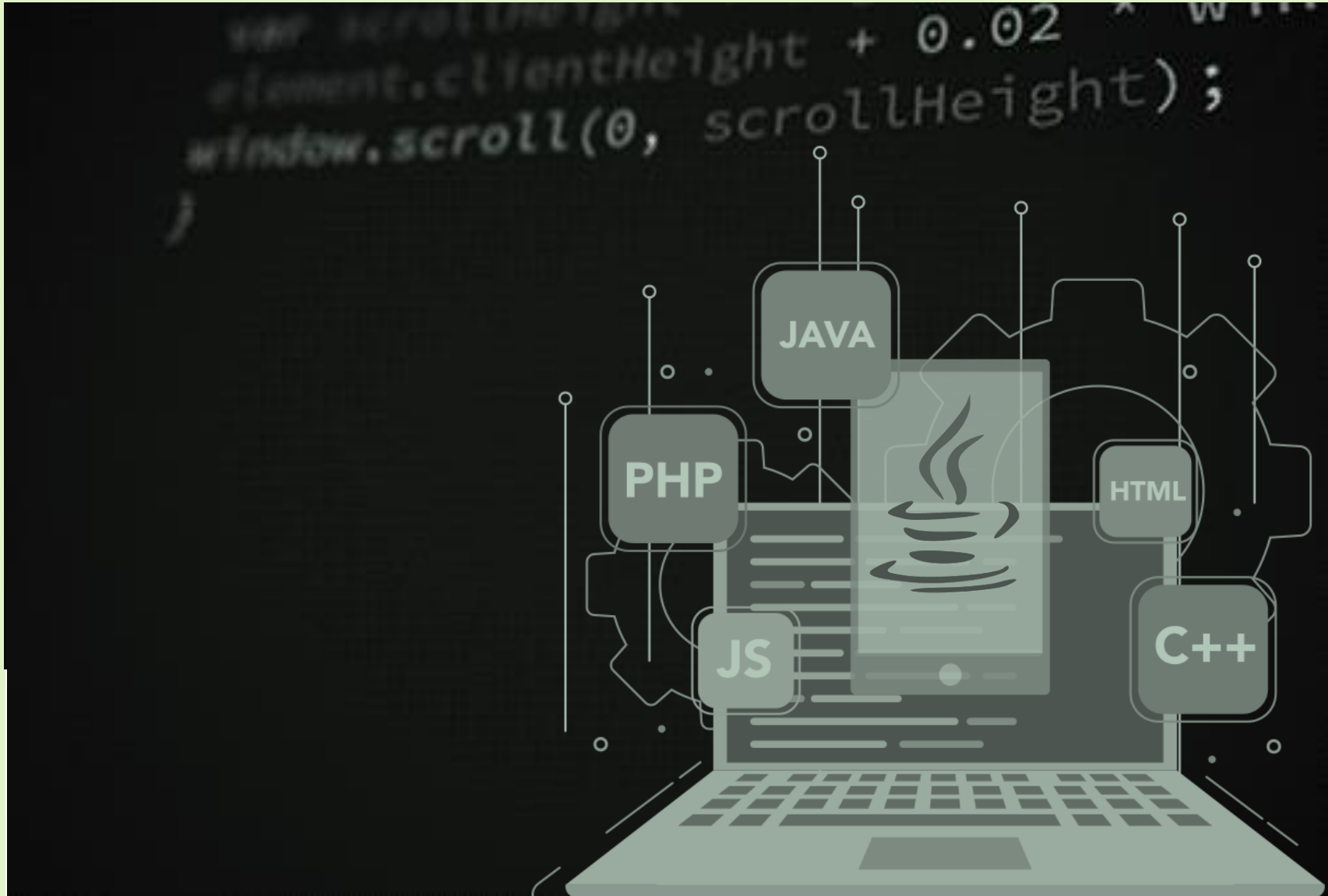
EFA
MORATALAZ

1º CFGS Desarrollo de Aplicaciones Web

PROGRAMACIÓN

DANIEL GONZÁLEZ-CALERO JIMÉNEZ

UT6 – PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA



EFA
MORATALAZ

1º CFGS Desarrollo de Aplicaciones Web

PROGRAMACIÓN

UT6 – PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

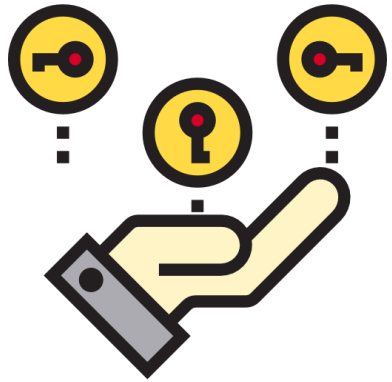
INDICE

INTRODUCCIÓN

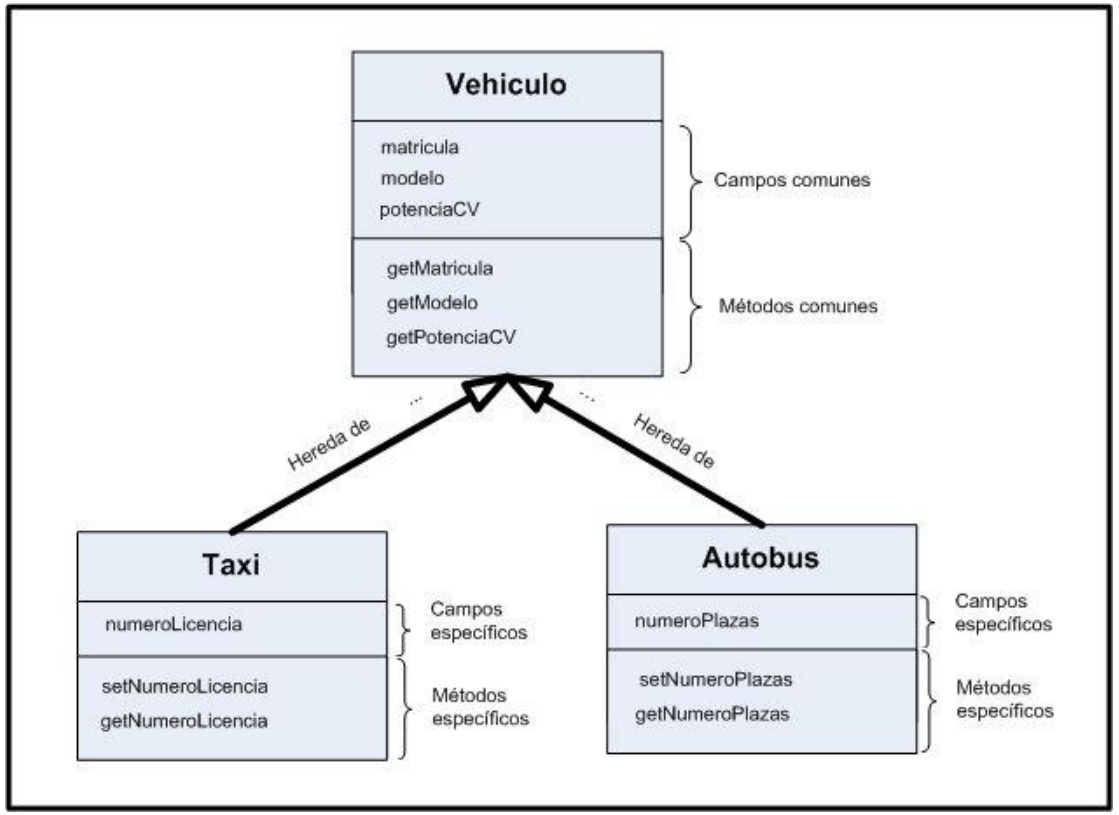
La **encapsulación** consiste en tener las variables de clase declaradas como **private** para evitar que cuando instanciemos un objeto podamos acceder a ellas (por seguridad).

Entonces, ¿cómo accedemos a los valores de esas variables de clase?

Mediante los métodos **getter** y **setter** que hemos declarado anteriormente.



La **herencia** en Java implica que una superclase hereda sus funciones y atributos a una subclase. La palabra reservada que nos permite realizar herencia entre clases es **extends.**




```
class Persona {

    // atributos de la clase padre
    String nombre;
    int edad;
    int telefono;

}
```

```
class Cliente extends Persona{

    //atributo de la clase hija
    int credito;
}
```

La **abstracción** es un concepto muy importante y se utiliza para reducir la complejidad de un programa, a través del uso de clases e interfaces abstractas, que permite la reutilización de código y la simplificación del mismo a través de la herencia.

Una **clase abstracta** es una clase base para otras clases llamadas “clases concretas”. La creación del objeto se hará a través de sus clases hijas, las cuales están obligadas a implementar los métodos abstractos definidos en la clase principal.

```
public abstract class Felino {
  // clase abstracta

  public abstract void Alimentarse(); //metodo abstracto

}
```

```
public  class Gato extends Felino
{
public void Alimentarse()
  {
  System.out.println(“El Gato come croquetas”);
  }
}
```

El **polimorfismo** es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros (diferentes implementaciones) utilizados durante su invocación. Dicho de otro modo, el objeto como entidad puede contener valores de diferentes tipos durante la ejecución del programa.

```
class Animal {
  public void makeSound() {
    System.out.println("Grr...");
  }
}
class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
}
class Dog extends Animal {
  public void makeSound() {
    System.out.println("Woof");
  }
}
```

```
public static void main(String[ ] args) {
  Animal a = new Dog();
  Animal b = new Cat();
}
```

```
a.makeSound();
//Outputs "Woof"

b.makeSound();
//Outputs "Meow"
```

ASOCIACIÓN, AGREGACIÓN Y COMPOSICIÓN DE CLASES

Los objetos de una o varias clases interactúan entre sí y cada uno puede existir por sí mismo sin depender del otro.

```java
public class Universidad {

    private String nombre;

    public Universidad(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}
```

```java
public class Alumno {

    private String nombre;

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

    public String getNombre() {
        return nombre;
    }
}
```

La relación existente entre los objetos Universidad y Alumno se da en el programa principal, de manera que ambos pueden existir en el sistema sin depender uno del otro.

La palabra reservada *this* representa al objeto desde el que se está ejecutando el código. De esta manera permite acceder a sus campos cuando existen parámetros con el mismo nombre.

```
public class Asociacion {
    public static void main(String[] args) {
        Universidad u = new Universidad("Universidad Oxford");
        Alumno a = new Alumno("Flora");
        System.out.println(a.getNombre()+" es alumn@ de "+u.getNombre());
    }
}
```

Es una *asociación* entre un objeto compuesto y objetos componentes, en la que los componentes pueden existir por sí mismos sin necesidad del compuesto. Este es un programa de ejemplo que utiliza agregación:

```
public class Persona {

    private Persona padre;

    Persona (Persona padre) {
        this.padre = padre;
    }

}
```

Un objeto persona podrá existir aunque no exista el objeto persona que represente a su padre.

Es una *asociación* entre un objeto compuesto y objetos componentes, en la que los componentes no existen si no existe el compuesto. Este es un programa ejemplo que utiliza composición:

```java
public class Punto {

    private int x;
    private int y;

    public Punto (int coord_X, int coord_Y) {
        x = coord_X;
        y = coord_Y;
    }

    public Punto() {
        x=0;
        y=0;
    }
}
```

```java
public class Circunferencia {

    private Punto centro = new Punto();
    private int radio;

    public Circunferencia(int radio) {
        this.radio = radio;
    }
}
```

No tiene sentido la existencia del objeto punto que formará el centro de la circunferencia en ausencia de ésta. Si el objeto circunferencia deja de existir, también dejará de existir el objeto punto que representa su centro.

HERENCIA

La *herencia* permitirá declarar una nueva clase (derivada, extendida, hija o subclase) a partir de otra existente (base, padre o superclase). Gracias a este mecanismo de herencia todos los miembros de la *superclase* son transferidos a la *subclase,* excepto los constructores.

```
class subclase extends superclase {

}
```

¡¡En Java únicamente se dispone de herencia simple!!

Esta es una *relación transitiva* mediante la cual, si una clase C hereda de la clase B, y la clase B hereda de la clase A, se puede decir que C también hereda de la clase A.

Por este motivo, todas las clases siempre heredarán de la superclase *Object* , a excepción de la clase *Object* que no hereda de ninguna otra clase.

1. Al contrario de lo que ocurre con los métodos y campos que son transferidos de una *superclase* a la *subclase*, los métodos constructores no se heredan. Por lo tanto, cada *subclase* tiene sus propios métodos constructores.

2. Aunque no se declare, cada clase tiene, de forma implícita, un constructor por defecto (sin parámetros ni código). Pero si se declara cualquier constructor, el constructor por defecto no se creará. Tal como se muestra en el siguiente ejemplo, es posible declarar tantos constructores parametrizados diferentes como sea necesario:

```
class Clase {
        Clase () {} // sobreescribe el constructor por defecto
        Clase (String parametro1) {}
        Clase (int parametro1, int parametro2) {}
        // … declaraciones de campos
}
class Superclase {}
class Subclase extends Superclase {}
```

```
public class Herencia {
        public static void main (String[] args) {
                Superclase sup = new Superclase();
                Subclase sub = new Subclase();
                Clase cla = new Clase();
        }
}
```

3 Cuando se crea un objeto de una clase, no se invoca únicamente a su constructor, sino que se ejecuta el de todas las *superclases*. Se recorrerán todas las clases de la jerarquía hasta llegar a la clase padre, formando una pila de llamadas y buscando el constructor correspondiente.

```java
class A {
    A() { System.out.println("En constructor A"); }
    A(String mensaje) { System.out.println(mensaje); }
}

class B extends A {
    B() { System.out.println("En constructor B"); }
    B(String mensaje) { System.out.println(mensaje); }
}

class C extends B{
    C() { System.out.println("En constructor C"); }
    C(String mensaje) { System.out.println(mensaje); }
}

public class Constructores {
    public static void main(String[]args) {
        C obj1 = new C();
        System.out.println("------");
        C obj2 = new C("Mensaje");
    }
```

```
En constructor A
En constructor B
En constructor C
------
En constructor A
En constructor B
Mensaje
```

POLIMORFISMO

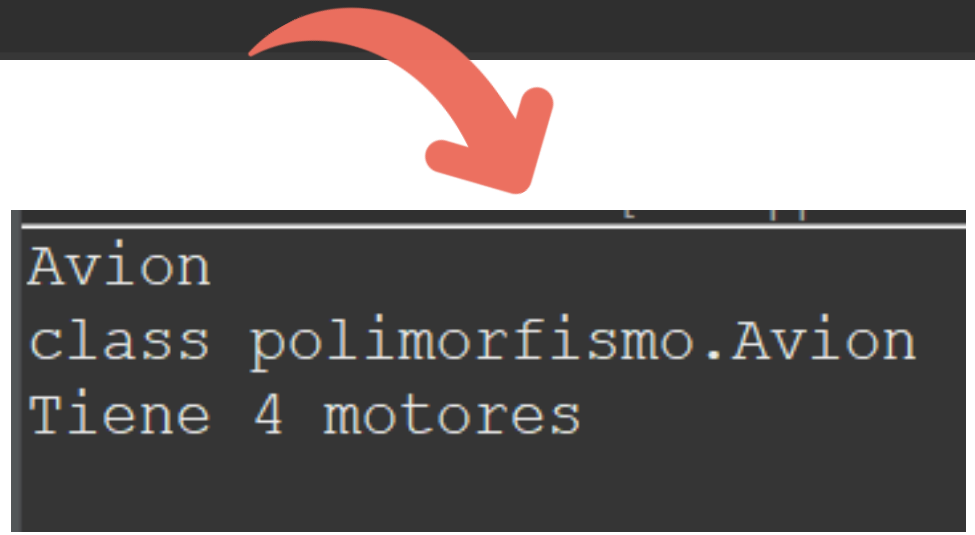
El **polimorfismo** es la propiedad de los objetos que les permite adoptar diferentes formas en tiempo de ejecución, así como que un objeto de una clase derivada sea utilizado como un objeto de la clase base.

El polimorfismo permite la invocación de métodos del mismo nombre sobre instancias de diferente clases.

```
class MedioTransporte {
    public void getInfo() {
        System.out.println("MedioTransporte");
    }
}

class Aereo extends MedioTransporte {
    public void getInfo() {
        System.out.println("Aereo");
    }
}
```

```
class Avion extends Aereo {
    private int numMotores;

    public Avion(int numMotores) {
        this.numMotores = numMotores;
    }
    public void getInfo() {
        System.out.println("Avion");
    }
    public void getNumMotores() {
        System.out.println("Tiene "+numMotores+" motores");
    }

}
```

```java
public class Polimorfismo {

    public static void main(String[] args) {
        MedioTransporte objMedioTransporte = new Avion (4);
        objMedioTransporte.getInfo();
        System.out.println(objMedioTransporte.getClass().toString());
        ((Avion) objMedioTransporte).getNumMotores();
    }

}
```



PARA PRACTICAR…

1

Codifica la jerarquía *ser vivo*, *animal* y *vegetal,* creando las estructuras que consideres necesario para almacenar cada elemento, como mínimo, nombre científico y nombre común. Para los animales almacenar peso y altura. Para los vegetales únicamente altura. Cada ser vivo se alimentará de una manera diferente, por lo que deberá reflejar este aspecto de alguna forma.

2

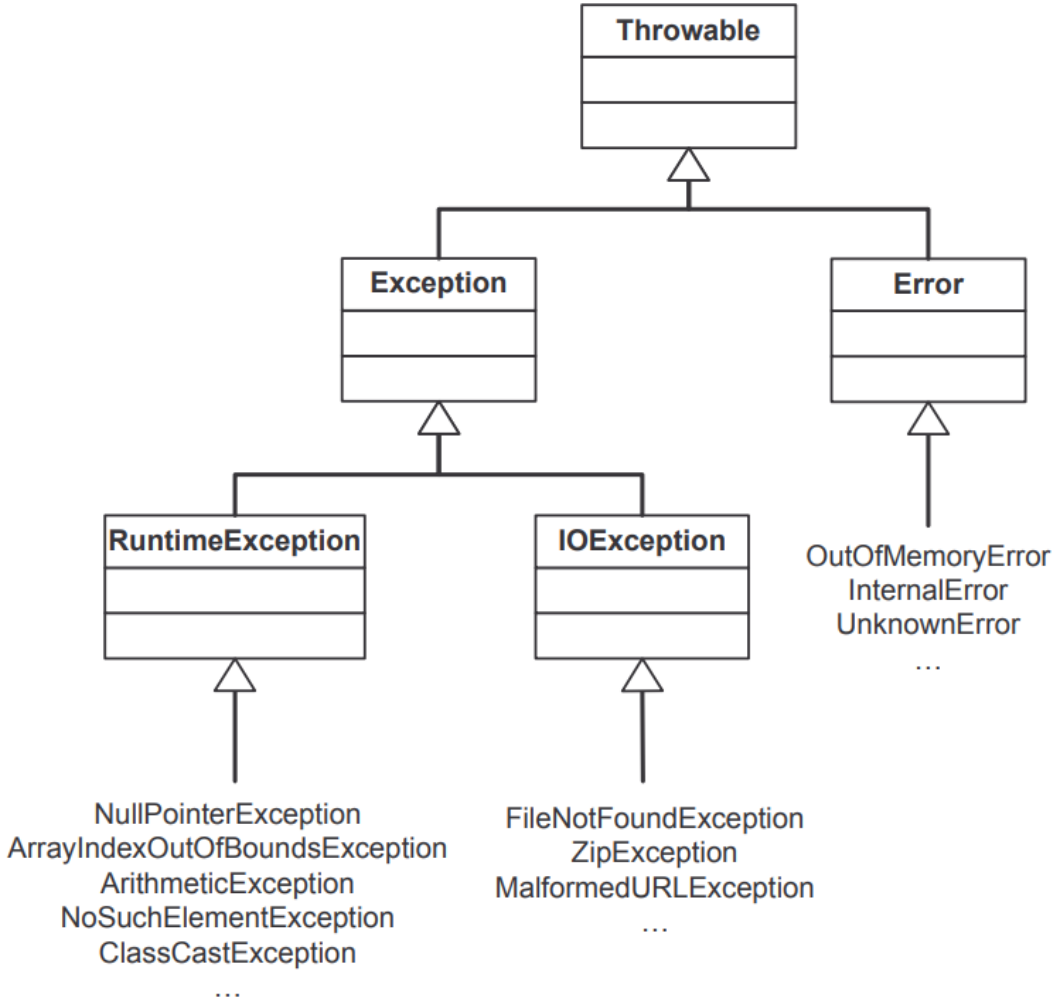
Codifica la jerarquía *empleado, senior y junior*, creando las estructuras que consideres necesario para almacenar cada elemento. El empleado deberá tener nombre, apellido y edad. El senior deberá tener responsabilidad. El junior deberá tener horas trabajadas. Cada empleado trabajará de una manera diferente, por lo que deberá reflejar este aspecto de alguna forma.

EXCEPCIONES

En Java, cuando se produce un error en un método “se lanza” un objeto Throwable.

Cualquier método puede “capturar la excepción” y tomar las medidas que estime oportunas.

Tras capturar la excepción, el control no vuelve al método en el que se produjo la excepción, sino que la ejecución del programa continúa en el punto donde se haya capturado la excepción.



THROWABLE

Clase base que representa todo lo que se puede “lanzar” en Java.

- Contiene una instantánea del estado de la pila en el momento en el que se creó el objeto.
- Almacena un mensaje que podemos utilizar para detallar qué error se produjo.

ERROR

Subclase de Throwable que indica problemas graves que una aplicación no debería intentar solucionar.

- Por ejemplo, memoria agotada, error interno de la JVM.

EXCEPTION

Exception y sus subclases indican situaciones que una aplicación debería tratar de forma razonable. Los dos tipos principales de excepciones son:

- *RuntimeException*: errores del programador (división por cero o acceso fuera de los límites del array)
- *IOException*: errores que no puede evitar el programador (relacionados con la entrada/salida del programa).

Todo el código que vaya dentro de la sentencia ***try*** será el código sobre el que se intentará capturar el error si se produce y, una vez capturado, hacer algo con él.

```
try {
    System.out.println("bloque de código donde pudiera saltar un error es este");
     }
```

En el bloque ***catch*** definimos el conjunto de instrucciones necesarias o de tratamiento del problema capturado con el bloque *try* anterior. Es decir, cuando se produce un error o excepción en el código que se encuentra dentro de un bloque *try*, pasamos a ejecutar el conjunto de sentencias que tengamos en el bloque *catch*.

```
catch (Exception e) {
    System.out.println("bloque de código donde se trata el problema");
}
```

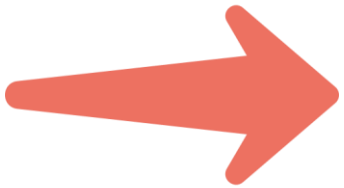
Fíjate que después del *catch* hemos puesto unos paréntesis donde pone "Exception e". Esto significa que cuando se produce un error, Java genera un objeto de tipo Exception con la información sobre el error y este objeto se envía al bloque *catch*.

EJEMPLO

```java
public static void main(String[] args) {
    int[] numeros = {1,2,3};
    System.out.println(numeros[10]);
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 3
        at Teoria.TeoriaExcepciones.main(TeoriaExcepciones.java:8)
```

```java
public static void main(String[] args) {
    try {
        int[] numeros = {1,2,3};
        System.out.println(numeros[10]);
    } catch (Exception e) {
        System.out.println("Fuera de limites...");
    }

}
```



```
Fuera de limites...
```

El bloque ***finally*** es un bloque donde podremos definir un conjunto de instrucciones necesarias tanto si se produce un error o excepción como si no y que por tanto se ejecuta siempre.

```java
public static void main(String[] args) {
    try {
        int[] numeros = { 1, 2, 3 };
        System.out.println(numeros[10]);
    } catch (Exception e) {
        System.out.println("Algo ha ido mal.");
    } finally {
        System.out.println("El 'try catch' ha terminado.");
    }
}
```

```
Algo ha ido mal.
El 'try catch' ha terminado.
```

La sentencia *finally* nos permite ejecutar código después del *try-catch* independientemente del resultado.

La declaración ***throw*** permite crear un error personalizado.
Esta instrucción se utiliza junto con un tipo de excepción. Hay muchos tipos de excepciones disponibles en Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityExcepction, etc.

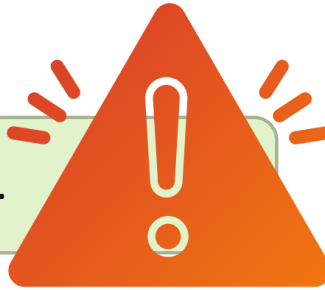
```
static void comprobarEdad(int edad) {
    if (edad < 18) {
        throw new ArithmeticException("Acceso denegado - Debes tener 18 años.");
    } else {
        System.out.println("Access confirmado - Eres mayor de edad.");
    }
}

public static void main(String[] args) {
    comprobarEdad(15);
}
```

```
Exception in thread "main" java.lang.ArithmeticException: Acceso denegado - Debes tener 18 años.
        at Teoria.TeoriaExcepciones.comprobarEdad(TeoriaExcepciones.java:7)
        at Teoria.TeoriaExcepciones.main(TeoriaExcepciones.java:14)
```

Si en el cuerpo de un método se lanza una excepción (de un tipo derivado de la clase Exception), en la cabecera del método hay que añadir una cláusula *throws* que incluye una lista de los tipos de excepciones que se pueden producir al invocar al método.

```
public String leerFichero (String nombreFichero)
                throws IOException
...
```

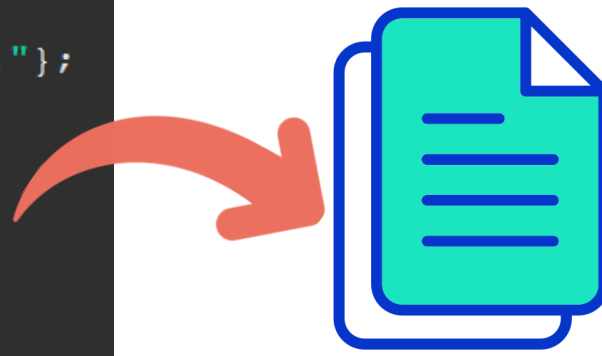
Las excepciones de tipo RuntimeException (muy comunes) no es necesario declararlas en la cláusula *throws*.



FICHEROS

ESCRITURA

```java
package ficheros;

import java.io.FileWriter;
import java.io.IOException;

public class EscrituraFicheros {

    public static void main(String[] args) throws IOException {

        String[] lineas = {"Uno","Dos","Tres","Cuatro","Cinco","Seis","Siete","..."};
        FileWriter fichero = null;
        try {
            fichero = new FileWriter("fichero_escritura.txt");
            // Escribimos linea a linea el fichero
            for (String linea : lineas) {
                fichero.write(linea + "\n");
            }
        } catch (Exception e) {
            System.out.println("Mensaje de la excepcion: " + e.getMessage());
        } finally {
            fichero.close();
        }

    }

}
```



```java
package ficheros;
import java.io.File;
import java.util.Scanner;
public class LecturaLineas {
    public static void main(String[] args) {
        // Fichero del que queremos leer
        File fichero = new File("fichero_leer.txt");
        Scanner s = null;
        try {
            // Leemos el contenido del fichero
            System.out.println("... Leemos el contenido del fichero ...");
            s = new Scanner(fichero);

            // Leemos linea a linea el fichero
            while (s.hasNextLine()) {
                String linea = s.nextLine();     // Guardamos la linea en un String
                System.out.println(linea);       // Imprimimos la linea
            }
        } catch (Exception ex) {
            System.out.println("Mensaje: " + ex.getMessage());
        } finally {
            try {
                if (s != null)
                    s.close();
            } catch (Exception ex2) {
                System.out.println("Mensaje 2: " + ex2.getMessage());
            }
        }
    }
}
```