Tema 03 - Introducción a la Programación Orientada a Objetos

Objetivos del Tema:

* Características de la POO
* Comparar POO con Programación Estructurada
* Concepto de clase
* Diferenciar entre clase y objeto (instanciación)
* Empezar a construir clases
* Organizar las clases en paquetes
* Diseñar métodos (parámetros)

Ejemplo antes de empezar: solución ecuación de 2º grado mediante una clase

---

1.- Concepto de objeto y clase (símil entre molde y el flan): las clases son los moldes con los cuales se generan los objetos. A la acción de generar un objeto se le llama ***Instanciación***.

- Cuando se utiliza POO diseñamos las clases de objetos dotándolas de un comportamiento y cuando se ejecute la aplicación se generarán los objetos dotándolos de un estado.
- Utilizaremos un editor de ***texto plano*** con codificación de caracteres utf8 (*notepad++, gedit, eclipse, netbeans, soportan texto plano y codificación utf8*), el nombre de cada clase debe comenzar por la primera letra en mayúscula y debe guardarse en un fichero con el mismo nombre y extensión ***.java***.
- Esqueleto de una clase, ejemplo:

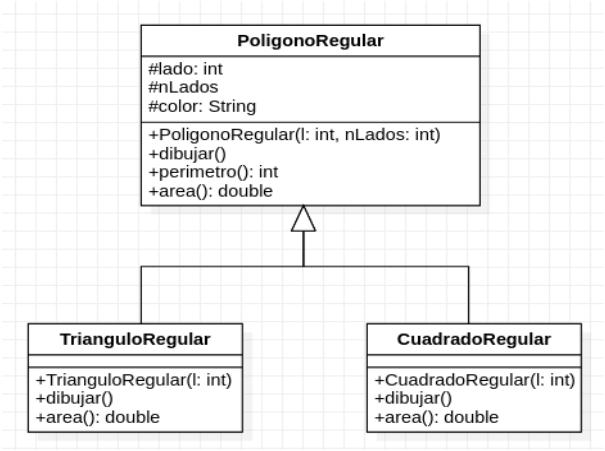
```
[public] class Pajaro [extends Animal] {
        [String nombre;
        String color;
        int edad;
        boolean domesticado;]
        ...
        [[public] void setDomesticado(boolean dom) {
        domesticado=dom;
        }]
        .....
}
```

- Características de un objeto:
  - ***Identidad*** cada objeto tiene un ID único (está almacenado en una zona única de memoria)
  - ***Estado*** (valor de los atributos)
  - ***Comportamiento***, se define en la clase y está determinado por los métodos que definimos en la misma.
- **Mensajes**. Los objetos interactúan unos con otros mediante el intercambio de mensajes (llamadas a métodos)
- **Métodos**. Los métodos definen el comportamiento de los objetos de una clase.
- Los **objetos** tienen dos características principales: ***Estado*** y ***Comportamiento***. El Estado viene determinado por el valor de los atributos y el comportamiento se define en la clase mediante los métodos.
- La programación orientada a objetos proporciona los siguientes beneficios:
  - **Modularidad**. El código fuente de una clase puede modificarse sin afectar al resto de la aplicación, siempre que se mantenga la *interfaz*.
  - **Reutilización**. Una vez diseñada e implementada una clase se puede utilizar en muchas aplicaciones distintas. No necesitamos conocer los detalles de implementación, sólo la interfaz.

- **Facilidad de depuración**. Si tenemos una clase de objetos que no está funcionando correctamente, podemos subsanar los errores de la misma sin que esto afecte al resto de la aplicación.
- **Ocultación de los detalles de implementación**. Se ocultan los detalles de implementación

2.- Características de POO

- **Abstracción**. Abstraer las propiedades que nos interesan del modelo real
- **Encapsulamiento**. Se ocultan los detalles de implementación de las clases y sólo se permite el acceso a los objetos de las mismas a través de su interfaz
- **Herencia**. Cuando una clase hereda de otra clase "adquiere" todas las características de la clase de la que hereda (atributos y métodos). (En java una clase sólo puede tener una superclase)
- **Polimorfismo**. Las clases que heredan puedan cambiar el comportamiento por defecto de la clase que heredan (ejemplo: PoligonoRegular, TrianguloRegular, CuadradoRegular)



3.- Propiedades y métodos de los objetos

- En una clase se engloban atributos y métodos. Los atributos pueden ser tanto tipos de datos básicos (char, int, double, ...) como objetos de otras clases.

4.- Entrada y salida de datos a través de la consola.

- La clase ***java.lang.System***, contiene tres objetos *static* que se corresponden con los "flujos estándar" del sistema y que son, respectivamente:
  - ***System.in***. Representa al teclado (/dev/stdin)
  - ***System.out***. Representa a la salida estándar (/dev/stdout)
  - ***System.err***. Representa a la salida estándar de errores (/dev/stderr)
- Para leer de la entrada utilizaremos la clase más moderna y sencilla de utilizar ***java.util.Scanner***, ejemplo:

java.util.Scanner teclado=new java.util.Scanner(System.in); //lee de la entrada estándar (teclado)
int n=teclado.nextInt(); // leer del teclado lo que haya tecleado el usuario e intentar convertirlo a int (puede producir una excepción)

- Para imprimir en la salida estándar utilizaremos ***System.out.println("...")*** o sin ln para no imprimir salto de línea
- Para imprimir en la salida estándar de errores utilizaremos: ***System.err.println("...")***

5.- Parámetros y valores devueltos

Los métodos pueden recibir y devolver (*return)* valores, por ejemplo el método ***area*** de la clase ***PoligonoRegular*** tiene el siguiente prototipo:
public ***double*** area(), quiere decir que no recibe ningún parámetro y que devuelve un double.

Si un método no recibe ningún parámetro los paréntesis tras su nombre no llevarán nada en medio.
Si un método no devuelve nada habrá que declararlo explícitamente diciendo que no devuelve nada mediante la palabra clave ***void***.

6.- Constructores y destructores de objetos

- El ***Constructor*** es un método especial que sirve para ***instanciar*** (construir) un objeto de la clase correspondiente e inicializar dicho objeto. Su nombre coincide con el nombre de la clase y no se indica que devuelva nada porque va implícito que se devuelve un objeto de la clase. Pueden existir tantos constructores como queramos, siempre que difieran en el número o en el tipo de los parámetros. Si no se especifica ningún constructor Java crea uno vacío de forma automática. Si se crea algún constructor, ya no existirá el predeterminado vacío. A la repetición de métodos con distinto prototipo se le llama ***sobrecarga***. Ejemplo:

```
    public TrianguloRegular(int l) {
        super(l, 3);
    }
```

- El ***Destructor***. En Java no existe método destructor ya que los objetos se destruyen de forma automática por parte del "*recolector de basura*". En otros lenguajes de programación, como C++, si existe el método destructor, que se encarga de liberar todos los recursos utilizados por el objeto correspondiente.

7.- Atributos Estáticos

Cuando un atributo o método de una clase se declara como ***static***, sólo existirá un ejemplar del mismo para todos los objetos que se creen de la clase.

```
public class Car {
    private String name;
    private String engine;

    private static int numberOfCars;
    public static void setNumberOfCars(int n) {
       numberOfCars=n;
    }
    public static int getNumberOfCars() {
       return numberOfCars;
    }
    public Car(String name, String engine) {
        this.name = name;
        this.engine = engine;
        numberOfCars++;
    }

    // getters and setters
}
```

* Razones por las que usar variables ***static***
    - Cuando el valor de la variable es independiente de los objetos de la clase
    - Cuando el valor de la variable va a ser compartido por todos los objetos de la clase

* Ideas clave para recordar sobre variables ***static***

- Al pertenecer a la clase, pueden ser accedidas directamente usando el nombre de la clase y no necesitan ninguna referencia a objeto (p.ej. Car.numberOfCars)
- Sólo pueden ser declaradas a nivel de clase (no se pueden declarar en un método)
- Puede ser accedidas sin que se haya creado ningún objeto de la clase
- Aunque pueden ser accedidas usando una referencia a objeto de la clase (p.ej. Car.numberOfCars++), deberíamos hacer uso de las mismas usando el nombre de la clase para evitar confusiones con las variables miembro de la clase no estáticas.

8.- Métodos Estáticos

De manera parecida a las variables ***static***, los métodos ***static*** también pertenecen a la clase en lugar de a los objetos de la clase. Están diseñados para hacer uso de ellos sin crear objetos de la clase donde se enmarcan.

- Ejemplo de método ***static***

Los métodos ***static*** son usados generalmente para realizar operaciones que no dependen de la creación de una instancia de la clase.

```
public static void setNumberOfCars(int numberOfCars) {
    Car.numberOfCars = numberOfCars;
}
```

Los métodos ***static*** son también muy usados para crear clases de *utilidades* (p.ej. la clase java.lang.Math), de forma que podemos usar sus métodos sin instanciar ningún objeto de la clase. P.ej. Math.sqrt(2);

- Razones por las que usar métodos ***static***
  - Para acceder y manipular variables ***static*** y otros métodos que no dependan de la creación de objetos de la clase
  - Crear clases de utilidades, como por ejemplo java.lang.Math, con métodos disponibles en cualquier otra clase de la aplicación sin tener que instanciar objetos.

* Ideas clave para recordar sobre métodos ***static***
  - Los métodos ***static*** en Java se resuelven en tiempo de compilación, mientras que el mecanismo de sobreescritura de métodos (polimorfismo) se resuelve en tiempo de ejecución, por ello los métodos ***static*** no pueden reescribirse.
  - Los métodos ***abstract*** no pueden ser ***static***
  - Los métodos ***static*** no pueden usar las palabras reservadas ***this*** ni ***super*** (no tiene sentido)
  - Las siguientes combinaciones de métodos de instancia (necesito tener un objeto creado), métodos de clase ((static) no necesitamos tener instancia del objeto) y variables son válidas:
    * Los métodos de instancia pueden directamente acceder a métodos y variables de instancia
    * Los métodos de instancia pueden también acceder a variables ***static*** y métodos ***static***
    * Los métodos ***static*** pueden acceder a variables y métodos ***static***
    * Los métodos ***static*** no pueden acceder a variables y métodos de instancia directamente; tienen que crear un objeto de la clase para ello.

https://www.baeldung.com/java-static

9.- Paquetes

Un ***Paquete (Package)*** en Java es un contenedor de clases que permite agrupar las distintas partes de un programa y que por lo general tiene una funcionalidad y elementos comunes, definiendo la ubicación de dichas clases en un directorio de estructura jerárquica.

El uso de paquetes proporciona las siguientes ventajas:

- Agrupamiento de clases con características comunes.
- Reutilización de código al promover principios de programación orientada a objetos como la encapsulación y modularidad.
- Mayor seguridad al existir niveles de acceso.
- Evita la colisión de clases que tengan el mismo nombre. Pueden existir clases con el mismo nombre siempre y cuando su *fully qualified class name* sean únicos.

Las clases en el código fuente deben incluir el nombre del paquete donde se sitúan: package persona.clientes;
./persona/clientes/ClientesPremium.java ./persona/clientes/ClientesVip.java
import persona.clientes.*;
ClientesPremium cp=new ClientesPremium();
ClientesVip cv=new ClientesVip();

*** Ideas clave sobre paquetes**

- Si no se define un paquete para un fichero de código Java se definirá un paquete llamado "default" automáticamente.
- Cada fichero java está incluido en un paquete (*default* si no se utiliza la sentencia *package)*
- Los nombres de paquete que comienzan con el nombre java.* y javax.* son reservados.
- El nombre del paquete equivale a una estructura de ficheros. Por ejemplo, el nombre de paquete com.empresa.utilidades debería ser igual al directorio ./com/empresa/utilidades. Si situamos una clase en un paquete (carpeta) y no incluimos en dicha clase la sentencia *package*, la clase no podrá utilizarse.

** Los siguientes son **convenciones** o **estándares** acordados en la definición de paquetes en Java:*

- El nombre del paquete se define de manera inversa al dominio de la organización o grupo. Por ejemplo, dominioempresa.com puede ser usado como nombre de paquete así: com.dominioempresa.utilidades.
- El nombre del paquete debería definirse en minúscula. Si existen varias palabras en el nombre, se pueden separar con guion bajo (_).

*** Uso de paquetes**

- En los ficheros de código Java se usa la palabra reservada *package* para especificar a qué paquete pertenecen. Suele indicarse como primera sentencia:

package java.awt.event;

- Para usar un paquete dentro del código se usa la declaración *import*. Si sólo se indica el nombre del paquete y añadimos un *, se importarán todas las clases que contiene:

import java.awt.event.;*

- Si además del nombre del paquete se especifica una clase, sólo se importa esa clase:

import java.awt.event.ActionEvent;

- Después de añadir la sentencia anterior, se puede hacer referencia a la clase *ActionEvent* usando su nombre:

ActionEvent myEvent = new ActionEvent();

- Si no se hubiera importado la clase o el paquete, cada vez que tuviéramos que usarla habría que especificarla por su ***fully qualified class name***, que no es más que el nombre del paquete seguido por el nombre de la clase:

java.awt.event.ActionEvent myEvent = new java.awt.event.ActionEvent();

- Si lo que se desea es importar todos los miembros estáticos de una clase, note la sentencia static después de import. (A partir de J2SE 5.0 en adelante)

import static java.awt.Color.;*

*** Librerías estándar de java**

Estas son la librerías estándar más importantes de la API de Java:

Paquete	Descripción
java.applet	*Contiene clases para la creación de applets.*
java.awt	*Contiene clases para crear interfaces de usuario con ventanas.*
java.io	*Contiene clases para manejar la entrada/salida.*
java.lang	*Contiene clases variadas pero imprescindibles para el lenguaje, como Object, Thread, Math...* *El paquete java.lang es importado por defecto en el ficheros de código Java.*
java.net	*Contiene clases para soportar aplicaciones que acceden a redes TCP/IP.*
java.util	*Contiene clases que permiten el acceso a recursos del sistema, etc.*
javax.swing	*Contiene clases para crear interfaces de usuario mejorando la AWT.*
java.sql	*Contiene utilidades para acceso a bases de datos*
java.security	*Relacionada con la seguridad*

10.- Localización de librerías

La localización de las clases de las que queremos hacer uso dependen del paquete donde se encuentren. Los paquetes se organizan en rutas relativas a partir de una carpeta raíz. Se pueden especificar más de una carpeta raíz haciendo uso de la variables del sistema ***CLASSPATH*** o en la línea de comandos haciendo uso del parámetro ***-cp*** o ***-classpath*** o ***--class-path***, en ambos casos se puede especificar una lista de directorios y de ficheros jar.

Imaginemos que tenemos una aplicación cuya clase principal es *Main.java* y que se encuentra en una carpeta de proyecto de Netbeans en */home/usuario/NetbeansProjects/proyecto/paquete1/Main.java y* que nos encontramos en nuestra carpeta personal, */home/usuario*, para ejecutar dicha clase podríamos hacerlo así:
java -cp /home/usuario/NetbeansProjects/proyecto paquete1.Main
o podríamos establecer la variables *CLASSPATH* y ejecutar posteriormente dicha clase:
export *CLASSPATH=/home/usuario/NetbeansProjects/proyecto && java paquete1.Main*
El contenido de la variables *CLASSPATH*, puede ser una lista de carpetas y ficheros *.jar* separados por *punto y coma (;)* en Windows o por *dos puntos (:)* en Unix.

BOLETÍN DE EJERCICIOS DEL TEMA

1. ¿Cuáles serían los atributos de la clase Ventana (de ordenador)? ¿cuáles serían los métodos? Piensa en las propiedades y en el comportamiento de una ventana de cualquier programa.

2. Realiza una clase *Finanzas* que convierta dólares a euros y viceversa. Codifica los métodos *dolaresToEuros* y *eurosToDolares*. Prueba que dicha clase funciona correctamente haciendo conversiones entre euros y dólares. La clase tiene que tener:

- Un constructor F*inanzas*() por defecto el cual establecerá el cambio Dólar-Euro en 1.36.
- Un constructor F*inanzas*(*double*), el cual permitirá configurar el cambio Dólar-Euro.

3. Realiza una clase *MiNumero* que proporcione el doble, triple y cuádruple de un número proporcionado en su constructor (realiza un método para doble, otro para triple y otro para cuádruple). Haz que la clase tenga un método main y comprueba los distintos métodos.

4. Realiza una clase número que almacene un número entero y tenga las siguientes características:

- Constructor por defecto que inicializa a 0 el número interno.
- Constructor que inicializa el número interno.
- Método *añade* que permite sumarle un número al valor interno.
- Método *resta* que resta un número al valor interno.
- Método *getValor*. Devuelve el valor interno.
- Método *getDoble*. Devuelve el doble del valor interno.
- Método *getTriple*. Devuelve el triple del valor interno.
- Método *setNumero*. Inicializa de nuevo el valor interno.

5. Modifica la clase *Satelite*, que se da a continuación y añádele los siguientes métodos:

- Método *void variaAltura(double desplazamiento)*. Este método acepta un parámetro que será positivo o negativo dependiendo de si el satélite tiene que alejarse o acercarse a La Tierra.
- Método *boolean enOrbita()*. Este método devolverá false si el satélite está en tierra y true en caso contrario.
- Método *void variaPosicion(double variap, double variam)*. Este método permite modificar los atributos de posición (paralelo y meridiano) mediante los parámetros variap y variam. Estos parámetros serán valores positivos o negativos relativos que harán al satélite modificar su posición.

```
public class Satelite {
	private double meridiano;
	private double paralelo ;
	private double distancia_tierra ;
	Satelite (double m, double p , double d) {
		meridiano=m;
		paralelo=p ;
		distancia_tierra=d ;
	}
	Satelite () {
		meridiano=paralelo =distancia_tierra=0;
	}
	public void setPosicion(double m,double p,double d) {
		meridiano=m;
		paralelo= p;
		distancia_tierra =d ;
	}
	public void printPosicion () {
		System.out.println("El satélite se encuentra en el paralelo "+
paralelo+" Meridiano "
				+meridiano+ " a una distancia de la tierra de
"+distancia_tierra+" Kilómetros");
```

```
        }
}
```

6. Vamos a crear una clase llamada **`Persona`**. Sus atributos son: **nombre**, **edad** y **DNI**. Construye los siguientes métodos para la clase:

- Un constructor, donde los datos pueden estar vacíos.
- Los setters y getters para cada uno de los atributos. Hay que validar las entradas de datos.
- `mostrar()`: Muestra los datos de la persona.
- `esMayorDeEdad()`: Devuelve un valor lógico indicando si es mayor de edad.

7. Crea una clase llamada `Cuenta` que tendrá los siguientes atributos: titular (que es una persona) y cantidad (puede tener decimales). El titular será obligatorio y la cantidad es opcional. Construye los siguientes métodos para la clase:

- Un constructor.
- Los setters y getters para cada uno de los atributos. El atributo cantidad no se puede modificar directamente, sólo ingresando o retirando dinero.
- `mostrar()`: Muestra los datos de la cuenta.
- `ingresar(cantidad)`: se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
- `retirar(cantidad)`: se retira una cantidad a la cuenta. La cuenta puede estar en números rojos.
- `estaEnNumerosRojos(): indicará si la cuenta está en números rojos o no`

8. Vamos a definir ahora una “Cuenta Joven”, para ello vamos a crear una nueva clase `CuantaJoven` que deriva de la anterior. Cuando se crea esta nueva clase, además del titular y la cantidad se debe guardar una bonificación que estará expresada en tanto por ciento. Construye los siguientes métodos para la clase:

- Un constructor.
- Los setters y getters para el nuevo atributo.
- En esta ocasión los titulares de este tipo de cuenta tienen que ser mayor de edad, por lo tanto hay que crear un método `esTitularValido()` que devuelve verdadero si el titular es mayor de edad pero menor de 25 años y falso en caso contrario.
- Además la retirada de dinero sólo se podrá hacer si el titular es válido.
- El método `mostrar()` debe devolver el mensaje de “Cuenta Joven” y la bonificación de la cuenta.

Piensa los métodos heredados de la superclase que hay que reescribir.

9. Crea la clase *Peso*, la cual tendrá las siguientes características:
    - Deberá tener un atributo donde se almacene el peso de un objeto en kilogramos.
    - En el constructor se le pasará el peso y la medida en la que se ha tomado ('Lb' para libras, 'Li' para lingotes, 'Oz' para onzas, 'P' para peniques, 'K' para kilos, 'G' para gramos y 'Q' para quintales).
    - Deberá de tener los siguientes métodos:
        - ***getLibras***. Devuelve el peso en libras.
        - ***getLingotes***. Devuelve el peso en lingotes.

- ***getPeso***. Devuelve el peso en la medida que se pase como parámetro ('Lb' para libras, 'Li' para lingotes, 'Oz' para onzas, 'P' para peniques, 'K' para kilos, 'G' para gramos y 'Q' para quintales).

- Para la realización del ejercicio toma como referencia los siguientes datos:
  1 Libra = 16 onzas = 0,453 Kgramos.
  1 Lingote= 32,17 libras= 14,59 kg.
  1 Onza = 0,0625 libras = 0,02835 Kgramos.
  1 Penique = 0,05 onzas = 0,00155 Kgramos.
  1 Quintal = 100 libras = 43,3 kg.
- Crea además un método main para testear y verificar los métodos de esta clase.

10. Crea una clase con un método *millasAMetrosQ* que toma como parámetro de entrada un valor en millas marinas y las convierte a metros. Una vez tengas este método escribe otro *millasAKilometrosQ* que realice la misma conversión, pero esta vez exprese el resultado en kilómetros.
Nota: 1 milla marina equivale a 1852 metros.

11. Crea la clase *Coche* con dos constructores. Uno no toma parámetros y el otro sí. Los dos constructores inicializarán los atributos *marca* y *modelo* de la clase. Crea dos objetos (cada objeto llama a un constructor distinto) y verifica que todo funciona correctamente.

12. Implementa una clase *Consumo,* la cual forma parte de la centralita electrónica de un coche y tiene las siguientes características:
Atributos:
kms - Kilómetros recorridos por el coche.
litros - Litros de combustible consumido.
vmed - Velocidad media.
pgas - Precio de la gasolina.
Métodos:
- *getTiempo* - indicará el tiempo empleado en realizar el viaje.
- *consumoMedio* - Consumo medio del vehículo (en litros cada 100 kilómetros).
- *consumoEuros* - Consumo medio del vehículo (en euros cada 100 kilómetros).

No olvides crear un constructor para la clase que establezca el valor de los atributos. Elige el tipo de datos más apropiado para cada atributo.

13. Para la clase anterior implementa los siguientes métodos, los cuales podrán modificar los valores de los atributos de la clase:
- *setKms*
- *setLitros*
- *setVmed*
- *setPgas*

14. El restaurante mejicano de Israel cuya especialidad son las papas con chocos nos pide diseñar un método con el que se pueda saber cuántos clientes pueden atender con la materia prima que tienen en el almacén. El método recibe la cantidad de papas y chocos en kilos y devuelve el número de clientes que puede atender el restaurante teniendo en cuenta que por cada tres personas, Israel utiliza un kilo de papas y medio de chocos.

15. Modifica el programa anterior creando una clase que permita almacenar los kilos de papas y chocos del restaurante. Implementa los siguientes métodos:

- *public void addChocos(int x)*. Añade x kilos de chocos a los ya existentes.
- *public void addPapas(int x)*. Añade x kilos de papas a los ya existentes.
- *public int getComensales()*. Devuelve el número de clientes que puede atender el restaurante (este es el método anterior).
- *public void showChocos()*. Muestra por pantalla los kilos de chocos que hay en el almacén.
- *public void showPapas()*. Muestra por pantalla los kilos de papas que hay en el almacén.

16. Crea la clase Pizza con los atributos y métodos necesarios. Sobre cada pizza se necesita saber el ***tamanio*** - *mediana* o *familiar* - el ***tipo*** - *margarita*, *cuatro quesos* o *funghi* - y su ***estado*** - *pedida* o *servida*. La clase debe almacenar información sobre el número total de pizzas que se han pedido y que se han servido. Siempre que se crea una pizza nueva, su estado es "*pedida*". El siguiente código del programa principal debe dar la salida que se muestra:

```
public class PedidosPizza {
	public static void main(String[] args) {
		Pizza p1 = new Pizza("margarita", "mediana");
		Pizza p2 = new Pizza("funghi", "familiar");
		p2.sirve();
		Pizza p3 = new Pizza("cuatro quesos", "mediana");
		System.out.println(p1);
		System.out.println(p2);
		System.out.println(p3);
		p2.sirve();
		System.out.println("pedidas: " + Pizza.getTotalPedidas());
		System.out.println("servidas: " + Pizza.getTotalServidas());
	}
}
```

Salida:
pizza margarita mediana, pedida
pizza funghi familiar, servida
pizza cuatro quesos mediana, pedida
esa pizza ya se ha servido
pedidas: 3
servidas: 1

17. Realizad una clase Complejo que implemente los métodos suma, resta, producto y división de números complejos, y un método numComplejos que nos devuelva el número de objetos Complejo creados en nuestra aplicación.

18. Realizad una clase VectorTridimensional que implemente la suma, resta, producto escalar, producto vectorial y módulo del vector, y un método numVectores que nos devuelva el número de objetos VectorTridimensional creados en nuestra aplicación.