelloWorldApp como nombre del proyecto y deja los parámetros por defecto en el resto de campos. Para finalizar, pulsa sobre el botón **Finish**:

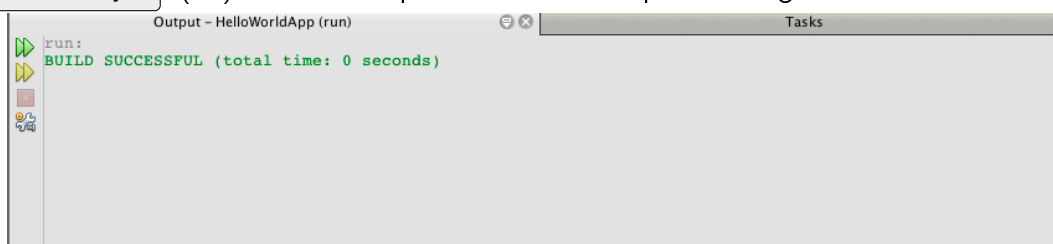


**Nota:** Los paquetes permiten agrupar clases y organizar nuestra aplicación, de manera similar a como agrupamos ficheros en carpetas en los sistemas de ficheros habituales.

4. Ahora entraremos en la pantalla principal de NetBeans, donde podremos visualizar los siguientes elementos:

- La *ventana del proyecto (project)*, que contiene en estructura de árbol los componentes del proyecto, incluyendo el código fuente, librerías de las cuales depende tu código, etc.
- La *ventana de edición del código fuente* con un fichero llamado HelloWorldApp abierto.
- La *ventana de navegación (navigator)*, que puedes usar para navegar rápidamente entre los elementos de las subclases.
- La *ventana de tareas (task)*, que muestra una lista de errores de compilación así como otras tareas marcadas con palabras claves tales como XXX y TODO (significa “por hacer” en inglés).

5. Compila y ejecuta el programa vacío que ha generado la aplicación. Para ello, elige la opción **Run** → **Run Main Project** (F6). Observarás que a continuación aparece lo siguiente:



**Nota:** Hasta ahora has generado el esqueleto de la clase principal con el IDE NetBeans. Esta clase no hace nada, como habrás podido comprobar al ejecutar el programa.

6. Ahora haremos que esta clase principal muestre texto en la consola. Esto lo podemos realizar reemplazando la línea:

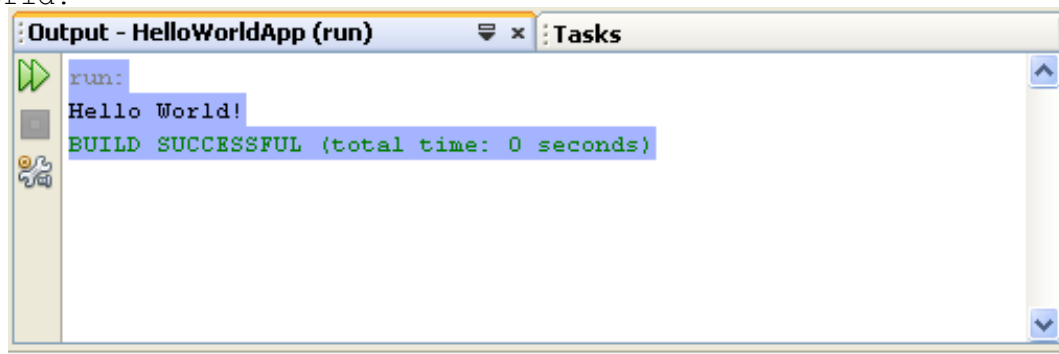
```
1 // TODO code application logic here
```

por la línea:

```
1 System.out.println("Hello World!");
```

Graba los cambios realizados con la opción **File** → **Save**. Una vez realizados estos cambios, el código de tu programa será el mostrado en el listado 1.

7. Por último, compila y ejecuta el programa de nuevo con la opción **Run** → **Run Main Project** (F6). Observarás que aparece una ventana tal como se muestra a continuación con el mensaje Hello World!



Listado 1: Código de la clase simple HelloWorld.java.

```

1  /*
2  * To change this template, choose Tools | Templates
3  * and open the template in the editor.
4  */
5  package helloworldapp;
6
7  /**
8   *
9   * @author nombre–apellidos
10  */
11  public class HelloWorldApp {
12      /**
13       * @param args the command line arguments
14       */
15      public static void main(String[] args) {
16
17          System.out.println("Hello World!");
18      }
19  }

```

### 3 Ejercicio 2 – Primera aplicación Java.

Una vez hemos aprendido las características básicas del entorno, desarrollaremos una pequeña aplicación Java. Esta aplicación nos ayudará a recordar algunos principios básicos de la programación orientada a objetos, al mismo tiempo que nos habituamos a la sintaxis del lenguaje. El objetivo del programa es permitir al usuario gestionar una lista de películas. En particular, la aplicación Java creará y mostrará por pantalla una lista de películas (nombre de la película, director y fecha de producción).

Recordamos que, mientras que en la programación procedimental el diseño viene dirigido por el procesamiento, en la programación orientada a objetos los datos dirigen el proceso de diseño. Si queremos almacenar una colección de películas, es lógico utilizar una clase para representar la película, y otra para representar la colección. La primera contendrá los campos necesarios que queramos almacenar sobre las películas y las operaciones básicas permitidas sobre objetos de este tipo. La segunda almacenará internamente un conjunto de referencias a objetos de tipo película y proporcionará las operaciones básicas que permitan interactuar con la colección. En UML (una notación orientada a objetos), esta relación se expresa como se ilustra a continuación.



En esta figura, cada clase viene representada por una caja, dividida en tres secciones. La primera contiene el nombre de la clase, la segunda los atributos y la tercera se reserva para los métodos. En nuestro caso,

hemos dejado las dos últimas secciones vacías porque aún no sabemos su contenido. La línea con el rombo oscuro en uno de sus extremos representa una relación de composición. Una colección está compuesta por cero o más películas.

Para incrementar la independencia entre clases, facilitar la reusabilidad y disminuir efectos laterales, es conveniente restringir la visibilidad de atributos y métodos (operaciones que proporciona la clase). En Java, tenemos tres modificadores de visibilidad:

- `private`. El elemento es únicamente visible desde dentro de la propia clase.
- `protected`. El elemento es visible desde la propia clase y desde cualquiera de sus subclases.
- `public`. El elemento es visible desde cualquier clase.

Cuando no se especifica un modificador, la visibilidad definida es “de paquete”. En este caso, el elemento es visible únicamente desde clases que se encuentren situadas en el mismo paquete.

Es una buena práctica de diseño nunca acceder a los atributos internos de una clase directamente desde otras clases poco relacionadas. En su lugar, deben crearse los denominados “getters y setters”. Esta práctica permite definir los métodos como públicos, pero mantener los atributos privados.

Para empezar el ejercicio, crea un nuevo proyecto de nombre `JavaApp1`. Ahora haz click sobre el nuevo proyecto con el botón derecho y en el menú contextual selecciona **New** → **Java Package...** y crea el paquete `es.uv.eu.javaapp1`.

**Nota:**

Nomenclatura de clases. En Java, el nombre completo de una clase está formado por el nombre del paquete al que pertenece más el nombre de la clase concatenados mediante un punto (“.”). Para garantizar que cada clase tiene un nombre único, es frecuente construir el nombre del paquete utilizando el nombre de dominio de la empresa que lo desarrolla (en nuestro caso, para la asignatura Entornos de Usuario en la Universitat de València hemos elegido `es.uv.eu`) seguido por el nombre del producto (`javaapp1`). En una aplicación Java se puede utilizar el nombre simple de la clase siempre que se importe el paquete mediante la sentencia `import` o bien el nombre completo (nombre del paquete + “.” + nombre de la clase) sin necesidad de importarlo previamente.

### 3.1 La clase `Pelicula`

La clase `Pelicula` será la utilizada para mantener la información relativa a una película concreta. En el listado 2 se muestra el código correspondiente a esta clase, que contiene información sobre el título de una película, su director y el año de producción. Observemos los siguientes detalles:

- La clase tiene un único constructor al que se le han de proporcionar todos los parámetros. Aunque podríamos haberlo hecho, no hemos proporcionado un constructor sin parámetros.
- Se proporcionan una serie de *getters*, que adoptan la forma `getNombreAtributo()`. Se trata de métodos que devuelven el valor del atributo `nombreAtributo`. Asimismo, se proporcionan *setters*. Los *setters* son métodos cuyo nombre tiene la forma `setNombreAtributo(...)` y que se utilizan para modificar el valor del atributo `nombreAtributo`. La existencia de *getters* y *setters* permite mantener los atributos de la clase como privados.

**Nota:**

El conjunto de *getters* y *setters* de una clase se conoce con el nombre de *actuators*. Su uso es tan generalizado que el IDE NetBeans los puede construir e integrar automáticamente en la clase.

- En algunas ocasiones, se ha utilizado la notación `this.nombreAtributo` para acceder a un atributo de la clase (por ejemplo, en los *setters*). En otras ocasiones, sin embargo, `this.` no se ha especificado explícitamente. La referencia `this` es una referencia al propio objeto. Cuando aparece `this.titulo`, se está haciendo referencia al atributo `titulo` del objeto sobre el que se ha invocado el método. Si no hay “ocultamiento”, `this.` puede omitirse. En el caso del método `setTitulo`, observa que existe un parámetro con el mismo nombre que el atributo. Si se omitiera `this.`, se

haría referencia al elemento más cercano con ese nombre, en este caso el parámetro del método. Sería absurdo escribir `titulo=titulo`.

- El nombre de la clase empieza con mayúscula. Métodos y atributos comienzan con minúscula. Todas las palabras que conforman un nombre aparecen concatenadas sin ningún carácter separador, poniendo en mayúscula la letra inicial de cada palabra a partir de la segunda. Esta forma de usar las letras mayúsculas y minúsculas no es casual. Obedece a unas reglas bien establecidas en Java y que es necesario respetar.
- Observa la forma de los comentarios que preceden la definición de la clase y la cabecera de cada método. Estos comentarios, que comienzan con `/**` y terminan con `*/` son comentarios “javadoc”. Javadoc es una aplicación que permite generar documentación automática en un formato estándar.

**Nota:**

En java, cada fichero puede contener como mucho una clase pública y además, el nombre del fichero debe coincidir con el de la clase. Sin embargo una clase puede contener otras clases internas (embebidas) o un mismo fichero puede contener varias clases que no sean públicas.

Listado 2: Código de la clase simple `Pelicula.java`.

```

1  /*
2  * Clase Pelicula
3  *
4  * Entornos de Usuario, Practica 1 (2012)
5  *
6  */
7  package es.uv.eu.javaapp1;
8
9  /**
10 *
11 * @author Entornos de Usuario
12 * @version 1.0
13 *
14 */
15 public class Pelicula {
16     /*
17     * Titulo de la pelicula.
18     */
19     private String titulo ;
20     /*
21     * Nombre del director.
22     */
23     private String director ;
24     /*
25     * Anyo de produccion.
26     */
27     private int anyo;
28
29     /**
30     * Constructor de Pelicula
31     * @param titulo Titulo de la pelicula .
32     * @param director Nombre del director.
33     * @param anyo Anyo de produccion.
34     */
35     public Pelicula(String titulo , String director , int anyo) {
36         this . titulo = titulo ;
37         this . director = director ;
38         this . anyo = anyo;
39     }
40
41     /**
42     * Devuelve el anyo de produccion de la pelicula.

```

```

43  * @return Anyo de produccion de la pelicula.
44  */
45  public int getAnyo() {
46      return anyo;
47  }
48
49  /**
50  * Asigna el anyo de produccion de la pelicula.
51  * @param anyo Anyo de produccion de la pelicula.
52  */
53  public void setAnyo(int anyo) {
54      this.anyo = anyo;
55  }
56
57  /**
58  * Obtiene el nombre del director de la pelicula.
59  * @return Nombre del director de la pelicula.
60  */
61  public String getDirector() {
62      return director;
63  }
64
65  /**
66  * Asigna el nombre del director de la pelicula.
67  * @param director Nombre del director de la pelicula.
68  */
69  public void setDirector(String director) {
70      this.director = director;
71  }
72
73  /**
74  * Obtiene el titulo de la pelicula.
75  * @return Titulo de la pelicula.
76  */
77  public String getTitulo() {
78      return titulo;
79  }
80
81  /**
82  * Asigna el titulo de la pelicula.
83  * @param titulo Titulo de la pelicula.
84  */
85  public void setTitulo(String titulo) {
86      this.titulo = titulo;
87  }
88  }

```

Antes de continuar con la siguiente clase, procedamos a incluir un pequeño código para comprobar el funcionamiento de nuestra clase. En general, es una buena práctica incluir un método “main” en cada clase que demuestre el buen funcionamiento de la misma. A modo de ejemplo, añadamos el código que aparece en el listado 3 a nuestra clase, que intenta crear dos películas y visualizarlas por pantalla:

Listado 3: Código del método main de la clase Pelicula.java. Primera versión.

```

1  public static void main(String[] argv){
2      Pelicula peli1 , peli2;
3      peli1 . setTitulo ("2001: Una Odisea en el Espacio");
4      peli2 . setTitulo ("2046");
5      System.out.println (peli1 );
6      System.out.println (peli2 );
7  }

```



El método `main` es un método más, y por lo tanto debe estar al mismo nivel que el resto de métodos (antes de la llave que denota el cierre de la clase).

Cuando introduzcamos el método `main` en el código de la clase, se producirán errores en la segunda y tercera línea. Mientras en C++ los elementos se instancian al declararse, en java es necesaria la operación `new`, utilizando alguno de los constructores declarados (o el constructor por defecto -sin parámetros- si no se ha definido ningún constructor). La declaración de `pel1` y `pel2` únicamente crea las referencias a los objetos, pero no los objetos en sí. Por lo tanto, NetBeans detecta que la operación `setTitulo` se está realizando sobre un objeto nulo (inexistente) y avisa de ello. Corrijamos el error cambiando estas dos líneas por operaciones de inicialización (ver listado 4)

Listado 4: Código del método `main` de la clase `Pelicula.java`. Segunda versión.

```
1 public static void main(String[] argv){
2     Pelicula peli1, peli2;
3     peli1=new Pelicula("2001: Una Odisea en el Espacio","Stanley Kubrick",1968);
4     peli2=new Pelicula("2046","Wong Kar Wai",2004);
5     System.out.println(peli1);
6     System.out.println(peli2);
7 }
```

Ahora el código compila, pero quizás el resultado no sea el esperado. En java, todo objeto hereda de la clase `Object` (cuando no se incluyen una cláusula `extends` para heredar de otra clase, `extends Object` queda implícito). La clase `Object` tiene definido un método llamado `toString` que devuelve una cadena con el nombre completo de la clase, seguido del carácter “@” y de la representación hexadecimal del código hash del objeto. Para mostrar un objeto por pantalla, se utiliza el método `System.out.println`. Este método llama al método `toString` del parámetro, y simplemente muestra por pantalla lo que devuelve. Si el método no se ha sobrescrito, su comportamiento será el definido en la clase `Object`. Esto explica la salida obtenida. Para poder visualizar las películas, sobrescribimos el método `toString`. Para ello, simplemente añadimos el nuevo método a la clase (listado 5).

Listado 5: Método `toString` propuesto.

```
1 public String toString(){
2     return "Pelicula: "+titulo+"\n"+"Director: "+director+"\n"+"Anyo: "+anyo+"\n";
3 }
```

**Nota:** Para sobrescribir un método, es necesario mantener la signatura (cabecera) de su superclase: el tipo de los argumentos y el tipo de retorno deben coincidir. Además, nunca podemos reducir la visibilidad del método.

Para terminar con esta clase, se nos ocurre que sería útil incluir dos atributos más:

- un código de película, que fuera generado automáticamente y de forma secuencial conforme se crean los objetos.
- el tipo de la película.

Para ello, necesitaremos crear dos campos más en la clase, que llamaremos `codigo` y `tipo`. Antes de continuar con la lectura, dedica 5 minutos a reflexionar como extenderías la clase para añadirle estos dos campos.

Quizás hayas encontrado dificultades para generar un código secuencial automático. Una manera de hacerlo es tener, además de un nuevo atributo para almacenar el código, un atributo de clase (estático). Un atributo de clase es un atributo que comparten todos los objetos de una misma clase (no se replica en cada instancia), cuya inicialización se produce cuando se instancia el primer objeto de esa clase. Para indicar que un atributo es de clase se utiliza el modificador `static`. La solución propuesta consiste en utilizar el atributo de clase para almacenar el próximo código a asignar. Cada vez que se instancia un nuevo objeto de la clase `Pelicula` se le asigna como código el valor almacenado en el atributo estático, y éste último se incrementa en una unidad.

Para ello, necesitamos primeramente añadir los atributos necesarios:

```

1  /*
2  * Codigo generado automaticamente para la pelicula
3  */
4  private int codigo;
5
6  /*
7  * Variable de clase para almacenar el proximo codigo que sera asignado a una pelicula
8  */
9  private static int proximoCodigo=0;

```

y posteriormente modificar el constructor para realizar la operación descrita arriba:

```

1  public Pelicula(String titulo , String director , int anyo) {
2      this.titulo = titulo ;
3      this.director = director ;
4      this.anyo = anyo;
5      codigo = proximoCodigo;
6      proximoCodigo++;
7  }

```

Por último, podríamos implementar el método getter correspondiente:

```

1  /**
2  * Devuelve un entero con el codigo de la pelicula
3  * @return El entero correspondiente
4  */
5  public int getCodigo() {
6      return codigo;
7  }

```

Sobre la representación del tipo de película, quizás has pensado utilizar un entero o un String para representar el tipo de película. Si queremos mantener un rango limitado de tipos, el uso de un String puede ser problemático (un carácter mal introducido es un tipo diferente). Si utilizamos un entero, el usuario de la clase deberá conocer los tipos disponibles y su correspondencia con el parámetro. Nosotros proponemos el uso de un enfoque híbrido: el uso de constantes con nombres que identifiquen cada uno de los géneros disponibles.

Si simplemente utilizáramos un entero, deberíamos añadir el siguiente código a nuestra clase:

```

1  /*
2  * Genero de la pelicula
3  */
4
5  private int genero;
6
7  /**
8  * Devuelve un entero con el codigo del genero de la pelicula
9  * @return El entero correspondiente
10 */
11 public int getGenero(){
12     return genero;
13 }
14
15 /**
16 * Asigna el genero a la pelicula
17 * @param genero
18 */
19 public void setGenero(int genero) {
20     this.genero=genero;
21 }

```

Además, sería conveniente modificar el constructor para incluir el nuevo atributo:

```

1 public Pelicula(String titulo , String director , int anyo, int genero) {
2     this.titulo = titulo ;
3     this.director = director ;
4     this.anyo = anyo;
5     this.genero = genero;
6     codigo = proximoCodigo;
7     proximoCodigo++;
8 }

```

El problema con este enfoque es que el programador que utilice la clase necesitará recordar el valor entero correspondiente a cada género. Para facilitar el trabajo del programador y evitar este problema, podemos utilizar constantes predefinidas. Primeramente definimos una serie de constantes con nombres fáciles de recordar y que se correspondan con los géneros disponibles. La palabra clave `final` sirve para definir constantes en java. Por ejemplo:

```

1 public final static int TERROR=0, CIENCIA_FICCION=1,
2     ACCION=2, SUSPENSE=3, COMEDIA=4;

```

Observa que también hemos utilizado el modificador `static`. De esta forma, las constantes se comparten por todos los objetos de la clase (no sería eficiente replicar y almacenar los mismos valores en todas las instancias). Además, estas constantes se han declarado como públicas, para poder utilizarlas desde fuera de la clase. De este modo, podemos utilizar estas constantes en lugar de su valor. Una vez hecho esto, podemos redefinir el método `main` de la siguiente forma:

```

1 public static void main(String[] argv){
2     Pelicula peli1 , peli2 ;
3     peli1=new Pelicula("2001: Una Odisea en el Espacio","Stanley Kubrick",1968,
4     Pelicula.CIENCIA_FICCION);
5     peli2=new Pelicula("2046","Wong Kar Wai",2004,Pelicula.CIENCIA_FICCION);
6     System.out.println(peli1);
7     System.out.println(peli2);
8 }

```

Observa la forma de hacer referencia a la constante. Como las constantes son de clase, hemos de anteponer el nombre de la clase en lugar del objeto (`Pelicula.`). En este caso, y dado que estamos dentro de la propia clase, no sería necesario (similar a como ocurre con la referencia `this`, estaría implícito).

Si ejecutamos este último método `main`, advertiremos que no se muestra el género de la película. Si intentamos mostrarlo, nos encontraremos con una nueva dificultad: hemos de convertir el valor entero que se almacena en el campo `genero` a una cadena que pueda interpretarse con facilidad. Sin duda, podríamos utilizar una sentencia `switch` para realizar esta operación.

```

1 public String toString(){
2     String cadenaGenero="";
3     switch (genero){
4         case 0: cadenaGenero="Terror";break;
5         case 1: cadenaGenero="Ciencia ficcion";break;
6         case 2: cadenaGenero="Accion";break;
7         case 3: cadenaGenero="Suspense";break;
8         case 4: cadenaGenero="Comedia"; break;
9     }
10    return "Codigo:"+codigo+"\n"+"Pelicula: "+titulo+"\n"+"Director: "+director+
11    "\n"+"Anyo:"+anyo+"\n"+"Genero: "+cadenaGenero+"\n";
12 }

```

También podríamos optar por una solución alternativa, definiendo un array de Strings para representar las cadenas que representan los géneros. Este array almacenaría, en su posición *i* el texto correspondiente a la constante cuyo valor es *i*:

```

1 private static final String[] generos = {
2     "Terror", "Ciencia ficcion", "Accion", "Suspense", "Comedia" };

```

Observa la forma de declarar e inicializar el array (en línea). `String[]` significa que el tipo es una array de Strings. Cada uno de los valores que aparecen entre llaves se asignarían a una posición del array, secuencialmente y empezando desde la posición 0.

Observa también los modificadores aplicados al array `generos`: `private` porque no queremos que se pueda acceder a este vector desde fuera de la clase, `static` porque no queremos replicar el array en cada instancia, y `final` porque no queremos que los valores del array puedan cambiar durante la ejecución.

En este caso, la implementación del método `toString` se simplificaría considerablemente:

```
1 public String toString(){
2     return "Codigo:"+codigo+"\n"+"Película: "+titulo+"\n"+"Director: "+director+
3     "\n"+"Anyo: "+anyo+"\n"+"Genero: "+generos[genero]+"\n";
4 }
```

Además, nos sería muy sencillo implementar un nuevo método para obtener el género de la película en forma de cadena:

```
1 /**
2  * Devuelve una cadena que contiene el genero de la pelicula
3  * @return La cadena
4  */
5 public String getGeneroComoCadena(){
6     return generos[genero];
7 }
```

## 3.2 La clase Coleccion

La clase `Coleccion` es una composición de objetos de la clase `Pelicula`. Una forma habitual de implementar esta relación es incluir un atributo en la clase `Coleccion` que permita almacenar un conjunto de referencias de tipo `Pelicula`. Podemos, por ejemplo, utilizar un array. Una posible implementación utilizando este enfoque se muestra a continuación. Recuerda que las clases `Coleccion` y `Pelicula` deben estar en el mismo paquete. Si estuvieran en diferentes paquetes, deberíamos utilizar una sentencia `import`, que en breve describiremos:

```
1 /**
2  *
3  * @author Miguel
4  */
5 public class Coleccion {
6     private Pelicula [] peliculas;
7     private static final int MAX=100;
8     private int posicionActual=0;
9     /**
10    * Constructor sin parametros
11    */
12    public Coleccion(){
13        peliculas=new Pelicula[MAX];
14    }
15    /**
16    * Anyade una pelicula a la coleccion
17    * @param p La pelicula que debe anyadirse
18    * @return verdadero si la operacion ha tenido exito o falso si no cabe
19    */
20    public boolean add(Pelicula p){
21        if (posicionActual<MAX) {
22            peliculas[posicionActual]=p;
23            posicionActual++;
24            return true;
25        } else
26            return false;
27    }
```

```

28 }
29
30 public String toString(){
31     String resultado="";
32     for (int i=0;i<posicionActual;i++)
33         resultado=resultado+películas[i].toString()+"\n";
34     return resultado;
35 }
36
37
38 public static void main(String[] argv) {
39     Coleccion coleccion=new Coleccion();
40     coleccion.add(new Pelicula("2001: Una Odisea en el Espacio","Stanley Kubrick",1968,
41     Pelicula.CIENCIA_FICCION));
42     coleccion.add(new Pelicula("2046","Wong Kar Wai",2004,Pelicula.CIENCIA_FICCION));
43     System.out.println(coleccion);
44 }
45 }

```

Un inconveniente de este código es definir el tamaño del array. En esta ocasión, se ha utilizado la constante `MAX`, definida con valor 100. El valor de esta constante impone una restricción importante sobre nuestra aplicación, al limitar el número de películas que pueden formar parte de la colección. Además, no facilita la implementación de otros métodos útiles, como el borrado de películas. Para borrar una película en una cierta posición, sería necesario desplazar a la izquierda todas las películas situadas a la derecha de la borrada, una operación poco eficiente desde un punto de vista computacional.

Afortunadamente, el API (Application Programming Interface) de java proporciona diversos contenedores de gran utilidad (similar a las STL -Standard Template Library- de C++). Puedes encontrar la documentación correspondiente en <http://docs.oracle.com/javase/7/docs/api/>. La documentación sobre el API que aparece en esta dirección está realmente bien estructurada. A la izquierda aparecen, en la parte de arriba, todos los paquetes que incluye en API. Justo debajo, las clases pertenecientes al paquete actualmente seleccionado (todas las clases si no se selecciona ningún paquete en concreto). Cuando se pulsa sobre cualquier clase, la documentación referente a la misma: jerarquía, constructores, métodos... aparece en la parte de la derecha. En formato utilizado es el mismo que el generado por la herramienta `javadoc`.

Dos de las clases que aparecen en la documentación del API son `Vector` y `HashSet`, incluidas en el paquete `java.util`. Una diferencia fundamental entre estas dos clases es que la primera admite elementos repetidos, mientras que la segunda no. Por esta razón, hemos decidido utilizar la clase `HashSet`, para así evitar posibles duplicidades en nuestra colección. Utilizando la clase `HashSet`, el código se simplifica notablemente:

```

1 import java.util.HashSet;
2 public class Coleccion {
3     private HashSet<Pelicula> peliculas;
4     public Coleccion(){
5         peliculas=new HashSet<Pelicula>();
6     }
7     /**
8     * Anyade una pelicula a la coleccion
9     * @param p La pelicula que debe anyadirse
10    * @return verdadero si la operacion ha tenido exito o falso si no cabe
11    */
12    public boolean add(Pelicula p){
13        return peliculas.add(p);
14    }
15
16    public String toString(){
17        String resultado="";
18        for (Pelicula p:peliculas)
19            resultado=resultado+p.toString()+"\n";
20        return resultado;

```

```

21 }
22
23 public static void main(String[] argv) {
24     Coleccion coleccion=new Coleccion();
25     coleccion.add(new Pelicula("2001: Una Odisea en el Espacio",
26     "Stanley Kubrick",1968,Pelicula.CIENCIA_FICCION));
27     coleccion.add(new Pelicula("2046","Wong Kar Wai",2004,
28     Pelicula.CIENCIA_FICCION));
29     System.out.println(coleccion);
30 }
31 }

```

En este código, es interesante observar la sintaxis utilizada para acceder a los elementos del `HashSet` `peliculas`. Esta construcción se llama *para-cada-uno* (*for-each*) . Además, obsérvese que para poder utilizar el `HashSet`, ha sido necesario importar la clase definida en el API. Para ello, se utiliza la sentencia `import`, similar al `include` de C/C++. Nótese que es necesario proporcionar el nombre completo de la clase que se quiere importar. Cuando se quieren importar varias clases de un mismo paquete, es posible utilizar un `*`. Por ejemplo, la sentencia `import java.util.*` permitiría utilizar todas las clases definidas en el paquete `java.util`. Sin embargo, no importaría aquellas que estuvieran en “subpaquetes” como `java.util.zip`.

También es interesante el uso parametrizado de la clase `HashSet`. Observa que el tipo de datos que contiene se especifica entre los símbolos `<` y `>`. En este caso particular, hemos especificado que el tipo de datos que almacenará el contenedor es de tipo `Pelicula`. Una vez se ha hecho esto, por ejemplo, el compilador no admitirá que se utilice el método `add` sobre el objeto `peliculas` con un parámetro que no sea de la clase `Pelicula`.

Almacena el código resultante. Te será necesario para la práctica 2.

### 3.3 Punto de entrada a la aplicación

La ejecución de una aplicación java comienza en el método `main` de la clase que se ejecuta. Es una buena práctica definir una clase que contenga únicamente un método `main`, y que constituya el punto de entrada de nuestra aplicación. Veamos un ejemplo de un aplicación sencilla que utilice las clases que hemos implementado. Esta aplicación simplemente solicita datos sobre películas, dando la opción al usuario de continuar en cada iteración. Cuando el usuario ha acabado de introducir los datos, simplemente muestra un listado con las películas introducidas.

```

1 import javax.swing.JOptionPane;
2
3 public class Starter {
4     public static void main(String[] args) {
5         String nombre,director;
6         int anyo,genero,continuar;
7         Coleccion coleccion=new Coleccion();
8         String [] generos={"Terror","Ciencia ficcion ","Accion","Suspense","Comedia"};
9         do {
10             nombre=JOptionPane.showInputDialog(null,"Introduce nombre de la pelicula ");
11             director=JOptionPane.showInputDialog(null,"Introduce el director de la pelicula ");
12             anyo=Integer.parseInt(JOptionPane.showInputDialog(null,
13             "Introduce el anyo de la pelicula " ));
14             genero=JOptionPane.showInputDialog(null,"Introduce genero de la pelicula ","Genero",
15             JOptionPane.DEFAULT_OPTION,JOptionPane.QUESTION_MESSAGE,null,
16             generos,generos[0]);
17             continuar= JOptionPane.showConfirmDialog(null,"Quieres continuar?",
18             "Seleccione la opcion deseada",
19             JOptionPane.YES_NO_OPTION);
20             coleccion.add(new Pelicula(nombre, director, anyo, genero));
21         } while (continuar!=0);
22         JOptionPane.showMessageDialog(null,coleccion);
23     }

```

24 }

En este código, es especialmente relevante la clase `JOptionPane` del paquete `javax.swing`. Esta clase proporciona una serie de métodos estáticos que permiten interactuar con el usuario utilizando cuadros de diálogo. Observa que la llamada a los métodos no se realiza sobre un objeto, sino sobre la propia clase. Los métodos estáticos son similares a las funciones de los lenguajes estructurados. Dado que no se ejecutan sobre un objeto, no es posible utilizar ningún atributo que no sea estático en su implementación.

Otro aspecto significativo se refiere a la posible ocurrencia de errores. Intenta introducir un valor no numérico cuando la aplicación solicite el año y observa los resultados: la aplicación finaliza su ejecución. Lo que ocurre es que el método `parseInt` de la clase `Integer` genera un error que no ha sido tratado. En java, un método puede devolver un error o un dato (nunca ambos), y esta información figura en la documentación del método. En el caso particular del método `parseInt` de la clase `Integer`, la documentación muestra su signature de la siguiente forma:

```
1 public static int parseInt(String s) throws NumberFormatException
```

Esto significa que el método puede devolver un error del tipo `NumberFormatException` o bien un entero. En general, los errores que puede devolver un método aparecen en su signature tras la palabra clave `throws`. Si en la cabecera del método no figura esta palabra clave, es porque el método siempre devuelve un dato (o nada si el tipo de retorno es `void`).

Para poder tratar un error, es necesario incluir una cláusula `try-catch` alrededor de la llamada que puede producirlo. A continuación mostramos un tratamiento básico para el error que nos ocupa (simplemente damos el valor cero al año):

```
1 try {
2     anyo=Integer.parseInt(JOptionPane.showInputDialog(null,
3         "Introduce el anyo de la pelicula "));
4 } catch (NumberFormatException e){ anyo=0; };
```

Indudablemente, serían posibles otros tratamientos más robustos, como solicitar el año hasta que no se produzca error en la conversión. La solución anterior se ha elegido con propósitos ilustrativos por su simpleza. En java, existen dos tipos de excepciones. Todas son subclases de la clase `Exception`, pero existen algunas especiales que derivan de `RuntimeException` (que también es una subclase de `Exception`). La diferencia entre ambos tipos está en la obligatoriedad de utilizar la sentencia `try-catch`, siendo opcional únicamente en este último caso. Precisamente `NumberFormatException` es una subclase de `RuntimeException`. Por esta razón, el código inicialmente propuesto no originó ningún error de compilación.

## 4 Ejercicio 3 – Contenedores, tipos primitivos y clases envoltorio

Crea una clase llamada `Test` con el siguiente código.

```
1 import java. util .HashSet;
2 public class Test {
3     public static void main(String[] argv)
4     {
5         HashSet<int> hs=new HashSet<int>();
6         hs.add(5);
7         hs.add(6);
8     }
9 }
```

Observarás que Netbeans informa de un error en la línea que define el contenedor de tipo `HashSet` `hs`. Muchos métodos de java, y entre ellos casi todos los relacionados con inclusión de datos en contenedores, exigen un objeto como parámetro. Igualmente, los tipos parametrizados requieren una clase como parámetro. Sin embargo `int` es un tipo primitivo, y no un objeto. Para resolver este problema, java proporciona una clase envoltorio (*Wrapper class*) para cada uno de los tipos primitivos existentes: `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Float` y `Double`. Cada una de estas clases proporciona el soporte necesario

para almacenar datos de tipo primitivo en objetos. Además, java realiza conversiones implícitas (no necesitan especificarse explícitamente) entre los tipos primitivos y las clases envoltorio. Por ejemplo, el siguiente código sería válido,

```
1 import java. util .HashSet;
2 public class Test {
3     public static void main(String[] argv)
4     {
5         HashSet<Integer> hs=new HashSet<Integer>();
6         hs.add(5);
7         hs.add(6);
8         int suma=0;
9         for (int numero:hs)
10            suma=suma+numero;
11        System.out.println("El valor de la suma es "+suma);
12    }
13 }
```

y equivalente a:

```
1 import java. util .HashSet;
2 public class Test {
3     public static void main(String[] argv)
4     {
5         HashSet<Integer> hs=new HashSet<Integer>();
6         hs.add(new Integer(5));
7         hs.add(new Integer(6));
8         int suma=0;
9         for (Integer numero:hs)
10            suma=suma+numero.intValue();
11        System.out.println("El valor de la suma es "+suma);
12    }
13 }
```

Puedes observar que, cuando java detecta que se está pasando un tipo primitivo a un método que acepta un objeto, automáticamente convierte el valor a la clase envoltorio correspondiente (en este caso `Integer`). Igualmente, cuando se asigna el valor devuelto por un método que retorna un objeto a un tipo primitivo, ocurre la conversión contraria.

## 5 Ejercicio 4 – Herencia. Construcción de objetos.

Una vez hemos hecho nuestra primera aplicación, revisaremos algunos conceptos de importancia sobre el funcionamiento de la herencia. Como primer ejercicio en esta dirección, predice la salida del siguiente código:

```
1 class Padre {
2     Padre(){
3         System.out.println("Creando Padre");
4     }
5     Padre(int a){
6         System.out.println("Creando Padre con un parametro");
7     }
8 }
9
10 class Hijo extends Padre {
11     Hijo () {
12         System.out.println("Creando Hijo");
13     }
14
15     Hijo (int a) {
16         System.out.println("Creando Hijo con un parametro");
17     }
18 }
```



```

18
19     Hijo(int a, int b) {
20         super(b);
21         System.out.println("Creando Hijo con dos parametros");
22     }
23 }
24
25 public class Starter {
26     public static void main(String[] args) {
27         Hijo h1=new Hijo();
28         Hijo h2=new Hijo(3);
29         Hijo h3=new Hijo(3,3);
30     }
31 }

```

**Nota:** Como la clase pública se llama `Starter`, el archivo que la contiene debería tener este mismo nombre (alternativamente puedes cambiar el nombre de la clase pública para adecuarlo al nombre del archivo). Observa que, aunque el archivo contiene tres clases, solamente una es pública.

Si ahora ejecutas el código, probablemente te sorprenda la salida. En java, la primera acción que realiza un constructor es siempre una llamada al constructor de la superclase. Si ésta no viene especificada explícitamente (como en el caso de los dos primeros constructores de la clase `Hijo`), se ejecuta la llamada implícita `super()`, siempre sin parámetros. Si por el contrario aparece una llamada explícita a `super`, no se ejecutará ninguna otra llamada implícita.

Elimina el constructor sin parámetros de la clase (Padre). Ejecuta de nuevo el programa principal y observa que ocurre.

Como último ejercicio de este apartado se propone realizar una clase (Nieta) que sea una subclase de la clase (Hijo). Implementa un constructor sin argumentos para esta nueva clase que únicamente imprima por pantalla: `Creando Nieta`. ¿A que constructores llamará? Ejecuta de nuevo el programa y observa que ocurre.

## 6 Uso del depurador

El hecho de que un programa compile no implica que no hayan errores que sólo se pueden ver en tiempo de ejecución. NetBeans tiene una herramienta de depuración para encontrar esos errores en tiempo de ejecución. Los pasos para aplicar el depurador son los siguientes


1. **Marcar punto de interrupción:** se hace click con el ratón sobre el número de línea donde queremos parar la ejecución del programa.

2. **Lanzar el programa en modo depuración:** se pulsa el botón "Debug Project" (CTRL+F5) 

3. **Consultar variables:** cuando la ejecución llegue al punto de interrupción se detendrá en ese punto. Se puede aprovechar para consultar el valor de las variables. Esta consulta se puede hacer pasando el ratón por encima de las variables y nos aparecerá su valor con un tooltip (Figura 2) o bien podemos indicar en el watchlist qué variables queremos consultar. Para añadir variables al watchlist se hace click derecho sobre la variable que queremos añadir a la lista, pulsamos sobre "New watch..." la variable nos aparecerá en whatlist junto con su valor actual (Figura 3).

4. **Seguir con la ejecución:** una vez he consultado las variables puedo seguir con la ejecución. Se me plantean varias opciones:

- seguir hasta el siguiente punto de interrupción (o hasta el final si ya no quedan más puntos).



Para ello se pulsa el botón 

```

15 public class Test {
16
17
18 public static void main(String[] argv)
19 {
20     HashSet<Integer> hs=new HashSet<Integer>();
21     hs.add(5);
22     hs.add(6);
23     int suma=0;
24     for ( int numero:hs)
25         suma=suma+numero;
26     mostrarResultado(suma);
27 }
28 public static void mostrarResultado(int suma){
29     System.out.println( "El valor de la suma es "+suma);
30 }
31 }

```

Figura 1: Ejemplo de punto de interrupción en la línea 25

- seguir hacia la siguiente línea pero sin entrar en las invocaciones que pudiera haber. Por ejemplo, no entraríamos dentro del código de "mostrarResultado". Para ello se pulsa el botón .
- seguir hacia la siguiente línea entrando en la siguiente invocación. Por ejemplo, sí entraríamos dentro del código de "mostrarResultado". Para ello se pulsa el botón .

## 7 Ejercicio 5 – Aplicación

Finalmente, se plantea la realización de una aplicación similar a la de gestión de películas presentada en el ejercicio 2. En este caso, la aplicación deberá gestionar el carro de la compra de una aplicación de comercio electrónico. Cada cliente tiene asociados un conjunto de productos que va a comprar. El sistema debe permitir introducir los siguientes datos sobre el cliente y los productos que conforman el carro de la compra:

- un identificador de cliente de 3 dígitos, que se asignará automáticamente y de forma secuencial conforme se crean los clientes.
- nombre
- dirección de envío
- si tiene activo o no el servicio de envío vip (utiliza una variable booleana).

Una vez introducidos los datos del cliente, se introducen los datos de los productos del carro de la compra. Para cada producto, debe almacenarse:

- un identificador de 7 dígitos. Los tres primeros se corresponderán con el identificador del cliente y los 4 siguientes con un identificador propio asignado de forma secuencial conforme se introducen los productos para un cliente (Por ejemplo, 1010001)
- descripción del producto
- cantidad
- precio del producto en Euros (utiliza una variable de tipo float)

Deberás utilizar un objeto de la clase HashSet para almacenar una lista de clientes y los productos asociados a cada uno. Como en el caso de la aplicación de películas, el programa permitirá al usuario introducir primero los datos del cliente y después la lista de productos. La introducción de datos se debe

```

15 public class Test {
16
17
18 public static void main(String[] argv)
19 {
20     HashSet<Integer> hs=new HashSet<Integer>();
21     hs.add(5);
22     hs.add(6);
23     int suma=0;
24     for ( int numero = (int) 5;
25         suma=suma+numero;
26         mostrarResultado(suma);
27     }
28     public static void mostrarResultado(int suma){
29         System.out.println( "El valor de la suma es "+suma);
30     }
31 }

```

Figura 2: Ejemplo de consulta de la variable "numero" al pasar el ratón por encima

Test > main > for (int numero : hs) >			
Variables Breakpoints			
Name	Type	Value	
numero	int	5	

Figura 3: Ejemplo de consulta de la variable "numero" desde el whatchlist

implementar en un bucle que permita introducir tantos clientes y tantos productos por cliente como desee el usuario. Al acabar de introducir la información, se debe mostrar por pantalla el listado de productos por cliente y el importe total (Se debe multiplicar la cantidad por cada precio introducido y sumar todos los productos de un cliente). Implementa únicamente los métodos que te sean necesarios para proporcionar esta funcionalidad.

## 8 Entrega de la Práctica

Para esta práctica se deberá entregar el código del ejercicio 5 debidamente documentado. Dicha entrega se realizará en Aula Virtual antes del comienzo de la siguiente sesión de prácticas.