Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

PRACTICA 7: HERENCIA

1. OBJETIVO

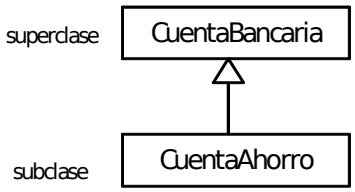
El objetivo de esta práctica es presentar la herencia en el entorno de programación elegido para realizar las prácticas de Fundamentos de Programación II. Descargue el código de la plataforma virtual para realizar la práctica 7. Se proporciona un makefile **`makeP7`** para la realización de la práctica 7.

2. HERENCIA

La herencia es una de las piedras angulares de la programación orientada a objetos ya que permite la creación de clasificaciones jerárquicas. Gracias a la herencia, se puede definir una clase general que define características comunes a un conjunto de elementos relacionados. Esta clase puede ser heredada por otras clases más específicas, añadiendo cada una de éstas aquellas cosas que son particulares a ella. En la terminología de Java, a la clase general se le llama **superclase**. A las clases más específicas que heredan de la general se les llama **subclases**. Una subclase es una versión especializada de una superclase, que hereda todas las variables de instancia y los métodos definidos por la superclase y que añade sus propios elementos.

Fundamentos

Para heredar una clase, simplemente es necesario incorporar su definición en la definición de otra clase utilizando la palabra clave **`extends`**. Para entender cómo se realiza esto comencemos con un ejemplo sencillo. El siguiente programa crea una superclase llamada **`CuentaBancaria`** y una subclase llamada **`CuentaAhorro`**. Esto lo representamos de forma gráfica de la siguiente forma:



Observe cómo se utiliza **`extends`** para crear la subclase **`CuentaAhorro`** de **`CuentaBancaria`**.

Primero la superclase:

```
package fp2.poo.practica7;


public class CuentaBancaria{

    private double saldo;

    public CuentaBancaria() {
        this.saldo = 0.;
    }

    public CuentaBancaria(double cantidadInicial) {
        System.out.println("Constructor de CuentaBancaria");
        this.saldo = cantidadInicial;
    }

    public double getSaldo(){
        return this.saldo;
    }
    public void setSaldo(double saldo){
        this.saldo = saldo;
    }

    public void retirar(double cantidad) throws NoFondosDisponiblesException {
        if( cantidad > 0 ){
            if (this.saldo >= cantidad) {
                this.saldo = this.saldo - cantidad;
            } else {
                throw new NoFondosDisponiblesException("No hay suficientes fondos");
            }
        }
    }

    public void depositar(double cantidad){
        if(cantidad > 0){
            this.saldo = this.saldo + cantidad;
        }
    }
}
```

CuentaBancaria.java

Segundo la subclase (en el fichero proporcionado se incluye mas código que se utilizará en ejemplos posteriores):

```
package fp2.poo.practica7;


public class CuentaAhorro extends CuentaBancaria {

    private double interes;

    public CuentaAhorro (){
        System.out.println("Constructor de CuentaAhorro");
        this.interes = 0.05d;
    }

    public void setInteres(double interes){
        this.interes = interes;
    }

    public double getInteres(){
        return this.interes;
    }
}
```

CuentaAhorro.java

Tercero la clase que contiene el método **`main`**:

```
package fp2.poo.practica7;

public class Practica7Ejercicio01 {
    public static void main (String args[]){
        CuentaAhorro   cuentaAhorro   = new CuentaAhorro();
        cuentaAhorro.setSaldo(1000.00d);
        System.out.println("Saldo   : " + cuentaAhorro.getSaldo());
        System.out.println("Interes : " + cuentaAhorro.getInteres());
    }
}
```

Practica7Ejercicio01.java

Para probar este ejemplo ejecute:

`make –f makeP7 prueba01`

La subclase **`CuentaAhorro`** incluye todos los miembros de su superclase **`CuentaBancaria`**, por lo que **`cuentaAhorro`** además del atributo de la superclase `saldo` tiene el atributo `interes`.
Aunque **`CuentaBancaria`** es una superclase de **`CuentaAhorro`**, también es una clase autónoma y totalmente independiente. Aunque una superclase tenga una subclase, esto no significa que la superclase no pueda ser utilizada por sí sola. Además, una subclase puede a su vez ser una superclase de otra subclase.

La forma general de la declaración de una **clase** que hereda de otra es la siguiente:

```
class NombreDeLaSubclase extends NombreDeLaSuperclase {
 //cuerpo de la clase
}
```

Una variable de la superclase puede referenciar a un objeto de la subclase

A una variable referencia de la superclase se le puede asignar una referencia a cualquier subclase derivada de dicha superclase. Esta es una característica de la herencia muy útil y que se suele utilizar en bastantes situaciones. Veamos el siguiente ejemplo:

```
package fp2.poo.practica7;

public class Practica7Ejercicio02 {
    public static void main (String args[]){
        /*
         * Referencia a la superclase
         */
        CuentaBancaria cuentaBancaria = null;
        /*
         * Creacion de un objeto de la subclase
         */
        CuentaAhorro   cuentaAhorro   = new CuentaAhorro();
        cuentaAhorro.setSaldo(1000.00d);
        System.out.println("Saldo   : " + cuentaAhorro.getSaldo());
        System.out.println("Interes : " + cuentaAhorro.getInteres());
        /*
         * Variable de la superclase que referencia a un objeto de la subclase
         */
        cuentaBancaria = new CuentaAhorro();
        System.out.println();
        System.out.println("Saldo   : " + cuentaBancaria.getSaldo());
        /*
         * Variable de la superclase no puede acceder a los
         * miembros añadidos en la subclase
         */
        //System.out.println("Interes : " + cuentaBancaria.getInteres());
    }
}
```

Practica7Ejercicio02.java

Para probar este ejemplo ejecute:

`make –f makeP7 prueba02`

La variable **`cuentaAhorro`** es una referencia de tipo **`CuentaAhorro`** (subclase)**,** y **`cuentaBancaria`** es una referencia de tipo **`CuentaBancaria`** (superclase).

Como **`CuentaAhorro`** es una subclase de **`CuentaBancaria`**, es posible asignar a **`cuentaBancaria`** una referencia del tipo **`CuentaAhorro`**.

El tipo de la variable referencia determina qué miembros son accesibles. En este caso no se puede acceder a **`getInteres()`**, mediante la referencia a la superclase (**`cuentaBancaria`**), ya que este método pertenece a la subclase, y no es visible desde una referencia a la superclase.

Quite el comentario de la última línea de código de la clase **`Practica7Ejercicio02`**, compile de nuevo y observe el resultado.

Uso de super en herencia

Cuando una subclase necesita referirse a su superclase inmediata, lo puede hacer utilizando la palabra clave **`super`**. La palabra reservada **`super`** se puede utilizar de dos formas:

1. La primera para llamar al constructor de la superclase.
2. La segunda se utiliza para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase.

A continuación vamos a ver el uso de **`super`** para llamar al constructor de la superclase. Una subclase puede llamar al constructor de la superclase utilizando **super** de la siguiente forma:

```
super (ListaDeParametros);
```

Aquí, **`ListaDeParametros`** especifica los parámetros del constructor de la superclase. Si se utiliza **`super()`**, tiene que ser la primera sentencia ejecutada dentro del constructor de la subclase.

Para ilustrar el ejemplo se ha añadido dos nuevos constructores a la clase `CuentaAhorro` en el primero se proporciona el saldo y en el segundo se proporciona el saldo y el interés.

```
package fp2.poo.practica7;


class CuentaAhorro extends CuentaBancaria {

    private double interes;

    public CuentaAhorro (){
        this.interes = 0.05d;
    }

    //Ejemplo de super para Practica7Ejercicio03
    public CuentaAhorro (double saldo){
        super(saldo);
        this.interes = 0.05d ;
    }

    public CuentaAhorro (double saldo, double interes){
        super(saldo);
        this.interes = this.interes;
    }

    public void setInteres(double interes){
        this.interes = interes;
    }

    public double getInteres(){
        return this.interes;
    }
}
```

`CuentaAhorro.java`

En este programa se utilizan los constructores añadidos:

```
package fp2.poo.practica7;


public class Practica7Ejercicio03 {
    public static void main (String args[]){
        /*
         * Llamada con un valor para saldo
         */
        CuentaAhorro   cuentaAhorro1   = new CuentaAhorro(6000.d);
        System.out.println();
        System.out.println("Saldo   de cuenta 1: " + cuentaAhorro1.getSaldo());
        System.out.println("Interes de cuenta 1: " + cuentaAhorro1.getInteres());
        System.out.println();
        /*
         * Llamada con un valor para saldo e interes
         */
        CuentaAhorro   cuentaAhorro2   = new CuentaAhorro(5000.d, 0.05);
        System.out.println("Saldo   de cuenta 2: " + cuentaAhorro2.getSaldo());
        System.out.println("Interes de cuenta 2: " + cuentaAhorro2.getInteres());
    }
}
```

Practica7Ejercicio03.java

Para probar este ejemplo ejecute:

`make –f makeP7 prueba03`

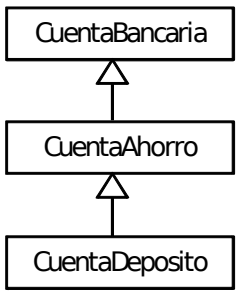
En este ejemplo se invoca al constructor de la superclase desde el constructor de la subclase mediante el uso de **`super(saldo)`**.

Para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase se utiliza super.miembro, siendo miembro, un atributo o un método visible a la superclase.

Creación de una jerarquía multinivel

Hasta ahora, sólo se han utilizado jerarquías de clases sencillas, formadas por una superclase y una subclase. Sin embargo, se pueden construir jerarquías que contengan tantos niveles de herencia como se desee.

Por ejemplo, dadas tres clases llamadas `CuentaBancaria`, `CuentaAhorro` y `CuentaDeposito`, `CuentaDeposito` es subclase de `CuentaAhorro`, `CuentaAhorro` es subclase de `CuentaBancaria`, tal y como se muestra a continuación.



Cuando se produce este tipo de situaciones, cada subclase hereda todas las características encontradas en todas sus superclases. En este caso, `CuentaDeposito` hereda todas las propiedades de `CuentaAhorro` y `CuentaBancaria`.

```
package fp2.poo.practica7;

/**
 * CuentaDeposito es una cuenta que por defecto si se deposita
 *  mas de 1000 se incrementa el 5% de lo ingresado.
 * Tanto el umbral a partir del cual se aplica el interes, como
 * el interes son configurables en el constructor de la clase.
 */
public class CuentaDeposito extends CuentaAhorro {
    private double minimoAIngresar;
    private double interesIngreso;

    public CuentaDeposito (){
        super( 0.d, .02d );
        this.minimoAIngresar = 1000.0;
        this.interesIngreso  = 5.0;
    }

    public CuentaDeposito (double  minimoAIngresar ){
        super( 0.d, .02d );
        this.minimoAIngresar = minimoAIngresar ;
        this.interesIngreso  = 5.0;
    }

    public CuentaDeposito (double  minimoAIngresar , double interesIngreso  ){
        super( 0.d, .02d );
        this.minimoAIngresar = minimoAIngresar ;
        this.interesIngreso  = interesIngreso  ;
    }

    public CuentaDeposito (double  minimoAIngresar , double interesIngreso  ){
        this.minimoAIngresar = minimoAIngresar ;
        this.interesIngreso  = interesIngreso  ;
    }

    public void depositar(double cantidad){
        if(cantidad >= this.minimoAIngresar){
            setSaldo( getSaldo() + cantidad + cantidad*this.interesIngreso/100);
        } else {
            super.depositar(cantidad);
        }
    }
}
```

CuentaDeposito.java

```
package fp2.poo.practica7;

public class Practica7Ejercicio04 {
    public static void main (String args[]){
      CuentaDeposito cuenta = new CuentaDeposito ();
      cuenta.depositar(10000.);
      System.out.println("Valor de saldo de la Cuenta Deposito "+cuenta.getSaldo() );
    }
}
```

Practica7Ejercicio04.java

Para probar este ejemplo ejecute:

make –f makeP7 prueba04

Gracias a la herencia, **`CuentaDeposito`** puede hacer uso de las clases **`CuentaAhorro`** y **`CuentaBancaria`**, definidas previamente, y añadir únicamente la información extra que necesita para su propio y específico uso. Ésta es una de las ventajas de la herencia: permite la reutilización de código.

Este ejemplo muestra otro aspecto importante: **`super()`** siempre hace referencia al constructor de la superclase más próxima. Dentro de **`CuentaDeposito`**, **super()** llama al constructor de **`CuentaAhorro`**. Dentro de **`CuentaAhorro`**, **`super()`** llama al constructor de **`CuentaBancaria`**.

Orden de ejecución de los constructores

Cuando se crea una jerarquía de clases, ¿en qué orden se ejecutan los constructores de cada una de las clases que constituyen la jerarquía?

La respuesta es que en una jerarquía de clases, **los constructores se ejecutan en orden de derivación** , desde la superclase a la subclase.

Además, como **`super()`** tiene que ser la primera sentencia que se ejecute dentro de constructor de la subclase, este orden es el mismo tanto si se utiliza **`super()`** como si no. Si no se utiliza **`super()`**, entonces se ejecuta el constructor por defecto o sin parámetros de cada superclase.

Sobreescritura de un método

En una jerarquía de clases, cuando un método de una subclase tiene el mismo nombre y tipo que un método de su superclase, entonces se dice que el método de la subclase **sobrescribe** al método de la superclase. Cuando se llama a un método sobrescrito dentro de una subclase, siempre se refiere a la versión del método definida por la subclase. La versión del método definida por la superclase está oculta. Si se desea acceder a la versión de la superclase de una función sobrescrita, se puede hacer utilizando **`super`**. La sobrescritura de métodos se produce **únicamente** cuando los nombres y tipos de los dos métodos son idénticos. Si no ocurre esto, el método está sobrecargado.

En el ejemplo anterior el método **`depositar`** de la clase **`CuentaDeposito`**, sobrescribe el método `depositar` de la clase **`CuentaBancaria`**, para llamar al método **`depositar`**, de **`CuentaBancaria`** se ha utilizado **`super`**.

Selección de método dinámica

La selección de método dinámica es el mecanismo mediante el cual una llamada a una función sobrescrita se **resuelve en tiempo de ejecución**, en lugar de durante la compilación. La selección de método dinámica es importante ya que es la forma que tiene Java de implementar el polimorfismo durante la ejecución.

Recordemos que **una variable de referencia de la superclase puede referirse a un objeto de la subclase.** Java utiliza esto para resolver durante la ejecución llamadas a métodos sobrescritos. Cuando un método sobrescrito se llama a través de una referencia de la superclase, Java determina la versión del método que debe ejecutar en función del tipo del objeto que está siendo referenciado en el momento en el que se produce la llamada. Esta decisión se realiza en tiempo de ejecución. Dependiendo del tipo de objeto referenciado se ejecutará una u otra versión de un método sobrescrito.

En otras palabras, es el **tipo del objeto que está siendo referenciado**, y no el tipo de la variable referencia, el que determina qué versión de un método sobrescrito será ejecutada. Así, si una superclase contiene un método que está sobrescrito en la subclase, entonces cuando son referenciados distintos tipos de objetos a través de una variable referencia de la superclase, se ejecutan distintas versiones del método.

A continuación se muestra un ejemplo que ilustra la selección de método dinámica.

```
package fp2.poo.practica7;

public class Practica7Ejercicio05 {
    public static void main (String args[]){
        /*
         * Referencia a la superclase
         */
        CuentaBancaria referenciaACuenta = null;
        /*
         * Objeto del tipo CuentaDeposito (subclase)
         */
        CuentaDeposito cuentaDeposito = new CuentaDeposito();
        /*
         * Objeto del tipo CuentaBancaria (superclase)
         */
        CuentaBancaria cuentaBancaria = new CuentaBancaria ();

        System.out.println("depositar en CuentaBancaria");
        referenciaACuenta= cuentaBancaria;
        referenciaACuenta.depositar(5000.);
        System.out.println(referenciaACuenta.getSaldo());

        System.out.println();
        System.out.println("depositar en CuentaDeposito ");
        referenciaACuenta= cuentaDeposito;
        referenciaACuenta.depositar(5000.);
        System.out.println(referenciaACuenta.getSaldo());

    }
}
```

Practica7Ejercicio05.java

Para probar este ejemplo ejecute:

```
make –f makeP7 prueba05
```

En este ejemplo se utiliza una referencia a la superclase (`CuentaBancaria`) para referirse a un objeto en primer lugar de la superclase (`CuentaBancaria`), y en segundo lugar para referirse a un objeto de la subclase (`CuentaDeposito`). En ambos la invocación al método es la misma (`referenciaACuenta.depositar(5000.)`). Sin embargo el resultado es diferente en cada llamada.

El tipo del objeto es el que determina qué método es invocado en la llamada (`referenciaACuenta.depositar(5000.)`). Cuando el tipo del objeto es `CuentaBancaria` se invoca al método `depositar` de `CuentaBancaria`. Cuando el tipo del objeto es `CuentaDeposito` se invoca al método `depositar` de `CuentaDeposito`.

Miembros static

Hay ocasiones en las que se necesita definir un miembro de una clase que será utilizado independientemente de cualquier objeto de esa clase. Normalmente, a un miembro de una clase se accede a través de la referencia a un objeto. Sin embargo, es posible crear un miembro que pueda ser utilizado por sí mismo sin referirse a una instancia específica. Para crear un miembro de ese tipo es necesario preceder su declaración con la palabra clave **`static`**.

Cuando se declara un miembro como **`static`**, se puede acceder a él antes de que se haya creado ningún objeto de esa clase, y sin hacer referencia a ningún objeto. Se pueden declarar **`static`** tanto los métodos como las variables. El ejemplo más común de un miembro **`static`** es **`main()`**. **`main()`**tiene que ser declarado como **`static`** ya que es llamado antes de que exista ningún objeto.

Las variables de instancia declaradas como **`static`** pueden ser accedidas mediante el nombre de la clase en vez de la referencia al objeto.

Cuando se declaran objetos de una clase, no se hace ninguna copia de las variables **`static`**. De hecho, todas las instancias de la clase comparten la misma variable **`static`**.

Clases abstractas

Una característica extremadamente útil de la programación orientada a objetos es cl concepto de **clase abstracta**. Utilizando clases abstractas podemos declarar clases que definen sólo parte de una implementación, dejando a las clases que heredan de la clase abstracta la tarea de proporcionar las implementaciones específicas de algunos métodos, o de todos.

Una clase **no abstracta** tiene implementados todos los métodos de las interfaces que implemente, y los métodos abstractos de las clases abstractas de las que herede, implementados en dicha clase no abstracta o en alguna de sus superclases. Además puede añadir nuevos métodos no abstractos.

Las clases abstractas definen una superclase que declara la estructura de la clase dada sin proporcionar la implementación completa de todos los métodos. Y por tanto, definen una superclase de forma generalizada que será compartida por todas las subclases, dejando a cada subclase la tarea de completar los detalles de la implementación.

Las clases abstractas son útiles cuando parte del comportamiento está definido para la mayoría o todos los objetos de un tipo dado, pero algo del comportamiento sólo tiene sentido para la clase particular y no para una superclase general. Una clase así se declara como **`abstract`**, y los métodos no implementados en esa clase se marcan también como **`abstract.`**

```
abstract tipo nombre(ListaDeParametros);
```

Como se puede observar, este método no tiene cuerpo (implementación).

Cualquier clase que contenga uno o varios métodos abstractos también se tiene que declarar como **`abstract`**. Para declarar una clase abstracta, simplemente se utiliza la palabra clave **`abstract`** al principio de la declaración de la clase.

No se pueden crear instancias de clases abstractas con el operador **`new`**. No se pueden declarar constructores **`abstract`** o métodos **`abstract`** estáticos. Cualquier subclase de una clase abstracta debe implementar todos los métodos abstractos de la superclase o ser declarada también como **`abstract`**.

Las clases abstractas pueden añadir tantos atributos o métodos como sea necesario. Aunque no se pueden crear instancias de las clases abstractas, éstas pueden ser utilizadas para crear referencias a objetos, ya que la aproximación de Java al polimorfismo dinámico se implementa utilizando referencias de la superclase. Es posible crear una referencia de una clase abstracta para que pueda ser utilizada como referencia a un objeto de una subclase.

Uso de final con la herencia

La palabra **`final`** tiene tres usos.

1. En primer lugar, se puede utilizar para crear el equivalente de una constante con nombre. Habitual se suelen utilizar las interfaces para este propósito. Por ejemplo,

```
package fp2.poo.practica7;


public interface Practica7Ejercicio06 {
    public final static int VALOR_MAXIMO = 100;
}

class Main{
    public static void main (String args[]){
        System.out.println("Valor de Practica7Ejercicio06.VALOR_MAXIMO :"
                           + Practica7Ejercicio06.VALOR_MAXIMO );
    }
}
```

`Practica7Ejercicio06.java`

2. El uso de **`final`** con la herencia tiene dos usos:

    A. Evitar la sobrescritura de método: Aunque la sobrescritura de métodos es una de las características más poderosas de Java, hay ocasiones en las que es conveniente evitar que esto ocurra. Cuando se desea que un método no pueda ser sobrescrito, es necesario utilizar la palabra clave **`final`** como modificador al principio de su declaración. Los métodos como **`final`** no pueden ser sobrescritos.

    Ejercicio: marque como **`final`** el metodo **`depositar (public final void depositar(double cantidad))`**, de la clase **`CuentaBancaria`** y ejecute de nuevo la prueba 4: Para probar este ejemplo ejecute:

    `make –f makeP7 prueba04`

    Obteniendo un error en compilación, ya que se intenta sobrescribir un método **`final`**.

    B. Uso de **`final`** para evitar la herencia. A veces se desea evitar que una clase pueda ser heredada. Para hacer esto, se utiliza la palabra clave **final** en la declaración de la clase. Cuando se declara una clase como **final**, implícitamente se están declarando todos sus métodos como **final**. Como se podría esperar, no es válido declarar una clase como **abstract** y **final**, ya que una clase abstracta es imcompleta por sí misma y necesita que sus subclases proporcionen las implementaciones completas.

Ejercicio a entregar:

Se proporciona la siguiente interfaz.

Figura.java

```
/**
 *  @(#)Figura.java
 *
 *  Fundamentos de Programacion II. GITT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *  Marzo-2016
 *  Modificado Abril 2018
 */


package fp2.poo.practica7.utilidades;

/**
 *  Descripción: interfaz para la gestión de figuras geométricas
 *
 *  @author Fundamentos de Programacion II. GITT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *  @since Marzo-2016
 *  @version 1.0
 *
 */
public interface Figura {

    /**
     * Devuelve el área de la figura como un float
     * @return area, tipo float
     */
   float getArea();
    /**
     * Devuelve el perímetro de la figura como un float
     * @return perimetro, tipo float
     */
   float getPerimetro();
    /**
     * Devuelve el color de la figura como un String
     * @return area, tipo String
     */
   String getColor();
} //Cierre de la interfaz
```

Se proporciona la siguiente clase abstracta

MiFigura.java

```
/*
 *  @(#)MiFigura.java
 *
 *  Fundamentos de Programacion II. GITT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *
 */


/**
 * Descripcion: Implementación de la interfaz Figura genérica, para métodos
  * comunes, en este caso solamente el método toString, para mostrar las
clases
 * como interesa.
 *
 * @version 1.0 Marzo 2016 (Modificado Abril 2018)
 * @author  Fundamentos de Programacion II
 */
package fp2.poo.practica7.utilidades;


public abstract class MiFigura implements Figura {

    /**
     * Sobreescritura del método toString, que facilita la depuración
     * y análisis del código.
     *
     * @return la figura como un String, tipo String
     * @see java.lang.Object#toString
     */
        public String toString(){
            int indicepunto=this.getClass().getName().lastIndexOf(".");
       return this.getClass().getName().substring(indicepunto+1)
                   + " de color " + this.getColor() + "\n";
        }

    /**
     * Devuelve el área de la figura como un float
     * @return area, tipo float
     */
    abstract float getArea();

    /**
     * Devuelve el perímetro de la figura como un float
     * @return perimetro, tipo float
     */
    abstract public float getPerimetro();

    /**
     * Devuelve el color de la figura como un String
     * @return area, tipo String
     */
    abstract public String getColor();
}
```

Complete la implementación de la clase **Circulo** que se muestra parcialmente a continuación.

Circulo.java

```
package fp2.poo.practica7.utilidades;
import java.awt.Color;
import java.util.*;

public class Circulo extends MiFigura{

   private float diametro;
   private float PI = 3.1416f;
   private Color color;

    // Resto del codigo
}
```

Complete la implementación de la clase **Cuadrado** que se muestra parcialmente a continuación.

Cuadrado.java

```
package fp2.poo.practica7.utilidades;
import java.awt.Color;

public class Cuadrado extends MiFigura {

    private float lado;
    private Color color;

    // Resto del codigo
}
```

Implemente la clase **Main** en el paquete **fp2.poo.practica7.XXX** (siendo XXX el uvus) para crear objetos del tipo **Circulo** y **Cuadrado**.

Comprima el código en un archivo de nombre el uvus del alumno y entréguelo en la Actividad de la práctica.