Programación Orientada a Objetos
Sesión 6: Herencia
Curso 2017-18

Félix Gómez Mármol, Santiago Paredes Moreno y Gregorio Martínez Pérez
{felixgm, chapu, gregorio}@um.es

Índice

I Teoría **1**

1. El concepto de herencia **1**

2. Herencia y constructores **2**

3. Sobrescritura de métodos en familias de clases **2**

4. La clase Object y sobrescritura de sus métodos **5**

5. Clases abstractas **6**

6. Herencia e interfaces **7**

7. Herencia de interfaces **8**

8. Uso de interfaces y de herencia **8**

II Prácticas **11**

9. Ejercicios sobre herencia **11**

Parte I
Teoría

1. El concepto de herencia

En Programación Orientada a Objetos, además de definir clases mediante la agregación de miembros de otros tipos de datos, también podemos definir subclases a partir de otras ya existentes mediante la **herencia**, esto es, heredando de éstas atributos y métodos, redefiniéndolos, o añadiendo otros nuevos.

La **subclase**, o **clase derivada**, representa en este sentido un subconjunto de los objetos modelados por la clase de la que hereda o **clase padre**. Los objetos de una clase derivada son, de hecho, objetos de la clase padre. Por lo tanto, pueden tener todos los miembros y métodos de la misma (aunque esto va a

depender de la visibilidad que apliquemos a cada atributo o método en concreto), y pueden asignarse a variables cuyo tipo sea un ancestro de ellas.

Una clase puede implementar múltiples interfaces, y también puede ser padre de múltiples subclases, pero sólo puede ser descendiente directa de una única clase, su clase padre o antecesor directo.

Para definir una subclase o clase derivada en Java usamos la palabra reservada `extends` seguida del nombre de la clase de la que se quiere heredar. En el listado 1 definimos dos subclases por herencia: `Circulo` y `Rectangulo`. Con la definición que estamos haciendo por herencia, ambas tendrán los atributos `color`, `posicionX` y `posicionY` y los métodos `get()` y `set()` asociados en la clase padre (`Figura`). Además, la clase `Circulo` tiene un atributo extra y la clase `Rectangulo` dos, así como los métodos `get()` y `set()` para acceder a cada uno de ellos.

Reseñar que los métodos y miembros privados sólo son accesibles en la clase donde están definidos. El resto de clases, incluidas las clases derivadas o hijas, no pueden acceder a ellos. Para conseguir que un miembro o método sea privado para cualquier clase excepto para las derivadas podemos usar el modificador de visibilidad especial denominado `protected`. Como regla general declararemos todos los miembros como privados excepto los que queramos que puedan ser utilizados por las subclases que los declararemos como `protected`.

Siendo así, en el ejemplo anterior si queremos que los atributos `color`, `posicionX` y `posicionY` estén accesibles a las clases hijas, esto es, a las clases `Circulo` y `Rectangulo`, debemos definirlos como `protected` (tal y como podemos ver en el listado 1) y no como `private`.

2. Herencia y constructores

Los métodos constructores no se heredan debido a la función tan específica que tienen. Una clase derivada puede, y de hecho suele, tener más miembros que su clase padre, y al crear un objeto de ésta hay que inicializarlos todos, los heredados y los propios. Sin embargo, el constructor de la clase padre sólo hace parte del trabajo necesario. Por esta razón no se hereda el constructor, pero sí que puede ser interesante poder reutilizarlo en algunas ocasiones para no repetir el código de inicialización de la clase padre en la derivada.

Java incluye un mecanismo para poder reutilizar el constructor de la clase padre evitando así duplicar código, y cometer errores. Se trata de la palabra reservada `super()` que nos permite invocar al constructor de la clase padre e incluso pasarle argumentos. Sólo es posible realizar una invocación dentro de cada constructor y debe estar en la primera línea del código del mismo.

En el listado 1 vemos como el constructor de `Figura` recibe el color de la figura, así como su posición en el eje X y en el eje Y como parámetros. El constructor de la clase `Circulo` tiene un parámetro adicional y el de la clase `Rectangulo` dos parámetros adicionales para recibir información propia de los objetos que modelan. En la primera línea de cada uno se invoca al constructor de la clase padre para que sea él quien dé valor a los miembros `color`, `posicionX` y `posicionY`. Después, se almacena el valor recibido como cuarto parámetro en el caso de la clase `Circulo` (que representa su radio) y los recibidos como cuarto y quinto parámetros en el caso de la clase `Rectangulo` (que representan la base y la altura, respectivamente) en los miembros propios de cada subclase.

Al menos uno de los constructores de una clase derivada (normalmente el que sea más genérico, ya que los demás se pueden invocar por sobrecarga de constructores con `this()`) debe contener una llamada a un constructor de su clase padre. Dicha llamada deberá ser siempre la primera instrucción del código de un constructor de la clase derivada y se hará utilizando `super(parámetros)`. Si no se pone nada, Java considera que hay una llamada sin parámetros de tipo `super()`. Esto asume que la clase padre tendrá un constructor sin parámetros. Si todos los constructores de la clase padre tuvieran parámetros, entonces se generará un error. Por lo tanto, para evitar errores, si la clase padre no tiene un constructor sin parámetros, se deberá invocar explícitamente algún constructor válido de la clase padre en al menos uno de los constructores de la clase derivada utilizando la invocación `super(parámetros)`.

3. Sobrescritura de métodos en familias de clases

Cuando trabajamos con una familia de clases existen acciones comunes a todas que suelen estar definidas en la clase padre. Sin embargo, en muchos casos la forma concreta en la que se desarrollan estas

```
1  public class Figura {
2     protected String color;
3     protected int posicionX;
4     protected int posicionY;

6     public Figura(String color, int posicionX, int posicionY) {
7        this.color = color;
8        this.posicionX = posicionX;
9        this.posicionY = posicionY;
10    }

12    public String getColor() {return this.color;}
13    public void setColor(String color) {this.color = color;}
14    public int getPosicionX() {return this.posicionX;}
15    public void setPosicionX(int posicionX) {this.posicionX = posicionX;}
16    public int getPosicionY() {return this.posicionY;}
17    public void setPosicionY(int posicionY) {this.posicionY = posicionY;}

19    public String posicionFigura() {
20       return ("La posicion de la figura es (" + this.getPosicionX() + "," +
             this.getPosicionY() + ")");
21    }
22 }

24 public class Circulo extends Figura {
25    private double radio;

27    public Circulo(String color, int posicionX, int posicionY, double radio) {
28       super(color, posicionX, posicionY);
29       this.radio = radio;
30    }

32    public double getRadio() {return this.radio;}
33    public void setRadio(double radio) {this.radio = radio;}
34 }

36 public class Rectangulo extends Figura {
37    private double base;
38    private double altura;

40    public Rectangulo(String color, int posicionX, int posicionY, double base, double
          altura) {
41       super(color, posicionX, posicionY);
42       this.base = base;
43       this.altura = altura;
44    }

46    public double getBase() {return this.base;}
47    public void setBase(double base) {this.base = base;}
48    public double getAltura() {return this.altura;}
49    public void setAltura(double altura) {this.altura = altura;}
50 }

52 public class Main {
53    public static void main(String[] args) {
54       Circulo circulo = new Circulo ("Blanco", 100, 100, 5.0);
55       Rectangulo rectangulo = new Rectangulo ("Gris", 50, 50, 10.0, 15.0);

57       System.out.println (circulo.posicionFigura());
58       System.out.println (rectangulo.posicionFigura());
59    }
60 }
```

Listado 1: Definición de subclases `Circulo` y `Rectangulo` por herencia de la clase padre `Figura`

acciones en cada subclase es distinta. Es decir, comparten el interface (el qué) pero no la implementación (el cómo). Por ejemplo, todos los objetos de la clase `Figura` pueden devolver la posición en la que se

encuentran usando el método `posicionFigura()`. Sin embargo, podríamos estar interesados en que los objetos de la clase `Circulo` indicaran que esta posición es la de su centro, mientras que los objetos de la clase `Rectangulo` podrían indicar que esa posición es la de su esquina superior izquierda.

En Programación Orientada a Objetos es posible ajustar las acciones genéricas definidas en una clase padre al comportamiento concreto de cada clase derivada reescribiendo los cuerpos de los métodos heredados.

Al definir una subclase no necesitamos escribir los métodos heredados. Sin embargo, podemos hacerlo para ajustar su comportamiento al de la clase derivada. Para ello basta con volver a escribirlos explícitamente. A esta técnica la llamamos **sobrescritura**. La invocación de un método sobre un objeto implicará la ejecución del método más cercano a la clase a la que pertenezca dicho objeto. Si el método es heredado pero la subclase lo tiene sobrescrito se ejecutará éste; si no, el del padre.

```
1  public class Circulo extends Figura {
2     ...
3     public String posicionFigura() {
4        return ("El centro del circulo es (" + this.posicionX + "," + this.posicionY +
             ")");
5     }
6  }

8  public class Rectangulo extends Figura {
9     ...
10    public String posicionFigura() {
11       return ("La esquina superior izquierda del rectangulo es (" + this.posicionX +
             "," + this.posicionY + ")");
12    }
13 }

15 public class Main {
16    public static void main(String[] args) {
17       Circulo circulo = new Circulo ("Blanco", 100, 100, 5.0);
18       Rectangulo rectangulo = new Rectangulo ("Gris", 50, 50, 10.0, 15.0);

20       System.out.println (circulo.posicionFigura());
21       System.out.println (rectangulo.posicionFigura());
22    }
23 }
```

Listado 2: Sobrescritura completa de métodos

En el listado 2 hemos sobrescrito el método `posicionFigura()` en las clases `Circulo` y `Rectangulo`. En el caso de los objetos de la clase `Circulo` la posición hace referencia ahora a la del centro del objeto círculo con el que se invoca a este método. Por su parte, en el caso de los objetos de la clase `Rectangulo` la posición hace referencia ahora a la esquina superior izquierda del objeto rectángulo con la que se invoca a este método.

En la clase `Main` declaramos dos variables, `circulo` de tipo `Circulo` y `rectangulo` de tipo `Rectangulo`. A la primera le asignamos un nuevo objeto de la clase `Circulo` y a la segunda uno de tipo `Rectangulo`. El método `posicionFigura()` está definido en la clase `Figura`, por lo tanto podemos aplicarlo a las dos variables; sin embargo, en cada caso el efecto es distinto.

Cuando la variable `circulo` recibe el método `posicionFigura()` (línea 20), ejecuta la versión de dicho método definida en la clase a la que pertenece el objeto real referenciado por dicha variable, es decir, el redefinido en la clase `Circulo`. En el caso de la variable `rectangulo`, el método ejecutado en la línea 21 es, por la misma razón, el redefinido en la clase `Rectangulo`.

Aunque sobrescribamos los métodos en las subclases, algunas veces resulta útil, o necesario, utilizar la versión definida en la clase padre. En este sentido, indicar que cuando la versión redefinida del método hace lo mismo que la del padre más algunas otras cosas y no queremos, o no podemos, repetir todo el código, Java nos permite invocar a la versión de un método definida en la clase padre mediante la palabra reservada `super` seguida de un punto y el nombre del método.

Mientras que en los constructores sólo podemos utilizar `super` en la primera línea de código, la invocación al método de la clase padre con `super` podemos hacerla tantas veces como queramos y en cualquier parte del código del método sobrescrito. Un ejemplo de ello se puede comprobar en el

listado 3 donde el método `colorFigura()` se define en la clase padre `Figura` y se sobrescribe en las clases derivadas `Circulo` y `Rectangulo`. Sin embargo, en este caso se hace uso de la implementación del método `colorFigura()` de la clase padre para completarlo indicando el tipo de figura del que se trata, en el primer caso de un círculo y en el segundo de un rectángulo y luego recuperando el color como un `String` devuelto por el método `colorFigura()` de la clase padre `Figura` mediante la invocación `super.colorFigura()`.

```
1  public class Figura {
2     ...
3      public String colorFigura(){
4         return this.getColor();
5      }
6  }

8  public class Circulo extends Figura {
9     ...
10    public String colorFigura() {
11       return ("El color del circulo es " + super.colorFigura());
12    }
13 }

15 public class Rectangulo extends Figura {
16    ...
17    public String colorFigura() {
18       return ("El color del rectangulo es " + super.colorFigura());
19    }
20 }

22 public class Main {
23    public static void main(String[] args) {
24       Circulo circulo = new Circulo ("Blanco", 100, 100, 5.0);
25       Rectangulo rectangulo = new Rectangulo ("Gris", 50, 50, 10.0, 15.0);

27       System.out.println (circulo.colorFigura());
28       System.out.println (rectangulo.colorFigura());
29    }
30 }
```

Listado 3: Sobrescritura parcial de métodos

4. La clase Object y sobrescritura de sus métodos

Con la herencia podemos crear **jerarquías de clases o de tipos** donde las clases que comparten padre, o clases hermanas, tienen atributos y/o métodos en común. En una organización jerárquica de clases, las situadas en la base de la pirámide representan conjuntos de objetos muy concretos, mientras que las de las capas superiores representan conjuntos más abstractos, o familias de objetos.

Además de las jerarquías de clases definidas por el programador, Java incluye una amplia biblioteca de clases organizadas también de forma jerárquica. En el punto más alto de esta pirámide se sitúa la clase más general de todas que representa a cualquier objeto. Esta superclase o **raíz de la jerarquía de clases** en Java se llama `Object`.

Cualquier clase que no sea derivada de otra de forma explícita, será descendiente directa de `Object`. En caso contrario será descendiente indirecto de ella a través de algún ancestro. Por lo tanto, podemos aplicar a cualquier objeto los métodos definidos en `Object` y, gracias al polimorfismo que estudiamos en la sesión anterior, una variable de la clase `Object` puede referenciar objetos de cualquier clase.

Veamos a continuación el comportamiento normal de algunos de éstos métodos y qué podemos conseguir sobrescribiéndolos.

- `int hashCode()` devuelve un valor único que identifica al objeto, normalmente, su dirección de memoria. Puede ser interesante sobrescribirlo si queremos que objetos distintos tengan el mismo identificador si tienen un contenido idéntico aunque, realmente, se refieran a objetos distintos en la memoria.

- `boolean equals(Object o)` devuelve `true` si el objeto pasado como parámetro es el mismo al que se refiere el objeto sobre el cual se invocó el método. Puede ser interesante sobrescribirlo para que objetos distintos se consideren el mismo si su contenido (no su referencia) es equivalente.
- `String toString()` devuelve una cadena formada por el nombre de la clase a la que pertenece el objeto seguida del caracter '@' y un código que representa al objeto. Este método normalmente se sobrescribe para que muestre una descripción textual del objeto. De hecho, el método `println` de `System.out` invoca a éste método cuando recibe un objeto como parámetro.

```
1     public String toString(){
2         return ("Objeto de la clase Circulo de color " + this.getColor()+ ", con centro
              en el punto (" + this.getPosicionX() + "," + this.getPosicionY() + ") y
              radio " + this.radio);
3     }
```

Listado 4: Sobrescritura de `toString()` para la clase `Circulo`

El listado 4 muestra cómo sobrescribir el método `toString()` en la clase `Circulo`. De esta manera cuando se ejecute, por ejemplo, `System.out.println(circulo);` aparecerá `Objeto de la clase Circulo de color Blanco, con centro en el punto (100,100) y radio 5.0` por pantalla.

5. Clases abstractas

Los interfaces representan clases abstractas ya que definen el comportamiento genérico de una familia de clases sin especificar los detalles de implementación concretos en cada una. Por otro lado hemos visto que al crear subclases podemos sobrescribir los métodos de la clase padre para ajustar el comportamiento de ciertos métodos comunes a las peculiaridades de cada subclase. Pero para poder hacer esto el método debe existir en la clase padre.

Sin embargo, hay acciones que debemos definir en una clase para asegurarnos de que también están en todas sus clases derivadas, pero para las cuales no podemos dar una implementación concreta en la clase padre. Por ejemplo, en nuestra jerarquía de clases que representa figuras resulta evidente que el método para calcular el área estará presente en todas las subclases, pero no podemos calcular el área de una figura en abstracto.

Al igual que hacemos con los interfaces, en Java podemos crear clases con métodos parcialmente definidos, es decir, sin cuerpo. Los métodos no implementados se denominan **métodos abstractos** y pueden devolver cualquier tipo y tener cualquier número de parámetros, pero sólo pueden ser públicos o protegidos. Una **clase abstracta** es aquella en la que hay uno o más métodos abstractos.

Para declarar un método abstracto anteponemos la palabra reservada `abstract` a su definición y no escribimos su código. Cualquier clase que contenga un método abstracto está incompleta, con lo que o lo indicamos de manera expresa anteponiendo la palabra reservada `abstract` a su definición, o se considerará un error.

Destacar que, al igual como no podemos crear objetos de una interfaz de las estudiadas en la sesión anterior, tampoco podemos crear un objeto (con el operador `new`) de una clase abstracta ya que, como acabamos de indicar, es una clase que está incompleta.

En el listado 5 hemos añadido a la clase `Figura` el método público `area()`. Éste es heredado por todas las subclases con lo que cualquier objeto de la clase `Circulo` o de la clase `Rectangulo` podrá calcular su área usando dicho método. Esto es equivalente a tener un interface denominado `AreaDeUnaFigura` cuyo único método fuera `area()` y forzar a que las clases `Circulo` y `Rectangulo` implementen dicho interface.

Al igual que pasa al implementar un interface, las subclases de clases abstractas se ven obligadas a implementar los métodos abstractos definidos por su padre. De no hacerlo se considerarán incompletas a no ser que las declaremos también abstractas añadiendo la palabra `abstract` a su definición entre las palabras `public` y `class`.

En nuestro ejemplo, cada subclase implementa de una forma distinta el método `area()`, esto es, haciendo una definición en base a la propia expresión de cálculo del área para esa figura concreta.

```
public abstract class Figura {
   ...
   public abstract double area();
}

public class Circulo extends Figura {
   ...
   public double area() {
      return (Math.PI * this.radio * this.radio);
   }
}

public class Rectangulo extends Figura {
   ...
   public double area() {
      return (this.base*this.altura);
   }
}
```

Listado 5: Ejemplo de clases abstractas

Podemos ver las clases abstractas como prototipos o plantillas de lo que deben ser sus clases derivadas. Las clases abstractas están incompletas y, por lo tanto, como hemos indicado antes, no podemos crear objetos de los tipos de datos que definen. Sin embargo, gracias al polimorfismo que estudiamos en la sesión anterior podemos declarar variables y parámetros cuyo tipo sea cualquier clase abstracta. A estas variables y parámetros podremos asignarles objetos de cualesquiera de las clases derivadas. Un ejemplo de ello lo podemos ver en el listado 6 donde el array `arrayFiguras` se declara del tipo abstracto `Figura`, y luego los objetos en sí que hay en cada posición del array son de tipo `Circulo` los dos primeros y `Cuadrado` los dos últimos. El polimorfimo y su uso con herencia nos permite pasar en el bucle `for` por todas las posiciones del array `arrayFiguras` y para cada una de ellas que se invoque al método `area()` particular dependiendo del tipo de objeto (`Circulo` o `Cuadrado`) del que se trate.

```
public static void main(String[] args) {
   Figura [] arrayFiguras = new Figura[4];

   arrayFiguras[0] = new Circulo ("Blanco", 3, 7, 4.5);
   arrayFiguras[1] = new Circulo ("Gris", 2, 3, 3.1);
   arrayFiguras[2] = new Cuadrado ("Azul", 10, 20, 10.2);
   arrayFiguras[3] = new Cuadrado ("Negro", 0, 0, 5.0);

   for (int i=0; i<arrayFiguras.length; i++){
      System.out.println(arrayFiguras[i].area());
   }
}
```

Listado 6: Ejemplo de polimorfismo haciendo uso de clases abstractas

Por su parte, indicar que los interfaces también definen clases abstractas y tampoco podemos crear objetos de los tipos que definen. La diferencia entre una clase abstracta y un interface es que la primera puede tener algunos métodos y miembros concretos, mientras que los interfaces definen clases abstractas *puras*, es decir, sin ningún método concreto y sin ningún miembro.

6. Herencia e interfaces

Las subclases heredan todos los métodos públicos de la clase padre, incluidos los métodos abstractos que no están definidos. De este modo, si una clase implementa un interface, las clases derivadas también lo tendrán. Si la clase padre no define el cuerpo de alguno de los métodos incluidos en alguno de los interfaces implementados se tiene que declarar como abstracta. De este modo dejaremos que sean las clases derivadas las encargadas de hacerlo.

Por otro lado, las clases derivadas pueden sobrescribir los métodos implementados por la clase padre que correspondan a cualesquiera de los interfaces que ésta implemente según lo indicado en la cláusula `implements` de la que se haga uso en la definición de la clase.

```
interface AreaDeUnaFigura {
   double area();
}

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

Listado 7: Ejemplo de clase que implementa un interface en Java convirtiéndose en abstracta

Si, como muestra el listado 7, hacemos que la clase `Figura` implemente el interfaz `AreaDeUnaFigura`, pero no diéramos cuerpo al método `area()`, las clases derivadas `Circulo` y `Rectangulo` ya no tendrían que indicar explícitamente que implementan este interface, pero estarán obligadas a incluir dicho método `area()` como público y darle una implementación.

Cuando una clase implementa un interface se ve obligada a darle una implementación concreta a los métodos abstractos definidos en el interface implementado. Sin embargo, una clase abstracta no tiene por qué implementar todos sus métodos. De este modo, cuando una clase implementa un interface, o bien da cuerpo a todos los métodos indicados en el interface o bien tendremos que declararla como abstracta.

7. Herencia de interfaces

Al igual que podemos definir clases nuevas a partir de otras existentes mediante la herencia, también podemos crear nuevos interfaces heredando de otros. Además, a diferencia de lo ocurrido con las clases, donde sólo se puede heredar de una única clase, no existe limitación en cuanto al número de interfaces de los que podemos heredar.

```
public interface MedidasDeUnaFigura extends AreaDeUnaFigura {
   double perimetro();
}

public abstract class Figura implements MedidasDeUnaFigura {
   public abstract double area();
   public abstract double perimetro();
}
```

Listado 8: Ejemplo de interface que hereda de otro

En el listado 8 vemos un nuevo interface denominado `MedidasDeUnaFigura` que hereda del interface `AreaDeUnaFigura`; por lo tanto, incluye todos los métodos y constantes definidos en éste (en este caso en concreto el método `double area()`) más los nuevos métodos definidos en el propio interface (en nuestro ejemplo, el método `double perimetro()`). Dado que la clase `Figura` implementa este último interface debe dar cuerpo o dejar como abstracta (y que sean las clases derivadas las que den cuerpo) a cada uno de los métodos incluidos en `MedidasDeUnaFigura` y, por extensión, en `AreaDeUnaFigura`.

8. Uso de interfaces y de herencia

En general, aplicaremos el siguiente principio a la hora de plantearnos cuándo usar interfaces frente a la opción que nos plantea la herencia: cuando necesitemos clases abstractas puras, cuando queramos heredar de múltiples conjuntos de comportamientos, y/o cuando no queramos forzar la herencia pero necesitemos asegurar la existencia de ciertos comportamientos en alguna clase, recurriremos a la definición y uso de interfaces.

La herencia, por su parte, la utilizaremos cuando queramos dejar constancia en una clase padre de un conjunto de atributos que deben compartir todas las clases derivadas de las que hagamos uso y/o cuando queramos que alguno de los métodos en la clase padre tenga una implementación concreta que luego se podría refinar, si ello fuera necesario, en las clases derivadas.

En el listado 9 se puede ver el contenido de todas las clases desarrolladas en esta sesión incluyendo el uso de interfaces y de herencia. También se incluye la definición de un array de objetos de la clase abstracta `Figura` y la invocación a los métodos `area()` y `perimetro()` sobre los objetos de ese array que son, en realidad, objetos de clases derivadas (`Circulo` y `Cuadrado`) de la clase padre `Figura`.

```
interface AreaDeUnaFigura {
   double area();
}

interface MedidasDeUnaFigura extends AreaDeUnaFigura {
   double perimetro();
}

public abstract class Figura implements MedidasDeUnaFigura {
   protected String color;
   protected int posicionX;
   protected int posicionY;

   public Figura(String color, int posicionX, int posicionY) {
      this.color = color;
      this.posicionX = posicionX;
      this.posicionY = posicionY;
   }

   public String getColor() {return this.color;}
   public void setColor(String color) {this.color = color;}
   public int getPosicionX() {return this.posicionX;}
   public void setPosicionX(int posicionX) {this.posicionX = posicionX;}
   public int getPosicionY() {return this.posicionY;}
   public void setPosicionY(int posicionY) {this.posicionY = posicionY;}

   public String posicionFigura() {
      return ("La posicion de la figura es (" + this.posicionX + "," + this.posicionY +
          ")");
   }

   public abstract double area();
   public abstract double perimetro();
}

public class Circulo extends Figura {
   private double radio;

   public Circulo(String color, int posicionX, int posicionY, double radio) {
      super(color, posicionX, posicionY);
      this.radio = radio;
   }

   public double getRadio() {return this.radio;}
   public void setRadio(double radio) {this.radio = radio;}

   public String posicionFigura() {
      return ("El centro del circulo es (" + this.getPosicionX() + "," +
          this.getPosicionY() + ")");
   }

   public String toString() {
      return ("Objeto de la clase Circulo de color " + this.getColor() + ", con centro
          en el punto (" + this.getPosicionX() + "," + this.getPosicionY() + ") y radio
          " + this.radio);
   }

   public double area() {
      return (3.14 * this.radio * this.radio);
   }

   public double perimetro() {
      return (2 * 3.14 * this.radio);
   }
}

public class Rectangulo extends Figura {
   private double base;
   private double altura;
```

```
67      public Rectangulo(String color, int posicionX, int posicionY, double base, double
            altura) {
68          super(color, posicionX, posicionY);
69          this.base = base;
70          this.altura = altura;
71      }

73      public double getBase() {return this.base;}
74      public void setBase(double base) {this.base = base;}
75      public double getAltura() {return this.altura;}
76      public void setAltura(double altura) {this.altura = altura;}

78      public String posicionFigura() {
79          return ("La esquina superior izquierda del rectangulo es (" + this.getPosicionX()
                + "," + this.getPosicionY() + ")");
80      }

82      public String toString() {
83          return ("Objeto de la clase Rectangulo de color " + this.getColor() + ", con
                esquina superior en el punto (" + this.getPosicionX() + "," +
                this.getPosicionY() + "), base " + this.base + " y altura " + this.altura);
84      }

86      public double area() {
87          return (this.base * this.altura);
88      }

90      public double perimetro() {
91          return (2 * this.base * 2 * this.altura);
92      }
93  }

95  public class Main {
96      public static void main(String[] args) {
97          Circulo circulo = new Circulo("Blanco", 100, 100, 5.0);
98          Rectangulo rectangulo = new Rectangulo("Gris", 50, 50, 10.0, 15.0);

100         System.out.println(circulo.posicionFigura());
101         System.out.println(rectangulo.posicionFigura());

103         System.out.println(circulo);
104         System.out.println(rectangulo);

106         System.out.println("El area del circulo es: " + circulo.area() + " y su
                perimetro: " + circulo.perimetro());
107         System.out.println("El area del rectangulo es: " + rectangulo.area() + " y su
                perimetro: " + rectangulo.perimetro());

109            Figura [] arrayFiguras = new Figura[2];

111         arrayFiguras[0] = circulo;
112            arrayFiguras[0] = rectangulo;

114         for (int i=0; i<arrayFiguras.length; i++){
115               System.out.print("Para la figura " + i);
116            System.out.println(" su area es: " + arrayFiguras[i].area());
117            System.out.println(" y su perimetro es: " + arrayFiguras[i].perimetro());
118         }
119     }
120 }
```

Listado 9: Ejemplo completo de interfaces y herencia

Parte II
Prácticas

9. Ejercicios sobre herencia

En esta sesión de prácticas, para la que debes haber leído y entendido en detalle la parte teórica de esta sesión, vamos a trabajar sobre el concepto de herencia y las implicaciones que tiene sobre otros aspectos ya estudiados en la asignatura como la visibilidad de los atributos y los métodos, los constructores, las interfaces o la sobrecarga, entre otros.

EJERCICIO 1. *Crea un nuevo proyecto para esta sesión 6 y copia todas las clases del proyecto de la sesión 5. Crea ahora una clase* `Persona` *que actúe como clase padre de las clases* `Contacto` *y* `Usuario` *que definimos en sesiones anteriores. Observa todos los atributos y métodos que sean comunes y ponlos en la nueva clase* `Persona` *y deja los que sean específicos en las clases derivadas o hijas* `Contacto` *y* `Usuario`*. Recuerda aplicar el tipo de visibilidad que sea más apropiado para cada tipo de atributo y método de la clase padre y de las clases derivadas o hijas. De igual manera, incluye la definición de* `super()` *cuando sea necesario para invocar a los constructores de la clase padre desde los constructores de las clases derivadas o hijas.*

EJERCICIO 2. *En la clase* `Fecha` *sobrescribe los métodos* `equals()` *y* `toString()` *de la clase* `Object`*, de la cual hereda directamente, de manera que se pueda determinar cuándo dos objetos de la clase representan la misma fecha (o no), así como su representación como cadenas de texto. Para hacer esto último te puedes valer de la implementación del método* `devolverFechaComoCadena()` *ya existente. Haz lo mismo en la clase* `Hora`*.*

EJERCICIO 3. *Borra el método* `main()` *actual y crea tres objetos de tipo Hora, dos de ellos iguales y el tercero no e invoca al método* `equals()` *entre el primero y el segundo y entre el primero y el tercero y muestra el resultado de la comparación por pantalla. De igual manera, invoca el método* `println()` *sobre el objeto que representa la pantalla (*`System.out`*) para cada uno de estos tres objetos directamente, esto es, haciendo* `System.out.println(hora1);`*, por ejemplo.*

EJERCICIO 4. *Crea una jerarquía de mensajes en base a una clase* `Mensaje` *general y abstracta que sea la raíz de la jerarquía, que implemente el interfaz* `MensajeAbstracto` *y que tenga el método abstracto* `buscarMensaje()` *que permita realizar una búsqueda dentro de un mensaje. En dicha clase se debe rescribir el método* `toString()` *para mostrar la información contenida en el mensaje.*

EJERCICIO 5. *Crea, como clases derivadas o hijas de la clase Mensaje definida en el ejercicio anterior, las clases* `MensajeConTexto` *(que es la clase* `Mensaje` *que teníamos hasta ahora y que debemos renombrar) y* `MensajeConFichero`*, que representa un mensaje con fichero adjunto que puede ser una nota de audio, un vídeo o una fotografía. De esta última clase* `MensajeConFichero` *heredarán las clases* `MensajeConAudio` *(que es la antigua clase* `MensajeDeAudio` *que debemos renombrar),* `MensajeConVideo` *y* `MensajeConFotografia`*. Para cada fichero guardaremos su nombre y una descripción de texto de lo que contiene ese fichero (por ejemplo, "fichero de audio con un aviso para hacer una reunión hoy a las 15:00 horas"). Invoca a los constructores de las clase padre cuando convenga. De igual manera, realiza tu propio ajuste entre los atributos y los métodos de las clases para indicar los que estarán en las clases padre y los que estarán en las clases hija, presentado especial atención a la visibilidad, e invocando al método de la clase padre siempre que sea necesario (por ejemplo, al implementar la sobrecarga de* `toString()`*). De igual manera, determina en qué clases el método* `buscarMensaje()` *debe ser abstracto y en qué clases debe tener una implementación final. Cambia igualmente las clases que antes hicieran referencia a* `Mensaje` *por la nueva clase con el nuevo constructor.*

EJERCICIO 6. *Añade al método* `main()` *un array de ocho objetos de la clase abstracta* `Mensaje` *y crea dicho array de manera conveniente. Crea los objetos asociados y guárdalos en el array, siendo dos de ellos de la clase* `MensajeConTexto`*, dos de ellos de la clase* `MensajeConAudio`*, otros dos de la clase* `MensajeConVideo` *y los dos últimos de la clase* `MensajeConFotografia`*.*

EJERCICIO 7. *Añade un bucle al método* `main()` *que permita recorrer los 8 objetos creados en el ejercicio anterior y permita mostrar el contenido de cada uno de ellos. Añade después otro bucle que permita*

invocar al método `buscarMensaje()` *sobre cada uno de los 8 objetos del ejercicio anterior. Para el mensaje de texto debe buscar una cadena contenida en el mismo y para los objetos de algunas de las clases que heredan de la clase* `MensajeConFichero` *debe buscar una cadena que forme parte del nombre del fichero del que estamos haciendo uso o en la descripción de texto que hemos asociado con cada fichero.*

EJERCICIO 8. *Representa en papel los objetos que aparecerán en la memoria del ordenador al ejecutar el método* `main()` *implementado en el apartado anterior sobre todo en relación con la herencia de los atributos entre las distintas clases de la jerarquía creada. Presta especial atención a las referencias existentes entre dichos objetos y al posible aliasing entre ellos, así como al efecto del polimorfismo. Se recomienda hacer uso del depurador de NetBeans para entender bien la evolución de los valores de los atributos existentes en cada uno de los objetos creados.*