

## 5.1. Creación de componentes: propiedades y atributos



# Índice

---

Objetivos .....	3
Clases, atributos y métodos .....	4
Crear clases personalizadas.....	4
Los atributos.....	5
Los métodos .....	7
Sobrecarga .....	8
Sobrecarga de constructores.....	9
Interfaz y modificaciones de acceso .....	9
Relaciones entre clases: jerarquía .....	11
Relaciones de agregación y composición.....	11
Agregación .....	11
Composición .....	12
Herencia.....	14
Ejemplo de herencia .....	15
Polimorfismo .....	17
Relación entre polimorfismo y excepciones.....	18
Uso de objetos de excepción .....	18
Tipos de excepciones (clases derivadas de <i>Exception</i> ).....	19
¿Cómo se trabaja con las excepciones comprobadas? .....	22
Provocando excepciones no comprobadas.....	23
Varios catch .....	25
<i>Multicatch</i> .....	27
Polimorfismo y excepciones .....	28
Relanzamiento o propagación de excepciones .....	29
Resumen de las excepciones más habituales .....	32
Despedida .....	34
Resumen.....	34

# Objetivos

Los temas tratados en esta lección ya lo estudiaste en la asignatura de Programación, pero te recomendamos que vuelvas a estudiarlo como repaso, ya que te ayudará a continuar con éxito el resto de contenidos y a realizar las actividades propuestas sin dificultad.

**Adelante, te costará poco esfuerzo recordar estos conceptos y volverlos a poner en la práctica.**

Con esta lección perseguimos los siguientes objetivos:

- Crear componentes software formados por jerarquías de clases Java construidas utilizando todas las características de la programación orientada a objetos: agregación, composición, herencia, abstracción y polimorfismo.
- Relacionar el concepto de polimorfismo con la gestión de excepciones.

# Clases, atributos y métodos

## Crear clases personalizadas

**Las clases son abstracciones de entidades del mundo real que sirven como modelos para construir objetos basados en ellas.**

Este es el formato de construcción de una clase:

```
modificadores class nombreClase {  
    // Cuerpo de la clase  
}
```

**Donde los modificadores definen el tipo de visibilidad** y comportamiento de la clase.

**Los modificadores que pueden aplicarse a una clase son:**

- ***final***: la clase no puede ser heredada por otras clases.
- ***public***: la clase es visible dentro de cualquier paquete del proyecto.
- ***protected***: la clase es visible solo dentro del mismo paquete donde está definida.
- ***private***: la clase solo es visible por otra clase que la contenga.
- ***abstract***: la clase es abstracta. No se pueden construir objetos a partir de ella. Está pensada solo para ser heredada.

Posiblemente te estés preguntando por qué existe un modificador de tipo *private* asociado a una clase. **¿Quién quiere una clase que no es accesible?**

Pues la respuesta está en las clases internas. A continuación te muestro un ejemplo:

```
public class Casa {  
    private int metros;  
    private int numHabitaciones;  
    private Propietario propietario;  
  
    public Casa(int metros, int numHabitaciones, String nombre, String telefono) {  
        this.metros = metros;  
        this.numHabitaciones = numHabitaciones;  
        this.propietario = new Propietario(nombre, telefono);  
    }  
  
    @Override  
    public String toString() {  
        return "Casa [metros=" + metros + ", numHabitaciones=" +  
            numHabitaciones + ", propietario=" +  
            propietario.toString() + " ]";  
    }  
}
```

## 5.1. Creación de componentes: propiedades y atributos

```
private class Propietario {
    private String nombre;
    private String telefono;

    public Propietario(String nombre, String telefono) {
        super();
        this.nombre = nombre;
        this.telefono = telefono;
    }

    @Override
    public String toString() {
        return "Propietario [nombre=" + nombre + ", telefono=" +
telefono
        + " ]";
    }
}
```

Observa que dentro de la clase *Casa* tenemos una clase interna llamada *Propietario*. Esta clase es privada, lo que significa que solo es visible por su clase contenedora *Casa*.

Puedes ponerlo en práctica con la siguiente clase *Principal*.

```
public class Principal {
    public static void main(String[] args) {
        Casa unaCasa = new Casa(100, 3, "Pepe", "Pérez");
        System.out.println(unaCasa);
    }
}
```

## Los atributos

**Las clases tienen atributos.** Los atributos pueden pertenecer al objeto (propiedades del objeto) o a la clase (propiedades que pertenecen a la clase).

Formato de creación de un atributo:

```
modificadores tipo nombreAtributo [= valor];
```

Los atributos admiten los siguientes modificadores:

### **private**

El atributo es accesible solo dentro de la clase donde está declarado.

### **public**

El atributo es accesible desde cualquier clase situada en cualquier paquete y también por las clases derivadas.

## 5.1. Creación de componentes: propiedades y atributos

### protected

El atributo es accesible solo desde otras clases situadas en el mismo paquete y también por las clases derivadas, independientemente del paquete en que se encuentren.

### default

Equivale a no poner ningún modificador de acceso. El atributo es accesible solo desde otras clases situadas en el mismo paquete y desde clases derivadas, siempre que se encuentren en el mismo paquete.

### static

El atributo pertenece a la clase, no al objeto. El valor del atributo es compartido por todos los objetos creados de la clase.

### final

El atributo es una constante. Su valor no puede ser modificado.

Veamos un ejemplo:

```
public class Triangulo {  
    // Constante publica, su valor no puede ser modificado.  
    // Además pertenece a la clase, no al objeto  
    public static final float PI = 3.141592F;  
  
    // Propiedades privadas.  
    private int lado1;  
    private int lado2;  
    private int lado3;  
}
```

**Los atributos *lado1*, *lado2* y *lado3* pertenecen al objeto.** Si creamos tres objetos de la clase *Triangulo*, cada uno de ellos tendrá distintos lados, es decir, existirán tres grupos de estas propiedades.

**La variable *PI*, sin embargo, pertenece a la clase.** Aunque creamos 20 objetos de la clase *Triangulo*, habrá un solo atributo *PI* compartido para todos ellos. Además *PI* es una constante, su valor no puede ser modificado.

Las propiedades *lado1*, *lado2* y *lado3*, además, son privadas, lo que significa que para tener acceso a ellas a través de los objetos, necesitamos implementar los métodos públicos *getLado1()*, *getLado2()* y *getLado3()*.



## Los métodos

Las clases sirven como abstracciones de objetos que además de tener atributos (características) pueden realizar acciones.

**Las acciones se implementan por medio de los métodos.**

Los métodos tienen la siguiente estructura:

```
modificadores tipo nombreMetodo([parámetros]) {  
  
}
```

- Los **modificadores** definen el tipo de visibilidad y comportamiento del método.
- El **tipo** define el tipo de valor que retorna el método. Se pone *void* si el método no retorna ningún valor.
- Los **parámetros** definen los datos que debe recibir el método.

Los modificadores aplicables a un método son:

### **private**

El método es accesible solo dentro de la clase donde está declarado.

### **public**

El método es accesible desde cualquier clase situada en cualquier paquete y también por las clases derivadas.

### **protected**

El método es accesible solo desde otras clases situadas en el mismo paquete y también por las clases derivadas, independientemente del paquete en que se encuentren.

### **default**

Equivale a no poner ningún modificador de acceso. El método es accesible solo desde otras clases situadas en el mismo paquete y desde clases derivadas siempre que se encuentren en el mismo paquete.

### **static**

El método pertenece a la clase, no al objeto. Los métodos estáticos solo pueden acceder a atributos estáticos, nunca a propiedades del objeto.

### **final**

El método no puede ser sobrescrito por las clases derivadas.

## 5.1. Creación de componentes: propiedades y atributos

Ejemplo:

```
public final String verTipoTriangulo() {
    if (this.lado1 == this.lado2 && this.lado2 == this.lado3) {
        return "Equilátero";
    } else if (this.lado1 == this.lado2 || this.lado2 == this.lado3
        || this.lado1 == this.lado3) {
        return "Isósceles";
    } else {
        return "Escaleno";
    }
}
```

Analicemos un poco el método:

- El método ***verTipoTriangulo()*** pertenece al objeto, es decir, trabaja con las propiedades de un triángulo concreto.
- El método ***verTipoTriangulo()*** no podrá ser sobrescrito por otras clases que hereden de *Triangulo*, ya que está declarado como *final*.

## Sobrecarga

La sobrecarga de métodos consiste en definir varias implementaciones para el mismo método, de modo que pueda ser utilizado de múltiples maneras. Veamos un ejemplo:

```
public class Compra {
    private String descripcion;
    private float precio;
    private float cantidad;

    public Compra(String descripcion, float precio, float cantidad) {
        this.descripcion = descripcion;
        this.precio = precio;
        this.cantidad = cantidad;
    }

    public float calcularPrecioVenta() {
        return this.cantidad * this.precio;
    }

    public float calcularPrecioVenta(float descuento) {
        return this.cantidad * this.precio - this.cantidad * this.precio
            * descuento;
    }

    @Override
    public String toString() {
        return "Compra [descripcion=" + descripcion + ", precio=" + precio
            + ", cantidad=" + cantidad + "]";
    }
}
```



## 5.1. Creación de componentes: propiedades y atributos

La clase *Compra* define dos implementaciones distintas para el método *calcularPrecioVenta()*, uno sin parámetros y otro que recibe un porcentaje de descuento.

Cada una de las implementaciones debe variar en el número y/o tipo de parámetros.

Ahora podemos calcular el precio de una compra de dos maneras distintas.

```
public class Principal {
    public static void main(String[] args) {
        Compra miCompra = new Compra("Peras", 1.25F, 3.5F);
        System.out.println(miCompra.toString());
        System.out.println("Precio: " + miCompra.calcularPrecioVenta());
        System.out.println("Precio con descuento: "
            + miCompra.calcularPrecioVenta(0.1F));
    }
}
```

## Sobrecarga de constructores

**Recuerda que el método constructor es un método especial que lleva el mismo nombre que la clase y es invocado en el momento de construir el objeto. También puede estar sobrecargado, así tenemos más posibilidades a la hora de construir los objetos.**

```
public class Compra {
    private String descripcion;
    private float precio;
    private float cantidad;

    public Compra(String descripcion, float precio, float cantidad) {
        this.descripcion = descripcion;
        this.precio = precio;
        this.cantidad = cantidad;
    }

    public Compra(String descripcion, float precio) {
        this.descripcion = descripcion;
        this.precio = precio;
        this.cantidad = 1;
    }

    .....
}
```

Ahora podemos construir un objeto de la clase *Compra* de dos maneras, indicando la cantidad o sin indicar la cantidad, en cuyo caso asigna una unidad por defecto.

## Interfaz y modificaciones de acceso

La interfaz de la clase está definida por lo que dejamos ver al exterior, de modo que está determinada por los modificadores de visibilidad que utilizamos.



La interfaz de un televisor son los botones que tenemos disponibles para interactuar con ella.

De la misma forma, la interfaz de los objetos de la clase *Compra* son las propiedades y los métodos públicos a los que el usuario puede acceder a través del objeto.

# Relaciones entre clases: jerarquía

## Relaciones de agregación y composición

**La agregación y la composición son tipos de relaciones entre clases donde una propiedad de una clase es un objeto de otra clase.**

Por ejemplo: un objeto *coche* podría tener las propiedades *matricula*, *marca*, *modelo* y *motor*, siendo *motor* un objeto con sus propiedades *tipoCombustible*, *caballos*, etc.

**Entonces, ¿dónde está la diferencia entre agregación y composición?**

La diferencia entre agregación y composición es bastante sutil.

## Agregación

Cuando existe una relación de agregación entre dos clases, **al eliminar la clase contenedora, no se elimina la clase contenida.**

Verás un ejemplo en el que una clase *Examen* agrega un objeto de la clase *Alumno*.

Las relaciones de agregación también se denominan de “**composición débil**”.

```
public class Alumno {
    private String nombre;
    private String telefono;

    public Alumno(String nombre, String telefono) {
        this.nombre = nombre;
        this.telefono = telefono;
    }

    @Override
    public String toString() {
        return "Alumno: " + this.nombre + " - " + this.telefono;
    }
}
```

```
public class Examen {
    private Alumno alumno;
    private String asignatura;
    private float nota;

    public Examen(Alumno alumno, String asignatura, float nota) {
        this.alumno = alumno;
        this.asignatura = asignatura;
        this.nota = nota;
    }
}
```

## 5.1. Creación de componentes: propiedades y atributos

```
public Alumno getAlumno() {
    return alumno;
}

@Override
public String toString() {
    return this.alumno.toString() + " " + this.asignatura + ":" +
this.nota;
}
}
```

Observa que el constructor de *Examen* recibe un objeto de tipo *Alumno* que podrá ser creado externamente y pasado como argumento.

Ahora puedes ponerlo en práctica con la clase *Principal*:

```
public class Principal {
    public static void main(String[] args) {
        Alumno al = new Alumno("Miguel Pérez", "616669966");
        Examen ex = new Examen(al, "Matemáticas", 7.5F);
        System.out.println(ex.toString());
        // Acceso al alumno a partir del examen.
        System.out.println(ex.getAlumno().toString());
        // Acceso al alumno como objeto autónomo
        System.out.println(al.toString());
    }
}
```

Observa que hemos creado dos objetos (*al* y *ex*) de las *clases* *Alumno* y *Examen*. El objeto *al* es completamente autónomo con respecto al objeto *ex*. No depende de él para su subsistencia.

El objeto *al* podría servir de componente para otros objetos diferentes que lo requieran.

## Composición

Cuando existe una relación de composición entre dos clases, **al eliminar la clase contenedora, se elimina la clase contenida**.

Para el ejemplo expuesto anteriormente, eliminar el objeto *Examen* supone la eliminación de su objeto *Alumno* contenido.

Las relaciones de composición también se denominan de “**composición fuerte**”.

Si quieres ponerlo en práctica puedes realizar una copia del proyecto anterior con otro nombre y realizar los cambios necesarios.

La clase *Alumno* no cambia.

Ahora modifica la clase *Examen* de la siguiente forma:

## 5.1. Creación de componentes: propiedades y atributos

```
public class Examen {
    private Alumno alumno;
    private String asignatura;
    private float nota;

    public Examen(String asignatura, float nota, String nombre, String tlf) {
        this.alumno = new Alumno(nombre, tlf);
        this.asignatura = asignatura;
        this.nota = nota;
    }

    public Alumno getAlumno() {
        return alumno;
    }

    @Override
    public String toString() {
        return this.alumno.toString() + " " + this.asignatura + ":" +
this.nota;
    }
}
```

Observa que ahora, en lugar de pasar como argumento al constructor un objeto *Alumno* ya creado, le pasamos los valores de nombre y teléfono para crear el objeto *Alumno* dentro del código interno de la clase *Examen*.

Para cada objeto *Examen* construido, se creará un objeto *Alumno* internamente.

El *Alumno*, al haberse construido dentro del ámbito del *Examen*, compartirá con él dicho ámbito y vigencia (tiempo de vida).

Como has podido comprobar, la diferencia entre agregación y composición es muy sutil y depende de la técnica de programación utilizada.

Ahora, la clase *Principal* quedará así:

```
public class Principal {
    public static void main(String[] args) {
        Examen ex = new Examen
            ("Matemáticas", 7.5F, "Miguel Pérez", "616669966");
        System.out.println(ex.toString());
        System.out.println(ex.getAlumno().toString());
    }
}
```

## Herencia

La herencia es una característica de la programación orientada a objetos que permite crear una clase que hereda las propiedades y métodos de otra clase madre.

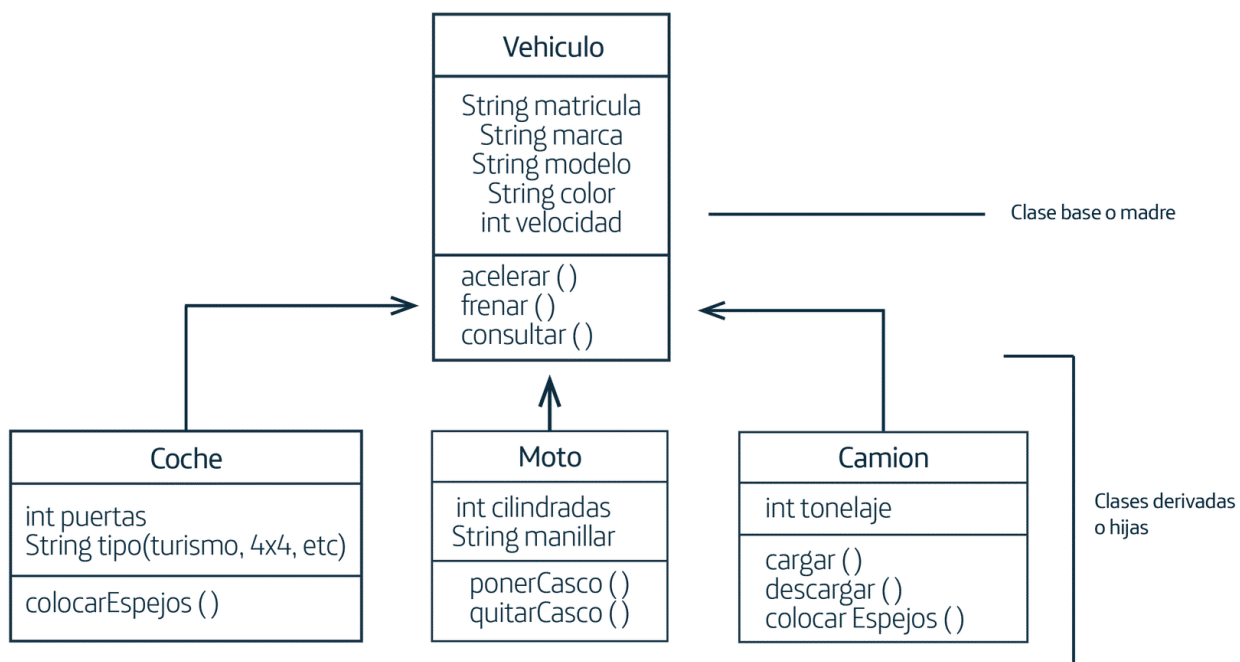
En la herencia se distinguen dos tipos de clase:

- Clase **base** o clase madre.
- Clase **derivada** o clase hija, que hereda las características de la madre.

La herencia es uno de los aspectos más importantes del paradigma de programación orientada a objetos y la distingue de otros paradigmas de programación.

**Una clase hija es una especialización de la clase madre**, por ejemplo: un coche es una especialización de un vehículo, pasamos de algo más abstracto a algo más concreto.

Tiene sentido aplicar la herencia a la hora de diseñar una nueva clase cuando existe una relación de tipo “**es un**”.



- Un coche **es un** vehículo.
- Una moto **es un** vehículo.
- Un camión **es un** vehículo.

## Ejemplo de herencia

Vamos a crear una clase llamada *Persona* con la misma estructura de la anterior clase *Alumno*.

```
public class Persona {
    private String nombre;
    private String telefono;

    public Persona(String nombre, String telefono) {
        this.nombre = nombre;
        this.telefono = telefono;
    }

    @Override
    public String toString() {
        return this.nombre + " - " + this.telefono;
    }
}
```

Ahora vamos a construir las clases derivadas *Alumno* y *Profesor*.

```
public class Alumno extends Persona {
    private int numMatricula;

    public Alumno(String nombre, String telefono, int numMatricula) {
        super(nombre, telefono);
        this.numMatricula = numMatricula;
    }

    @Override
    public String toString() {
        return "Alumno: " + super.toString() +
            " N° matrícula " + this.numMatricula + "];"
    }
}
```

```
public class Profesor extends Persona {
    private String especialidad;

    public Profesor(String nombre, String telefono, String especialidad) {
        super(nombre, telefono);
        this.especialidad = especialidad;
    }

    @Override
    public String toString() {
        return "Profesor: " + super.toString() + " Especialidad "
            + this.especialidad;
    }
}
```



## 5.1. Creación de componentes: propiedades y atributos

Vamos a analizar el código:

1. El constructor de una clase derivada se complica un poquito. El método constructor de *Alumno* no solo tiene que ocuparse de inicializar los valores de sus propiedades (en este caso la propiedad *numMatricula*), sino que también debe suministrar al constructor de la clase madre *Persona* los valores que necesita. Para ello se invoca al constructor de la clase madre de la siguiente manera:

```
super(argumentos);
```

Para nuestro ejemplo:

```
super(nombre, telefono);
```

El constructor de *Alumno* tiene 3 parámetros, los 2 primeros son suministrados al constructor de la clase *Persona* y el último es asignado directamente al constructor de la clase *Alumno*.

**La palabra *super* debe ser lo primero que aparezca en el constructor.**

2. La clase *Alumno* también ha heredado el método *toString()* de *Persona* y lo ha sobrescrito para especializarlo más. A su vez, en la clase *Persona* ya estaba sobrescrito del original de la clase *Object* (superclase de la que heredan todas las clases).

```
@Override
```

```
public String toString() {
    return "Alumno " + super.toString() + " N° matrícula = " + this.numMatricula + "];"
}
```

La anotación **@Override** informa de que este método ya existe en la clase base (*Persona*) pero está siendo sobrescrito en la clase derivada. **Entonces, ¿cuál de los dos métodos ejecuta cuando lo invocamos desde la clase *Alumno*?** La respuesta está de nuevo en la utilización de la palabra *super*, que nos permite el acceso a los métodos de la clase base.

*super.toString()* - Ejecuta el método *toString()* de *Persona*.

*toString()* - Ejecuta el método *toString()* de *Alumno*.

Ahora, en la clase *Principal* puedes crear objetos más especializados para representar un alumno o un profesor en lugar de una persona sin más.

```
public class Principal {
    public static void main(String[] args) {
        Profesor pro = new Profesor("Luis Pérez", "913332211",
"Matemáticas");
        Alumno alu = new Alumno("Alicia Robles", "914445566", 3899);
        System.out.println(pro.toString());
        System.out.println(alu.toString());
    }
}
```

## Polimorfismo

En el contexto de la programación orientada a objetos, el polimorfismo se refiere a la capacidad de un objeto para adoptar distintos comportamientos y se puede lograr de varias formas.

Permite que una referencia a un objeto pueda apuntar a distintos tipos de objetos. Una referencia a un objeto de tipo *Figura* puede apuntar a objetos de tipo *Triangulo*, *Rectangulo* o *Circulo*, igual que una referencia a un objeto *Animal* puede apuntar a un objeto *Pulga* o *Tiranosaurio*.

Para nuestro ejemplo anterior, podemos tener una referencia a un objeto *Persona* que apunte indistintamente a un objeto *Profesor* o a un objeto *Alumno*.

El polimorfismo facilita la reutilización de código, ya que podemos utilizar el mismo método para procesar distintos tipos de objetos.

```
public class Principal {  
    public static void main (String[] args) {  
        Profesor pro = new Profesor("Luis Pérez","913332211","Matemáticas");  
        Alumno alu = new Alumno("Alicia Robles", "914445566",3899);  
        procesar(pro);  
        procesar(alu);  
    }  
  
    public static void procesar(Persona p) {  
        System.out.println(p.toString());  
        System.out.println(p.saludar());  
    }  
}
```

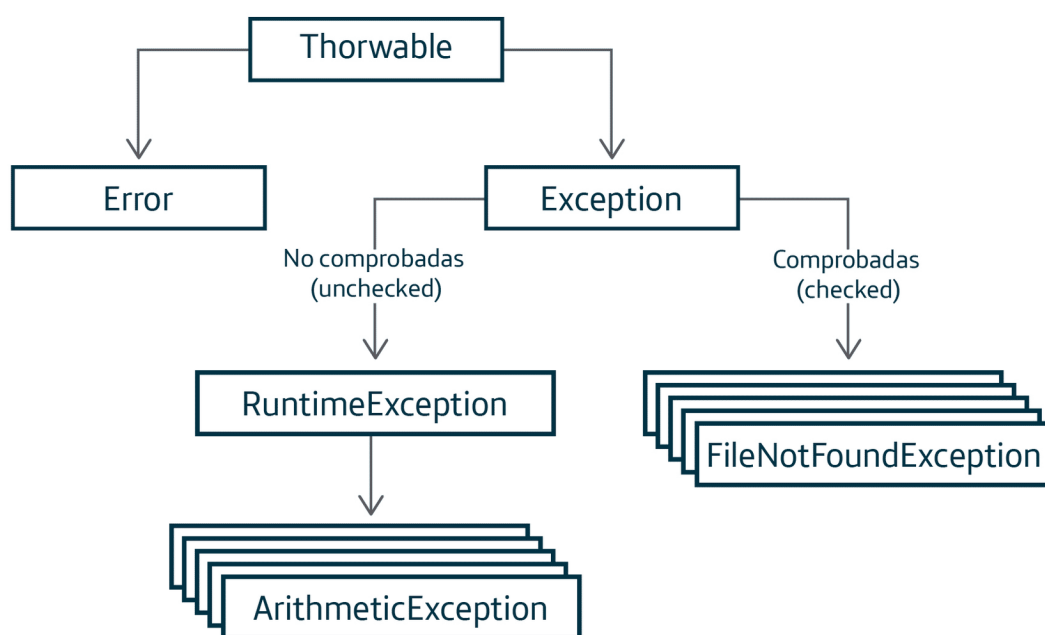
El método *procesar* es capaz de trabajar igualmente con un objeto *Profesor* que con un objeto *Alumno*, ya que ambos son de tipo *Persona*.

Esta es una de las grandes ventajas del polimorfismo.

# Relación entre polimorfismo y excepciones

## Uso de objetos de excepción

Cuando se produce una condición de error (excepción para Java), la máquina virtual genera un tipo de objeto especial con información sobre el suceso ocurrido; un objeto de tipo *Exception*. La jerarquía de clases para la gestión de excepciones en Java se puede resumir con la siguiente imagen:



Jerarquía de clases de excepción en Java.

La clase principal de la que heredan todas las clases que representan excepciones es ***Throwable***.

Observando la estructura jerárquica de la imagen, puedes comprobar que *Throwable* se descompone en dos ramas, que representan dos clases de situaciones de error:

- ***Error***: representan situaciones que se escapan al control del programador, por lo que no deberíamos hacer nada con ellas. Por ejemplo, un error grave en el funcionamiento de la máquina virtual de Java que escapa a nuestro control.
- ***Exception***: representan situaciones que el programador sí puede gestionar y por lo tanto, será de este tipo de clases de las que nos ocupemos en esta lección. Los objetos de este tipo son los que realmente denominamos excepciones.

## Ejemplo de programa que genera un error o excepción

Prueba a ejecutar este pequeño programa, donde realizamos una operación claramente incorrecta; una división cuyo divisor es un cero:

```
public class Principal {  
    public static void main(String args[]) {  
        int cero=0;  
        int resul=6/cero;  
        System.out.println(resul);  
    }  
}
```

Programa que genera una situación de error o excepción.

Como resultado has obtenido un mensaje de error o excepción como el siguiente:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at DivCero.main(DivCero.java:4)
```

La máquina virtual de Java (JVM) ha generado un objeto de la clase *ArithmeticException* y ha abortado el programa. A lo largo de la unidad aprenderás cómo puedes capturar dicho objeto para recuperar el control sobre la ejecución del programa y evitar que aborte la ejecución.

## Tipos de excepciones (clases derivadas de *Exception*)

Volviendo de nuevo a observar la imagen, puedes comprobar que existen dos tipos de clases que derivan de *Exception*:

- **Excepciones no comprobadas que derivan de *RuntimeException*:** no estamos obligados a controlar estas excepciones. Tienen que ver con situaciones en que existe un mal planteamiento en el código. Puede evitarse que lleguen a ocurrir y no hay necesidad de tener que controlarlas, aunque podemos hacerlo. Nuestro ejemplo anterior de la división entre cero ha provocado una *exception* de tipo *ArithmeticException*, este es un ejemplo de excepción que no es obligatorio controlar. También está claro que podría evitarse esta situación antes de realizar la división.
- **Excepciones comprobadas que derivan directamente de *Exception*:** son situaciones de error de las que podemos recuperarnos, pero que no son fruto de un mal planteamiento del código, sino de una situación inesperada, tal como cuando abrimos un fichero de texto y de repente alguien ha borrado dicho fichero del disco. Debemos controlar este tipo de excepciones para que el programa pueda recuperarse de la situación de error sin necesidad de abortarlo.

## 5.1. Creación de componentes: propiedades y atributos

### ¿Y cómo se controlan las excepciones?

Un bloque de código susceptible de producir una excepción debe encerrarse en un bloque **try {...}** seguido de un bloque **catch {...}**, que capturará el objeto de excepción. Debes utilizar el siguiente formato:

```
try {
    // Sentencias que pueden provocar excepción
}
catch (Exception e) {
    // Respuesta a la situación de excepción
}
finally {
    // Sentencias que se ejecutan incondicionalmente
}
```

Formato *try ... catch ... finally*.

- **Bloque try:** donde se encierran las instrucciones que pueden provocar una situación de excepción, ya sea comprobada o no comprobada. La diferencia está en que si se trata de excepciones comprobadas, es obligatorio usar un bloque *try* y si se trata de excepciones no comprobadas, el uso de *try* es opcional.
- **Bloque catch:** donde se captura el objeto de excepción. Si dentro del bloque *try* ocurre una situación de excepción, el control pasa al siguiente bloque *catch*, capaz de recoger dicha excepción.
- **Bloque finally:** este bloque es opcional y contiene sentencias que se ejecutarán de manera incondicional ocurra o no una excepción.

```
public class Principal {
    public static void main(String args[]) {
        int cero=0;
        int resul;
        try {
            resul=6/cero;
            System.out.println(resul);
        }
        catch (ArithmeticException e) {
            System.out.println("Se ha producido una excepción");
            System.out.println(e.getMessage());
        }
        finally {
            System.out.println("Hasta pronto");
        }
    }
}
```

## Paso a paso

**Vamos ver paso a paso lo que ocurre con el programa anterior:**

- La sentencia `resul=6/cero;` provoca una excepción, por lo que la máquina virtual de Java genera un objeto de tipo *ArithmeticException*. Al encontrarse dentro de un bloque *try*, el control pasará al primer bloque *catch* que pueda recoger un objeto de tipo *ArithmeticException* (pueden existir varios bloques *catch*). En el ejemplo, el objeto es recogido en la variable `e`.
- Dentro del bloque *catch*, el programa informa de lo ocurrido y utiliza el método `getMessage()` de la clase *Exception* para mostrar la descripción del error o excepción. Recuerda que *ArithmeticException* hereda de *Exception*, luego el objeto `e` es tanto una *ArithmeticException* como una *Exception*.
- La sentencia `System.out.println(resul);` será pasada por alto, no llegará a ejecutarse, ya que tras ocurrir la excepción en la línea anterior, el control pasó al bloque *catch*.
- Por último se ejecutará el bloque *finally*.

**¿Y para qué sirve el bloque *finally*? Vas a descubrirlo con una pequeña práctica.**

Pega este pequeño programa en un proyecto Eclipse y ejecútalo:

```
import java.util.Scanner;

public class Principal {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce radio de la circunferencia: ");
        String num = lector.nextLine();
        lector.close();
        int radio;
        try {
            radio = Integer.parseInt(num);
        } catch (NumberFormatException e) {
            System.out.println("Ha ocurrido una excepción de tipo
NumberFormatException");
            System.out.println(e.getMessage());
            return;
        }
        System.out.println("Longitud: " + (2*Math.PI*radio));
        System.out.println("Area: " + (Math.PI*radio*radio));
        System.out.println("Fin del programa");
    }
}
```

Introducimos por teclado el radio de una circunferencia como un texto, dentro de un bloque *try* lo convertimos a valor numérico. Al final del programa estamos calculando y mostrando en pantalla la longitud y el área de la circunferencia, pero como no tiene sentido que estas sentencias se ejecuten si hay error en la entrada de datos, hemos colocado un *return* al final del bloque *catch*, lo que provoca que termine la ejecución del programa si se desencadenó la excepción y no ejecuta el resto del código que queda en la función. Sin embargo, nos gustaría

que pase lo que pase aparezca el mensaje de "*Fin del programa*", y es aquí donde viene a salvarnos el bloque *finally*.

```
import java.util.Scanner;

public class Principal {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce radio de la circunferencia: ");
        String num = lector.nextLine();
        lector.close();
        int radio;
        try {
            radio = Integer.parseInt(num);
        } catch (NumberFormatException e) {
            System.out.println("Ha ocurrido una excepción de tipo
NumberFormatException");
            System.out.println(e.getMessage());
            return;
        } finally {
            System.out.println("Fin del programa");
        }
        System.out.println("Longitud: " + (2*Math.PI*radio));
        System.out.println("Area: " + (Math.PI*radio*radio));
    }
}
```

El bloque *finally* se ejecuta siempre, independientemente del código que esté encerrado en cada uno de los *catch*.

## ¿Cómo se trabaja con las excepciones comprobadas?

Las excepciones comprobadas son las que pueden producir determinados métodos, cuyas llamadas deben estar obligatoriamente encerradas dentro de un bloque *try* o, en su defecto, estas excepciones también pueden ser relanzadas para que las gestione el método superior dentro de la cadena de llamadas.

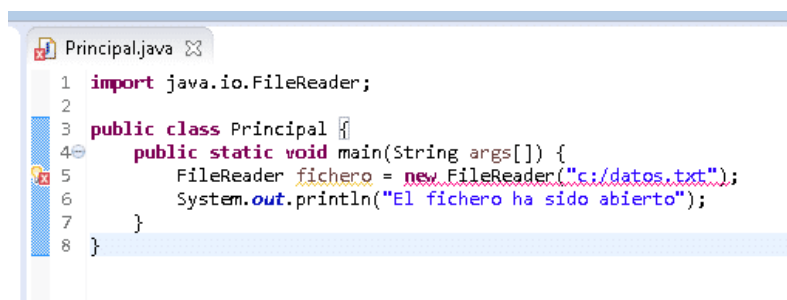
En nuestro ejemplo de la división entre 0 no estábamos obligados a poner un bloque *try*. El programa no tenía errores de compilación, aunque al ejecutar, nos lanzaba la excepción, pero hay determinadas situaciones en que estamos obligados a gestionar las situaciones de error. Vamos a ver un pequeño ejemplo para que puedas comprenderlo mejor.

Prueba a crear este pequeño programa dentro de Eclipse:  
`import java.io.FileReader;`

```
public class Principal {
    public static void main(String args[]) {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```



Verás que ni siquiera has intentado ejecutar y ya estás teniendo problemas, Eclipse te está indicando que tienes errores de compilación, tu programa no puede ni siquiera ser ejecutado.



El error de compilación viene porque el método constructor de *FileReader* abre un fichero para lectura cuya ruta y nombre se especifica como argumento. El fichero especificado podría no existir, y eso es algo que escapa a nuestro control, porque no podemos predecir si el usuario va a borrar el archivo o lo va a mover de sitio. Por esta razón, estamos obligados a controlar dicha excepción.

Para poder ejecutar el programa tienes que usar obligatoriamente un bloque *try*. Cambia ahora el código de esta manera:

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        try {
            FileReader fichero = new FileReader("c:/datos.txt");
            System.out.println("El fichero ha sido abierto");
        } catch (FileNotFoundException e) {
            System.out.println("Error al abrir el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

En resumen:

- **Excepciones comprobadas:** *try ... catch* obligatorio.
- **Excepciones no comprobadas:** *try ... catch* opcional.

## Provocando excepciones no comprobadas

En este apartado tendrás oportunidad de provocar excepciones de tipo "no comprobadas". Como no es obligatorio usar *try ... catch* en este tipo de excepciones, no lo usaremos, solo vamos a familiarizarnos con más tipos de excepciones, a parte de la ya conocida *ArithmeticException*.

Ve probando con Eclipse los pequeños programas que te vamos a proponer. Ejecuta cada ejemplo para que veas con tus propios ojos lo que ocurre.

## 5.1. Creación de componentes: propiedades y atributos

### 1. Salida de los límites de un *array*

```
public class Principal {
    public static void main(String args[]) {
        int nums[] = new int[3];
        nums[4]=25;
    }
}
```

Estamos accediendo a una posición inexistente del *array* y se produce la excepción *ArrayIndexOutOfBoundsException*.

### 2. Salida de los límites de una colección

```
import java.util.ArrayList;

public class Principal {
    public static void main(String args[]) {
        ArrayList<Integer> nums = new ArrayList<Integer>();
        nums.add(25);
        nums.add(56);
        nums.add(18);
        System.out.println(nums.get(4));
    }
}
```

Estamos intentando recuperar un elemento de la colección de una posición inexistente. Se produce la excepción *IndexOutOfBoundsException*.

### 3. Error en la conversión entre tipos de datos

```
public class Principal {
    public static void main(String args[]) {
        String texto = "pepe";
        int num = Integer.parseInt(texto);
    }
}
```

Estamos intentando convertir el valor de tipo *String* "pepe" a formato numérico, cosa que no es posible y provoca una excepción de tipo *NumberFormatException*.

### 4. Intento de utilización de una referencia con valor *null*

```
public class Principal {
    public static void main(String args[]) {
        String texto=null;
        System.out.println(texto.toString());
    }
}
```

Estamos intentando utilizar una referencia a un *String* que en realidad no apunta a ningún objeto, tiene el valor *null*. Se produce una excepción de tipo *NullPointerException*.

Recuerda que aunque no tienes obligación de controlar estas excepciones con un *try ... catch*, aunque puedes hacerlo si lo consideras necesario.

## Varios catch

En esta ocasión probaremos con un pequeño programa que puede producir dos tipos distintos de excepciones.

Este pequeño programa permite al usuario introducir por teclado los valores de un dividendo y un divisor, almacenándolos en variables tipo *String*. Luego, convierte ambos valores a datos numéricos de tipo *int* y por último calcula la división y el resto, mostrándolos en pantalla.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce dividendo: ");
        String texto = lector.nextLine();
        int dividendo = Integer.parseInt(texto);
        System.out.println("Introduce divisor: ");
        texto = lector.nextLine();
        int divisor = Integer.parseInt(texto);
        int resultado = dividendo/divisor;
        int resto = dividendo%divisor;
        System.out.println("Resultado división: " + resultado);
        System.out.println("Resto: " + resto);
        lector.close();
    }
}
```

Esto puede dar lugar a tres situaciones diferentes:

### 1. Que el usuario introduzca correctamente los dos valores y el programa funcione sin excepciones.

```
<terminated> Principal (16) [Java App]
Introduce dividendo:
16
Introduce divisor:
3
Resultado división: 5
Resto: 1
```

### 2. Que el usuario introduzca un valor de texto que no pueda ser convertido a número.

```
<terminated> Principal (16) [Java App]
Introduce dividendo:
16
Introduce divisor:
pepe
Exception in thread "main" java.lang.NumberFormatException: For input string: "pepe"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at Principal.main(Principal.java:11)
```

### 3. Que el usuario introduzca 0 como divisor.

```

Introduce dividendo:
16
Introduce divisor:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Principal.main(Principal.java:12)

```

## ¿Cómo podemos tener controladas todas estas situaciones?

Tras un bloque *try* pueden existir tantos bloques *catch* como sean necesarios:

```

import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        try {
            Scanner lector = new Scanner(System.in);
            System.out.println("Introduce dividendo: ");
            String texto = lector.nextLine();
            int dividendo = Integer.parseInt(texto);
            System.out.println("Introduce divisor: ");
            texto = lector.nextLine();
            int divisor = Integer.parseInt(texto);
            int resultado = dividendo/divisor;
            int resto = dividendo%divisor;
            System.out.println("Resultado división: " + resultado);
            System.out.println("Resto: " + resto);
            lector.close();
        }
        catch (ArithmeticException e1) {
            System.out.println("Se ha producido una ArithmeticException");
            System.out.println(e1.getMessage());
        }
        catch (NumberFormatException e2) {
            System.out.println("Se ha producido un NumberFormatException");
            System.out.println(e2.getMessage());
        }

        System.out.println("El programa sigue aquí, no se ha abortado");
    }
}

```

El objeto de excepción será recogido por el bloque *catch* cuyo tipo de argumento coincida con el tipo de excepción que se ha producido.

Cuando colocamos varios *catch* es importante tener en cuenta el orden en que se colocan. Deben ir situados en orden de las clases más específicas a las más genéricas, o lo que es lo mismo, desde las inferiores en el árbol jerárquico a las superiores.

## 5.1. Creación de componentes: propiedades y atributos

Observa este código:

```
try {
    FileReader f = new FileReader("C:/datos.txt");
    int x = f.read();
    System.out.println(x);
    f.close();
} catch (FileNotFoundException e1) {
    System.out.println("El fichero no se encuentra");
} catch (IOException e2) {
    System.out.println(e2.getMessage());
} catch (Exception e3) {
    System.out.println(e3.getClass().getName());
    System.out.println(e3.getMessage());
}
```

Dentro del bloque *try* intentamos abrir un fichero para lectura y después el programa realiza alguna operación de lectura. No te preocupes por comprender el código encerrado en el *try*, lo único que nos ocupa ahora es la comprensión de las excepciones. Si el fichero no puede abrirse por cualquier circunstancia, irá recorriendo secuencialmente los bloques *catch* en busca de la primera referencia que pueda capturar la excepción que se ha producido. Si el primer bloque *catch* fuese el de *Exception*, hubiera capturado la excepción sea cual sea, ya que un objeto *FileNotFoundException* o *IOException* también es *Exception*.

### Multicatch

Una de las novedades a partir de la versión 8 de Java son los llamados *multicatch*, capaces de recoger con un solo bloque *catch* distintos tipos de excepciones. Para separar cada uno de los tipos de excepción se utiliza el carácter `|`.

El programa anterior utilizando *multicatch* quedaría así:

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        try {
            Scanner lector = new Scanner(System.in);
            System.out.println("Introduce dividendo: ");
            String texto = lector.nextLine();
            int dividendo = Integer.parseInt(texto);
            System.out.println("Introduce divisor: ");
            texto = lector.nextLine();
            int divisor = Integer.parseInt(texto);
            int resultado = dividendo/divisor;
            int resto = dividendo%divisor;
            System.out.println("Resultado división: " + resultado);
            System.out.println("Resto: " + resto);
            lector.close();
        }
        catch (NumberFormatException | ArithmeticException e) {
            System.out.println("Se ha producido una excepción de tipo " +
                e.getClass().getName());
            System.out.println(e.getMessage());
        }

        System.out.println("El programa sigue aquí, no se ha abortado");
    }
}
```

Utilizamos `e.getClass().getName()` para averiguar cuál de las dos excepciones ocurrió.

## Polimorfismo y excepciones

Recuerda que todas las clases que representan excepciones heredan directa o indirectamente de la clase *Exception*.

Esto significa que los objetos de tipo *NumberFormatException*, *ArithmeticException*, *IndexOutOfBoundsException*, *FileNotFoundException*, etc., son a la vez objetos *Exception*. Esto nos permite aplicar las ventajas del polimorfismo, es decir, tratar todos estos objetos de manera genérica como *excepciones*.

Sea cual sea el tipo de excepción que produzca nuestro programa, podrá ser capturada de esta manera:

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        try {
            Scanner lector = new Scanner(System.in);
            System.out.println("Introduce dividendo: ");
            String texto = lector.nextLine();
            int dividendo = Integer.parseInt(texto);
            System.out.println("Introduce divisor: ");
            texto = lector.nextLine();
            int divisor = Integer.parseInt(texto);
            int resultado = dividendo/divisor;
            int resto = dividendo%divisor;
            System.out.println("Resultado división: " + resultado);
            System.out.println("Resto: " + resto);
            lector.close();
        }
        catch (Exception e) {
            System.out.println("Se ha producido una Excepción de tipo " +
e.getClass().getName());
            System.out.println(e.getMessage());
        }

        System.out.println("El programa sigue aquí, no se ha abortado");
    }
}
```

Una referencia de tipo *Exception* puede apuntar a cualquier tipo de objeto de excepción. Utilizamos *getClass().getName()* para averiguar qué excepción ocurrió.

Aunque este sistema resulta muy cómodo, no hay que abusar de su uso, ya que evita el hecho de poder tratar cada tipo de excepción de manera más especializada y profesional.

## Relanzamiento o propagación de excepciones

Aunque estamos obligados a controlar las excepciones de tipo "comprobadas", también podemos propagarlas o relanzarlas hacia arriba para que sean controladas en el método superior en la pila de llamadas.

Observa este ejemplo y prueba a añadirlo a un proyecto Eclipse:

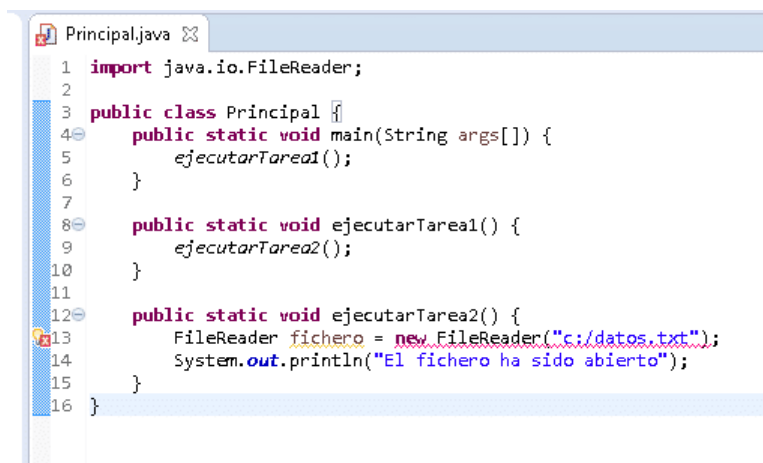
```
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

El método *ejecutarTarea2()* es invocado desde el método *ejecutarTarea1()*, que es llamado desde el método *main()*. Si ya has pegado el código en Eclipse estarás comprobando que de nuevo te da error de compilación porque es obligatorio controlar la posible excepción de tipo *FileNotFoundException* que puede provocar el constructor de *FileReader*. Deberíamos añadir el bloque *try ... catch* en el método *ejecutarTarea2()*.



Sin embargo, también **es posible que el método *ejecutarTarea2()* propague o relance la excepción hacia arriba** para responsabilizar al método superior. **Esto se hace con una declaración *throws* en la cabecera del método, indicando las excepciones que vayan a relanzarse.**



## 5.1. Creación de componentes: propiedades y atributos

```
import java.io.FileNotFoundException;
import java.io.FileReader;

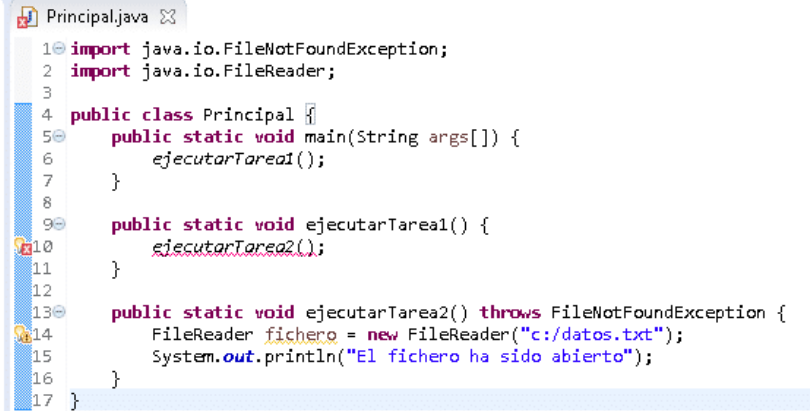
public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

Propagación de la excepción desde *ejecutarTarea2()* hasta *ejecutarTarea1()*.

En este caso le hemos pasado la responsabilidad al método *ejecutarTarea1()*, que es el que ahora tiene el problema, tal y como puedes apreciar en la siguiente imagen:



```
Principal.java
1 import java.io.FileNotFoundException;
2 import java.io.FileReader;
3
4 public class Principal {
5     public static void main(String args[]) {
6         ejecutarTarea1();
7     }
8
9     public static void ejecutarTarea1() {
10        ejecutarTarea2();
11    }
12
13    public static void ejecutarTarea2() throws FileNotFoundException {
14        FileReader fichero = new FileReader("c:/datos.txt");
15        System.out.println("El fichero ha sido abierto");
16    }
17 }
```

Por otro lado, el método *ejecutarTarea1()* también podría "escurrir el bulto" pasando el problema al método *main()* de la misma forma:

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

Propagación de la excepción desde *ejecutarTarea1()* hasta *main()*.

## 5.1. Creación de componentes: propiedades y atributos

Ahora es el método *main()* el que tiene el problema.

```

1 import java.io.FileNotFoundException;
2 import java.io.FileReader;
3
4 public class Principal {
5     public static void main(String args[]) {
6         ejecutarTarea1();
7     }
8
9     public static void ejecutarTarea1() throws FileNotFoundException {
10        ejecutarTarea2();
11    }
12
13    public static void ejecutarTarea2() throws FileNotFoundException {
14        FileReader fichero = new FileReader("c:/datos.txt");
15        System.out.println("El fichero ha sido abierto");
16    }
17 }

```

En el método *main()* podemos quitar los problemas de compilación de dos formas:

**1.** De nuevo propagando la excepción hacia arriba, pero como ya no hay más métodos para recogerla, será recibida por la máquina virtual de Java, que abortará el programa en el caso de que realmente se produzca la excepción, es decir, en el caso de que no exista el fichero que estamos intentando abrir.

```

import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) throws FileNotFoundException {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}

```

## 2. Colocar un *try ... catch*:

```

import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        try {
            ejecutarTarea1();
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
            return;
        }
    }

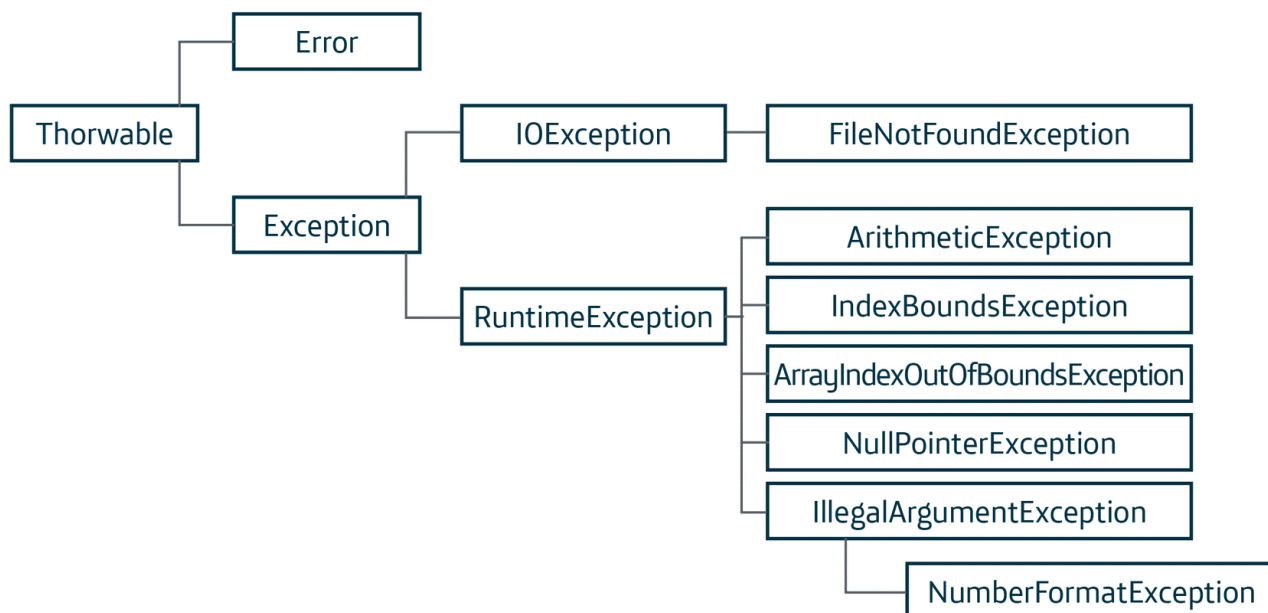
    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}

```

## Resumen de las excepciones más habituales

Pulsa sobre los círculos para obtener información adicional sobre las clases de excepción que hemos visto durante la unidad.



### Throwable

Clase base de la que derivan todas las demás clases de excepción.

### Error

Errores graves que escapan a nuestro control, tal como un fallo de la máquina virtual de Java.

### Exception

Excepciones que sí podemos controlar y que se clasifican en:

- Comprobadas: las que derivan directamente de *Exception*. Es obligatorio gestionirlas.
- No comprobadas: las que derivan de *RuntimeException*. No es obligatorio gestionirlas.

### IOException

Excepciones producidas durante operaciones de entrada y/o salida.

### RuntimeException

Clase base de la que derivan todas las excepciones no comprobadas.

### FileNotFoundException

Se está intentando acceder a un fichero que no existe.

### ArithmeticException

Errores en operaciones aritméticas, como por ejemplo una división entre 0.

### **IndexOutOfBoundsException**

Intento de acceso a una posición inexistente de una colección.

### **ArrayIndexOutOfBoundsException**

Intento de acceso a un elemento de un *array* que existe, es decir, intento de salirse de los límites del *array*.

### **NullPointerException**

Intento de acceder a una propiedad o a un método a partir de una referencia nula.

### **IllegalArgumentException**

Se pasa un argumento a una función que resulta inválido para la operación que la función debe realizar.

### **NumberFormatException**

Formato de número incorrecto. Ocurre cuando realizamos una conversión de texto a número y el texto no puede interpretarse como un número.

# Despedida

## Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Las **clases son abstracciones de entidades en el mundo real** que sirven como modelos para construir objetos basados en ellas.
- Las clases **tienen atributos**. Los atributos pueden pertenecer al objeto (**propiedades del objeto**) o a la clase (**propiedades que pertenecen a la clase**). Los atributos que pertenecen a la clase se declaran con el modificador *static*.
- Los objetos **realizan acciones** que se implementan por medio de **métodos**. Los métodos pueden recibir argumentos y retornar un valor.
- **Los métodos pueden estar sobrecargados**, es decir, estar implementados de distintas maneras variando el número y/o tipo de parámetros.
- La interfaz de una clase está definida por las propiedades y métodos que dejamos ver al exterior.
- La **agregación y la composición son tipos de relaciones entre clases, donde una propiedad de una clase es un objeto de otra clase**.
- La **herencia** es una característica de la programación orientada a objetos que permite **crear una clase que hereda las propiedades y métodos de otra clase madre**.
- En el contexto de la programación orientada a objetos, **el polimorfismo se refiere a la capacidad de un objeto para adoptar distintos comportamientos y se puede lograr de varias formas**.
- Cuando se produce una **condición de error o excepción**, se crea un objeto que representa esa excepción y se envía al método en el que se ha producido el error para que lo maneje.
- **Todas las excepciones** que podemos capturar **heredan de la clase *Exception***.