Tema 6: Herencia y Polimorfismo

6.1. Herencia.-

Se puede construir una clase a partir de otra mediante el mecanismo de la herencia. Para indicar que una clase deriva de otra se utiliza la palabra **extends**, como por ejemplo:

```
class CirculoGrafico extends Circulo {...}
```

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser redefinidas (overridden) en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la sub-clase (la clase derivada) "contuviera" un objeto de la super-clase; en realidad lo "amplía" con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de Java creadas por el programador tienen una super-clase. Cuando no se indica explícitamente una super-clase con la palabra extends, la clase deriva de **java.lang.Object,** que es la clase raíz de toda la jerarquía de clases de Java. Como consecuencia, todas las clases tienen algunos métodos que han heredado de **Object**.

La composición (el que una clase contenga un objeto de otra clase como variable miembro) se diferencia de la herencia en que incorpora los datos del objeto miembro, pero no sus métodos o interface (si dicha variable miembro se hace private).

6.2 La clase Object.-

Como ya se ha dicho, la clase Object es la raíz de toda la jerarquía de clases de Java. Todas las clases de Java derivan de Object. La clase Object tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase. Entre ellos se pueden citar los siguientes:

Métodos que pueden ser redefinidos por el programador:

- **clone()** Crea un objeto a partir de otro objeto de la misma clase. El método original heredado de Object lanza una CloneNotSupportedException. Si se desea poder clonar una clase hay que implementar la interface Cloneable y redefinir el método clone(). Este método debe hacer una copia miembro a miembro del objeto original. No debería llamar al operador new ni a los constructores.

- **equals()** Indica si dos objetos son o no iguales. Devuelve true si son iguales, tanto si son referencias al mismo objeto como si son objetos distintos con iguales valores de las variables miembro.

- **toString()** Devuelve un String que contiene una representación del objeto como cadena de caracteres, por ejemplo para imprimirlo o exportarlo.

Métodos que no pueden ser redefinidos (son métodos final):

- **getClass()** Devuelve un objeto de la clase Class, al cual se le pueden aplicar métodos para determinar el nombre de la clase, su super-clase, las interfaces implementadas, etc. Se puede crear un objeto de la misma clase que otro sin saber de qué clase es.

6.3 Redefinición de métodos heredados.-

Una clase puede redefinir (volver a definir) cualquiera de los métodos heredados de su super-clase que no sean final. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Las métodos de la super-clase que han sido redefinidos pueden ser todavía accedidos por medio de la palabra super desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Los métodos redefinidos pueden ampliar los derechos de acceso de la super-clase (por ejemplo ser public, en vez de protected o package), pero nunca restringirlos.

Los métodos de clase o static no pueden ser redefinidos en las clases derivadas.

6.4 Clases y métodos abstractos.-

Una clase abstracta (abstract) es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra **abstract**, como por ejemplo,

```
public abstract class Geometria { ... }
```

Una clase abstract puede tener métodos declarados como abstract, en cuyo caso no se da definición del método. Si una clase tiene algún método abstract es obligatorio que la clase sea abstract. En cualquier sub-clase este método deberá bien ser redefinido, bien volver a declararse como abstract (el método y la sub-clase).

Una clase abstract puede tener métodos que no son abstract. Aunque no se puedan crear objetos de esta clase, sus sub-clases heredarán el método completamente a punto para ser utilizado.

Como los métodos static no pueden ser redefinidos, un método abstract no puede ser static.

6.5 Constructores en clases derivadas.-

Ya se comentó que un constructor de una clase puede llamar por medio de la palabra this a otro constructor previamente definido en la misma clase. En este contexto, la palabra this sólo puede aparecer en la primera sentencia de un constructor.

De forma análoga el constructor de una clase derivada puede llamar al constructor de su super-clase por medio de la palabra **super()**, seguida entre paréntesis de los argumentos apropiados para uno de los constructores de la super-clase. De esta forma, un constructor sólo tiene que inicializar directamente las variables no heredadas.

La llamada al constructor de la super-clase debe ser la primera sentencia del constructor, excepto si se llama a otro constructor de la misma clase con this(). Si el programador no la incluye, Java incluye automáticamente una llamada al constructor por defecto de la super-clase, super().

Esta llamada en cadena a los constructores de las super-clases llega hasta el origen de la jerarquía de clases, esto es al constructor de Object.

Como ya se ha dicho, si el programador no prepara un constructor por defecto, el compilador crea uno, inicializando las variables de los tipos primitivos a sus valores por defecto, y los Strings y demás referencias a objetos a null. Antes, incluirá una llamada al constructor de la super-clase.

En el proceso de finalización o de liberación de recursos (diferentes de la memoria reservada con new, de la que se encarga el garbage collector), es importante llamar a los finalizadores de las distintas clases, normalmente en orden inverso al de llamada de los constructores. Esto hace que el finalizador de la sub-clase deba realizar todas sus tareas primero y luego llamar al finalizador de la super-clase en la forma super.finalize(). Los métodos finalize() deben ser al menos protected, ya que el método finalize() de Object lo es, y no está permitido reducir los permisos de acceso en la herencia.

6.6 Interfaces.-

Una interface es un conjunto de declaraciones de métodos (sin definición). También puede definir constantes, que son implícitamente public, static y final, y deben siempre inicializarse en la declaración. Estos métodos definen un tipo de conducta. Todas las clases que implementan una determinada interface están obligadas a proporcionar una definición de los métodos de la interface, y en ese sentido adquieren una conducta o modo de funcionamiento.

Una clase puede implementar una o varias interfaces. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las interfaces, separados por comas, detrás de la palabra **implements**, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la super-clase en el caso de herencia. Por ejemplo,

```
public class CirculoGrafico extends Circulo
implements Dibujable, Cloneable {
...
}
```

¿Qué diferencia hay entre una interface y una clase abstract? Ambas tienen en común que pueden contener varias declaraciones de métodos (la clase abstract puede además definirlos). A pesar de esta semejanza, que hace que en algunas ocasiones se pueda sustituir una por otra, existen también algunas diferencias importantes:

- Una clase no puede heredar de dos clases abstract, pero sí puede heredar de una clase abstract e implementar una interface, o bien implementar dos o más interfaces.

- Una clase no puede heredar métodos -definidos- de una interface, aunque sí constantes.

- Las interfaces permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo comportamiento, inpendientemente de su situación en la jerarquía de clases de Java.

- Las interfaces permiten “publicar” el comportamiento de una clase desvelando un mínimo de información.

- Las interfaces tienen una jerarquía propia, independiente y más flexible que la de las clases, ya que tienen permitida la herencia múltiple.

De cara al polimorfismo, las referencias de un tipo interface se pueden utilizar de modo similar a las clases abstract.

Una interface se define de un modo muy similar a las clases. A modo de ejemplo se reproduce aquí la definición de la interface Dibujable:

```
// fichero Dibujable.java
import java.awt.Graphics;
public interface Dibujable {
public void setPosicion(double x, double y);
public void dibujar(Graphics dw);
}
```

Cada interface public debe ser definida en un fichero *.java con el mismo nombre de la interface.

Los nombres de las interfaces suelen comenzar también con mayúscula.

Las interfaces no admiten más que los modificadores de acceso public y package. Si la interface no es public no será accesible desde fuera del package (tendrá la accesibilidad por defecto, que es package). Los métodos declarados en una interface son siempre public y abstract, de modo implícito

6.7 Herencia en interfaces.-

Entre las interfaces existe una jerarquía (independiente de la de las clases) que permite herencia simple y múltiple. Cuando una interface deriva de otra, incluye todas sus constantes y declaraciones de métodos.

Una interface puede derivar de varias interfaces. Para la herencia de interfaces se utiliza asimismo la palabra extends, seguida por el nombre de las interfaces de las que deriva, separadas por comas.

Una interface puede ocultar una constante definida en una super-interface definiendo otra constante con el mismo nombre. De la misma forma puede ocultar, re-declarándolo de nuevo, la declaración de un método heredado de una super-interface.

Las interfaces no deberían ser modificadas más que en caso de extrema necesidad. Si se modifican, por ejemplo añadiendo alguna nueva declaración de un método, las clases que hayan implementado dicha interface dejarán de funcionar, a menos que implementen el nuevo método.

6.8 Utilización de interfaces.-

Las constantes definidas en una interface se pueden utilizar en cualquier clase (aunque no implemente la interface) precediéndolas del nombre de la interface, como por ejemplo (suponiendo que PI hubiera sido definida en Dibujable):

```
area = 2.0*Dibujable.PI*r;
```

Sin embargo, en las clases que implementan la interface las constantes se pueden utilizar directamente, como si fueran constantes de la clase. A veces se crean interfaces para agrupar

constantes simbólicas relacionadas.

De cara al polimorfismo, el nombre de una interface se puede utilizar como un nuevo tipo de referencia. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la interface.

Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

6.9 Transformaciones de tipo (Casting).-

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de int a double, o de float a long. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia. En este apartado se explican brevemente estas transformaciones de tipo.

La conversión entre tipos primitivos es más sencilla. En Java se realizan de modo automático conversiones implícitas de un tipo a otro de más precisión, por ejemplo de int a long, de float a double, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto (más amplio) que el resultado de evaluar el miembro derecho.

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son conversiones inseguras que pueden dar lugar a errores (por ejemplo, para pasar a short un número almacenado como int, hay que estar seguro de que puede ser representado con el número de cifras binarias de short). A estas conversiones explícitas de tipo se les llama **cast**.

El cast se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

6.10 Polimorfismo.-

El polimorfismo tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama vinculación (binding). La vinculación puede ser temprana (en tiempo de compilación) o tardía (en tiempo de ejecución). Con funciones normales o sobrecargadas se utiliza vinculación temprana (es posible y es lo más eficiente). Con funciones redefinidas en Java se utiliza siempre vinculación tardía, excepto si el método es final. El polimorfismo es la opción por defecto en Java.

La vinculación tardía hace posible que, con un método declarado en una clase base (o en una interface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea el tipo de objeto y no el tipo de la referencia lo que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el polimorfismo necesita evaluación tardía.

El polimorfismo permite a los programadores separar las cosas que cambian de las que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los

programas.

El polimorfismo puede hacerse con referencias de super-clases abstract, super-clases normales e interfaces. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las interfaces permiten ampliar muchísimo las posibilidades del polimorfismo.

6.11 Conversión de objetos.-

El polimorfismo visto previamente está basado en utilizar referencias de un tipo más "amplio" que los objetos a los que apuntan. Las ventajas del polimorfismo son evidentes, pero hay una importante limitación: el tipo de la referencia (clase abstracta, clase base o interface) limita los métodos que se pueden utilizar y las variables miembro a las que se pueden acceder. Por ejemplo, un objeto puede tener una referencia cuyo tipo sea una interface, aunque sólo en el caso en que su clase o una de sus super-clases implemente dicha interface. Un objeto cuya referencia es un tipo interface sólo puede utilizar los métodos definidos en dicha interface. Dicho de otro modo, ese objeto no puede utilizar las variables y los métodos propios de su clase. De esta forma las referencias de tipo interface definen, limitan y unifican la forma de utilizarse de objetos pertenecientes a clases muy distintas (que implementan dicha interface).

Si se desea utilizar todos los métodos y acceder a todas las variables que la clase de un objeto permite, hay que utilizar un cast explícito, que convierta su referencia más general en la del tipo específico del objeto. De aquí una parte importante del interés del cast entre objetos (más bien entre referencias, habría que decir).

Para la conversión entre objetos de distintas clases, Java exige que dichas clases estén relacionadas por herencia (una deberá ser sub-clase de la otra). Se realiza una conversión implícita o automática de una sub-clase a una super-clase siempre que se necesite, ya que el objeto de la sub-clase siempre tiene toda la información necesaria para ser utilizado en lugar de un objeto de la super-clase. No importa que la super-clase no sea capaz de contener toda la información de la subclase.

La conversión en sentido contrario -utilizar un objeto de una super-clase donde se espera encontrar uno de la sub-clase- debe hacerse de modo explícito y puede producir errores por falta de información o de métodos. Si falta información, se obtiene una ClassCastException.

No se puede acceder a las variables exclusivas de la sub-clase a través de una referencia de la super-clase. Sólo se pueden utilizar los métodos definidos en la super-clase, aunque la definición utilizada para dichos métodos sea la de la sub-clase.

Por ejemplo, supóngase que se crea un objeto de una sub-clase B y se referencia con un nombre de una super-clase A,

```
A a = new B();
```

en este caso el objeto creado dispone de más información de la que la referencia a le permite acceder (podría ser, por ejemplo, una nueva variable miembro j declarada en B). Para acceder a esta información adicional hay que hacer un cast explícito en la forma (B)a. Para imprimir esa variable j habría que escribir (los paréntesis son necesarios):

```
System.out.println( ((B)a).j );
```

Un cast de un objeto a la super-clase puede permitir utilizar variables -no métodos- de la super-

clase, aunque estén redefinidos en la sub-clase.

Considérese el siguiente ejemplo: La clase C deriva de B y B deriva de A. Las tres definen una variable x. En este caso, si desde el código de lasub-clase C se utiliza:

```
x // se accede a la x de C
this.x // se accede a la x de C
super.x // se accede a la x de B. Sólo se puede subir un nivel
((B)this).x // se accede a la x de B
((A)this).x // se accede a la x de A
```

EJERCICIOS RESUELTOS.-

- *Escribir un programa que permita convertir horas en notación de 24 horas a su equivalente en notación de 12 horas. Por ejemplo, `23:50` debe ser convertida en `11:50 PM`.*

  En primer lugar intentamos crear la clase *Reloj* que representará en reloj en formato de 24 horas y tendrá los atributos y operaciones necesarios para mantener una hora en dicho formato. Su código es el siguiente:

```
// Reloj.java

class Reloj
{
      // Estado de la clase
      private int horas, minutos;

      // Metodos de la clase

      // Constructores
      public Reloj ()
      {
            horas = 0;
            minutos = 0;
      }

      public Reloj (int horas, int minutos)
      {
            this.setHoras (horas);
            this.setMinutos (minutos);
      }

      // Selectores
      public void setHoras (int horas)
      {
            this.horas = horas;
            this.comprobar ();
      }

      public void setMinutos (int minutos)
      {
```

```
            this.minutos = minutos;
            this.comprobar ();
      }

      public int getHoras ()
      {
            return this.horas;
      }

      public int getMinutos ()
      {
            return minutos;
      }

      // Metodos propios
      private void comprobar ()
      {
            while (this.minutos > 59)
            {
                  this.minutos -= 60;
                  this.horas++;
            }

            while (horas > 23)
                  horas -= 24;
      }
}// Reloj
```

A continuación basándonos en esta clase se creará la clase *Reloj12* para dar soporte a las horas en formato de 12. Su código es el siguiente:

```
// Reloj12.java

import Reloj;

class Reloj12 extends Reloj
{

      // Estado de la clase
      String indicador;

      // Metodos de la clase
```

```
// Constructores
Reloj12 ()
{
      super (); // llamada al constructor base
      indicador = new String ("AM");

}

Reloj12 (int horas, int minutos)
{
      super (horas, minutos);
      this.setIndicador ();
}

Reloj12 (int horas, int minutos, String indicador)
{
      super (horas,minutos);
      this.setIndicador (); // ajustar las horas a formato 12
      this.setIndicador (indicador);
}


// Selectores
public void setHoras (int horas)
{
      super.setHoras (horas);
      this.setIndicador ();
}

public void setMinutos (int minutos)
{
      super.setMinutos (minutos);
      this.setIndicador ();
}

private void setIndicador ()
{
      while (super.getMinutos () > 59)
      {
            super.setMinutos (super.getMinutos () - 60);//minutos -= 60
            super.setHoras (super.getHoras () +1 ); // horas++;
      }
```

```
            while ( super.getHoras () > 11)
            {
                  this.setIndicador ("PM");
                  super.setHoras (super.getHoras () - 12); // horas -= 12
            }
      }

      public void setIndicador (String indicador)
      {
            if (indicador.equals ("AM")|| indicador.equals ("am"))
                  this.indicador = indicador;
            else
                  if (indicador.equals ("PM") || indicador.equals ("pm"))
                        this.indicador = indicador ;
      }

      public String getIndicador ()
      {
            return indicador;
      }

} // Reloj12
```

Finalmente nos plantemos resolver el problema propuesto utilizando las clases anteriores. Para ello creamos la clase aplicación *HorasApp*, que tendrá un método *main ()* encargado de resolver el problema. Su código es el siguiente:

```
// HorasApp.java
import Reloj12;

class HorasApp
{

      public static void main (String args[])
      {
            int h,m;
            int horas,minutos;
            Reloj12 reloj ;

            if (args.length != 2) // comprobar el numero de parametros
```

```
            {
                  System.out.println ("Argumentos no validos");
            }
            else
            {
                  h= Integer.valueOf (args[0]).intValue(); // convertir a int
                  m = Integer.valueOf(args[1]).intValue (); // convertir a int

                  reloj = new Reloj12 (h, m); // crear el relor de 12 horas


                  // Sacar resultados
                  System.out.println ("La hora en formato 12 horas es:");
                  System.out.println (reloj.getHoras()+":"+reloj.getMinutos()
                                          + " " + reloj.getIndicador ());
            }
      }
}// HorasApp
```