Tema 10: Estructuras de datos avanzadas

¿Qué aprenderás?

- Usar las estructuras ArrayList y HashMap.
- Usar los métodos típicos para trabajar con estas estructuras.
- Modificar los métodos hashCode y equals del HashMap.

¿Sabías que…?

- Tanto los ArrayList como los HashMap se incluyeron en la especificación 1.2 de Java.
- Estas dos estructuras no pueden trabajar con tipos de datos simples. Debemos usar sus Wrappers, en un proceso que se le conoce como Boxing.
- Normalmente los métodos hashCode y equals se sobreescriben de manera que compartan los criterios para considerar cuándo dos objetos son iguales.

10. Estructuras de datos avanzados

Hasta ahora nuestros programas constaban de variables cuyos tipos de datos nos permitían guardar un único valor alfanumérico. Disponemos también de los arrays y las cadenas de caracteres (String) para poder almacenar varios valores de un mismo tipo. Incluso los objetos nos ofrecen una forma de almacenamiento de información, mediante los atributos.

Estos no son los únicos mecanismos que nos brinda Java para poder almacenar información. Hay otras estructuras avanzadas, como pueden ser las listas, colas, pilas, árboles, etc., cada una de ellas con sus métodos de acceso a los datos. En este capítulo nos centraremos en el funcionamiento de las listas (ArrayList) y colecciones (HashMap).

10.1. La clase ArrayList

Un ArrayList lo podemos comparar con un array estático de los que se han visto en temas anteriores, por el hecho de permitirnos guardar varios valores de un mismo tipo bajo un único nombre de variable. Pero hay una diferencia muy importante entre los array y los ArrayList: éste último es dinámico. Esto quiere decir que su tamaño puede cambiar a lo largo de la ejecución del programa.

Si recordamos los arrays, en su declaración debíamos especificar el tamaño que tendrá, y este no cambiará en todo el programa. Esto puede suponer un gasto innecesario de memoria, o puede ocurrir que nos quedemos cortos al querer añadir un nuevo elemento al array. Con los arrays dinámicos, los ArraysList, este problema desaparece, ya que su tamaño se adaptará a su contenido. Durante la ejecución del programa podemos añadir y eliminar tantos elementos como queramos.

Para usar ArrayList en nuestro programa, tendremos que importar la librería java.util.ArrayList.

10.1.1. Declaración y método add

La declaración del ArrayList es la siguiente:

```
ArrayList miLista = new ArrayList();
```

Declaración de un ArrayList vacío

La instrucción anterior crea un ArrayList vacío. Creándolo de esta forma, el ArrayList puede contener objetos de cualquier tipo. Para añadir elementos, usaremos el método add():

```
miLista.add(“Hola”);
miLista.add(31);
miLista.add(‘d’);
miLista.add(5.5);
```

Adición de elementos en un ArrayList.

El método add añade al final del ArrayList un nuevo elemento. En este ejemplo, el primer objeto que se añade al ArrayList es el String “Hola”. El resto no son objetos. Son datos de tipos básicos pero el compilador los convierte automáticamente en objetos de su clase contenedora antes de añadirlos al ArrayList.

Un ArratList al que se le pueden asignar elementos de distinto tipo puede tener alguna complicación a la hora de trabajar con él. Por eso, una alternativa a esta declaración es indicar el tipo de objetos que contiene. En este caso, el ArrayList sólo podrá contener objetos de ese tipo.

```
ArrayList<tipo> miLista = new ArrayList<tipo>();
```

Declaración de un ArrayList especificando el tipo de datos que contendrá.

En la instrucción anterior, “tipo” debe ser una clase. Indica el tipo de objetos que contendrá el array.

No se pueden usar tipos primitivos. Para un tipo primitivo se debe utilizar su clase contenedora. Por ejemplo, si queremos que el ArrayList contenga enteros:

```
ArrayList<Integer> miLista = new ArrayList<Integer>();
```

Declaración de un ArrayList que guardará números enteros.

Hemos visto que podemos añadir elementos a un ArrayList con el método add. Los elementos nuevos siempre se añadirán al final del ArrayList. Si queremos añadir un elemento en un lugar concreto, podemos usar la misma función add con otros parámetros:

```
miLista.add(0, 1000);
```

Adición de un valor en una posición concreta de un ArrayList.

La instrucción anterior añadirá el entero 1000 en la primera posición del ArrayList (posición 0). Hay que decir que la posición que especifiquemos debe estar comprendida entre 0 y ArrayList.size()-1, sino se producirá una excepción.

10.1.2. Métodos de la clase ArrayList

En el apartado anterior ya hemos visto cómo crear un ArrayList y cómo añadir elementos en él, mediante el método add. Vamos a ver otros métodos igualmente útiles para trabajar con nuestros ArrayList.

Para saber el tamaño del ArrayList, es decir, cuántos elementos tiene actualmente, usaremos el método size:

```
miLista.size();
```

Consulta del tamaño de un ArrayList.

Si queremos acceder al elemento que ocupa la posición x en el ArrayList, usaremos el método get. Éste nos devuelve un objeto del mismo tipo que el que hay en la posición accedida del ArrayList.

```
ArrayList<String> listaPalabras = new ArrayList<String>();
…
String palabra = listaPalabras.get(6);
```

Declaración de un ArrayList y acceso a la posición 6.

La instrucción anterior devuelve el String que hay almacenado en la posición 6 del ArrayList, y lo asigna a una variable “palabra”. Hay que tener en cuenta que la primera posición de un ArrayList es la 0. Por tanto, en el método get las posiciones válidas van de 0 a ArrayList.size()-1. Cualquier posición fuera de este rango lanzará una excepción.

A modo de resumen, en la siguiente tabla podemos ver los principales métodos de la clase ArrayList, algunos de ellos ya comentados:

size()	Devuelve un entero conel número de elementos del ArrayList.
add(X)	Añade el objeto X al final del ArrayList.
add(posición, X)	Inserta el objeto X en la posición indicada del ArrayList.
get(posición)	Devuelve el elemento que está en la posición indicada.
remove(posición)	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.

remove(X)	Elimina la primera ocurrencia del objeto X. Devuelve true si el elemento está en la lista.
clear()	Elimina todos los elementos del ArrayList.
set(posición, X)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X. Devuelve el elemento sustituido.
contains(X)	Comprueba si la colección contiene al objeto X. Devuelve true o false.
indexOf(X)	Devuelve la posición del objeto X. Si no existe devuelve -1.

10.2. La clase HashMap

Un HashMap es una implementación de la interfaz Map de Java. Es una colección de objetos. El HashMap nos permite almacenar pares clave/valor, de tal manera que una clave sólo puede tener un valor, es decir, las claves no se repiten. Si añadimos un elemento cuya clave ya existe, no se generará un nuevo elemento en el HashMap, sino que se reescribe el valor que pertenece a la clave existente. Hay que tener en cuenta que el HasMap no admite valores nulos.

La particularidad de los HashMap radica en cómo se ordenan sus elementos. Se utiliza un algoritmo que, a partir de la clave, genera otra clave "hash", que será la que determine el orden en que se almacenarán en memoria los elementos.

Esta forma de ordenar los objetos hace que el tiempo de ejecución de operaciones como coger un elemento o añadir uno nuevo permanezca constante independientemente del tamaño del HashMap.

10.2.1. Declaración

Para usar HashMap en nuestro programa, tendremos que importar la librería java.util.HashMap. La declaración del HashMap es la siguiente:

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
```

Declaración de un HashMap en que las claves serán de tipo String y los valores de tipo Integer.

La instrucción anterior crea un HashMap cuyos elementos tendrán una clave de tipo String y un valor de tipo Integer. Como pasa con los ArrayList, en los HashMap no se pueden usar tipos primitivos (int), se debe utilizar su clase contenedora (Integer).

Podemos declarar un HashMap sin especificar el tipo de datos de las claves y los valores a almacenar. Esto nos permite añadir cualquier tipo de objeto, lo cual no es recomendable, pues para mostrar los datos del HashMap podemos tener problemas de conversión de tipos de datos.

10.2.2. Añadir elementos a un HashMap

Una vez se ha creado un HashMap, vamos a introducir elementos en él. Para ello usaremos el método put de la siguiente manera:

```
map.put("Pedro", 12345);
map.put("Luis", 45321);
map.put("Susana", 99999);
map.put("Maria", 34521);
map.put("Manuel", 123);
map.put("Manuel", 12333);
```

Adición de claves y valores en un HashMap cpn el método put.

Siguiendo con el HashMap creado de ejemplo, con un String como clave y un Integer como valor, las líneas de código anterior almacenarían 5 nuevos elementos en el HashMap. Observad que los dos últimos elementos comparten la misma clave “Manuel”. Recordemos que el HashMap no contiene claves repetidas, por tanto, al ejecutarse la última instrucción, no se crea un nuevo elemento en el HashMap, sino que se modifica el valor del elemento con la clave “Manuel”.

El método put recibe dos parámetros, el primer indica la clave y el segundo el valor del objeto que se va a insertar en el HashMap. Los tipos de ambos parámetros deben concordar con los empleados en la declaración del HashMap. Si no se produce un error de compilación.

10.2.3. Recorrer los elementos de un HashMap

Una operación típica que se hace con los HashMap es recorrerlos y mostrar los datos que almacena. El siguiente fragmento de código lo hace:

```
Iterator it = map.keySet().iterator();
while(it.hasNext()){
        String key = (String) it.next();
        System.out.println("Clave: " + key + " -> Valor: " + map.get(key));
}
```

Uso de la clase Iterator para recorrer los elementos de un HashMap.

Para recorrer el HashMap se ha declarador un iterador en la primera instrucción. La interfaz Iterator nos ofrece un mecanismo para acceder secuencialmente a los objetos de una colección. Su uso extendido radica en la gran variedad de tipologías de colecciones, además de que los elementos almacenados en ellas no son homogéneos.

Volviendo al código de ejemplo, el iterador lo creamos sobre una colección que contiene las claves de los elementos del HashMap. La función keySet() nos devuelve el conjunto de claves que hay en el HashMap.

Para movernos por la colección de claves usando el iterador creado, se usa un bucle while con la condición de entrada "it.hasNext()". El método hasNext() de la interfaz Iterator devuelve true si la iteración tiene más elementos.

Dentro del bucle, movemos el iterador al elemento siguiente con la instrucción it.next(). Como estamos recorriendo la colección de claves del HashMap, y éstas son de tipo String, almacenamos el contenido del iterador en una variable "key" de tipo String.

Para acceder al valor de cada elemento del HashMap usamos el método get(key). Como parámetro debemos especificar la clave que queremos buscar en el HashMap. El método nos devolverá el valor asociado a dicha clave, si existe, nulo en otro caso.

Si se ejecuta el código anterior, se puede observar que los datos se muestran sin un orden previsible, como podría ser alfabéticamente. Esto es debido al algoritmo usado por el HashMap para almacenar eficientemente sus elementos.

10.2.4. Métodos de la clase HashMap

En la siguiente tabla podemos ver los principales métodos de la clase HashMap:

clear()	Vacía el HashMap.
containsKey(key)	Indica si el HashMap contiene un elemento con la clave key.
containsValue(value)	Indica si el HashMap contiene un elemento con el valor value.
entrySet()	Devuelve una copia del HashMap en forma de colección. Cualquier cambio en la colección afecta directamente al HashMap.
get(key)	Devuelve el valor asociado a la clave key, o nulo si no existe la clave.
isEmpty()	Indica si el HashMap está vacío.
keySet()	Devuelve una colección con las claves del HashMap.

put(key, value)	Añade un elemento al HashMap con la clave key y el valor value. Si ya existe un elemento con clave key, sustituye su valor a value.
putAll(map)	Copia todos los elementos de map en el HashMap que invoca el método.
remove(key)	Elimina el elemento con clave key del HashMap si existe.
size()	Devuelve un número entero que es el número de elementos que tiene el HashMap
values()	Devuelve una colección con los valores que contiene el HashMap.

10.2.5. Métodos equals y hashCode

Estos dos métodos son los que debemos usar si queremos tener control sobre cuándo dos elementos se consideran iguales. Ambos métodos son proporcionados por la interfaz Map, los cuales tendremos que sobrescribir para programar el comportamiento que queremos que tengan.

El método equals nos devuelve un booleano que indica si dos elementos son iguales o no. Por defecto, se considera que dos elementos son iguales si tienen la misma clave. Podemos ir más allá y hacer que, por ejemplo, dos elementos se consideren iguales si tienen el mismo valor. Por ejemplo, consideramos dos personas iguales si tienen el mismo nombre y apellidos.

El método hashCode viene a hacer lo mismo que el método equals, es decir, compara dos elementos y nos dice si son iguales. Pero la comparación la hace más rápida, ya que utiliza un número entero. Dos objetos que sean iguales devuelven el mismo valor hash.

Cuando comparamos dos objetos en estructuras de tipo hash (HashMap, HashSet, etc.) primero se invoca al método hashCode y luego al equals. Si los métodos hashCode de cada objeto devuelve un entero diferente, se consideran los objetos distintos. En caso contrario, en que ambos objetos tengan el mismo código hash, Java ejecutará el método equals y revisará en detalle si se cumple la igualdad.

Veamos un ejemplo en el que creamos la clase Persona e implementamos los métodos hashCode y equals:

```
public class Persona {
        String nombre;
        String apellido;
        int edad;
        public Persona(String nombre, String apellido, int edad) {
                this.nombre = nombre;
                this.apellido = apellido;
                this.edad = edad;
        }
        //Getters y setters
        …
}
```

Clase Persona.

Vemos que tenemos los atributos nombre, apellido y edad. Vamos a considerar que dos personas son iguales si tienen el mismo valor en los atributos nombre y apellidos. Implementamos el método hashCode:

```
@Override
public int hashCode() {
        int result = this.nombre.hashCode() + this.apellido.hashCode();
        return result;
}
```

Ejemplo de método hashCode en la clase Persona.

Lo que hacemos es declarar un entero, cuyo valor vendrá de sumar el número hash del nombre y del apellido. Se puede ver que invocamos el método hashCode sobre los Strings nombre y apellidos. La clase String tiene el método hashCode implementado, como todas las clases de Java (lo heredan de la clase Object). Lo que nosotros podemos hacer es sobreescribirlo, como estamos haciendo en la clase Persona.

El método equals considerará los mismos criterios que hashCode, es decir, dos personas son iguales si tienen el mismo nombre y apellido.

```
@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	Persona other = (Persona) obj;
	if (apellido == null) {
		if (other.apellido != null)
			return false;
	} else if (!apellido.equals(other.apellido))
		return false;
	if (nombre == null) {
		if (other.nombre != null)
			return false;
	} else if (!nombre.equals(other.nombre))
		return false;
	return true;
}
```

Ejemplo de método equals en la clase Persona.

Test de autoevaluación

¿Qué hace la siguiente instrucción? ArrayList lista = new ArrayList()

a) Se crea un ArrayList de una posición

b) Se crea un ArrayList que puede almacenar cualquier tipo de objeto

c) Da un error de compilación, ya que no se ha especificado el tipo de dato que almacenará el ArrayList

d) Da un error de compilación, ya que no se ha especificado un tamaño inicial para el ArrayList

¿Cuál de las siguientes es una declaración errónea de un ArrayList?

a) ArrayList lista = new ArrayList()

b) ArrayList<String> lista = new ArrayList<String>()

c) ArrayList<Double> lista = new ArrayList<Double>()

d) ArrayList<int> lista = new ArrayList<int>()

¿Qué ocurre si intentamos añadir a un HashMap un elemento con una clave que ya existe?

a) Se modificará el valor del elemento con la clave que se intenta añadir

b) Se inserta a continuación del elemento con la misma clave

c) No se inserta nada

d) Se produce un error

Recursos y enlaces

- Clase ArrayList Java 10:
  https://docs.oracle.com/javase/10/docs/api/java/util/ArrayList.html

- Clase HashMap Java 10:
  https://docs.oracle.com/javase/10/docs/api/java/util/HashMap.html

Conceptos clave

- **ArrayList**: estructura que permite almacenar varios datos del mismo tipo. El tamaño de esta estructura es variable en función del número de elementos que contenga en cada momento.
- **HashMap**: colección de objetos que almacena pares clave/valor. Podemos acceder a los elementos del HashMap mediante la clave, ya que éstas tienen un valor único.
- **hashCode**: es un valor numérico entero que sirve para identificar un objeto. En las colecciones de tipo hash tiene especial importancia para identificar objetos iguales de forma rápida.
- **Método equals**: método que compara dos objetos y devuelve un booleano indicando si son iguales o no.

Ponlo en práctica

Actividad 1

Crea una aplicación que use un ArrayList para almacenar números enteros y añade varios valores.

A continuación, muestra el número total de elementos del ArrayList, y recórrelo.

Luego, pide al usuario que introduzca un valor para buscarlo en el ArrayList. Si lo encuentra, el programa preguntará si se quiere cambiar el valor. Si el usuario contesta afirmativamente, se le pedirá el nuevo valor y se introducirá en el ArrayList modificando el anterior.

Al final, se debe volver a mostrar el número total de elementos del ArrayList, y cuáles son.

Los solucionarios están disponibles en el aula virtual.

Actividad 2

Crea la clase Alumno con los atributos nombre, apellidos y edad. Deberás implementar el constructor, todas los getters y setters, el método toString y los métodos hashCode y equals. El código hash de cada alumno se calculará sumando el código hash del nombre, el código hash del apellido y la edad. Por otro lado, dos alumnos se consideran iguales si tienen el mismo valor en sus atributos.

Crea una aplicación que use un HashMap para almacenar los alumnos (clave) y el curso al que van (valor).

Añade varios alumnos al HashMap y luego muestra el número total de alumnos y sus valores (datos del alumno y curso al que van).

Crea un nuevo alumno cuyos datos coincidan con alguno de los ya dados de alta en el HashMap, pero en un curso diferente, e intenta insertarlo.

Comprueba que el HashMap sigue teniendo el mismo tamaño y muestra todos los datos que contiene.

Los solucionarios están disponibles en el aula virtual.

Tema 11: Control de excepciones

¿Qué aprenderás?

- Gestionar los errores que se pueden producir cuando se ejecutan las aplicaciones.
- Crear tus propios tipos de errores.
- Dar diferentes soluciones a los errores de ejecución de la aplicación según dónde se produzcan y se capturen.

¿Sabías que…?

- La mayoría de los errores en un programa se originan en las primeras fases del desarrollo. Conforme avanza el proyecto, los errores suelen ser más costosos de arreglar.
- El mecanismo de control de excepciones de Java se amplió considerablemente con el lanzamiento del JDK 7 con las siguientes mejoras: (1) gestión automática de liberación de recursos, como archivos cuando dejan de ser necesarios; (2) la captura múltiple mediante la cual un mismo catch puede capturar dos o más excepciones; (3) regeneración más precisa.
- “El testing puede probar la presencia de errores, pero no la ausencia de ellos” (Edsger Dijkstra).

11. Control de excepciones

Durante la ejecución de nuestros programas se pueden producir diversos errores, que deberemos controlar para evitar fallos. Algunos ejemplos de estos errores en tiempo de ejecución son:

- El usuario introduce datos erróneos (letras cuando se esperan números).
- Se intenta realizar una división entre cero.
- Se quiere abrir un archivo que no existe.

Todas estas situaciones se pueden controlar para que el programa no se cierre de manera inesperada, y actúe de la forma que nosotros le digamos.

Los errores antes comentados son lo que se conoce como excepciones en Java.

Java clasifica las excepciones en dos grupos:

- Error: producidos por el sistema y de difícil tratamiento.
- Exception: errores que se pueden llegar a dar por diversas situaciones, y que se pueden controlar y solucionar (formato numérico incorrecto, ausencia de fichero, error matemático, etc.).

11.1. Captura de excepciones

En nuestros programas debemos identificar situaciones en que se puedan producir errores, con el fin de desviar el flujo de ejecución y realizar las operaciones necesarias para solucionar el problema.

Esto se consigue con los bloques try/catch. La sintaxis es la siguiente:

```
try{
        /*Instrucciones que pueden producir un error*/
}
catch(TipoExcepcion e){
        /*Instrucciones a ejecutar si se da algún error en alguna de
        las instrucciones que hay en try de tipo TipoExcepcion*/
}
finally{
        /*Instrucciones que se ejecutarán siempre*/
}
```

Sintaxis de los bloques try-catch-finally.

En el momento en que se produjera un error de tipo TipoExcepcion en alguna de las instrucciones del bloque try, pasaría a ejecutarse el bloque catch.

Si no hay error en las instrucciones del bloque try, las del bloque catch no se ejecutan.

Por cada bloque try podemos poner varios bloques catch asociados. Así podremos dar diferente trato a ciertos tipos de excepciones, ejecutando código distinto. También podemos poner un único catch que capture todos los tipos de excepciones, usando la clase Exception, que es la clase de la que heredan el resto de excepciones.

En caso de poner varios bloques catch, el que capture la clase Exception deberá ir en último lugar, ya que, de otra forma, no se llegarían a ejecutar nunca las excepciones tratadas en los bloques catch siguientes.

Al final de los bloques catch podemos poner opcionalmente un bloque finally. En él incluimos instrucciones que queremos que se ejecuten siempre, independientemente de si se ha producido un error o no.

Veamos un programa de ejemplo que usa el bloque try-catch vista antes:

```
import java.io.*;
public class Excepciones {
    public static void main(String[] args) {
            int dividendo, divisor, result;
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            try{
                System.out.println("Introduce el dividendo de la división: ");
                dividendo = Integer.parseInt(br.readLine());
                System.out.println("Introduce el divisor de la división: ");
                divisor = Integer.parseInt(br.readLine());
                result = dividendo/divisor;
                System.out.println("La división" + dividendo + " entre " + divisor + " da " + result);
            }
            catch(ArithmeticException e){
                System.out.println("No se puede dividir entre 0");
            }
            catch(NumberFormatException e){
                System.out.println("Has insertado letras en vez de números");
            }
            catch(Exception e){
                System.out.println("Se ha producido un error inesperado.");
            }
            finally{
                System.out.println(“Gracias por utilizar este programa.");
            }
    }
```

Ejemplo uso try-catch-finally.

Es importante poner el bloque catch correspondiente al tipo general Exception en último lugar. Así nos aseguramos que se ejecutarán los bloques anteriores en caso de error (siempre que el tipo de error corresponda con el de algún bloque catch).

Otra forma de capturar excepciones es poner las palabras throws Exception en la cabecera de un método (en el main por ejemplo). De esta forma no necesitamos poner los bloques try-catch, pero cuando se produzca un error éste no será capturado, y no podremos tratarlo como deseemos.

Un método puede lanzar más de una excepción:

```
public static void main(String args[]) throws IOException, NumberFormatException, …
```

Cabecera de función que puede lanzar varios tipos de excepciones

11.2. Creación de excepciones propias

Java nos proporciona una estructura de clases para tratar diversos tipos de excepciones. Exception es la superclase que gestiona cualquier tipo de excepción. De ella heredarán sus métodos un gran número de subclases encargadas en la gestión de excepciones específicas.

El programador puede crear sus propias clases de excepciones para tratar errores específicos que Java no contempla. Para ello hay que usar el concepto de herencia sobre la clase Exception. Normalmente, la nueva clase contendrá un único constructor con un parámetro de tipo String, que será el mensaje de error que queremos mostrar. Este mensaje lo mostraremos con el método getMessage() de la clase Exception.

Veamos un ejemplo de programa que crea y lanza excepciones propias:

```
public class MiExcepcion extends Exception{
    public MiExcepcion (String mensaje){
            super(mensaje);
    }
}
```

```
import java.io.*;
public class Main {
    public static void main(String[] args) {
            int nota;
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            try{
                System.out.println("Introduce una nota entre 0 y 10:");
                nota = Integer.parseInt(br.readLine());
                if(nota<0 || nota>10){
                        throw new MiExcepcion("Nota fuera de rango");
                }
            }
            catch(MiExcepcion e){
                System.out.println(e.getMessage());
            }
            catch(Exception e){
                System.out.println(e.getMessage());
            }
    }
}
```

Ejemplo de creación y uso de una excepción propia

11.3. Propagación de excepciones

Cuando se produce una excepción dentro de un método, se puede capturar dentro de éste y tratarlo. De no hacerse, la excepción se propaga automáticamente al método que lo llamó, y así sucesivamente hasta que un método captura la excepción o llegamos al main.

IMPORTANTE: los errores de la clase Exception no se propagan.

Observemos el siguiente ejemplo. ¿Qué ocurre cuando se produce un error de tipo NumberFormatException y no lo capturamos en el método leerNumero? Saldrá el mensaje de error que tenemos en el bloque catch en el main.

```
public static void main(String[] args) {
    int num;
    try{
            System.out.println("Introduce un número: ");
            num = leerNumero();
            System.out.println("El número es " + num);
    }
    catch(NumberFormatException e){
            System.out.println("Has insertado letras en vez de números");
    }
}
static int leerNumero(){
    int numero = 0;
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    try{
            numero = Integer.parseInt(br.readLine());
    }
    catch(IOException e){
            System.out.println("ERROR");
    }
    return numero;
}
```

Código de ejemplo de propagación de excepciones

¿Qué ocurrirá ahora si capturamos la excepción en el método leerNumero? En este caso no debe ejecutarse el bloque catch del main, por tanto, se ejecutará todo el código del try, incluido el System.out.println. Se mostrará un mensaje cuando no debería hacerse.

```
public static void main(String[] args) {
    int num;
    try{
            System.out.println("Introduce un número: ");
            num = leerNumero();
            System.out.println("El número es " + num);
    }
    catch(NumberFormatException e){
            System.out.println("Has insertado letras en vez de números");
    }
}
static int leerNumero(){
    int numero = 0;
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    try{
            numero = Integer.parseInt(br.readLine());
    }
    catch(Exception e){
            //CAPTURAMOS EN ESTA FUNCIÓN CUALQUIER TIPO DE EXCEPCIÓN
            System.out.println("ERROR");
    }
    return numero;
}
```

Código de ejemplo de propagación de excepciones (II)

Con estos ejemplos vemos la utilidad de no capturar el error dentro del método donde se produce, sino dejarlo que se propague, debido a la necesidad de tener que controlarlo fuera.

Test de autoevaluación

Al crear excepciones propias, ¿qué forma tendrá el constructor?

a) No tendrá constructor, se usa el de la clase Exception

b) Un parámetro de tipo String y una llamada a la función getMessage de la clase Exception

c) Una llamada a la función System.out.println con el texto que queramos. No recibe ningún parámetro

d) Un parámetro de tipo String y una llamada a la función System.out.println para mostrar el String parámetro

¿Qué ocurre si no se captura una excepción en un método?

a) Que se propaga al método que lo llamó

b) Que se produce un error de compilación

c) Que ese tipo de excepción no se contempla en el programa

d) Que al ejecutarse, el método que lo llama da un error en tiempo de ejecución

Recursos y enlaces

- Clase Exception Java 10:
  https://docs.oracle.com/javase/10/docs/api/java/lang/Exception.html

- Clase Error Java 10:
  https://docs.oracle.com/javase/10/docs/api/java/lang/Error.html

Conceptos clave

- **Exception**: clase de Java que se encarga de la gestión de los errores en tiempo de ejecución que se pueden producir en los programas.
- **Propagación de excepciones**: cuando se produce una excepción y no se captura, ésta se propagará hacia el método que hizo la llamada. Esto se hará hasta llegar a un método que capture la excepción, o hasta el main.

Ponlo en práctica

Actividad 1

Crear un programa que pida los datos de dos alumnos, que son: nombre (letras), edad (entero) y altura (decimal). Se debe realizar un control de la entrada de datos, de tal forma que, si el usuario introduce números en el nombre, o letras en la edad o la altura, se vuelva a pedir el dato correspondiente hasta que la entrada sea correcta.

NOTA: puedes usar el método “matches” de la clase String para ver si una cadena de caracteres posee números o no.

Los solucionarios están disponibles en el aula virtual.

Actividad 2

Modificar el programa anterior de manera que, si el usuario comete más de 5 errores al introducir datos, se muestre un mensaje informando de este hecho y se cierre el programa.

Los solucionarios están disponibles en el aula virtual.

Tema 12: Almacenamiento de objetos en ficheros. Serialización

¿Qué aprenderás?

- Guardar objetos en ficheros como una secuencia de bytes
- Recuperar esta secuencia de bytes y transformarlas en objetos.

¿Sabías que…?

- Con la serialización podemos crear objetos sin ejecutar el constructor de la clase.
- El concepto marshalling se suele usar como sinónimo de serialización, pero no es lo mismo. Con el marshalling se almacena el estado de un objeto junto con su código.

12. Almacenamiento de objetos en ficheros. Serialización

En lecciones anteriores has visto las herramientas que ofrece Java para trabajar con ficheros. Has visto su utilidad para ofrecer persistencia y poder así guardar los datos con los que trabaja nuestra aplicación. Estos datos acostumbraban a ser cadenas de caracteres y otros tipos de datos simples.

Ahora hemos hecho aplicaciones que trabajan con clases y objetos. ¿Es posible guardar un objeto entero en un fichero? La respuesta es sí. Para ello vamos a ver el concepto de serialización.

12.1. Serialización

La serialización es un proceso mediante el cual un objeto es transformado en una secuencia de bytes que representa el estado de dicho objeto, es decir, el valor de sus atributos, para luego ser guardado en un fichero, ser enviado por la red, etc. Un objeto serializado se puede recomponer luego sin ningún problema.

Para hacer que una clase pueda ser serializable hay que hacer que implemente la interfaz Serializable. Esta interfaz no define ningún método, por lo que no tendremos que implementar nada nuevo en nuestra clase.

```
import java.io.Serializable;
public class Empleado implements Serializable{
    private String nombre;
    private int edad;
    private double sueldo;

    public Empleado(String nombre, int edad, double sueldo) {
            super();
            this.nombre = nombre;
            this.edad = edad;
            this.sueldo = sueldo;
    }
    public String getNombre() {
            return nombre;
    }
    public void setNombre(String nombre) {
            this.nombre = nombre;
    }
    public int getEdad() {
            return edad;
    }
    public void setEdad(int edad) {
            this.edad = edad;
    }
    public double getSueldo() {
            return sueldo;
    }
    public void setSueldo(double sueldo) {
            this.sueldo = sueldo;
    }
    @Override
    public String toString() {
            return "Empleado [nombre=" + nombre + ", edad=" + edad + ", sueldo=" + sueldo + "]";
    }
}
```

Declaración de una clase serializable

Una vez la clase se ha marcado como serializable, ya podemos guardar los objetos de esta clase en un fichero como una secuencia de bytes. Java se encarga de este proceso.

12.1.1. Clase ObjectOutputStream y clase ObjectInputStream

La clase ObjectOutputStream es la que usaremos para guardar objetos en un fichero. Por su parte, la clase ObjectInputStream será la encargada de recuperar los datos de los objetos del fichero.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Principal {
    public static void main(String[] args) throws FileNotFoundException, IOException,
    ClassNotFoundException {
            Empleado e1 = new Empleado("e1", 44, 30000);
            File archivo = new File("Empleados");
            //ESCRIBIR EN EL ARCHIVO
            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(archivo));
            oos.writeObject(e1);
            oos.close();
            //LEER EL ARCHIVO
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream(archivo));
            Empleado e2 = (Empleado) ois.readObject();
            ois.close();
            System.out.println(e2);
    }
}
```

Ejemplo de escritura y lectura de un objeto serializable en un fichero

En el código anterior vemos que creamos un objeto de la clase Empleado que se ha implementado antes y que se ha definido como serializable. Es importante destacar este hecho ya que, si intentásemos guardar en un fichero un objeto que no se haya definido como serializable, obtendríamos un error de compilación.

Para guardar el objeto en el fichero “Empleados” se crea un objeto de la clase ObjectOutputStream sobre el fichero, y se llama al método writeObject, pasándole como parámetro el objeto a guardar.

Para leer el objeto del fichero, se crea un objeto de la clase ObjectInputStream, y se obtienen los datos del empleado con el método readObject. Para realizar la asignación se hace un casting indicando entre paréntesis que los datos leídos deben convertirse en un objeto de tipo Empleado.

No hay que olvidarse de cerrar los flujos de salida y entrada de datos una vez ya hemos escrito o leído todo lo que queríamos. Para ello usamos el método close que está implementado tanto en la clase ObjectOutputStream como en la clase ObjectInputStream.

12.1.2. El modificador transient

En algunas ocasiones querremos que uno o varios atributos de una clase serializable no se incluyan en la secuencia de bytes que representa el estado del objeto. Es decir, no queremos que se guarde el valor de un atributo. Pensemos, por ejemplo, en el caso de guardar los datos de los usuarios en un fichero. ¿Debemos guardar también sus contraseñas?

Para que el valor de un atributo no se guarde cuando serializamos un objeto usaremos el modificador transient.

```
import java.io.Serializable;
public class Empleado implements Serializable{
    private String nombre;
    private int edad;
    private double sueldo;
    private transient String clave = "12345";
    public Empleado(String nombre, int edad, double sueldo) {
            super();
            this.nombre = nombre;
            this.edad = edad;
            this.sueldo = sueldo;
    }
    …
}
```

Ejemplo de uso del modificador transient

En el código anterior vemos el atributo clave de la clase Empleado declarado como transient. Esto hace que, en caso de que un empleado se serializase, no se guardaría el valor del atributo clave.

12.1.3. Serialización a medida

Hemos visto que Java se encarga del proceso de serialización de un objeto, es decir, del proceso de convertirlo en una secuencia de bytes. Pero también tenemos la posibilidad de controlar nosotros esta acción. Para ello disponemos de los métodos readObject y writeObject que tendremos que añadir en la clase serializable e implementar en ellos las acciones que queremos que se realicen en el proceso de serialización del objeto.

Test de autoevaluación

¿Qué clases utilizaremos para el almacenamiento y la recuperación de objetos en ficheros?

a) DataInputStream y DataOutputStream
b) FileInputStream y FileOutputStream
c) FilterInputStream y FilterOutputStream
d) ObjectInputStream y ObjectOutputStream

¿Qué tendremos que añadir en la cabecera de las clases que queremos serializar?

a) extends java.io.Serializable
b) implements java.io.Serializable
c) extends java.io.transient
d) implements java.io.transient

¿Qué modificador se utiliza para hacer que un atributo de una clase no sea serializable?

a) protected serializable
b) static-serializable
c) no-serializable
d) transient

Recursos y enlaces

- Documentación oficial sobre la serialización: https://docs.oracle.com/javase/10/docs/api/java/io/Serializable.html

Conceptos clave

- **Serialización:** proceso mediante el cual un objeto es convertido en una secuencia de bytes para poder ser guardado en un fichero, enviado por la red, etc.
- **Serializable**: interface que hace que los objetos de una clase puedan ser serializados.
- **transient**: modificador que se puede añadir a un atributo de una clase serializable que hace que el valor de éste no se incluya en la secuencia de bytes del objeto.

Ponlo en práctica

Actividad 1

Crea la clase Contacto con los atributos nombre (cadena de caracteres) y teléfono (long). Implementa el constructor, los métodos getters y setters, y el método toString. Esta clase debe ser serializable.

En el programa principal, crea una ArrayList y almacena en él 4 contactos. A continuación, deberás crear el archivo “Agenda” y recorrer el ArrayList, guardando los objetos que contiene dentro del archivo.

Por último, lee el archivo para recuperar los contactos, y muestra por pantalla el total de contactos recuperados y sus datos.

Los solucionarios están disponibles en el aula virtual.

Tema 13: Creación de interfaces de usuario

¿Qué aprenderás?

- Crear aplicaciones con interfaz gráfica: ventanas, botones, cajas de texto…
- Programar eventos a determinadas acciones sobre componentes gráficos.

¿Sabías que…?

- La primera interfaz gráfica de usuario (GUI) fue presentada con el ordenador personal XEROX Alto en el año 1973.
- La librería Swing nació en 1997 y se incluyó en la versión 1.2 de Java.
- Inicialmente, Java contaba con el kit AWT para implementar la interfaz gráfica. Swing surgió más adelante como respuesta a las deficiencias de AWT.

13. Creación de interfaces de usuario

En los capítulos anteriores hemos ido haciendo programas basados en la consola. Esto quiere decir que no usan interfaz gráfica de usuario (GUI). Pese a que los programas hechos en la consola son completamente operativos, la mayoría de aplicaciones tienen una interfaz gráfica. En este capítulo, veremos un kit de herramientas GUI que ofrece Java, llamado Swing.
Cabe destacar que Swing ofrece una amplia variedad de componentes, por lo que en este capítulo se tratará el tema de forma superficial. Será un punto de partida para entender Swing y saber utilizarlo, siendo capaces de crear aplicaciones con una interfaz gráfica completa y funcional.

13.1. Componentes

Los componentes en Swing son clases que representan un elemento visual independiente: un botón, un campo de text, etc.

En este documento vamos a ver algunos de los componentes que ofrece Swing, los que se suelen usar más. Son tantas las posibilidades de Swing, que ver todo en detalle daría para un libro aparte.

JButton	Botón
JLabel	Etiqueta de texto
JTextField	Cuadro de texto
JCheckBox	Casilla de verificación
JRadioButton	Botón de opción
JComboBox	Lista desplegable

Todas las clases de componentes empiezan por la letra J. Para incorporar alguno de estos componentes en nuestro programa simplemente se debe instanciar un objeto de uno de estas clases. Su uso no difiere del resto de objetos ya vistos en Java. Más adelantes veremos un ejemplo completo de creación de una aplicación que usará interfaz gráfica y se mostrará el código para crear diversos componentes y cómo editarlos a través de sus propiedades y métodos.

Hay un grupo de componentes especiales que sirven para agrupar otros componentes, y poder así organizarlos dentro de nuestra ventana: los contenedores.

13.1.1. Contenedores

Un contenedor es un tipo especial de componente cuyo propósito es agrupar un conjunto de componentes. Todas las GUI de Swing deben tener como mínimo un contenedor que almacenará el resto de componentes de nuestra interfaz gráfica. Sin un contenedor, el resto de componentes no se pueden mostrar.

Swing ofrece varios tipos de contenedores. Podemos clasificarlos en dos tipos: los de nivel superior (o pesados) y los de nivel inferior (o ligeros).

Los contenedores de nivel superior ocupan el primer lugar en la jerarquía, es decir, no se incluyen en ningún otro contenedor. Es más, en nuestra GUI siempre debemos empezar por un contenedor de nivel superior que incluirá el resto de componentes.

Entre los contenedores de nivel superior podemos destacar:

- JFrame: implementa una ventana. Sería la ventana principal de nuestra aplicación.
- JDialog: implementa una ventana de tipo diálogo. Serían las ventanas secundarias de nuestra aplicación, que se llamarían desde la ventana principal (JFrame).
- JApplet: implementa una zona dentro de un Applet donde poder incluir componentes de Swing.

El que más usaremos en este curso será JFrame. Podemos destacar varios métodos, de los muchos que tiene la clase JFrame, por el uso que les daremos:

setDefaultCloseOperation(int)	Especifica la acción a realizar cuando cerramos la ventana (pulsando sobre el icono 'X'). Las opciones son: • DO_NOTHING_ON_CLOSE: no hace nada. • HIDE_ON_CLOSE: esconde el JFrame. • DISPOSE_ON_CLOSE: borra el JFrame de memoria. La aplicación puede seguir en funcionamiento si tiene más operaciones ejcutándose. • EXIT_ON_CLOSE: equivale a un System.exit. Es decir, cierra la aplicación entera.
setVisible(boolean)	Hace que la ventana se muestre. Por defecto no lo hace (el booleano es false). Es un método heredado de la clase Window.
setSize(int, int)	Establece el tamaño de la ventana. El primer número especifica el ancho y el segundo el alto. Se puede usar el método pack para ajustar el tamaño al contenido de la ventana. Es un método heredado de la clase Window.

setLocationRelativeTo(Component)	Establece la posición de la ventana relativa al componente especificado. Si se pone "null", la ventana se centra en la pantalla. Es un método heredado de la clase Window.

Los contenedores de nivel inferior, o contenedores ligeros, se usan para organizar grupos de componentes relacionados entre sí. Algunos ejemplos de contenedores ligeros son JPanel, JScrollPane o JRootPane.

A modo de resumen, para tener una idea más clara de cómo funcionan los diferentes tipos de contenedores y cómo se relacionan entre sí, podemos tener una aplicación con una ventana principal, JFrame. Dentro de este contenedor, podríamos tener un JRootPane, que sirve para gestionar otros paneles y además se encarga de gestionar la barra de menús opcional de nuestra aplicación. Luego podríamos tener otros contenedores ligeros para agrupar diferentes componentes de nuestra aplicación, que se podrían ir mostrando u ocultando según ciertos eventos, como podrían ser seleccionar una opción concreta de la barra de menús.

Para terminar con el concepto de contenedor, debemos conocer los administradores de diseño, que sirven para ubicar los componentes gráficos de nuestra aplicación dentro del contenedor.

En la siguiente tabla vemos los administradores de diseño que nos ofrece Swing.

FlowLayout	Los componentes se ubican de izquierda a derecha y de arriba abajo.
BorderLayout	Los componentes se ubican en el centro o los bordes del contenedor. Divide el contenedor en cinco partes (norte, sur, este, oeste y centro)
GridLayout	Los componentes se organizan en una cuadrícula (filas y columnas).
GridBagLayout	Los componentes se organizan en una cuadrícula flexible (un componente puede ocupar más de una fila o columna).
BoxLayout	Los componentes se organizan horizontal o verticalmente en un cuadro.
SpringLayout	Los componentes se organizan con respecto a una serie de restricciones.

No siempre tendremos que usar uno de los layouts anteriores en nuestra aplicación. Una práctica bastante habitual es la de no usar ningún layout, es decir, usar el layout NULL. Con esto tenemos completa libertad para ubicar los componentes en la posición que queramos y con el tamaño deseado.

13.2. Controladores de eventos

En todas las aplicaciones hay componentes que reaccionan a algún evento. Por ejemplo, cuando pulsamos un botón, cuando cerramos una ventana, cuando presionamos la tecla Intro sobre un campo de texto, etc.

Java ofrece una serie de controladores de eventos, llamados listeners, especializados en un evento concreto sobre un componente. En estos listeners podemos implementar las acciones que se llevarán a cabo después de dicho evento.

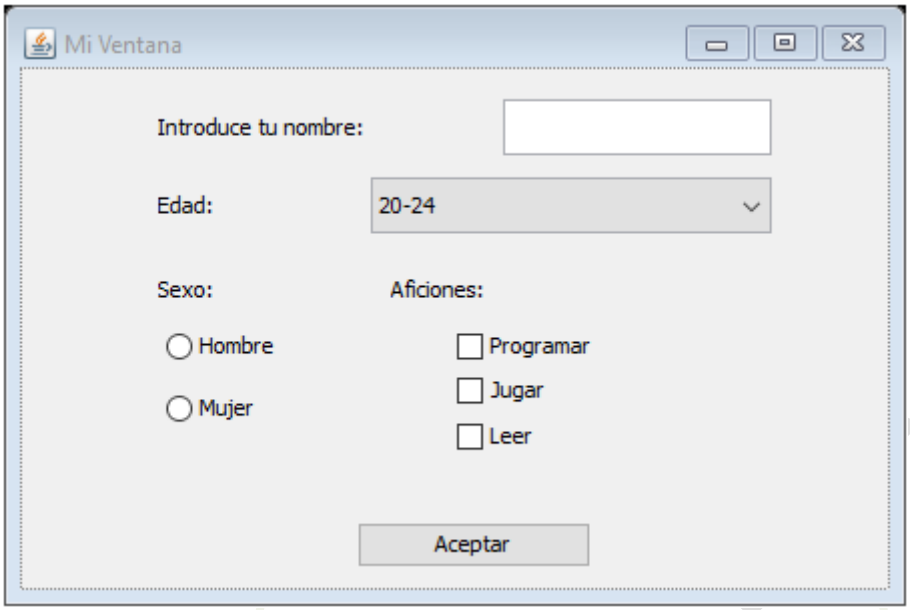
En la siguiente tabla se pueden ver algunos listeners y la acción a la que responden.

ActionListener	Hace referencia a la acción más típica sobre un componente. Por ejemplo, sobre un JButton será presionarlo, sobre un JTextField será pulsar Intro, sobre un JComboBox será seleccionar una opción, etc.
FocusListener	Ejecuta acciones cuando un componente obtiene o pierde el foco (colocarnos sobre el componente o irnos cuando éste estaba activo)
KeyListener	Responde a la acción de pulsar una tecla cuando un componente tiene el foco.
ItemListener	Hace referencia a la acción de seleccionar o deseleccionar una opción (en un JCheckBox, por ejemplo)
MouseListener	Responde al click sobre el ratón.
MouseMotionListener	Responde a acciones como arrastrar (drag) un elemento o pasar por encima.
WindowListener	Responde a acciones sobre una ventana como, por ejemplo, cerrarla.

Para incorporar listeners a nuestro programa, debemos crear primero el componente. Podemos añadir varios listeners a un mismo componente. Por ejemplo, podemos querer que un botón realice una acción al pulsarlo con el ratón, pero también queremos que reaccione al pulsar Intro cuando el botón tenga el foco.

13.3. Ejemplo de uso de Swing

Ahora que hemos visto qué componentes podemos usar en nuestras aplicaciones, y cómo programar eventos sobre éstos, sólo nos falta ver código de ejemplo. Vamos a crear una aplicación con una ventana que contendrá varios componentes. La apariencia de la aplicación será la siguiente:



En el siguiente vídeo puedes ver el proceso de creación de la aplicación:

UF5_13_3_ Ejemplo de uso de Swing

13.4. Window Builder en Eclipse

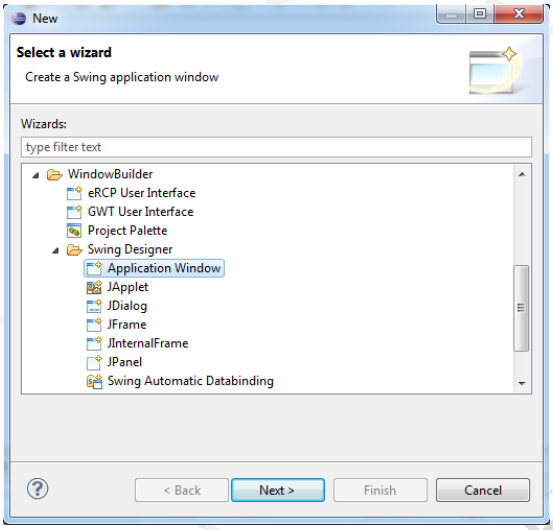
Window Builder es un plugin de Eclipse que nos permite desarrollar de forma rápida y cómoda la GUI (interfaz gráfica de usuario) de nuestras aplicaciones Java.

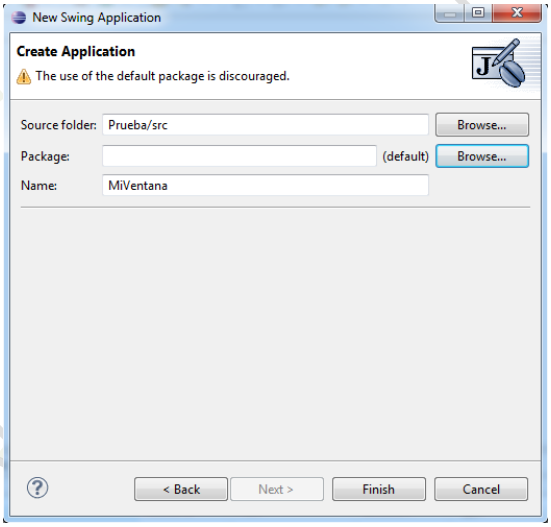
El proceso de instalación de Window Builder tiene dos posibilidades. Ambas las encontraremos en la página de descarga del plugin (ver apartado Recursos y enlaces).

Una posibilidad es descargarnos un ZIP que contendrá el plugin. Si abrimos el archivo veremos que tiene varias carpetas y archivos. Lo que nos interesan son las carpetas “features” y “plugins”. Descomprimiremos el contenido de ambas dentro de las carpetas con el mismo nombre en el directorio donde hagamos instalado Eclipse.

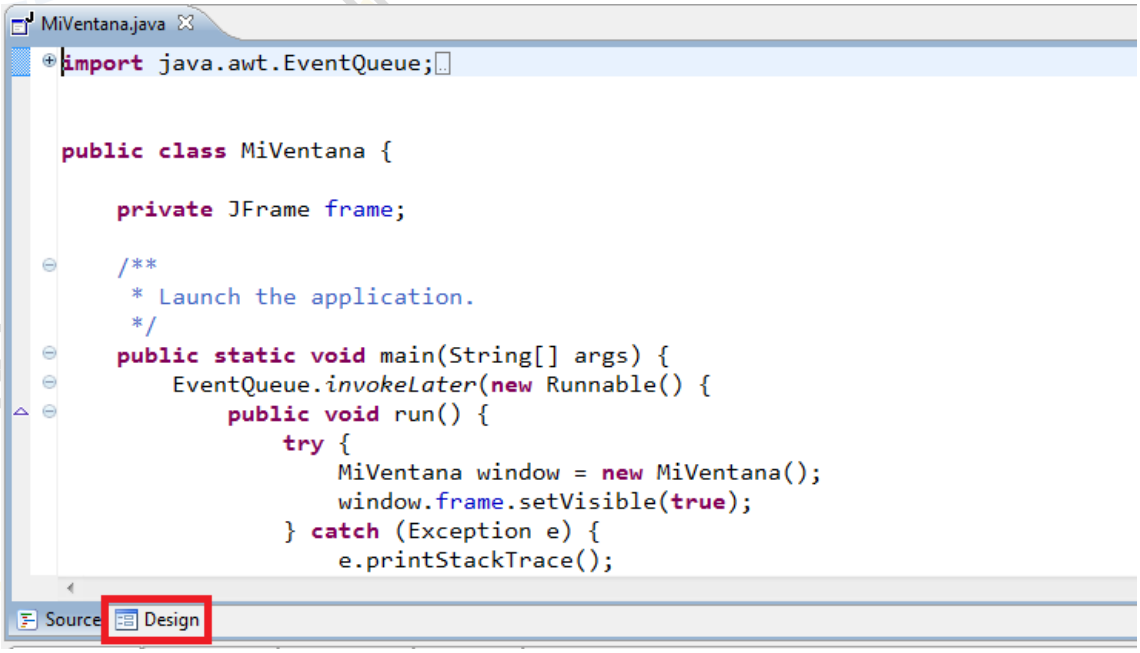
La otra forma de instalar Window Builder es desde el propio Eclipse, con su herramienta de instalación de software que encontrarás en el menú “Help > Install new software…”. Los pasos a seguir están explicados en la página de descarga de Window Builder.

Vamos a crear una sencilla aplicación Java con una ventana. Para ello ejecutamos Eclipse y creamos un proyecto Java al que llamaremos “Prueba”. A continuación, vamos al menú “File > New > Other…” y seleccionamos “WindowBuilder > Swing designer > Application Window”. Crearemos una interfaz gráfica para el proyecto “Prueba” creado anteriormente, tal y como se muestra en las siguientes imágenes:





Obtendremos un archivo de código autogenerado como el que se muestra a continuación. Pero para trabajar más cómodamente, usaremos la propiedad de WindowBuider “Design”:



Vemos que de esta forma se abre un editor gráfico que nos permitirá retocar las propiedades de nuestras ventanas y añadir nuevos elementos (menús, cuadros de texto, botones, etc.). Cada vez que cambiemos algo desde esta interfaz, se generará automáticamente el código fuente asociado a las acciones que realicemos.

Test de autoevaluación

¿Cuál de los siguientes NO es uno de los contenedores superiores de la librería Swing?

a) JPanel
b) JFrame
c) JDialog
d) JApplet

¿Qué objeto de la librería Swing permite crear listas desplegables?

e) JList
f) JSelect
g) JComboBox
h) JScrollBar

Si quiero que un JButton realice una acción cuando el usuario presiona el botón Intro, ¿qué tipo de listener tengo que programar?

a) ActionListener
b) ItemListener
c) FocusListener
d) KeyListener

Recursos y enlaces

- Descarga plugin WindowBuilder para Eclipse: http://www.eclipse.org/windowbuilder/download.php

- Documentación oficial sobre Swing: https://docs.oracle.com/javase/tutorial/uiswing/index.html

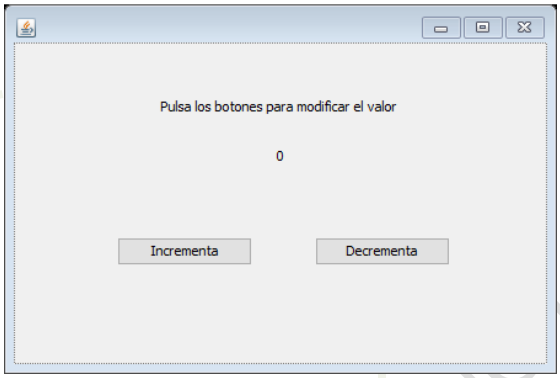
Conceptos clave

- **Swing:** kit de herramientas GUI que ofrece Java.
- **Componente**: un componente es una clase que representa un elemento visual independiente: un botón, un campo de texto, etc.
- **Contenedor**: un contenedor es un tipo especial de componente cuyo propósito es agrupar un conjunto de componentes.
- **Controlador de eventos**: es una clase predefinida de Java que se encarga de controlar y administrar que interacciones que realice el usuario con un elemento de la interfaz gráfica.

Ponlo en práctica

Actividad 1

Crea una aplicación que conste de una ventana con el siguiente contenido:



El funcionamiento de la aplicación será el siguiente: cuando pulses el botón “Incrementa” se debe sumar uno al valor que contiene la etiqueta con el valor 0 en la imagen. Al pulsar el botón “Decrementa”, se debe restar uno al valor de la etiqueta.

Hay que hacer que, al pulsar sobre el botón de cerrar de la ventana, el programa se cierre.

Los solucionarios están disponibles en el aula virtual.

Actividad 2

Crea una aplicación con una ventana en la que el usuario introduzca su nombre en una caja de texto y seleccione su edad de un desplegable con varias opciones.

Si el usuario ha introducido su nombre, y ha seleccionado una edad igual o mayor a 18, al darle a un botón se cerrará la ventana actual y se abrirá otra en la que se mostrará un mensaje de bienvenida que contendrá el nombre introducido por el usuario en la ventana anterior.

Si la edad seleccionada es menor a 18, en la segunda ventana se mostrará el mensaje “Eres menor de edad”.

Si el usuario dejara vacío el campo de texto, se debería mostrar un mensaje de error al pulsar el botón para ir a la siguiente ventana.

Los solucionarios están disponibles en el aula virtual.

SOLUCIONARIOS

Test de autoevaluación Tema 10

¿Qué hace la siguiente instrucción? ArrayList lista = new ArrayList()

e) Se crea un ArrayList de una posición

f) Se crea un ArrayList que puede almacenar cualquier tipo de objeto

g) Da un error de compilación, ya que no se ha especificado el tipo de dato que almacenará el ArrayList

h) Da un error de compilación, ya que no se ha especificado un tamaño inicial para el ArrayList

¿Cuál de las siguientes es una declaración errónea de un ArrayList?

i) ArrayList lista = new ArrayList()

j) ArrayList<String> lista = new ArrayList<String>()

k) ArrayList<Double> lista = new ArrayList<Double>()

l) ArrayList<int> lista = new ArrayList<int>()

¿Qué ocurre si intentamos añadir a un HashMap un elemento con una clave que ya existe?

m) Se modificará el valor del elemento con la clave que se intenta añadir

n) Se inserta a continuación del elemento con la misma clave

o) No se inserta nada

p) Se produce un error

Ponlo en práctica Tema 10

Actividad 1

Crea una aplicación que use un ArrayList para almacenar números enteros y añade varios valores.

A continuación, muestra el número total de elementos del ArrayList, y recórrelo.

Luego, pide al usuario que introduzca un valor para buscarlo en el ArrayList. Si lo encuentra, el programa preguntará si se quiere cambiar el valor. Si el usuario contesta afirmativamente, se le pedirá el nuevo valor y se introducirá en el ArrayList modificando el anterior.

Al final, se debe volver a mostrar el número total de elementos del ArrayList, y cuáles son.

Los solucionarios están disponibles en el aula virtual.

Actividad 2

Crea la clase Alumno con los atributos nombre, apellidos y edad. Deberás implementar el constructor, todas los getters y setters, el método toString y los métodos hashCode y equals. El código hash de cada alumno se calculará sumando el código hash del nombre, el código hash del apellido y la edad. Por otro lado, dos alumnos se consideran iguales si tienen el mismo valor en sus atributos.

Crea una aplicación que use un HashMap para almacenar los alumnos (clave) y el curso al que van (valor).

Añade varios alumnos al HashMap y luego muestra el número total de alumnos y sus valores (datos del alumno y curso al que van).

Crea un nuevo alumno cuyos datos coincidan con alguno de los ya dados de alta en el HashMap, pero en un curso diferente, e intenta insertarlo.

Comprueba que el HashMap sigue teniendo el mismo tamaño y muestra todos los datos que contiene.

Los solucionarios están disponibles en el aula virtual.

Test de autoevaluación Tema 11

Al crear excepciones propias, ¿qué forma tendrá el constructor?

q) No tendrá constructor, se usa el de la clase Exception

r) **Un parámetro de tipo String y una llamada a la función getMessage de la clase Exception**

s) Una llamada a la función System.out.println con el texto que queramos. No recibe ningún parámetro

t) Un parámetro de tipo String y una llamada a la función System.out.println para mostrar el String parámetro

¿Qué ocurre si no se captura una excepción en un método?

u) **Que se propaga al método que lo llamó**

v) Que se produce un error de compilación

w) Que ese tipo de excepción no se contempla en el programa

x) Que al ejecutarse, el método que lo llama da un error en tiempo de ejecución

Ponlo en práctica Tema 11

Actividad 1

Crear un programa que pida los datos de dos alumnos, que son: nombre (letras), edad (entero) y altura (decimal). Se debe realizar un control de la entrada de datos, de tal forma que, si el usuario introduce números en el nombre, o letras en la edad o la altura, se vuelva a pedir el dato correspondiente hasta que la entrada sea correcta.

NOTA: puedes usar el método “matches” de la clase String para ver si una cadena de caracteres posee números o no.

Los solucionarios están disponibles en el aula virtual.

Actividad 2

Modificar el programa anterior de manera que, si el usuario comete más de 5 errores al introducir datos, se muestre un mensaje informando de este hecho y se cierre el programa.

Los solucionarios están disponibles en el aula virtual.

Test de autoevaluación Tema 12

¿Qué clases utilizaremos para el almacenamiento y la recuperación de objetos en ficheros?

y) DataInputStream y DataOutputStream

z) FileInputStream y FileOutputStream

aa) FilterInputStream y FilterOutputStream

bb) ObjectInputStream y ObjectOutputStream

¿Qué tendremos que añadir en la cabecera de las clases que queremos serializar?

cc) extends java.io.Serializable

dd) implements java.io.Serializable

ee) extends java.io.transient

ff) implements java.io.transient

¿Qué modificador se utiliza para hacer que un atributo de una clase no sea serializable?

gg) protected serializable

hh) static-serializable

ii) no-serializable

jj) transient

Ponlo en práctica Tema 12

Actividad 1

Crea la clase Contacto con los atributos nombre (cadena de caracteres) y teléfono (long). Implementa el constructor, los métodos getters y setters, y el método toString. Esta clase debe ser serializable.

En el programa principal, crea una ArrayList y almacena en él 4 contactos. A continuación, deberás crear el archivo "Agenda" y recorrer el ArrayList, guardando los objetos que contiene dentro del archivo.

Por último, lee el archivo para recuperar los contactos, y muestra por pantalla el total de contactos recuperados y sus datos.

Los solucionarios están disponibles en el aula virtual.

Test de autoevaluación Tema 13

¿Cuál de los siguientes NO es uno de los contenedores superiores de la librería Swing?

kk) JPanel

ll) JFrame

mm) JDialog

nn) JApplet

¿Qué objeto de la librería Swing permite crear listas desplegables?

oo) JList

pp) JSelect

qq) JComboBox

rr) JScrollBar

Si quiero que un JButton realice una acción cuando el usuario presiona el botón Intro, ¿qué tipo de listener tengo que programar?

ss) ActionListener

tt) ItemListener

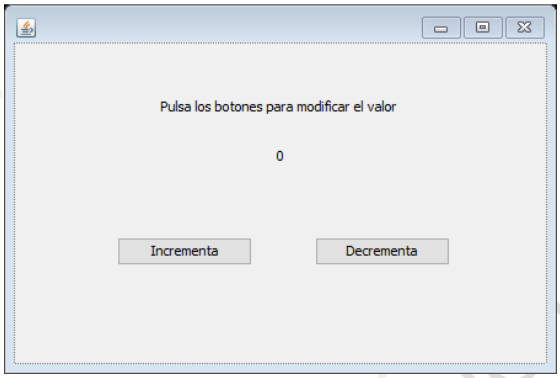
uu) FocusListener

vv) KeyListener

Ponlo en práctica Tema 13

Actividad 1

Crea una aplicación que conste de una ventana con el siguiente contenido:



El funcionamiento de la aplicación será el siguiente: cuando pulses el botón “Incrementa” se debe sumar uno al valor que contiene la etiqueta con el valor 0 en la imagen. Al pulsar el botón “Decrementa”, se debe restar uno al valor de la etiqueta.

Hay que hacer que, al pulsar sobre el botón de cerrar de la ventana, el programa se cierre.

Los solucionarios están disponibles en el aula virtual.

Actividad 2

Crea una aplicación con una ventana en la que el usuario introduzca su nombre en una caja de texto y seleccione su edad de un desplegable con varias opciones.

Si el usuario ha introducido su nombre, y ha seleccionado una edad igual o mayor a 18, al darle a un botón se cerrará la ventana actual y se abrirá otra en la que se mostrará un mensaje de bienvenida que contendrá el nombre introducido por el usuario en la ventana anterior.

Si la edad seleccionada es menor a 18, en la segunda ventana se mostrará el mensaje “Eres menor de edad”.

Si el usuario dejara vacío el campo de texto, se debería mostrar un mensaje de error al pulsar el botón para ir a la siguiente ventana.

Los solucionarios están disponibles en el aula virtual.