



## FUNDAMENTOS DE INFORMÁTICA

### Ingeniería en Electrónica y Automática (18/19)

#### ***Práctica 5***

#### ***POO en el entorno BlueJ. Diseño de clases sencillas***

Alumnos:		Grupo:

Un primer objetivo de esta práctica es familiarizarse con el entorno de programación en Java BlueJ, en lo que se refiere a su entorno gráfico de evaluación de objetos y otras herramientas que proporciona orientadas a la POO. Otro objetivo es repasar todos los conceptos estudiados en actividades anteriores y familiarizarse con el diseño de clases sencillas, sus componentes fundamentales (campos o atributos, constructores y métodos), el desarrollo de los mismos y la creación de nuevas clases utilizando otras para resolver problemas más complejos, haciendo hincapié en los distintos tipos de interrelación que pueden aparecer entre clases y cómo utilizar los elementos de unas clases en otras.

#### ***FASE 1. HERRAMIENTAS POO DEL ENTORNO GRÁFICO BLUEJ***

Para realizar la práctica descarga a tu equipo el contenido del archivo “*proyectos.zip*” que está disponible en el Anillo Digital Docente (asegúrate de descomprimirla en el equipo local para poderla utilizar).

**Ejercicio 1.1.-** Abre el proyecto denominado “*proyectoFiguras*”. Para ello tienes que ir al menú “*Project | Open Project...*”, y buscarlo dentro de la carpeta “*proyectos*” que habéis copiado a vuestro equipo. Se abrirá una ventana que mostrará el diagrama UML de las clases contenidas en dicho proyecto: Punto2D, Triangulo y Teclado. Puede abrirse, también, haciendo doble clic en el icono denominado “*package.bluej*” existente en la carpeta descargada. Visualiza el aspecto de la ventana de proyecto que se ha abierto y fíjate en los detalles de los elementos que aparecen ya que su significado es importante.

En la barra de título de la ventana de proyecto que se ha abierto puede observarse algo similar a “BlueJ: proyectoFiguras [figuras]”. Ello significa que la ventana de proyecto que se ha abierto corresponde al paquete “*figuras*” perteneciente al proyecto “*proyectoFiguras*”. Haciendo doble click en el icono denominado “go up” se abre la ventana de proyecto correspondiente al proyecto “*proyectoFiguras*”. Fíjate que un proyecto puede contener cero, uno o varios paquetes. Observa, en la ventana del explorador de archivos del sistema operativo, la correspondencia que existe entre cómo se organizan los elementos del proyecto y cómo se organizan los archivos en el disco; por cada paquete del proyecto se habrá generado un directorio con su mismo nombre.

Es posible que los iconos de las clases Punto2D, Triangulo y Teclado estén sombreados, indicando que el código Java que les corresponde no ha sido compilado. Si es el caso, compílalas seleccionándolas y eligiendo la opción “*compile*” que aparecerá en el menú emergente al hacer clic con el botón derecho del ratón o con el botón “*compile*” del entorno. Observa el cambio de imagen del icono.

Haciendo doble clic sobre el icono de cada una de las clases, es posible acceder al código fuente en Java que corresponde a cada una de ellas. Lee atentamente las descripciones de cada una de las clases y echa un primer vistazo a su código Java intentando comprenderlo mediante su análisis. **Dedícale el tiempo necesario hasta conseguir una comprensión básica del mismo si algunos elementos del código que aparece no te resultan conocidos.**

**Ejercicio 1.2.-** Con BlueJ, es posible crear objetos individuales de cualquier clase e interactuar con ellos ejecutando sus métodos **no privados**. El objetivo que se persigue con ello es poder desarrollar sin errores partes del código del programa sin la obligación de tener que completar dicho programa para comprobar que es correcto y funciona. Selecciona la clase Punto2D y haz clic con el botón derecho del ratón. En el menú emergente, ahora que la clase está compilada, **aparecen también las operaciones disponibles para la clase**. En nuestro caso, además, pueden observarse los constructores de objetos de la clase y los encabezamientos de los métodos de clase (*static*).

Selecciona el primer constructor para la clase Punto2D (*new Punto2D()*) para crear un nuevo objeto de esta clase. En el cuadro diálogo que aparece, además de **mostrar los comentarios de método escritos en el código fuente**, se te solicitará un nombre para el objeto que se va a crear (aparecerá por defecto el nombre *punto2D1*, lo puedes dejar tal cual).

Una vez que se ha creado el objeto, aparecerá en el **banco de objetos** con el nombre que se le haya dado (**punto2D1** si no se ha modificado). BlueJ ofrece la posibilidad, seleccionando un objeto del banco de objetos con el botón derecho, de inspeccionar dicho objeto, viendo el valor que tienen sus atributos en un determinado momento. También permite ejecutar sus métodos no privados a través del menú contextual que aparece. Inspecciona el objeto **punto2D1**. ¿Cuáles son los valores de sus atributos? ¿Por qué?

Para el punto creado ejecuta de forma similar los métodos que devuelven los valores de las coordenadas y el cuadrante al que pertenece el punto. Observa lo que aparece en los cuadros de diálogo. Analiza el código correspondiente a los métodos si es necesario y responde a las preguntas: ¿Cómo se llaman? ¿Qué hacen? ¿Qué valores devuelven?

Utiliza los otros dos constructores para generar dos objetos de clase **Punto2D**. El primero para generar el punto de coordenadas (2.5, -1.5) y el segundo para generar un clon de éste último recién creado. Observa los cuadros de diálogo que se han mostrado. ¿Qué información has tenido que introducir? Para comprobar que efectivamente los dos puntos son iguales, ¿qué has hecho? ¿Hay otras posibilidades para hacerlo?

**Ejercicio 1.3-** Localiza e invoca desde el banco de objetos los métodos que permiten modificar los valores de las coordenadas de un punto a valores nuevos y utilízalos para cambiar las coordenadas del tercer punto creado de forma que sus valores sean  $(-2.0, 3.5)$ . Localiza el código de dichos métodos en el código fuente y explica brevemente qué es lo que hacen.

**Ejercicio 1.4-** Localiza e invoca desde el banco de objetos el método que muestra por pantalla las coordenadas del punto. Observa el cuadro de diálogo que aparece. Localiza el código de dicho método en el código fuente y explica brevemente qué es lo que hace. Aplícalo a los tres objetos de clase **Punto2D** creados. ¿Cómo se llama el método? ¿Qué información has tenido que introducir en el cuadro de diálogo? ¿Cómo la has tenido que introducir? ¿Aparece en la ventana de Terminal dicha información? ¿Qué valores de coordenadas se han mostrado?

**Ejercicio 1.5-** Localiza e invoca desde el banco de objetos el método que permite mover un punto de una posición del plano a otra en relación a la que tenía anteriormente. Utilízalo para mover el punto que estaba en el origen de coordenadas 2 unidades a la derecha y 3 unidades hacia abajo. Localiza el código de dicho método en el código fuente y explica brevemente qué es lo que hace. ¿Cómo se llama el método? ¿Qué información has tenido que introducir en el cuadro de diálogo? ¿Cuáles son las nuevas coordenadas del punto movido?

**Ejercicio 1.6.-** Crea un nuevo objeto, esta vez de la clase **Triangulo**, utilizando el constructor apropiado y da valores a sus atributos a través del cuadro de diálogo que aparece. Utiliza el inspector de objetos y localiza los objetos de la clase **Punto2D** con los que has construido el objeto de la clase **Triangulo** (tendrás que hacer clic varias veces en los iconos con flechas que representan el valor de los atributos del objeto). Comprueba que los valores que se observan se corresponden con las coordenadas de los puntos utilizados como vértices.

**Ejercicio 1.7.-** Ejecuta el método *escribir(String mensaje)* del objeto triángulo y escribe lo que aparece en la ventana:

**Ejercicio 1.8.-** Ejecuta el método *perimetro()* del objeto triángulo y escribe el resultado obtenido y describe cómo lo comunica el entorno BlueJ.

**Ejercicio 1.9.-** Modifica el tercer punto creado para que se convierta en el origen de coordenadas. ¿Cómo lo has hecho? Descríbelo brevemente.

**Ejercicio 1.10.-** Ejecuta de nuevo el método *escribir(String mensaje)* del objeto triángulo y escribe lo que aparece en la ventana. Compáralo con el resultado obtenido en el ejercicio 1.7. ¿Qué puede observarse? ¿Por qué? ¿Qué está ocurriendo?

## ***FASE 2. EDICIÓN Y COMPILACIÓN CON BLUEJ***

**Ejercicio 2.1.-** Vas a modificar el código fuente Java del proyecto editando un método de la clase **Triangulo** para que realice una nueva tarea. Dicho método comprobará si los vértices que forman el triángulo son válidos o no. Para que sean válidos deberían ser puntos distintos y no estar alineados aunque se puede pensar en una forma más sencilla, teniendo en cuenta las longitudes de los lados, si se cumple que ninguno de los lados sea mayor o igual en longitud que la suma de los otros dos que forman el triángulo.

El código que tenemos que desarrollar es, por tanto, el del método *esTriangulo()* de la clase **Triangulo**. El esqueleto del método ya se encuentra en el código fuente de la clase **Triangulo** y está preparado para ser editado. Edita el método y compila la clase **Triangulo** hasta que, en la línea de estado en la parte inferior de la ventana de edición, BlueJ te informe de que la clase ha sido compilada y de que no hay errores sintácticos. Mientras haya errores sintácticos, el compilador de Java indicará en esa misma línea la causa del error y, en la mayor parte de las ocasiones, BlueJ dará información adicional si se hace clic en el enlace que aparece mostrando el número de errores en la esquina inferior derecha de la ventana de edición.

Una vez que hayas compilado sin errores, crea nuevos objetos de la clase **Punto2D** y **Triangulo** (recuerda que cada compilación que se realiza elimina los objetos existentes) y comprueba que *esTriangulo()* se ejecuta de acuerdo con la nueva especificación. Cuando sea así, escribe a continuación las líneas de programa que hayas desarrollado en el código fuente:

**Ejercicio 2.2.-** Vas a modificar de nuevo el código fuente Java del proyecto editando otro método de la clase `Triangulo` para que realice otra nueva tarea. Dicho método calculará el área del triángulo. Localiza en el código fuente de la clase `Triangulo` el esqueleto incompleto del método que tienes que diseñar. Para poderlo hacer ten en cuenta que el problema puede resolverse aplicando la fórmula de Herón que permite calcular el área del triángulo conociendo las longitudes de sus lados. La fórmula de Herón es la siguiente:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

donde  $a$ ,  $b$  y  $c$  son los valores de las longitudes de sus lados y  $s$  es el *semiperímetro* del triángulo.

Repite el mismo proceso de edición que en el ejercicio anterior y una vez que hayas compilado sin errores, crea nuevos objetos de la clase `Punto2D` y `Triangulo` y comprueba que el método `area()` se ejecuta adecuadamente y realiza el cálculo de forma correcta. Cuando sea así, escribe a continuación las líneas de programa que hayas desarrollado en el código fuente:

**Ejercicio 2.3.-** De manera análoga a los apartados anteriores, desarrolla los métodos `baricentro()` y `esRectangulo()` cuyo esqueleto está contenido de manera incompleta en el código fuente de la clase `Triangulo`. Una vez concluidos, compruébalos introduciendo los valores que sean necesarios.



### ***FASE 3. CREACIÓN DE CLASES***

Utilizando como base las clases `Punto2D` y `Triangulo` desarrolladas en apartados anteriores, vamos a implementar, de forma completa desde cero, la clase `Circunferencia`, para representar circunferencias caracterizadas por su **centro** y **radio**.

**Ejercicio 3.1.-** Genera una nueva clase denominada `Circunferencia`. En la plantilla de código que aparece deberás modificar los elementos para que se adecuen a lo que perseguimos. Elimina el atributo que aparece y sustitúyelo por los atributos necesarios para representar una circunferencia. Siguiendo el principio de ocultación, defínelos para que sean privados. ¿Qué atributos has puesto?

**Ejercicio 3.2.-** Vamos a añadir ahora constructores en la clase. Primero escribiremos un constructor con dos parámetros que transmitirán valor a los atributos del objeto creado. Una vez completado y comprobado, escribiremos otro constructor que sirva para clonar objetos de clase `circunferencia`. El código fuente de la clase `Triangulo` puede servir de orientación. ¿Qué cabeceras has puesto a los constructores?

**Ejercicio 3.3.-** El siguiente paso podría ser escribir métodos observadores y modificadores. Hay que recordar que cuando tenemos atributos `private` los métodos observadores pueden ser necesarios pero, por el contrario, los métodos modificadores no siempre lo van a ser. A pesar de ello, en este ejercicio los vamos a añadir. Escribe dichos métodos para todos los atributos.

**Ejercicio 3.4.-** Basándonos en el método *LeerTriangulo()* de la clase *Triangulo*, vamos a escribir el método *LeerCircunferencia()* y lo vamos a añadir a la clase que estamos desarrollando.

**Ejercicio 3.5.-** De la misma forma que en el ejercicio anterior, basándonos en el método `toString()` de la clase `Triangulo`, vamos a escribir el método `toString()` aplicado a circunferencias y lo vamos a añadir a la clase que estamos desarrollando.

**Ejercicio 3.6.-** Desarrolla el método *escribir()* en la clase *Circunferencia*, de forma similar al método del mismo nombre desarrollado en la clase *Triangulo*.

**Ejercicio 3.7.-** Desarrolla el método *area()* en la clase *Circunferencia*, que calcule el área que encierra la circunferencia objeto dueño.

**Ejercicio 3.8.-** Desarrolla el método *perimetro()* en la clase *Circunferencia*, que calcule el perímetro de la circunferencia objeto dueño.

**Ejercicio 3.9.-** De manera **opcional**, desarrolla los métodos necesarios para calcular la posición relativa (interior, exterior) de un punto respecto de una circunferencia. Observa que habrá diferentes formas de afrontar el problema y darle solución. Describe brevemente la estrategia que has seguido.

--

## **FASE 4. HERENCIA Y POLIMORFISMO**

En esta fase vamos a modificar las clases desarrolladas hasta el momento para añadir algunos conceptos de la POO en nuestra implementación, en concreto la **herencia** y el **polimorfismo**.

**Ejercicio 4.1.-** Como primer paso en esta fase, añadiremos a nuestro proyecto la clase **Principal**, cuyo código nos hemos descargado del ADD a principio de la práctica. Dicha clase se encuentra en el fichero “**principal.java**”.

Una vez añadida, dedica el tiempo necesario para analizar su código. Aunque es relativamente corto, contiene elementos que pueden resultar algo confusos al principio. La clase **Principal** contiene sólo un método (**main**) que lanzará la ejecución de nuestro programa. De forma resumida, el programa pide al usuario que seleccione entre un triángulo y una circunferencia y, una vez hecho, lee valores para construirlo y, después de mostrarlo por pantalla, calcula su área y su perímetro. El objetivo que perseguimos en este apartado de la actividad es **conseguir que funcione dicho programa**.

**Ejercicio 4.2.-** Intenta compilar la clase **Principal**. El compilador lanzará un mensaje de error en el que se nos avisa que se ha declarado un objeto **fig** de la clase **Figura** que no es conocida. Analizando el código se observa que dicho objeto se utiliza **para asignarle indistintamente** un objeto de la clase **Triangulo** o un objeto de la clase **Circunferencia**, objetos que devuelven, respectivamente, los métodos *LeerTriangulo()* y *LeerCircunferencia()*.

Este hecho nos debe llevar a pensar que la clase **Figura**, que debemos diseñar y añadir a nuestro proyecto, debe ser una **superclase** de las clases **Triangulo** y **Circunferencia**, estableciéndose una interrelación de herencia entre éstas y la primera.

Genera una nueva clase denominada **Figura** y edita su contenido para dejar vacío su cuerpo. Al compilar la clase **Principal** ya no mostrará el problema anterior sino otros diferentes que hacen referencia a la incompatibilidad entre el objeto de clase **Triangulo** que genera *LeerTriangulo()* y el objeto **fig** de la clase **Figura**. Para resolver este problema tenemos que convertir la clase **Figura** en **superclase** de las clases **Triangulo** y **Circunferencia** y, para ello, debemos modificar la cabecera de éstas de forma que queden respectivamente:

```
public class Triangulo extends Figura {...}
public class Circunferencia extends Figura {...}
```

Una vez modificado el código de estas clases, compila ambas (no debería comunicar ningún error) y observa en la ventana de proyecto la forma que expresa gráficamente la relación de herencia entre las clases que hemos provocado con los cambios en el código.

**Ejercicio 4.3.-** Intenta compilar la clase **Principal**. Al generar herencia hemos conseguido que los objetos **Triangulo** y **Circunferencia** sean objetos de clase **Figura** por lo que las asignaciones que nos daban error antes, ahora son completamente correctas. En cambio, ahora el compilador mostrará otro mensaje de error distinto en el que se nos avisa que el método *escribir()* no está en el código de la clase **Figura**.

Al contrario, el método *escribir()* sí se encuentra en las clases derivadas **Triangulo** y **Circunferencia**, y nos interesa que sean dichos métodos los que se ejecuten, de forma que muestre por pantalla los valores del triángulo o de la circunferencia, según sea el caso. Por ello, tendremos que forzar **polimorfismo** para este método y para conseguirlo añadiremos el siguiente método en la clase **Figura**:

```
protected void escribir(String mensaje){}
```

Aunque no sea demasiado trascendente, optamos por dar visibilidad protegida a este método que, como recordatorio, hará que sea visible en clases derivadas y a nivel de paquete. Definiendo el método en la clase **Figura** habremos conseguido nuestro objetivo pero el compilador mostrará, en esta ocasión, un error cuya causa es similar a la anterior aunque referente al método *area()* y posteriormente al método *perimetro()*. Resuelve ambos errores hasta que compile correctamente la clase **Principal**. ¿Qué código has tenido que escribir?

Una vez resueltos los problemas, comprueba que el programa se ejecuta adecuadamente. Observa el resultado de la ejecución. Recuerda que hemos utilizado herencia y polimorfismo para conseguir los objetivos.

**Ejercicio 4.4.-** Hasta ahora hemos utilizado la herencia y el polimorfismo para hacer funcionar el código de la clase **Principal**. La clase **Figura** sólo contiene los encabezamientos de los métodos que se redefinen en las clases derivadas **Triangulo** y **Circunferencia**, de forma que mediante polimorfismo conseguimos que el funcionamiento de los mismos se adapte a la clase a la que pertenece cada objeto distinto. Vamos a dar otro paso más y vamos a añadir un atributo, denominado **color**, a la superclase **Figura**. Por sencillez será de tipo entero y por el principio de ocultación lo definiremos privado. ¿Cómo lo has declarado en el código fuente?

**Ejercicio 4.5.-** Con la definición anterior hemos conseguido que los objetos de las clases **Triangulo** y **Circunferencia** tengan, por herencia, el atributo **color**. Se puede observar creando un objeto de clase **Circunferencia** mediante el método *LeerCircunferencia()*. Ejecuta este método invocándolo desde el menú contextual que aparece pulsando el botón derecho sobre el icono de la clase **Circunferencia** en la ventana de proyecto y, después de la interacción que se establece con el usuario en la ventana de terminal, aparecerá un cuadro de diálogo en el cual podemos pulsar en el botón **“Get”**. Al hacerlo, se creará un objeto que se almacenará en el banco de objetos y lo podremos inspeccionar como en otras ocasiones. Realiza esta tarea. ¿Qué valor presenta el atributo **color** del objeto creado? ¿A qué se debe?

**Ejercicio 4.6.-** Con la última modificación de código hemos conseguido que los objetos **triangulo** y **circunferencia** tengan un nuevo atributo (**color**) definido en la clase **Figura** y transmitido por herencia. Esto nos va a obligar a realizar algunas modificaciones en el código para poder acceder a su valor y poderlo procesar. Para conseguir esto podríamos optar por escribir métodos observador y modificador para el atributo **color** en la clase **Figura**. Escribe ambos métodos en la clase **Figura**. Denomínalos *setColor()* y *getColor()* y añade parámetros si fuera necesario. Obliga a que tengan visibilidad **protected**. Realiza pruebas para comprobar que es correcto el código generado y funciona adecuadamente.

**Ejercicio 4.7.-** Con la adición de los métodos observador y modificador, podremos tener acceso al atributo `color` de los triángulos y las circunferencias que se vayan generando teniendo en cuenta que, para ello, deberemos adaptar el código de las clases derivadas para incluir aquellas instrucciones que nos permitan dicho acceso en aquellos puntos que sea necesario.

Otra forma, quizá más elegante, de tratar el atributo `color` que hemos añadido a la clase `Figura` es **darle valor en el momento en el que se genera un objeto** de clase `Figura`. Para ello debemos desarrollar un constructor que dé valor a dicho atributo en el momento de la construcción del objeto. Desarrolla un constructor para la clase `Figura` que reciba un parámetro entero y asigne dicho valor recibido al atributo `color`. Defínelo con visibilidad `protected`. Compila la clase `Figura` y genera un objeto de esta clase para comprobar que funciona. ¿Qué código has generado?

**Ejercicio 4.8.-** Con la última modificación de código, las clases `Principal`, `Triangulo` y `Circunferencia` quedarán sombreadas, lo que quiere decir que hay que recompilarlas. Intenta compilar la clase `Circunferencia`. El compilador mostrará mensajes de error que **debemos comprender para resolverlos**. En este caso, uno de los errores que se muestran hace referencia a la no coherencia entre los parámetros de los constructores de las clases `Figura` y `Circunferencia`. Para resolverlo debemos añadir un parámetro entero, denominado `color`, en la lista de parámetros formales de la cabecera del constructor de la clase `Circunferencia` que se ha marcado como errónea y, para poder dar valor al atributo `color` cuando se crea un objeto de la clase `Circunferencia`, en el cuerpo del constructor deberá aparecer una invocación al constructor de la superclase, `super(color)`, de forma que el código queda:

```
public Circunferencia(Punto2D cen, double rad, int color) {  
    super(color);  
    this.centro = new Punto2D(cen);  
    this.radio = rad;  
}
```

Hay que observar que la instrucción que hemos añadido en el cuerpo del constructor debe figurar como **primera línea del código de dicho constructor** (en caso contrario da un error).

Con estas modificaciones el error anterior deja de mostrarse pero siguen apareciendo otros mensajes de error. El siguiente que se muestra es del mismo tipo que el anterior pero el compilador señala el constructor que clona circunferencias. En este caso, para resolverlo, añadiremos en la primera línea del cuerpo de dicho constructor, de nuevo, una invocación al constructor de la superclase de la siguiente forma:

```
super(cir.getColor())
```

Hechas las modificaciones anteriores, si volvemos a compilar el código, el siguiente mensaje de error mostrado señala la invocación a la generación de un nuevo objeto de la clase en el método *LeerCircunferencia()*. El error se debe a la no concordancia de parámetros actuales de la invocación *new Circunferencia( ... )* con respecto a la definición formal del constructor, por lo que tendremos que añadir un valor para **color** en dicha invocación. Habría varias formas de obtener dicho valor pero es aconsejable modificar el código de dicho método de forma que quede parecido a lo siguiente:

```
public static Circunferencia LeerCircunferencia() {  
    Punto2D centro = Punto2D.LeerPunto2D();  
    double radio = Teclado.LeerReal("Valor del radio? ");  
    int color = Teclado.LeerEntero("Valor de color? ");  
    return new Circunferencia( centro , radio, color );  
}
```

Una vez realizados estos cambios, podremos compilar la clase **Circunferencia** y, para probar su funcionamiento, vamos a invocar el método *LeerCircunferencia()* y guardar en el banco de objetos el objeto que se genera con su ejecución. Podremos inspeccionar dicho objeto para comprobar sus valores y observar que ha funcionado correctamente.

**Ejercicio 4.9.-** Recordando todo lo realizado en el ejercicio anterior, realiza lo mismo aplicado a la clase `Triangulo`, de forma que al final del proceso ya no se den errores de ningún tipo. Prueba los cambios realizados. Describe brevemente lo que has tenido que realizar.

**Ejercicio 4.10.-** Una vez realizado los cambios anteriores, compila la clase `Principal` y ejecuta el programa para comprobar que funciona correctamente. Analiza detenidamente los resultados de la ejecución. ¿Qué podemos observar?

**Ejercicio 4.11.-** Vamos a resolver el problema de que no se muestre el valor de `color` en los resultados de la ejecución. Aunque podría hacerse de maneras diferentes según sean nuestros intereses o problemas a resolver, vamos a hacer las siguientes modificaciones. En primer lugar podemos modificar el método `escribir()` de la clase `Figura` para que quede de la siguiente manera:

```
protected void escribir(String mensaje){
    System.out.println("El color es: " + this.color);
}
```

Por otra parte, modificaremos el código del método `escribir()` de la clase `Circunferencia` para que quede de la siguiente manera:

```
public void escribir(String mensaje) {
    System.out.println(mensaje);
    super.escribir("");
    System.out.println(this.toString());
}
```



Analiza el código detenidamente y comprueba que los cambios han tenido efecto. ¿Cómo has realizado dicha comprobación?

Realiza el mismo cambio aplicándolo a la clase `Triangulo`. Compila todas la clases y realiza una o más ejecuciones para comprobar que el resultado es el correcto.

## ***FASE 5. CLASES ABSTRACTAS***

Las clases abstractas suponen otra herramienta adicional que podemos utilizar en nuestra programación. De forma simplificada, *las podremos usar cuando no tenga sentido hablar de objetos de la superclase o, también, cuando no existan en el contexto que manejemos ningún objeto que no esté dentro de alguna de las clases derivadas*. Vamos a aplicar algunos cambios a nuestro programa para ver qué consecuencias tenemos que abordar.

**Ejercicio 5.1.-** Supongamos que en nuestro contexto de problema **sólo van a existir figuras que sean triángulos o circunferencias**. En esta situación se podría justificar que la clase `Figura` la definiéramos como abstracta. Para ello cambiaremos la cabecera de la clase para dejarla como sigue:

```
public abstract class Figura {...}
```

Hecha la modificación podemos compilar la clase y observaremos que ha habido cambios en la ventana de proyecto, en diagrama UML que representa dicha clase. ¿Qué se observa?

**Ejercicio 5.2.-** Las clases **Principal**, **Triangulo** y **Circunferencia** aparecen sombreadas lo que quiere decir que deben ser recompiladas. Vuélvelas a compilar y ejecuta de nuevo el programa. Comprueba que funciona adecuadamente.

**Ejercicio 5.3.-** Hemos definido la clase **Figura** como abstracta. No se pueden construir objetos de clases abstractas directamente, cosa que en BlueJ podemos observar en las opciones que aparecen en el menú contextual que surge pulsando el botón derecho sobre el icono de clase y que en este caso no aparece la opción de crear objeto (llamada a constructor). A pesar de ello, las clases abstractas pueden contener constructores explícitamente en su código, aunque no pueden ser invocados directamente sino mediante los constructores de las clases derivadas.

En cualquier caso, **una clase es abstracta cuando tiene métodos abstractos**. En nuestro caso hay dos métodos que claramente deben serlo y que podemos identificar porque su cuerpo no contiene instrucciones que se vayan a ejecutar y puede parecer que no tienen sentido. Convierte en abstractos los métodos *area()* y *perimetro()*.

¿Qué código has escrito para ello? ¿Cómo quedan dichos métodos en la clase **Figura**?

**Ejercicio 5.4.-** Si intentamos hacer lo mismo con el método *escribir()*, ¿qué ocurriría? ¿Por qué no puede ser abstracto dicho método?