Universidad Nacional de Educación a Distancia
Departamento de Lenguajes y Sistemas Informáticos

FUNDAMENTOS DE INFORMÁTICA

Práctica Obligatoria
Curso 2024/2025

Índice general

Introducción

La práctica se realizará utilizando *Java* como lenguaje de Programación Orientado a Objetos. El primer concepto que se debe conocer es el de *objeto*, el ladrillo con el que se construye un programa bajo este paradigma de programación. Los objetos tienen vida cuando nuestro programa está en ejecución y pueden relacionarse entre sí de diversas maneras, proporcionando al programador las funcionalidades necesarias para resolver una amplia gama de problemas.

La definición de un objeto viene dada por lo que se conoce como *clase*, siendo ésta la entidad que en realidad diseñaremos y programaremos en términos de sus *campos* o *atributos* y *métodos*. Estos campos y métodos nos permiten definir las características y funcionalidades de la clase. Una vez definida, a partir de la clase podremos crear objetos, que son los que realmente llevarán a cabo la funcionalidad de nuestro programa en tiempo de ejecución.

Por tanto, podemos ver los objetos como los componentes que nos servirán para diseñar nuestro programa, y las clases como las definiciones de dichos objetos. Una misma clase puede, por tanto, dar lugar a muchos objetos diferentes cada uno con características diferentes que vendrán dadas por los valores de sus campos.

1.1. Descripción general de la práctica

Esta práctica consistirá en crear un sistema informático simplificado para gestionar una empresa de reparto de leña a domicilio. Para ello, el programa realizará tres tipos de funcionalidades: la gestión de clientes, la gestión del almacén con el stock de leña que tiene y, por último, la gestión de los pedidos que realizan los clientes. Este proyecto busca familiarizar al estudiante con la creación de aplicaciones informáticas, llevando a la práctica los principios de programación en Java.

En general, el proceso que se quiere informatizar es el siguiente:

1. Después de cada llamada de un cliente, el recepcionista introduce en el sistema los detalles del pedido, como la cantidad de leña solicitada, si se requieren una o varias bolsas de astillas, además del nombre y la dirección del cliente. Entre estos datos, se incluye el pueblo al que pertenece el cliente. El sistema calcula automáticamente el precio del pedido teniendo en cuenta la cantidad de leña, el coste del transporte, que varía según el pueblo, y si se han solicitado bolsas de astillas. Es importante destacar que el almacén únicamente suministra leña; las astillas son un complemento adicional, aunque también tienen un coste asociado. Además, es fundamental evitar la duplicación de datos, de manera que no sea necesario introducir nuevamente la información de clientes que ya han realizado compras previamente.

2. Durante el proceso de instalación y en la explotación del sistema, será posible añadir o actualizar la lista de pueblos atendidos, incluyendo sus nombres y el coste de transporte asociado. Asimismo, se podrá modificar tanto el precio por kilo de leña como el de cada bolsa de astillas.

3. Además, será necesario mantener un control preciso de la cantidad de leña disponible en el almacén. Cada vez que se realice un pedido, se deberá restar del stock el peso correspondiente. De igual manera, cuando se reciba un nuevo suministro de leña por parte del proveedor, habrá que actualizar el stock disponible en el sistema.

Este ejercicio de programación se centrará únicamente en el *back-end*, es decir, en la parte del desarrollo encargada de la lógica necesaria para que el sistema funcione, omitiendo la interacción real con una interfaz de usuario y las comunicaciones de red; todo ello será obviado en esta práctica. De este modo, el objetivo es completar la programación de la lógica que subyace al sistema planteado, poniendo a prueba la capacidad de los estudiantes para estructurar y organizar el código de una manera que facilite el crecimiento iterativo y la adición de nuevas funcionalidades en futuras versiones.

Para la realización de esta práctica se seguirá un enfoque iterativo, de forma que el desarrollo se dividirá en etapas. En cada etapa se partirá del diseño y desarrollo de la etapa inmediatamente anterior, añadiéndole nuevas funcionalidades y llevando a

la práctica los conceptos del temario que se van estudiando progresivamente.

Además, el enunciado de la práctica se acompaña de distintos fragmentos de código, que servirán para validar cada una de las etapas del enunciado, lo que permitirá comprobar si la solución que se está dando en cada momento es correcta o no. Ese código de comprobación estará encapsulado en distintas clases, una por cada etapa, planteando un conjunto de pruebas que permitirán validar la solución propuesta hasta ese momento. Las pruebas consistirán en llamadas a métodos de los objetos creados en cada etapa, por lo que **será muy importante respetar y seguir los identificadores que se proponen para los distintos elementos; deberán respetarse el nombre de las clases, atributos, métodos, tipos de datos, etc.**

RECOMENDACIÓN: La práctica se realiza a lo largo del cuatrimestre y forma parte del estudio de la asignatura. Es conveniente hacer una primera lectura completa de este documento para tener una visión global de lo que se pide, pudiéndose organizar así apropiadamente el estudio de la asignatura.

1.2. Diseño de la práctica: Diagrama de clases

Cuando afrontamos el desarrollo de una aplicación informática es indispensable y necesario realizar una fase previa donde se analicen los requisitos y se diseñen las entidades; en nuestro caso, las clases necesarias para llevar a cabo las funcionalidades requeridas y la relación entre las mismas. Para ello, antes de implementar cada etapa en las que se ha dividido la práctica, se deberá **realizar una lectura completa de la etapa a abordar**, realizando un diagrama donde se representen las clases necesarias con sus atributos y métodos, así como las relaciones entre las mismas, para posteriormente comenzar la fase de implementación. Así, se deberá realizar un diagrama de clases desde la primera etapa, que luego posteriormente se irá refinando y/o modificando en cada una de las siguientes etapas, siempre como requisito previo a comenzar a implementar el código de cada etapa.

El diagrama de clases obtenido en la última etapa debe ser entregado junto al código de la práctica.

Etapa 1

2.1. ETAPA 1.1 Orientación a Objetos: La clase Pueblo

2.1.1. Objetivos

En esta primera etapa de la práctica se van a afianzar los conceptos básicos de la programación orientada a objetos: **clase**, **objeto**, **atributos** y **métodos** (constructores, métodos que ayudan a la encapsulación de la clase y métodos de visualización).

Veamos un ejemplo que nos permita iniciar esta primera etapa de la práctica. Como ya hemos mencionado, este año modelaremos una empresa de reparto de leña. Esta empresa organiza sus repartos en función del conjunto de pueblos a los que ofrece servicio, y este conjunto debe ser actualizable.

En este apartado, analizaremos cómo modelar la entidad básica *pueblo*. Siguiendo el principio de modularización y abstracción descrito en el libro base, el problema debe descomponerse en subproblemas, abordando cada uno de forma independiente.

Para la gestión de pueblos, definiremos la clase *Pueblo*, que encapsulará el coste de transporte asociado a cada uno. Esto permitirá mantener esta información aislada, facilitando la gestión de clientes y pedidos y garantizando la coherencia del sistema.

Cada pueblo registrado en el sistema se modelará como un objeto de la clase Pueblo, con características comunes (atributos o campos). Sin embargo, los valores de estos atributos podrán ser específicos para cada pueblo, permitiendo representar y gestionar de manera eficiente cada localidad dentro del sistema.

2.1.2. Requisitos previos

Para la realización de esta primera etapa se requiere haber estudiado los temas 4, 5 y 6 del temario detallado de la asignatura, correspondientes a los capítulos 1, 2 y 3, así como los apéndices A, B, C, D y F del libro de referencia de la Unidad Didáctica II.

2.1.3. Implementación de la clase Pueblo

Uno de los elementos principales del sistema es la noción de pueblo, que será modelada como una clase a partir de la cual se podrán crear instancias u objetos de la misma. En esta fase del desarrollo de la práctica, cada pueblo se describe por la siguiente información, que modelaremos como atributos de la clase `Pueblo`:

- `nombre`, que representa el nombre del pueblo. El valor de este atributo debe introducirse obligatoriamente en el constructor para que se cree el objeto.
- `precioTransporte`, representando el coste de transporte de cualquier pedido a dicho pueblo. El coste por defecto del transporte en cualquier pueblo es de 10 euros, por lo que será este el valor que se asigne en el constructor.

Además, la clase `Pueblo` debe permitir consultar y modificar estos campos a través de una serie de métodos creados a tal efecto: por cada atributo debe haber dos métodos, uno para consultar su valor y otro para modificarlo. Por ejemplo, para el campo `nombre` tendremos los métodos `getNombre()` y `setNombre()` para consultar su valor y modificarlo, respectivamente.

Finalmente, la clase `Pueblo` dispondrá de un método que imprima en pantalla toda su información, incluyendo nombre y precio del transporte. En la figura 2.1 se muestra un ejemplo de formato que debería tener la salida en pantalla de este método.

RECOMENDACIÓN: jugar con BlueJ creando objetos de la clase Pueblo, modificando el valor de sus atributos, mostrándolos por pantalla, etc.

```
Población: <nombre del pueblo>
Precio del transporte: <precio> euros
```

Figura 2.1: Ejemplo de formato de salida por pantalla mostrando la información de un pueblo

2.2. ETAPA 1.2 Ejecución de Aplicaciones: La clase Principal

2.2.1. Objetivos

Es importante comprender que *Java* y *BlueJ* son cosas diferentes. Java es un lenguaje de programación, mientras que BlueJ es un entorno de programación que nos permite programar en Java. BlueJ está pensado para el aprendizaje de Java y proporciona distintas herramientas que permiten inspeccionar las partes de un programa, tanto en la parte de diseño (cuando estamos escribiendo el código) como en la de ejecución (cuando el código se ejecuta, se crean los objetos en memoria, etc.)

Una de las herramientas de BlueJ que pueden resultar más útiles es el banco de objetos u object bench (ver capítulo 1 del libro), que junto con otras herramientas como el inspector de objetos, resulta muy interesante para "jugar" con los objetos en tiempo de ejecución de forma interactiva. Podemos ver un ejemplo de estas herramientas en la Figura 2.2, donde se muestran varios objetos de una clase Pelicula en el banco de objetos;

Una de las herramientas de BlueJ que pueden resultar más útiles es el banco de objetos u *object bench* (capítulo 1 del libro), junto con otras herramientas como el *inspector de objetos*, resulta muy interesante para "jugar" con los objetos en tiempo de ejecución de forma interactiva. Podemos ver un ejemplo de los mismos en la figura 2.2, en el ejemplo, uno de ellos está siendo inspeccionado, lo que nos permite visualizar los valores de sus atributos. La clase `Película` no está incluida en la práctica; es solo un ejemplo para mostrar el funcionamiento de BlueJ.

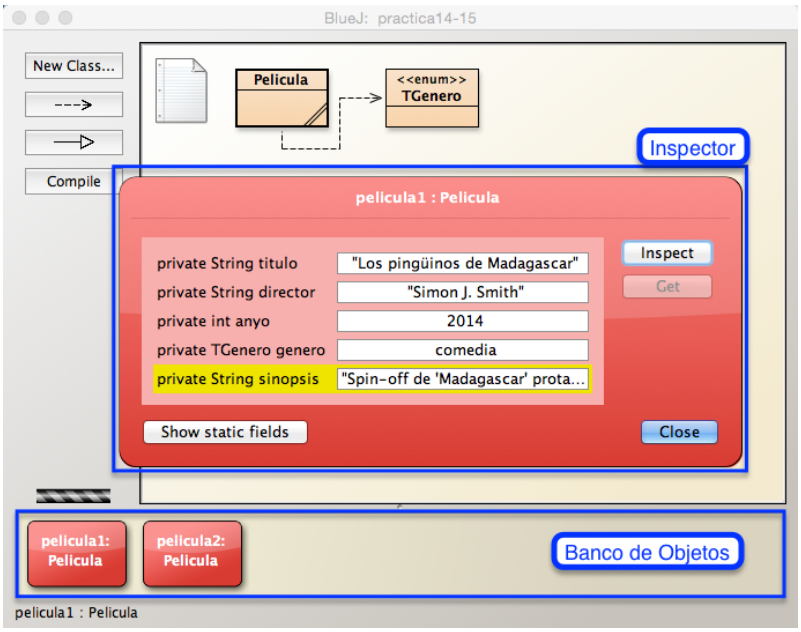

Figura 2.2: BlueJ con varias películas creadas en el banco de objetos. La clase `Película` no está incluida en la práctica. Es solo un ejemplo.

BlueJ nos permite ejecutar nuestras aplicaciones mediante la invocación de los métodos de sus objetos, pero no es esta la única manera de hacerlo. Java, más allá de BlueJ, nos permite ejecutar los programas que creamos de forma independiente. A estas alturas sabemos que cada clase tiene unos métodos y campos propios de los objetos creados a partir de dicha clase, que para poderlos usar necesitamos crear primero un objeto. Por ejemplo, la clase `Pueblo` que hemos creado tiene un método para consultar su nombre, que sólo podría ser invocado una vez tengamos creado algún objeto de esta clase.

RECOMENDACIÓN: Al final de este apartado se debería entender la diferencia entre ejecutar un programa en BlueJ y hacerlo de forma independiente, esto es, por ejemplo, ejecutarlo desde la línea de comandos (símbolo de sistema en Windows/Linux/MacOS).

2.2.2. Requisitos previos

Además de los expuestos en la sección 2.2.1, deben consultarse capítulos 6.15 y 6.16 del libro de texto, sobre métodos de clase. Además, en el apéndice E puede encontrar más información sobre el método `main()`, en concreto en el apartado E1. Puede consultar también el capítulo 3.10 "Objetos que crean objetos" si necesita más información acerca de la creación de objetos desde el código.

2.2.3. Enunciado: Creación de instancias

Para ejecutar nuestro programa de manera independiente de BlueJ y crear el flujo principal del sistema, crearemos una clase llamada `EjemploEtapa1`, donde implementaremos el método `main()`. Esta clase contendrá el código en el que se crean las instancias de objetos necesarias y las llamadas para poder comprobar el funcionamiento del sistema.

El ejercicio a realizar en este punto consiste en crear dicha clase `EjemploEtapa1`, que contenga un método `main()` e implementar en este método el siguiente código[1]:

```
public class EjemploEtapa1 {
      public static void main (String [ ] args) {
            Pueblo p=new Pueblo(''Villanueva'');
            p.imprimeInformacion();
            p.setPrecioTransporte(20);
            p.imprimeInformacion();
      }
}
```

Este código debería crear una instancia de la clase `Pueblo`. Posteriormente, el código imprime por pantalla la información siguiente:

```
Población: Villanueva
precio del transporte: 10.0 euros
Población: Villanueva
precio del transporte: 20.0 euros
```

Como ejercicio, debe ejecutarse esta clase `EjemploEtapa1`, comprobando que compila y ejecuta correctamente, visualizándose los datos de los usuarios. A continuación, modifique el método `main()` para crear nuevos objetos de la clase `Pueblo`, asignando los valores que desee para sus atributos y visualícelos de manera análoga. Ejecute el programa desde BlueJ y desde la línea de comandos (consola o símbolo de sistema, según el sistema operativo que utilice).

Por último, es importante entender que estamos implementando clases y métodos que servirán de base para la implementación de un sistema con interfaz de usuario. Un requisito de la práctica implementada es que permita ejecutar el programa `InterfazEtapa1` que se proporciona junto a este enunciado. Al ejecutar este programa, se pide por linea de commandos el nombre de un pueblo y su coste de transporte asociado. El programa crea el objeto correspondiente e imprime la información. Este programa realiza estas operaciones mediante llamadas a constructores y métodos de la práctica implementada. Es importante que la práctica siga las especificaciones y cubra los ejemplos de instanciación y llamadas a métodos.

NOTA IMPORTANTE: La presente práctica debe poderse ejecutar independientemente de BlueJ de cara a su entrega y corrección.

[1] En el curso virtual, junto con el enunciado de la práctica puede descargar el código fuente de esta clase EjemploEtapa1

ETAPA 2

3.1. ETAPA 2.1: Uso de la composición. Las clases Cliente y Pedido

3.1.1. Objetivos

Una manera muy usual de relacionar los objetos en la programación orientada a objetos es mediante la **composición**. Esta relación se da cuando un atributo o campo de una clase toma como valor un objeto de otra clase. Esta relación permite la modularización del problema. En nuestro caso, por ejemplo, un cliente reside en un pueblo. Hemos encapsulado el concepto de pueblo en la clase correspondiente. Asignar como atributo una instancia de la clase pueblo a un cliente permite encapsular la noción de *pueblo* sin necesidad de duplicar código cuando modelamos la noción de *cliente*. De la misma forma, un pedido tiene como atributo un cliente. Mediante esta relación de composición podemos encapsular también la noción de cliente como atributo de la clase `Pedido`. De esta forma, trabajamos las nociones de *abstracción* y *modularización* descritas en el tema 6 de la asignatura (Interacción entre objetos) y más concretamente en la sección 3.2 del libro base.

Además, en esta etapa deberán ponerse en práctica los principios de encapsulación de la programación orientada a objetos tal y como se describe en el libro base de la asignatura. Esto implica que los campos de las clases deben estar definidos como privados, siendo los métodos de la propia clase los únicos que tienen acceso a dichos campos.

Finalmente, es una buena práctica de programación documentar todo el código que se va realizando, no sólo con el objetivo de que terceras personas puedan comprender nuestro programa, sino también para que nosotros mismos podamos entender nuestro código de cara a futuras extensiones, mejoras o modificaciones. Para la generación de documentación, Java nos ofrece la herramienta **Javadoc**, que por medio de determinado tipo de comentarios añadidos en nuestro código fuente, nos permite la generación posterior de toda la documentación de las distintas clases.

3.1.2. Requisitos previos

Para la realización de esta etapa se deben estudiar los temas 7, 8 y 9 del temario detallado de la asignatura. Encontrará información más detallada sobre el mecanismo de composición y la generación de documentación en el libro de texto de la asignatura en lo capítulos 4, 5 y 6. La lectura de las secciones de 8.3 a 8.6 (ambas inclusive) sobre buenas prácticas en el diseño de aplicaciones es también recomendable.

3.1.3. Implementación de la clase Cliente

En el tema 6 de la asignatura (requisito previo para la realización de esta parte de la práctica) se describen los conceptos de tipos primitivos y tipos de objeto. Es decir, a parte del tipo `int` o `boolean`, las clases que definamos, como por ejemplo la clase `Pueblo` que hemos implementado puede constituir un tipo (ver secciones 3.5 a 2.8 del libro base). Es decir, el atributo `pueblo` de la clase `Cliente` que estamos diseñando será un tipo de objeto de la clase `Pueblo` y su valor será un objeto de dicha clase.

Cada cliente se describe por la siguiente información, que modelaremos como atributos de la clase `Cliente`:

- `nombre`, que representa el nombre del usuario. El valor de este atributo deberá introducirse obligatoriamente en el constructor para que se cree el objeto.
- `dirección`, que representa la calle y número donde reside el cliente. El valor de este atributo no se inicializará por defecto en el constructor sino que se introducirá mediante un método diseñado para ello.

- `pueblo`, representando el pueblo al que pertenece el cliente. El valor de este atributo deberá introducirse obligatoriamente en el constructor para que se cree el objeto.

Igual que en la clase anterior, la clase `Cliente` debe permitir consultar y modificar todos estos campos a través de una serie de métodos creados a tal efecto: por cada atributo debe haber dos métodos, uno para consultar su valor y otro para modificarlo. Por ejemplo, para el campo `nombre` tendremos los métodos `getNombre()` y `setNombre()` para consultar su valor y modificarlo, respectivamente. Los métodos `getPueblo()` y `setPueblo()` recibirán o devolverán un objeto de la clase `Pueblo`.

La clase `Cliente` dispondrá también de un método que imprima en pantalla toda su información, es decir, nombre, dirección y el pueblo al que pertenece. Hay que tener en cuenta que para imprimir el nombre del pueblo al que pertenece el cliente es necesario invocar al método `getNombre` del objeto `Pueblo` que da valor al campo del cliente. En la figura 3.1 presentamos un ejemplo de formato que debería tener la salida en pantalla de este método. Es importante tener en cuenta en la implementación de este método la modularización. Esto quiere decir que ya disponemos en la clase `Pueblo` de un método encargado de imprimir la información relativa al pueblo (nombre y precio del transporte). Por tanto, dentro del método `imprimirInformacion` de la clase `Cliente`, tras imprimir el nombre del cliente y su dirección, se hará una llamada al método `imprimirInformacion` del objeto `Pueblo` correspondiente para que imprima el nombre de la población y el precio del transporte.

```
Cliente: <nombre del cliente>
    Dirección: <dirección>
    Población: <nombre del pueblo>
    Precio del transporte: <precio> euros
```

Figura 3.1: Ejemplo de formato de salida por pantalla mostrando la información de un cliente

RECOMENDACIÓN: jugar con BlueJ creando objetos de la clase Cliente, modificando el valor de sus atributos, mostrándolos por pantalla, etc.

El ejercicio a realizar en este punto consiste en crear una clase `EjemploEtapa21`, que contenga un método `main()` e implementar en este método el siguiente código[1]:

```
public class EjemploEtapa21 {
      public static void main (String [ ] args) {
            Pueblo p=new Pueblo(``Villanueva'');
            p.setPrecioTransporte(20.0);
            Cliente c=new Cliente(``Juan López'',p);
            c.setDireccion(``C/ Quijote 3'');
            c.imprimeInformación();
      }
}
```

Este código debería crear una instancia de la clase `Pueblo` y otra de la clase cliente. Posteriormente, el código imprime por pantalla la información siguiente:

```
Cliente: Juan López
     Dirección: C/ Quijote 3
     Población: Villanueva
     Precio del transporte: 20.0 euros
```

3.1.4. Implementación de la clase Pedido: Comportamientos mas sofisticados

En este punto vamos a modelar el concepto fundamental de nuestro problema, es decir, los pedidos. Teniendo en cuenta las especificaciones del enunciado, cada pedido se describe por la siguiente información, que modelaremos como atributos de la clase `Pedido`:

[1] En el curso virtual, junto con el enunciado de la práctica puede descargar el código fuente de esta clase EjemploEtapa21

- `cliente`, que representa al cliente. Toma como valor un objeto de la clase `Cliente`. El cliente debe de ser un parámetro obligatorio en el constructor para que se cree el objeto.
- `fecha`, en la que se ha realizado el pedido. Toma como valor un objeto de la clase `Date`. El valor por defecto dado en el constructor será la fecha actual.
- `cantidad`, representando la cantidad de madera solicitada. El valor inicial por defecto será cero.
- `bolsasAstillas`, representando la cantidad de bolsas de astillas solicitada. El valor inicial por defecto será cero.
- `servido`, representando si el pedido ya ha sido servido, o no, al cliente. El valor inicial por defecto será `false`.

De nuevo, estamos aplicando composición al definir los campos de cliente, y fecha como instancias de las clases `Cliente` y `Date` respectivamente. En cambio, los campos `cantidad`, `bolsasAstillas` y `servido` pueden modelarse como tipos predefinidos, en concreto, enteros y booleanos.

Igual que en las clases anteriores, la clase `Pedido` debe permitir consultar y modificar todos estos campos a través de una serie de métodos creados a tal efecto. Los métodos `getCantidad()`, `setCantidad()`, `getBolsasAstillas()` y`setBolsasAstillas()`, recibirán o devolverán un valor tipo `int`.

Conviene realizar algunas observaciones en relación al atributo fecha. Java dispone de una clase `Date` que permite el manejo de fechas. En el constructor de la clase `Pedido`, que se ejecuta cada vez que se instancia un objeto de dicha clase (ver sección 2.4 del libro base), debería crearse un objeto date (`Date date = new Date();`) que automáticamente toma el valor de la fecha actual. Para imprimir el valor de esta fecha basta con dárselo como parámetro a la instrucción de escritura en pantalla (`System.out.println(``Fecha: '' + date);`) Para poder hacer uso de esta clase es necesario importar la librería correspondiente añadiendo al comienzo del código la instrucción `import java.util.Date;` Los métodos `getFecha()`, `setFecha()`, recibirán o devolverán un valor tipo `Date`.

Además, la clase dispondrá de un método `calcularPrecio` que compute el precio del pedido en función de la cantidad de leña, el coste de transporte al pueblo de destino en donde reside el cliente y el número de bolsas de astillas solicitada. Este método devolverá dicho precio como un valor tipo `float`.

Sin embargo, en realidad el método `calcularPrecio` aun no dispone de toda la información necesaria, dado que también tiene que tener en cuenta el precio por kilo de leña y por bolsa de astillas. Para ello, tendremos que estudiar el concepto de *variable de clase* expuesto en la sección 6.14 del libro base. Mientras que campos como `Cliente` o `Cantidad` adquieren valores distintos para cada instancia de `Pedido`, el precio por kilo de leña y bolsa de astillas es variable (puede modificarse según las especificaciones de nuestro problema), pero el valor es común a todos los pedidos. Esto se modela mediante un tipo de atributo que se especifica mediante la palabra reservada `static`. Cuando un atributo queda definido como `static` cualquier modificación de éste afecta a todas las instancias de la clase, es decir, a todos los pedidos. Por ejemplo, el atributo `precioPorKilo` debería definirse en la clase `Pedido` como:

```
private static double precioPorKilo;
```

A su vez, el método que modifica esta variable debe tambien quedar definido como `static`. Por ejemplo, mediante la cabecera:

```
public static void setPrecioPorKilo(double p) {...
```

Es decir, si un método necesita modificar o consultar un campo estático, ese método también debe ser declarado como estático. Esto se debe a que los métodos estáticos pertenecen a la clase y no a una instancia específica de la clase, al igual que los campos estáticos.

En definitiva, en esta etapa tendremos que definir dos variables de clase `precioPorKilo` y `precioPorBolsa` con sus métodos get y set correspondientes. Cuando se invoque al método set correspondiente se deberá además indicar en pantalla la modificación realizada. Además tendremos que implementar un método `CalcularPrecio` que a partir de toda la información necesaria encapsulada en el objeto calcule el precio del pedido.

Finalmente, la clase `Pedido` dispondrá también de un método que imprima en pantalla toda su información, incluyendo nombre, dirección y el pueblo al que pertenece. Para respetar el principio de modularización, este método deberá invocar al método `imprimeInformacion` del objeto de tipo `Cliente` correspondiente. Además, imprimirá también el precio del pedido haciendo uso del método correspondiente. La figura 3.2 muestra un ejemplo de formato que debería tener la salida en pantalla de este método.

```
INFORMACIÓN DE PEDIDO
Fecha: <fecha del pedido>
Cantidad: <Cantidad>
Número de bolsas de astillas: <Número de bolsas de astillas>
Cliente: <nombre del cliente>
     Dirección: <dirección>
     Población: <nombre del pueblo>
     Precio del transporte: <precio> euros
Precio del pedido: <precio total>
Estado del pedido: <si ha sido servido o no>
```

Figura 3.2: Ejemplo de formato de salida por pantalla mostrando la información de un pedido

RECOMENDACIÓN: jugar con BlueJ creando objetos de la clase Pedido, modificando el valor de sus atributos, mostrándolos por pantalla, etc.

El ejercicio a realizar en este punto consiste en ampliar la clase `EjemploEtapa22`, ya desarrollada en la sección anterior, que contenga un método `main()` e implementar en este método el siguiente código[2]:

```
    public class EjemploEtapa22 {
       public static void main (String [ ] args) {
            Pueblo p=new Pueblo(''Villanueva'');
            p.setPrecioTransporte(20);
            Cliente c=new Cliente(''Juan López'',p);
            c.setDireccion(''C/ Quijote 3'');
            Pedido.setPrecioPorBolsa(7);
            Pedido.setPrecioPorKilo(0.75);
            Pedido pe=new Pedido(c);
            pe.setCantidad(100);
            pe.setBolsasAstillas(2);
            pe.imprimeInformacion();
       }
}
```

Como puede verse en el código anterior, los métodos `setPrecioPorBolsa` y `setPrecioPorKilo` se invocan antes de haber instanciado ningún objeto de la clase pedido. Esto se debe a que es un método asociado a la clase y no a ninguna instancia en particular. Este es el motivo por el que la sintaxis se conforma desde el nombre `Pedido` de la propia clase (`Pedido.setPrecioPorBolsa(7)`).

Este código debería crear una instancia de la clase `Pueblo` y otra de la clase `Cliente`. Posteriormente, el código imprime por pantalla la información siguiente:

```
PRECIO POR KILO DE LEÑA MODIFICADO A 0.75 EUROS

PRECIO POR BOLSA DE ASTILLAS MODIFICADO A 7 EUROS

INFORMACIÓN DE PEDIDO
Fecha: Dec 18 10:30:45 GMT 2024
Cantidad: 100 kilos
Número de bolsas de astillas: 2
Cliente: Juan López
     Dirección: C/ Quijote 3
     Población: Villanueva
     Precio del transporte: 20.0 euros
Precio del pedido: 109.0 euros
Estado del pedido: no servido
```

[2]En el curso virtual, junto con el enunciado de la práctica puede descargar el código fuente de esta clase EjemploEtapa22

Además, una vez implementadas las clases `Pueblo`, `Cliente` y `Pedido` ya puede ejecutarse el programa `InterfazEtapa22`. Este programa crea una instancia de la clase `Pueblo`, de la clase `Cliente` y de la clase `Pedido` preguntando el nombre del pueblo, el coste de transporte, el nombre del cliente, el precio por kilo de leña, el precio por bolsa de astillas, la cantidad de leña solicitada, la cantidad de bolsas de astillas, y muestra la información usando el método `imprimeInformacion()` de la clase Pedido.

3.1.5. Creación de la clase `AlmacenLenya`: Incorporando estructuras de datos

RECOMENDACIÓN: Lea primero los requisitos completos antes de tomar decisiones acerca de cómo desarrollar la solución.

Hasta ahora hemos modelado relaciones de composición simples. Por ejemplo, la clase `Cliente` tiene como atributo un objeto de la clase `Pueblo` y a su vez, la clase `Pedido` tiene como atributo un objeto de la clase `Cliente`. Sin embargo, en algunos casos es necesario modelar relaciones de composición múltiple. Por ejemplo, nuestra empresa de leñas se caracteriza por tener asociada una lista de pedidos, de clientes y de pueblos. Para modelar estas relaciones, necesitamos estructuras de datos como listas o arrays, que se describen en el capítulo 4 del libro base. En lugar de tener un objeto como atributo, la clase tendrá una lista de objetos de cierto tipo como atributo. Estas listas pueden implementarse mediante la clase *ArrayList*. En la sección 4.4 del libro se describe el manejo de esta clase para la creación y manipulación de listas.

Nuestra aplicación va a gestionar un conjunto de pueblos, clientes y pedidos, que hemos modelado como diferentes clases. Los objetos de estas clases se gestionarán por medio de una clase `AlmacenLenya`. En esta etapa, los procesos de la empresa de leña que queremos modelar mediante esta clase son los siguientes:

1. **Incluir un pueblo, un cliente o un pedido en el sistema.** Los métodos encargados de añadir cada uno de estos elementos al sistema recibirán un objeto de la clase correspondiente. Además, el sistema debe restar la cantidad del pedido a la cantidad disponible en el almacén. Se asume que siempre hay astillas por lo que no vamos a llevar un control de la cantidad de bolsas de astillas en el almacén.

2. **Añadir una cantidad de leña al almacén.** El método encargado de esta funcionalidad recibirá como entrada la cantidad de leña que se va a recargar en el almacén. El sistema debe imprimir en pantalla un mensaje indicando que se ha añadido al almacén la cantidad de leña correspondiente e indicar también la cantidad actual.

3. **Anotar un pedido como servido.** Dado el nombre del cliente, si existe un pedido no servido a su nombre, lo anota como servido e imprime en pantalla la información del pedido indicando que ya ha sido servido. Se asume que una vez que se atiende a un cliente y se ejecuta esta llamada, quedan servidos todos sus pedidos. Si no hay ningún pedido por servir a nombre de dicho cliente lo indica en pantalla.

Además, como en las clases anteriores, ésta deberá incluir un método get y set para cada atributo. Los métodos `getPueblos`, `setPueblos`, `getClientes`, `setClientes`, `getPedidos`, `setPedidos` recibirán o devolverán un objeto `ArrayList`.

En esta etapa se deben documentar todas las clases creadas hasta ahora. En el capítulo 6, concretamente en la sección 6.11 se describen los pasos a seguir para documentar las clases correctamente.

Por último, vamos a crear ahora una nueva clase llamada `EjemploEtapa23`, que contenga un método `main()` e implementar en este método el siguiente código[3]:

```
public class EjemploEtapa23 {
      public static void main (String [ ] args) {
            Pueblo p=new Pueblo("Villanueva");
            p.setPrecioTransporte(20);
            Cliente c1=new Cliente("Juan Lopez",p);
            c1.setDireccion(Ç/ Quijote 3");
            Cliente c2=new Cliente("Luis Ramirez",p);
            c2.setDireccion(Ç/ Dulcinea 9");
            Pedido.setPrecioPorBolsa(7);
            Pedido.setPrecioPorKilo(0.75);
            Pedido pe1=new Pedido(c1);
            pe1.setCantidad(100);
            pe1.setBolsasAstillas(2);
```

[3]En el curso virtual, junto con el enunciado de la práctica se pueden descargar el código fuente de esta clase EjemploEtapa23

```
            AlmacenLenya g=new AlmacenLenya();
            g.anyadirPueblo(p);
            g.anyadirCliente(c1);
            g.anyadirCliente(c2);
            g.anyadirLenya(1000);
            g.anyadirPedido(pe1);
            g.anotarServido("Juan Lopez");
            g.anotarServido("Luis Ramirez");
      }
}
```

Al ejecutar el main con este código, el sistema deberá imprimir en pantalla lo siguiente:

```
Precio por bolsa de astillas modificado a: 7.0 euros
Precio por kilo modificado a: 0.75 euros

PUEBLO AÑADIDO
      Población: Villanueva
      Precio del transporte: 20.0 euros

CLIENTE AÑADIDO
Cliente: Juan Lopez
      Dirección: C/ Quijote 3
      Población: Villanueva
      Precio del transporte: 20.0 euros

CLIENTE AÑADIDO
Cliente: Luis Ramirez
      Dirección: C/ Dulcinea 9
      Población: Villanueva
      Precio del transporte: 20.0 euros

SE HA AÑADIDO 1000 KILOS DE LEÑA AL ALMACÉN
CANTIDAD ACTUAL: 1000 KILOS

PEDIDO AÑADIDO
INFORMACIÓN DE PEDIDO
Fecha: Wed Feb 05 14:35:58 GMT 2025
Cantidad: 100
Número de bolsas de astillas: 2
Cliente: Juan Lopez
      Dirección: C/ Quijote 3
      Población: Villanueva
      Precio del transporte: 20.0 euros
Estado del pedido: No servido
Precio total del pedido: 109.0 euros

PEDIDO SERVIDO:
INFORMACIÓN DE PEDIDO
Fecha: Wed Feb 05 14:35:58 GMT 2025
Cantidad: 100
Número de bolsas de astillas: 2
Cliente: Juan Lopez
      Dirección: C/ Quijote 3
      Población: Villanueva
      Precio del transporte: 20.0 euros
Estado del pedido: Servido
Precio total del pedido: 109.0 euros
```

```
NO EXISTE NINGÚN PEDIDO POR SERVIR A NOMBRE DE: Luis Ramirez
```

Es importante probar el sistema ante otras situaciones, como diferentes datos del cliente, nuevos pueblos, etc.

En este punto, ya debe de poder ejecutarse el programa `InterfazFase23`. Este programa debería de poder ejecutarse sobre las clases, pero es necesario que la definición de métodos y atributos se ajuste a los ejemplos anteriores. Este programa instancia un objeto de la clase AlmacenLenya y ofrece las siguientes opciones:

- Actualizar el precio de la leña introduciendo un valor y llamando al método estático setPrecioPorKilo de la clase pedido
- Actualizar el precio de la bolsa de astillas introduciendo un valor.
- Añadir una cierta cantidad de leña al almacén.
- Crear una instancia de la clase Pueblo preguntando por el coste del transporte y añadir el pueblo a la lista de pueblos.
- Actualizar el coste de transporte de un pueblo, eligiendo el pueblo por menú.
- Crear una instancia de la clase Cliente, introduciendo su nombre y dirección y eligiendo por menú el pueblo al que pertenece. Añadir el cliente al sistema.
- Crear una instancia de la clase Pedido, introduciendo por menú el cliente de entre los clientes que hay creados y preguntando la cantidad de leña y de bolsas de astillas. Además, añadir el pedido a la lista de pedidos.
- Anotar como servido un pedido, mostrando los pedidos pendientes de ser servidos y eligiendo uno de ellos por menu.

ETAPA 3

4.1. ETAPA 3.1: Uso de la herencia y polimorfismo

4.1.1. Objetivos

En esta etapa vamos a analizar un aspecto fundamental de la programación orientada a objetos. A lo largo del ciclo de vida de un sistema informático pueden surgir nuevos requisitos, es decir, la necesidad de cubrir nuevas funcionalidades. Antes de que se popularizara este paradigma de programación, era necesario replantear el sistema y revisar el código ya implementado. Sin embargo, la programación orientada a objetos ofrece mecanismos de *herencia*, de forma que, a partir del código ya existente, pueden definirse nuevas clases que *heredan* las propiedades de las clases ya existentes pero incorporando nuevos atributos o métodos.

En esta parte de la práctica vamos a simular la situación en la que se quiere extender el sistema para que se contemplen pedidos personalizados, con un coste adicional, en los que se especifica la proporción de leña de tamaño pequeño, mediano y grande. Para ello, sin necesidad de modificar el código ya implementado, tendremos que extender la clase `Pedido` a una clase `PedidoPersonalizado` que cubra estas características.

Por otro lado, una vez que el sistema pasa a gestionar distintos tipos de pedidos, la clase que gestiona el almacén no necesita saber si cada uno de los pedidos es del tipo original o del nuevo, dado que los métodos a los que tendrá que llamar serán los mismos. Con todo esto, además de la herencia, estamos haciendo uso de un mecanismo llamado polimorfismo para que el sistema siga funcionando sin tener que modificar el código que ya teníamos. Es decir, todas las llamadas a métodos de objetos *Pedido* seguirán siendo las mismas. De este modo, estudiaremos el uso y posibilidades que nos ofrecen este tipo de mecanismos de la programación orientada a objetos.

4.1.2. Requisitos previos

Esta etapa requiere haber estudiado los capítulos del libro base de las etapas anteriores, así como los temas 10, 11 y 12 del temario detallado de la asignatura, correspondientes a los Capítulos 10 y 11, secciones 12.3 y 12.4 y el capítulo 14 y 7 del libro base para la Unidad Didáctica II.

4.1.3. Enunciado: Añadiendo pedidos personalizados.

A diferencia del pedido que hemos modelado, el pedido con personalizado incluirá un coste adicional por la colocación de la leña y por el hecho de asegurar una proporción de leña de tamaño grande, mediano y pequeño. Además, el sistema deberá registrar la proporción en porcentaje de leña pequeña, mediana y grande. Al crear un pedido de este tipo, el sistema deberá asegurarse de que las tres proporciones sumen 100. El coste adicional es el mismo para cualquier pedido personalizado, pero este coste puede modificarse en cualquier momento.

En base a estas especificaciones, el sistema deberá extenderse incluyendo los siguientes elementos:

1. **Extender la clase `Pedido`** con una nueva clase hija `PedidoPersonalizado`. Esta clase debe disponer de tres nuevos campos con la proporción de cada uno de los tamaños de leña. Además, al igual que en el caso del coste de las bolsas de astillas, el coste adicional por pedido personalizado deberá representarse como una variable o atributo de clase o estática de la clase `PedidoPersonalizado`, dado que este valor es compartido por todas las instancias de la clase.

Como en las etapas previas, vamos a crear una nueva clase llamada `EjemploEtapa31`, que contenga un método `main()`, cuyo cuerpo contenga el código fuente que se muestra a continuación[1]:

[1] En el curso virtual, junto con el enunciado de la práctica puede descargar el código fuente de esta clase EjemploEtapa31

```
public class EjemploEtapa3 {
	public static void main (String [ ] args) {
		Pueblo p=new Pueblo("Villanueva");
		p.setPrecioTransporte(20);
		Cliente c1=new Cliente("Juan Lopez",p);
		c1.setDireccion(Ç/ Quijote 3");
		Cliente c2=new Cliente("Luis Ramirez",p);
		c2.setDireccion(Ç/ Dulcinea 9");
		Pedido.setPrecioPorBolsa(7);
		Pedido.setPrecioPorKilo(0.75);
		Pedido pe1=new Pedido(c1);
		pe1.setCantidad(100);
		pe1.setBolsasAstillas(2);
		PedidoPersonalizado pe2=new PedidoPersonalizado(c2);
		pe2.setCantidad(50);
		pe2.setProporciones(50,25,25);
		AlmacenLenya g=new AlmacenLenya();
		g.anyadirPueblo(p);
		g.anyadirCliente(c1);
		g.anyadirCliente(c2);
		g.anyadirLenya(1000);
		g.anyadirPedido(pe1);
		g.anyadirPedido(pe2);
		g.anotarServido("Juan Lopez");
		g.anotarServido("Luis Ramirez");
	}
```

} Al ejecutar el main con este código, el sistema deberá imprimir en pantalla una serie de mensajes indicando:

```
Precio por bolsa de astillas modificado a: 7.0 euros
Precio por kilo modificado a: 0.75 euros

PUEBLO AÑADIDO
	Población: Villanueva
	Precio del transporte: 20.0 euros

CLIENTE AÑADIDO
Cliente: Juan Lopez
	Dirección: C/ Quijote 3
	Población: Villanueva
	Precio del transporte: 20.0 euros

CLIENTE AÑADIDO
Cliente: Luis Ramirez
	Dirección: C/ Dulcinea 9
	Población: Villanueva
	Precio del transporte: 20.0 euros

SE HA AÑADIDO 1000 KILOS DE LEÑA AL ALMACÉN
CANTIDAD ACTUAL: 1000 KILOS
PEDIDO AÑADIDO
INFORMACIÓN DE PEDIDO
Fecha: Wed Feb 05 14:49:02 GMT 2025
Cantidad: 100
Número de bolsas de astillas: 2
Cliente: Juan Lopez
	Dirección: C/ Quijote 3
	Población: Villanueva
	Precio del transporte: 20.0 euros
Estado del pedido: No servido
```

```
Precio total del pedido: 109.0 euros

PEDIDO AÑADIDO
INFORMACIÓN DE PEDIDO
Fecha: Wed Feb 05 14:57:31 GMT 2025
Cantidad: 50
Número de bolsas de astillas: 0
Cliente: Luis Ramirez
       Dirección: C/ Dulcinea 9
       Población: Villanueva
       Precio del transporte: 20.0 euros
Estado del pedido: No servido
Precio total del pedido: 87.5 euros
Proporción de leña pequeña: 50%
Proporción de leña mediana: 25%
Proporción de leña grande: 25%
Coste adicional por pedido personalizado: 30.0 euros
PEDIDO SERVIDO:
INFORMACIÓN DE PEDIDO
Fecha: Wed Feb 05 14:57:31 GMT 2025
Cantidad: 100
Número de bolsas de astillas: 2
Cliente: Juan Lopez
       Dirección: C/ Quijote 3
       Población: Villanueva
       Precio del transporte: 20.0 euros
Estado del pedido: Servido
Precio total del pedido: 109.0 euros
PEDIDO SERVIDO:
INFORMACIÓN DE PEDIDO
Fecha: Wed Feb 05 14:57:31 GMT 2025
Cantidad: 50
Número de bolsas de astillas: 0
Cliente: Luis Ramirez
       Dirección: C/ Dulcinea 9
       Población: Villanueva
       Precio del transporte: 20.0 euros
Estado del pedido: Servido
Precio total del pedido: 87.5 euros
Proporción de leña pequeña: 50%
Proporción de leña mediana: 25%
Proporción de leña grande: 25%
Coste adicional por pedido personalizado: 30.0 euros
```

4.2. ETAPA 3.2: Extendiendo la funcionalidad del sistema (opcional)

Esta etapa de la práctica es opcional. La correcta solución de esta etapa permitirá al estudiante optar a la máxima calificación de la práctica (sobresaliente), no permitiendo compensar deficiencias en otras partes obligatorias del enunciado, siendo un notable la máxima calificación a la que se puede optar, sin realizar esta parte opcional.

4.2.1. Objetivos

Hasta llegar a esta última etapa hemos hecho un recorrido por los aspectos más importantes de la programación orientada a objetos en Java. Para finalizar el diseño e implementación de la aplicación que tenemos entre manos, vamos a introducir nuevas funcionalidades de diferente dificultad.

4.2.2. Requisitos previos

Esta etapa requiere haber estudiado los capítulos del libro base de las etapas anteriores, así como los temas 10, 11 y 12 del temario detallado de la asignatura, correspondientes a las Secciones de la 8.3 a la 8.6, y los Capítulos 9, 10 y 11 del libro base para la Unidad Didáctica II. En concreto la sección 10.7 y relacionadas, así como el capítulo 11 resultarán útiles en la resolución de esta etapa.

4.2.3. Enunciado

Para abordar esta etapa tendremos que extender la clase `AlmacenLenya` con una nueva clase `AlmacenLenyaV2` que incluya algunas de las siguientes funcionalidades. Esta parte de la práctica sera puntuada en función de la cantidad de funcionalidades cubiertas:

1. Dificultad baja. Comprobar al añadir un nuevo pueblo que no exista ya un pueblo con el mismo nombre, en cuyo caso, debe notificarse en pantalla.
2. Dificultad baja. Comprobar al añadir un nuevo pedido, que el objeto `Pueblo` exista en la lista de pueblos del sistema. En caso contrario, debe notificarse en pantalla y no añadir el pedido.
3. Dificultad baja. Comprobar al añadir un nuevo cliente que no exista ya un cliente con el mismo nombre, en cuyo caso, debe notificarse en pantalla.
4. Dificultad baja. Comprobar al añadir un nuevo pedido que no exista ya un pedido no servido para un cliente con el mismo nombre, en cuyo caso, debe notificarse en pantalla.
5. Dificultad media. Comprobar al añadir un nuevo pedido que la cantidad solicitada no exceda la cantidad disponible en el almacén, en cuyo caso, debe notificarse en pantalla.
6. Dificultad media. Sumar las ganancias de los precios de los pedidos servidos hasta el momento.
7. Dificultad alta. Calcular la cantidad total de leña pendiente de servir para cada uno de los pueblos del sistema. Nota: para ello, puede recorrerse la lista de pueblos dados de alta en el sistema, y para cada uno de ellos, recorrer los pedidos sumando las cantidades de leña con servicio pendiente al pueblo correspondiente.
8. Dificultad muy alta. Calcular la cantidad de leña que hay que pedir al proveedor para poder cubrir tantos pedidos como los que se han realizado en el último mes. Nota: Para ello, se debe recorrer la lista de pedidos servidos tales que la fecha se corresponda al intervalo entre la fecha actual y el último mes y sumar las cantidades servidas.

En este ejercicio se debe extender la clase con las funcionalidades correspondientes, describir brevemente la implementación realizada, las llamadas de prueba y los resultados obtenidos.

Fechas y normas de entrega

5.1. Fechas

La realización de la práctica se llevará a cabo en los Centros Asociados, siendo las sesiones organizadas y supervisadas por el tutor de la asignatura. Habrá como mínimo tres sesiones presenciales de obligatoria asistencia. **Los estudiantes deberán ponerse en contacto con su centro asociado para informarse acerca de cuándo tendrán que asistir a las sesiones.**

Las fechas orientativas para la realización de cada una de las etapas serán:

- Finales de marzo. Realización de la primera etapa.
- Mediados de abril. Realización de la segunda etapa.
- Principios de mayo. Realización de la tercera etapa.

5.2. Normas de entrega

La práctica se entregará a través de una tarea definida en el entorno virtual de la asignatura. Cada estudiante creará y comprimirá una carpeta nombrada con su DNI y primer apellido separados por un guion (”8345385X-gonzalez”). Esta carpeta contendrá:

- Una memoria de **no más de 6 páginas** donde se expliquen las decisiones tomadas, así como el diseño (etapa 0, ver 1.2) realizado en cada parte de la práctica.
- Los ficheros *.java*, sin caracteres especiales; por ejemplo “ñ” o tildes finales, es decir, únicamente los de la última etapa.

NOTA IMPORTANTE: Los nombres de los ficheros y carpetas/paquetes que compongan la práctica entregada deben contener SÓLO caracteres correspondientes a las letras de la A a la Z, tanto mayúsculas como minúsculas, números del 0 al 9 y los caracteres especiales ’-’ y ’_’. **No deben utilizarse otros, tales como tildes o símbolos**.

NOTA IMPORTANTE: Los tutores tienen que cumplir una serie de requisitos ante los estudiantes debido a que la práctica cuenta para la calificación de la asignatura. Por tanto, antes de entregar las calificaciones definitivas al equipo docente deberán:

1. Informar de la nota de las prácticas a los estudiantes.
2. Establecer un día de revisión de prácticas (previo al período de entrega de las calificaciones al equipo docente), dado que éstas forman parte de la evaluación del estudiante.

Es importante que se mantengan todos los identificadores definidos en el enunciado, es decir, el nombre de las clases, atributos y métodos deben ser tal y como se definen en este enunciado.

5.3. Fechas de entrega

Para la convocatoria ordinaria (junio) todos los estudiantes deberán entregar su práctica en el entorno virtual antes del **jueves 8 de mayo**.

Los tutores deberán evaluar las prácticas y subir las notas al entorno virtual antes del **viernes 16 de mayo**. La fecha correspondientes a la convocatoria extraordinaria (septiembre) es el **martes 15 de julio** para estudiantes.

5.4. Evaluación de la práctica

Las prácticas tienen carácter **INDIVIDUAL**, para evitar posibles copias todas las prácticas pasarán por un software detector de ejemplares similares. La detección de prácticas copiadas implicará un **SUSPENSO en TODO el curso, es decir, convocatorias de junio y septiembre, para todos los implicados.**
Es requisito indispensable para aprobar el examen la asistencia a las 3 sesiones obligatorias de la práctica, así como superar la propia práctica.
Las prácticas NO se guardan de un curso para otro.

5.5. Preguntas al equipo docente

Las preguntas relativas a la instalación del entorno de desarrollo, puesta en funcionamiento y errores de compilación deben ser remitidas a los tutores de los centros asociados.

El equipo docente atenderá preguntas de carácter metodológico y de diseño, preferentemente a través de los foros del curso virtual de la asignatura y del siguiente correo electrónico:

fund_inf@lsi.uned.es

Los datos de contacto de cada uno de los miembros del equipo docente son:

Roberto Centeno Sánchez, *Profesor Titular de Universidad*
Tutorías: Jueves de 11:00 a 13:00h. y de 15:00 a 17:00 h.
Teléfono: 91 398 96 96
Mail: rcenteno@lsi.uned.es

Víctor Fresno Fernández, *Profesor Titular de Universidad*
Tutorías: Martes y Miércoles de 11:30 a 13:30
Teléfono: 91 398 82 17
Mail: vfresno@lsi.uned.es

Enrique Amigó Cabrera, *Profesor Titular de Universidad*
Tutorías: Jueves de 15:00 a 19:00
Teléfono: 91 398 86 51
Mail: enrique@lsi.uned.es

Alberto Pérez García Plaza, *Profesor Ayudante*
Tutorías: Jueves de 15:00 a 19:00
Teléfono: (+34) 91398-8412
Mail: alberto.perez@lsi.uned.es