Tema 6: Utilización avanzada de clases

1. Herencia de clases

1.1 Herencia

- La palabra clave utilizada para indicar que una clase hereda de otra es *extends*.
- Crear una clase Persona con atributos *nombre*, *apellidos* y *dni*. Y los métodos de acceso.
- Crear dos clases Cliente y Empleado que hereden de Persona.
- Podemos acceder a los atributos de Persona si son *protected*. Si son private, tenemos que usar los métodos de acceso.

- Añadir un atributo extra diferente a Cliente y Empleado:
  - Cliente: visa (el número de su tarjeta de crédito).
  - Empleado: categoriaEmpleado (un número que indicará el sueldo que va a cobrar).

- Añadir un método mostrar( ) a la clase Persona, que muestre el nombre, los apellidos y el DNI.
- Comprobar que se puede acceder a él desde Cliente y Empleado usando *super* (en realidad, también sin usarlo).
- Sobreescribir el método *mostrar( )*, en Cliente y Empleado, para que llame al método de la superclase y muestre además el atributo propio de Cliente y de Empleado.

- Anotación *@Override*.

Actividad: Proyecto Concesionario. Realizar una jerarquía de clases similar para Vehículo, Coche y Moto.

- Vehiculo: Matrícula, marca, modelo.
- Coche: Potencia.
- Moto: Cilindrada.

Actividad: Práctica de medios digitales. Realizar la jerarquía de clases que aparece en el ejercicio 1 de la práctica 1 de clases avanzadas.

1.2 Constructores

- Recordar el modificador *protected* (representado en UML como #) que permite que los atributos o métodos de una clase sean accedidos desde una clase hija (y desde cualquier clase del mismo paquete, por cierto).
- Observar que todas las clases heredan de otra. Si no se indica explícitamente, entonces heredan de *Object*.
- Acceder al constructor de la superclase mediante *super( )*. Esta instrucción debe ser la primera del constructor de la subclase.
- Si no llamamos a *super( )*, se ejecutará el constructor por defecto de la clase padre (sin argumentos). En caso de que no exista, aparecerá un error.

- **Ejemplo**:
  - Añadir un constructor a la clase Persona que tome como parámetros: dni, apellidos y nombre.
  - Añadir a la clase Cliente otro que llame al constructor de Persona y luego inicialice el campo visa.
  - Añadir a la clase Empleado un constructor que llame al constructor de Persona y luego actualice el campo categoriaEmpleado.

Actividad: Hacer lo mismo que con el ejemplo, para el proyecto del Concesionario.

1.3 Clases abstractas

- Una clase abstracta no se puede instanciar (no se pueden crear objetos de esa clase).
- Está pensada para que otras clases hereden de ella.
- Se declara con el modificador *abstract*.
- Se utiliza para representar un concepto.
- En UML se representa con el nombre en cursiva.
- **Ejemplo**:
  - La clase Persona puede ser declarada como abstracta, ya que realmente en la empresa no vamos a tener una categoría que sea "Persona".
- **Ejemplo**:
  - La clase Vehiculo puede ser declarada también como abstracta.

1.4 Métodos abstractos

Podemos declarar un método en una clase abstracta como abstracto sin codificarlo. De esta manera las clases hijas estarán obligadas a implementarlo. Esto se hace si el método no tiene sentido para la clase padre pero sí para las subclases (por ejemplo, la clase padre no tiene suficiente información).

- **Ejemplo**: Un método mostrar() en la clase Persona, puede ser interesante que sea abstracto, y que se implemente en las clases hijas.

- **Ejemplo**: En la jerarquía de medios digitales, en la clase Medio no tiene sentido implementar un método *obtenDatos( )* puesto que esta operación depende del tipo de medio. Pero sí podemos tener claro que todos los tipos de medio deberán implementarlo.

Recordar: Una clase que tenga algún método abstracto debe ser a su vez abstracta.

Si una subclase no implementa todos los métodos abstractos de la superclase, entonces deberá ser declarada también como abstracta.

- **Ejemplo**: Para las clases Cliente y Empleado tiene sentido un método *darDeAlta( )*, que los dé de alta en el array *clientes[ ]* o *empleados[ ]*. Pero para la clase Persona no. Por tanto, podemos crear este método como abstracto en la clase Persona e implementarlo en las clases Cliente y Empleado.

```
public abstract void darDeAlta();       // Dentro de Persona
```

1.5 Clases finales

Una clase también se puede declarar como *final*. Esto significa que no se podrá heredar de ella.

```
public final class Coche extends Vehiculo{

}
```

1.6 Herencia múltiple

- En qué consiste.
- No existe herencia múltiple en java. Lo más parecido son las interfaces.
- **Ejemplo**: Una clase MotorElectrico podría tener sentido que heredara de Motor y de ComponenteElectrico. Esto sería posible hacerlo en ciertos lenguajes, como C++.

2. Polimorfismo

2.1 Polimorfismo

El polimorfismo consiste en la posibilidad de aplicar una misma operación a objetos de diferentes clases, pero llamando a una implementación diferente según la clase a la que pertenece el objeto sobre el cual se llama. Esto se consigue mediante sobreescritura de métodos.

Gracias al polimorfismo podemos crear una variable del tipo de la superclase y asignarle un objeto de una subclase. Al llamar a un método de este objeto, se ejecuta la versión sobreescrita por la subclase.

No obstante, si un objeto de una subclase es guardado en una variable de tipo superclase, sólo se podrá acceder a los métodos que estén definidos en la superclase.

Ejemplo 1: Crear tres variables de la clase Persona. Asignarles respectivamente un objeto de tipo Persona, Cliente y Empleado. Llamar al método *mostrar( )* de cada uno de ellos.

Ejemplo 2: Crear un método *saluda( )* en la clase Cliente, que muestre un saludo por pantalla. Crear un objeto Cliente, c1, y guardarlo en una variable de tipo Persona, p1. Comprobar que no se puede llamar al método *saluda( )* de p1.

```
p1.saluda(); // Esto no puede hacerse.
```

Ejemplo 3: Siguiendo el ejemplo anterior, comprobar que si hacemos un cast sí podremos llamar al método saluda( ):

```
p1 = c1;
p1.mostrar();
((Cliente) p1).saluda();  // Para acceder al método saluda() hago un cast
                          // a (Cliente).
```

2.2 Sobreescritura de métodos de la clase Object

Sobreescritura del método void finalize( )

Llamado por el recolector de basura cuando va a destruir un objeto.

Sobreescritura del método boolean equals(Object obj)

Cuando utilizamos el operador == entre dos variables que referencian objetos, sólo indica si ambas apuntan al mismo objeto. Para comparar dos objetos en función de su contenido podemos sobreescribir el método *equals( )*.

Ejemplo: La clase String dispone de este método.

Ejemplo: Aquí tenemos sobreescrito el método *equals( )* en nuestra clase Persona. Se entiende que dos personas son la misma si tienen el mismo DNI.

```
public boolean equals(Object obj) {
	if (obj == this)
		return true;
	if (obj == null)
		return false;
	if (obj instanceof Persona)
		return dni.equals(((Persona) obj).dni);
	return false;
}
```

Ejemplo: Para ahorrarnos la comprobación de si el objeto que nos pasan es de tipo Persona, podríamos haber declarado el parámetro de tipo Persona en vez de Object. Sin embargo de esa manera ya no estaríamos sobreescribiendo el método *equals( )*, sino creando uno aparte (tendríamos realmente dos). Esto no se recomienda.

```
public final boolean equals (Persona obj) {
	if (obj == this) return true;
	if (obj == null)
		return false;
	return dni.equals(obj.dni);
}
```

Sobreescritura del método String toString(Object obj)

El método *toString( )* de un objeto es lo que devuelve la representación como cadena del mismo cuando hacemos:

```
System.out.println(objeto);
```

3. Interfaces

Una interface es un grupo de métodos relacionados con cuerpos vacíos. Se utilizan para dejar constancia de que una clase tiene que implementar una funcionalidad determinada.

Los atributos en una interface son implicitamente considerados *static* y *final*. Los métodos, *public*.

Ejemplo:

```
public interface Figura2D {
	double area(); // Área de la figura
	double perimetro(); // Perímetro de la figura
}

public class Circulo implements Figura2D {
	int radio;

	@Override
	public double area() {
		return Math.PI * radio * radio;
	}

	@Override
	public double perimetro() {
		return 2 * Math.PI * radio;
	}

}
```

3.1 Utilidades de las interfaces

- Una interface puede ser implementada por múltiples clases, de manera similar a como una clase puede ser superclase de múltiples clases.

- Las clases que implementen una interface están obligadas a sobreescribir todos los métodos definidos en la interface.

- Una clase puede implementar múltiples interfaces, a diferencia de la derivación (herencia), que sólo se permite de una única clase base.

- Una interface no se puede instanciar, pero sí se puede hacer referencia. Así, si I es una interface y C es una clase que implementa la interface, se pueden hacer declaraciones del tipo:

  I objeto = new C (<parámetros>);

  Con la orden anterior estaríamos creando un objeto de clase C. A su vez, la clase C debe implementar la interface I.

- Las interfaces pueden heredar de otras interfaces y, a diferencia de la herencia de clases, pueden heredar de más de una interface.

3.2 Declarar una variable de tipo interface

Ejemplo:

```
public class Inicio {
	public static void main(String[] args) {
		Figura2D fig = new Circulo();
		System.out.println(fig.area());
	}
}
```

3.3 Declarar un parámetro de tipo interface

```
public class Inicio {
	public static void main(String[] args) {
		Circulo c1 = new Circulo();
		System.out.println("Área del círculo: " + calcularArea(c1));
	}

	/*No tenemos que crear un método calcularArea() distinto para cada tipo de
	figura. En lugar de eso, simplemente hacemos que tome como parámetro un
	objeto de tipo Figura2D.*/
	public static double calcularArea(Figura2D fig) {
		return fig.area();
	}
}
```

3.4 Clases que implementan varias interfaces

```
public interface Figura2D {
	double area(); // Área de la figura
	double perimetro(); // Perímetro de la figura
}

public interface Mostrable {
	void mostrar();
}

public class Circulo implements Figura2D, Mostrable {
	int radio;

	@Override
	public double area() {
		return Math.PI * radio * radio;
	}

	@Override
	public double perimetro() {
		return 2 * Math.PI * radio;
	}

	@Override
	public void mostrar() {
		System.out.println("Radio del círculo: " + radio);
	}
}
```

3.5 Interfaces para crear grupos de constantes

Como sus atributos son estáticos y constantes, las interfaces son aptas para agrupar constantes:

```
	public interface DiasSemana {
		int LUNES = 1;
		int MARTES = 2;
		int MIERCOLES = 3;
		int JUEVES = 4;
		int VIERNES = 5;
		int SABADO = 6;
		int DOMINGO = 7;
	}
```

4. Manejo de excepciones

4.1 Excepciones en Java

El siguiente código de ejemplo pide un número al usuario y divide 20 entre este número:

```
public static void main(String[] args) {
	Scanner entrada = new Scanner(System.in);

	System.out.println("Programa para dividir 20 entre un n°");
	System.out.println("Escribe un n° entero: ");
	String numero = entrada.nextLine();

	int num = Integer.parseInt(numero);

	System.out.println("La división de 20 entre " + num + " es " + 20 / num);
}
```

Sin embargo encontramos un error si introducimos una letra en lugar de un número entero, en el momento en que intentamos convertir el String a entero:

```
Programa para dividir 20 entre un nº
Escribe un nº entero:
carlos
Exception in thread "main" java.lang.NumberFormatException: For input string: "carlos"
	at java.lang.NumberFormatException.forInputString(Unknown Source)
	at java.lang.Integer.parseInt(Unknown Source)
	at java.lang.Integer.parseInt(Unknown Source)
	at Errores1.main(Errores1.java:13)
```

En Java cuando se produce un error se dice que se *lanza* una excepción. Y al hecho de ejecutar un código para manejarla se le llama *capturar la excepción*. El mecanismo que usamos es la intrucción *try-catch-finally*.

4.2 Bloque try - catch

Colocaremos las instrucciones que pueden dar error dentro de un bloque try. A continuación, capturaremos el error escribiendo un bloque catch, con las instrucciones que han de cumplirse si se produce una excepción.

En la captura anterior vemos que el tipo de excepción producida es *NumberFormatException*. Esto es lo que escribiremos entre paréntesis al lado del catch.

```
public static void main(String[] args) {
	Scanner entrada = new Scanner(System.in);

	System.out.println("Programa para dividir 20 entre un n°");
	System.out.println("Escribe un n° entero: ");
	String numero = entrada.nextLine();

	try {

		// La siguiente línea puede producir un error si el String no puede
		// convertirse en double.
		int num = Integer.parseInt(numero);
		System.out.println("La división de 20 entre " + num + " es " + 20 /
		num);

	} catch (NumberFormatException e) {

		// Este bloque se ejecuta si se produce una NumberFormatException,
		// al escribir el usuario una palabra en lugar de un número.
		System.out.println("Tienes que escribir un número entero!!!");

	}
}
```

4.3 Obtener información de una excepción

Cuando se lanza una excepción, Java nos proporciona un objeto con información sobre la misma. Es el que aparece en el catch junto al tipo de excepción. Podemos usar algunos de sus métodos para obtener información. Por ejemplo:

- toString(): Devuelve una cadena con información sobre el error.
- getMessage(): Devuelve una cadena con información sobre el error.
- printStackTrace(): Muestra en la consola la traza del error, tal como se ve en la ventana de Eclipse.

Un ejemplo de uso puede ser añadir un print con la salida del método toString, dentro del catch:

```
public static void main(String[] args) {
	Scanner entrada = new Scanner(System.in);

	System.out.println("Programa para dividir 20 entre un nº");
	System.out.println("Escribe un nº entero: ");
	String numero = entrada.nextLine();

	try {

		// La siguiente línea puede producir un error si el String no puede
		// convertirse en double.
		int num = Integer.parseInt(numero);
		System.out.println("La división de 20 entre " + num + " es " + 20 /
		num);

	} catch (NumberFormatException e) {

		// Este bloque se ejecuta si se produce una NumberFormatException,
		// al escribir el usuario una palabra en lugar de un número.
		System.out.println("Tienes que escribir un número entero!!!");
		System.out.println(e.toString());

	}
}
```

Así obtendríamos en caso de error:

```
Programa para dividir 20 entre un nº
Escribe un nº entero:
wq
Tienes que escribir un número entero!!!
java.lang.NumberFormatException: For input string: "wq"
```

4.4 Capturar varias excepciones

En el programa anterior obtendremos una excepción diferente si introducimos un 0, ya que estaremos intentando hacer una división por cero:

```
Programa para dividir 20 entre un nº
Escribe un nº entero:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Errores2.main(Errores2.java:20)
```

Esta excepción no la hemos capturado, ya que es de un tipo distinto (ArithmeticException). Pero podemos añadir otro bloque catch para ello:

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

```
        Scanner entrada = new Scanner(System.in);

        System.out.println("Programa para dividir 20 entre un nº");
        System.out.println("Escribe un nº entero: ");
        String numero = entrada.nextLine();

        try {

                // La siguiente línea puede producir un error si el String no puede
                // convertirse en double.
                int num = Integer.parseInt(numero);

                // La siguiente línea puede producir un error si intentamos dividir
                // entre 0.
                System.out.println("La división de 20 entre " + num + " es " + 20 /
                num);

        } catch (NumberFormatException e) {

                // Este bloque se ejecuta si se produce una NumberFormatException,
                // al escribir el usuario una palabra en lugar de un número.
                System.out.println("Tienes que escribir un número entero!!!");
                System.out.println(e.toString());

        } catch (ArithmeticException e) {

                // Este bloque se ejecuta si se produce una ArithmeticException,
                // al intentar dividir entre cero.
                System.out.println("No se puede dividir por cero!!!");

        }
}
```

Ahora obtendremos este mensaje de error:

```
Programa para dividir 20 entre un nº
Escribe un nº entero:
0
No se puede dividir por cero!!!
```

Todas las excepciones heredan de la clase *Exception*. Esto nos permite añadir un último catch para capturar cualquier otro tipo de excepción que no sea *NumberFormatException* o *ArithmeticException*:

```
public static void main(String[] args) {
	Scanner entrada = new Scanner(System.in);

	System.out.println("Programa para dividir 20 entre un n°");
	System.out.println("Escribe un n° entero: ");
	String numero = entrada.nextLine();

	try {

		// La siguiente línea puede producir un error si el String no puede
		// convertirse en double.
		int num = Integer.parseInt(numero);

		// La siguiente línea puede producir un error si intentamos dividir
		// entre 0.
		System.out.println("La división de 20 entre " + num + " es " + 20 /
		num);

	} catch (NumberFormatException e) {

		// Este bloque se ejecuta si se produce una NumberFormatException,
		// al escribir el usuario una palabra en lugar de un número.
		System.out.println("Tienes que escribir un número entero!!!");
		System.out.println(e.toString());

	} catch (ArithmeticException e) {

		System.out.println("No se puede dividir por cero!!!");

	} catch (Exception e) {

		System.out.println("Se ha producido una excepción en el programa.");
		System.out.println("Más información:");
		e.printStackTrace();

	}
}
```

4.5 Combinar varias excepciones

Puede ocurrir que queramos realizar las mismas acciones para un par de excepciones distintas. Por ejemplo, podríamos combinar las dos del ejemplo anterior en un simple mensaje que diga "Tienes que introducir un número entero distinto de cero".

Para ahorrar espacio podemos combinar las dos excepciones en un mismo *catch*, separándolas con una barra vertical (|).

```
public static void main(String[] args) {
	Scanner entrada = new Scanner(System.in);

	System.out.println("Programa para dividir 20 entre un nº");
	System.out.println("Escribe un nº entero: ");
	String numero = entrada.nextLine();

	try {

		// La siguiente línea puede producir un error si el String no puede
		// convertirse en double.
		int num = Integer.parseInt(numero);

		// La siguiente línea puede producir un error si intentamos dividir
		// entre 0.
		System.out.println("La división de 20 entre " + num + " es " + 20 /
		num);

	} catch (NumberFormatException | ArithmeticException e) {

		// Este bloque se ejecuta si se produce una NumberFormatException o
		// una ArithmeticException.
		System.out.println("Tienes que introducir un número entero distinto
		de cero!!!");

	} catch (Exception e) {

		System.out.println("Se ha producido una excepción en el programa.");
		System.out.println("Más información:");
		e.printStackTrace();

	}
}
```

4.6 Sentencia finally

Después de un bloque *try-catch* la ejecución del método donde está puede continuar. Sin embargo hay ocasiones en que esta ejecución puede detenerse.

Si queremos asegurarnos de que algunas instrucciones se ejecuten antes de terminar el método, podemos colocarlas añadiendo un bloque *finally*. En este bloque colocaremos típicamente instrucciones para liberar recursos, como conexiones a bases de datos, archivos, etc.

Las instrucciones dentro de un *finally* se ejecutarán siempre, tanto si se produce una excepción como si no, incluso si dentro del *try-catch* aparece una instrucción *return*.

```
public static void main(String[] args) {
	Scanner entrada = new Scanner(System.in);

	System.out.println("Programa para dividir 20 entre un nº");
	System.out.println("Escribe un nº entero: ");
	String numero = entrada.nextLine();

	try {

		// La siguiente línea puede producir un error si el String no puede
		// convertirse en double.
		int num = Integer.parseInt(numero);
		// La siguiente línea puede producir un error si intentamos dividir
		// entre 0.
		System.out.println("La división de 20 entre " + num + " es " + 20 /
		num);

	} catch (NumberFormatException | ArithmeticException e) {

		// Este bloque se ejecuta si se produce una NumberFormatException o
		// una ArithmeticException.
		System.out.println("Tienes que introducir un número entero distinto
		de cero!!!");

	} catch (Exception e) {

		System.out.println("Se ha producido una excepción en el programa.");
		System.out.println("Más información:");
		e.printStackTrace();

	} finally {

		// Las instrucciones que aparezcan aquí se ejecutarán tanto si se
		// lanza una excepción como si no.
		System.out.println("Instrucciones finales ...");

	}
}
```

Referencias

- Draw.io, herramienta online para crear toda clase de diagramas.
  - https://www.draw.io/
- Lucid Chart, herramienta online para crear diagramas UML (entre otros).
  - https://www.lucidchart.com/