TEMA 5: INTERFACES

Podría suceder que varias clases no relacionadas necesariamente compartan un mismo conjunto de operaciones. Para ello utilizamos las **interfaces**, ya que nos permiten especificar un conjunto de operaciones, y dependiendo de la clase, cada operación se realizará de una manera diferente. **Ejemplo**: las clases Parcela, Foto, Cuadro, EspejoCircular, … incluyen los métodos calcularArea, calcularPerimetro, etc. En este caso podríamos definir una **interfaz** que agrupe todos los métodos comunes que contenga tan solo las cabeceras de estos métodos y luego definir varias clases que implementen los métodos de la interfaz.

Una interfaz se declara como ***public interface*** *Nombre {...}*. Suele incluir un conjunto de cabeceras de métodos abstractos que deben ser implementados (todos) en la clase que implemente dicha interfaz. **IMPORTANTE**: una interfaz NO tiene atributos, aunque sí puede incluir definiciones constantes públicas (static final).

NOTA: una misma clase puede implementar más de una interfaz → *herencia múltiple de interfaces*

Ejemplo:

```
public interface Figura {   // Define una interfaz
   double area();
   double perimetro();
}

public class EspejoCircular implements Figura {  // Implementa una interfaz
   private double radio;
   ..............
   public double area() { return Math.PI*radio*radio; }
   public double perimetro() { return 2*Math.PI*radio; }
}

public class Foto implements Figura { // Implementa una interfaz
   private double lado1, lado2;
   ..............
   public double area() { return lado1*lado2; }
   public double perimetro() { return 2*(lado1+ lado2); }
}
```

Se pueden declarar referencias a objetos que implementen una cierta interfaz. Esto permite definir un método que sea aplicable a todos los objetos de clases que implementen una cierta interfaz.

```
double totalArea( Figura v[] ) {
  double t=0;                                   // Array de instancias de clases que
  for (int i=0; i<v.length; i++)                //  implementan la interfaz figura
  t += v[i].area(); // enlace dinámico
  return t;
}
```

Para poder ordenar un array es necesario saber cómo comparar 2 elementos para averiguar cuál es mayor o si son iguales. Java proporciona la interfaz **Comparable<T>** Es necesario que una clase A que quiera poder comparar objetos de la propia clase A implemente esa interfaz así: ***public class A implements Comparable<A> {…}*** Esto obliga a tener implementado el método ***public int compareTo(A parametro)*** que retorna < 0 si this es menor que parametro, > 0 si this es mayor que parametro e = 0 si son iguales.

NOTA: El método de Java que permite ordenar un array es: **Arrays.sort (Array de objetos)**

Ejercicio:
Dada una clase Persona y una clase FechaComparable que implementa la interfaz **Comparable<FechaComparable>**, modificar la clase Persona de forma que implemente la interfaz **Comparable<Persona>** (la comparación se hará teniendo en cuenta sólo la edad).

Escribir, además, un programa principal que ejecute:
→ Crear un array de 5 personas y ordenar el array (con **Arrays.sort**)

Diferencias básicas y más destacables entre:

Interfaces	Clases Abstractas
Una clase puede implementar múltiples interfaces	Una clase solo puede extender a otra clase (un único padre)
Las clases que lo implementan no tienen porqué estar relacionadas entre sí	La clase que extienda a la clase abstracta tiene una relación es-un con ella y es-hermano con otras subclases de la abstracta
Solo permite constantes públicas (static final)	Permite atributos de clase y de instancia
Todos los métodos son públicos	Sus métodos no tienen por qué ser públicos

TEMA 6: EXCEPCIONES

Las **excepciones** sirven para notificar una situación anómala, la cual se presenta durante la ejecución de un programa. Algunos ejemplos de situaciones anómalas podrían ser: acceso a una posición de un array que no existe, división por cero, agotamiento de la memoria, fichero no encontrado, problemas al construir una instancia de una clase, conexión por red no lograda, etc.

Tipos de excepciones:

1. **Específicas** de un programa: pueden ser de dos tipos (1) las definidas en la API de Java (fichero no encontrado, conexión de red no lograda, etc), o (2) las definidas por el propio programador, que se lanzan cuando se llama a un método con unos datos de entrada que no cumplen la PRE.

Ejemplo:
Crear una clase Vaso para representar objetos vasos y sus contenidos en mililitros:
Atributos: contenido (double) y capacidadMax (double)
Constructor: asigna el valor de la capacidad máxima del vaso
Métodos: llenar (double cantidad) para llenar el vaso de una cantidad de líquido, vaciar (double cantidad) para extraer del vaso una cantidad de líquido y getCantidad() que dice la cantidad de líquido que tiene un vaso.
→ llenar (double cantidad): PRE: contenido+cantidad <= capacidadMax
→ vaciar (double cantidad): PRE: cantidad <= contenido
Crear una clase de prueba que cree un objeto de tipo Vaso y lo llene y vacíe con diferentes cantidades.
¿Qué ocurre si intentamos llenar el vaso por encima de su capacidad máxima?
El valor del contenido no se ve afectado (no cambia), cosa que no es correcta ya que se vertería dicho contenido encima de la mesa (o donde lo estemos llenando).

Una posible "solución" podría ser comprobar que cabe la cantidad en el vaso y, en caso contrario, imprimir por la pantalla de error el mensaje "*No cabe cantidad*"

En realidad, debemos crear una excepción para ello. En Java las excepciones también son objetos; y se debe definir una clase para cada tipo de excepción que queramos generar como subclase (clase hija) de la clase Exception:

```
class MiExcepcion extends Exception {

   public MiExcepcion () {  }

   public MiExcepcion(String msg) {   // si se quiere mostrar un cierto mensaje se
       super(msg);                    // debe definir este segundo constructor
   }

}
```

Para generar una excepción debemos (1) crear la clase para crear objetos excepción y (2) generar o lanzar la excepción desde un método.

El método que genera la excepción tiene que (1) declarar que lanza una excepción añadiendo el siguiente código en su cabecera `throws MiExcepcion`, y (2) lanzar la excepción, creando el objeto excepción con `throw new MiExcepcion(mensaje)` *

* Donde mensaje es opcional, pero una buena práctica de programación. En él se indica la causa de la excepción.

En cuanto al tratamiento de las excepciones, tenemos tres opciones:

(1) No tratar las posibles excepciones que se lancen desde el main(), añadiendo en su propia cabecera un throws MiExcepcion()

(2) Tratar las posibles excepciones que se lancen desde el main() mediante un try-catch

```
try {                                   → resuelve un problema
      código que puede generar una excepción (o varias)
} catch (tipo_excepcion1 parámetro1) {
      "tratamiento de la excepción1"
      parámetro1.printStackTrace(); // por norma general
      parámetro1.getMessage();
} catch (tipo_excepcion2 parámetro2) {     →      trata      situaciones
especiales
      ..................................
} catch (tipo_excepcionN parámetroN) {
      "tratamiento de la excepciónN"
} [ finally {                           → se ejecuta en cualquier caso
      "bloque que se ejecuta siempre (cierre de recursos)"
} ]
```

Podemos encontrarnos con una (o varias) excepciones diferentes que queramos capturar en nuestro código. Cada excepción se capturará en un catch diferente*. La opción finally no es obligatoria y, en caso de estar definida, se ejecuta siempre (tanto si hay excepción como si no).

* **NOTA:** No es recomendable escribir ramas catch vacías que enmascaren excepciones durante la ejecución del programa.

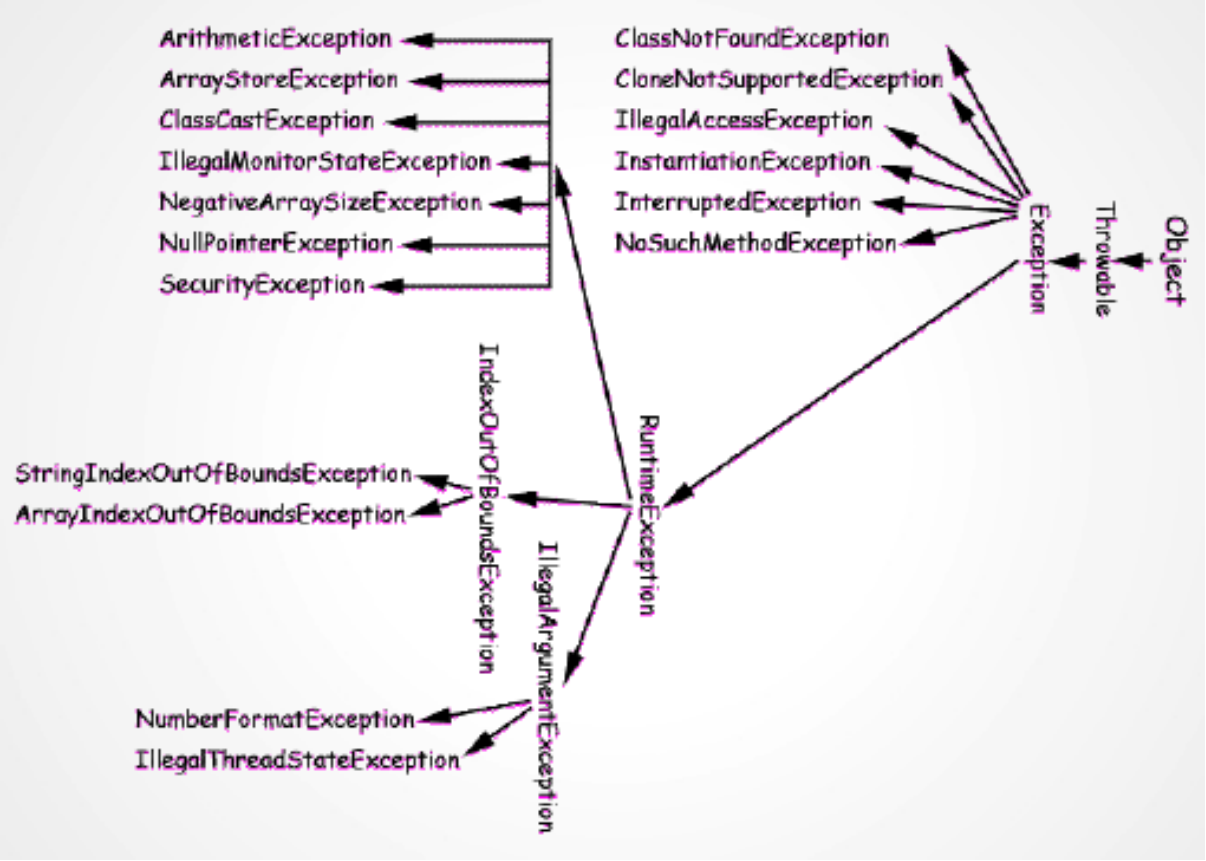
Esta segunda opción ofrece varias ventajas: resolver el problema que causó la excepción; informar al usuario del error y terminar el programa "limpiamente"; o permitir que la ejecución del programa pueda proseguir aunque no se haya resuelto el problema, entre otras.

Ejemplo continuación:
Crear en la clase Vaso el siguiente método:
prepararVasoDesayuno (double leche, double cacao, double azucar) que llama al método llenar de la clase Vaso con los valores de leche, cacao y azúcar que representan los valores de estos ingredientes respectivamente
En la clase de prueba crear un objeto vaso con una capacidad máxima de 200 ml. Llamar al método prepararVasoDesayuno con 250 ml de leche, 30 ml de cacao y 10 ml de azúcar
¿Qué pasa al hacer esto? → Salta la excepción y es imposible llenar el vaso con esas cantidades

(3) Tratar la excepción en el método y evitar de esta forma que se propague al nivel superior (main). Esta opción es la más idónea y la que vamos a seguir.

Ejemplo continuación:
Crear una clase con main que pida por teclado una cantidad positiva y llame al método llenar de la clase Vaso anterior. Si la cantidad introducida provoca que el método llenar lance la excepción, el main debe pedir otra cantidad al usuario. En caso contrario, el programa debe terminar.

2. **Generales** o **predefinidas**: pueden ocurrir en cualquier lugar del código. Las lanza la JVM. **Ejemplos**: acceso a una posición de un array que no existe, división por cero o agotamiento de memoria. Estas excepciones no hace falta declararlas ni tratarlas como hemos visto anteriormente con las específicas. Las más habituales son:



Ejercicio:
Dada una clase Cuenta con los siguientes métodos:
Atributos: cliente (String), saldo, gastosApertura (inicialmente los gastos de apertura son 10€)
Constructor (cliente, saldoInicial): resta al valor del saldo inicial gastos de apertura
void sacarDinero(cantidad)
void ingresarDinero(cantidad)
Crear una excepción SaldoInicialInsuficiente
lanzarla desde el constructor si el saldo inicial es menor que los gastos de apertura
Crear una excepción SaldoInsuficiente
lanzarla desde sacarDinero() si el saldo es menor que la cantidad que se desea retirar.