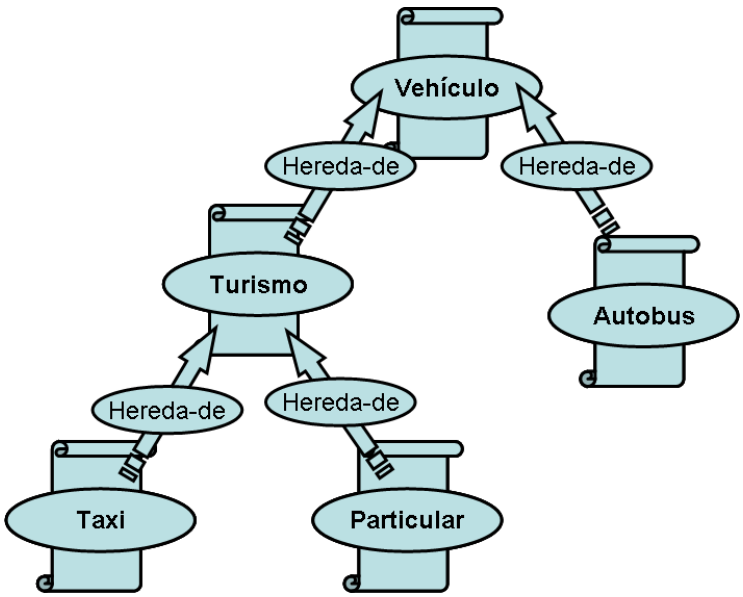
Tema 7

Definición de clases: Herencia, polimorfismo, ligadura dinámica

Con alguna frecuencia es necesario definir clases de objetos entre las cuales hay elementos comunes. En una aplicación en la cual intervengan taxis, turismos particulares, autobuses, aviones, etc, seguramente encontraremos que todos ellos comparten la capacidad de transportar pasajeros, de conocer su posición, etc. Ahora bien, quizás no sea adecuado agrupar a todos ellos en una única categoría, viéndolos como vehículos. Al margen de los aspectos comunes que puedan existir entre todos ellos, algunos aspectos de interés en un taxi podrían no tener sentido referidos a un vehículo particular; por ejemplo, la tarifa a aplicar en un recorrido urbano. Algo parecido podríamos decir si imaginamos una aplicación en la cual intervengan profesores universitarios, estudiantes universitarios de grado, estudiantes universitarios de master, personal administrativo, etc. En tanto que personas, compartirán muchas características, pero también habrá funciones que sólo sean razonables para algunos de ellos. Para evitar la repetición de fragmentos de código en diferentes clases, en P.O.O. se permite establecer una relación entre clases de objetos y organizarlas jerárquicamente, como se muestra en la figura siguiente:



Esta relación entre clases se define explícitamente al escribir el texto de cada una de ellas. Cuando entre dos clases **A** y **B**, se da la relación **A hereda-de B**, todas las variables y métodos declarados en la segunda se consideran automáticamente parte también de la primera. De esta manera, los métodos y declaraciones de variables relacionados con las características comunes a taxis, turismos de uso particular, y autobuses, aparecerán solo en la clase `Vehiculo`. Análogamente, el código relacionado con las características comunes a taxis y turismos de uso particular, pero que no sean compartidas por todos los vehículos, estará en la clase `Turismo`. A continuación, veremos los aspectos más importantes de la relación *hereda-de* en *java*.

7.1. Definición de clases y herencia

Programa 28 Definición de la clase `Coche`

```
1  package ehu.student;
2
3  public class Coche
4  {
5      private String propietario;
6      private String matricula;
7      private double cuentaKilometros;
8
9      public void vender(String elPropietario) {
10         propietario = elPropietario;
11     }
12
13     public void matricular(String laMatricula) {
14         matricula = laMatricula;
15     }
16
17     public void recorrer(double kms){
18         cuentaKilometros = cuentaKilometros + kms;
19     }
20
21     public void printInfo(){
22         String tmp =
23             "Propietario: " + propietario + "; " +
24             "Matricula: " + matricula + "; " +
25             "Kms recorridos: " + cuentaKilometros + ";";
26         System.out.println(tmp);
27     }
28 }
```

Los objetos de la clase `Coche`, definida en el programa 28, representan coches con tienen funciones para cambiar el nombre del propietario, o el código de matrícula. Además, un coche puede recorrer la cantidad de kilómetros que se le indique, y proporcionar información diversa, como la distancia total recorrida desde que fué creado.

Para definir un nuevo tipo de coches, con algunas prestaciones adicionales, podríamos copiar en una nueva clase todas las variables y métodos declarados en el programa 28, pero

esa forma de proceder es muy pobre. Otra posibilidad, más interesante, es definir la nueva clase de manera que sea una **subclase** de la anterior; es decir, estableciendo la relación *hereda-de* con la clase `Coche`. En *java* esto se hace como se indica en el programa siguiente:

```
1  package ehu.student.herencia;
2
3  import ehu.student.Coche;
4
5  /**
6   * Una clase definida mediante herencia
7   *
8   * Un CocheConGPS es un Coche que conoce
9   * las coordenadas de su posicion: latitud y longitud
10  */
11 public class CocheConGPS extends Coche
12 {
13     private double latitud = 0;
14     private double longitud = 0;
15
16     public void cambiarCoordenadas(double deltaLatitud,
17                                    double deltaLongitud)
18     {
19         latitud = latitud+deltaLatitud;
20         longitud = longitud+deltaLongitud;
21     }
22
23     public double latitud(){
24         return latitud;
25     }
26
27     public double longitud(){
28         return longitud;
29     }
30
31     public void printInfoPosicion(){
32         String tmp = "Latitud: " + latitud + "; Longitud: " + longitud;
33         System.out.println(tmp);
34     }
35 }
```

El texto:

`CocheConGPS` **`extends`** `Coche`

que aparece en la línea 9 es lo que establece la relación *hereda-de* entre las clases `CocheConGPS` y `Coche`. La consecuencia de ello es que cada objeto `CocheConGPS` poseerá las variables de instancia declaradas en `Coche`, y se le podrán enviar los mismos mensajes que a cualquier instancia de `Coche`. Es como si en el programa 28 hubiésemos copiado las declaraciones de métodos y variables que aparecen en la clase `Coche`.

El programa 29 es un ejemplo sencillo de uso de la clase `CocheConGPS`: crea un objeto y le envía diferentes mensajes. Obsérvese que las líneas 16 a 18 son llamadas a métodos que

Programa 29 Uso de la clase `CocheConGPS`

```
1  package ehu.student.herencia;
2
3  /*
4   * Ejemplo de uso de una clase definida por herencia
5   */
6  public class DemoCocheConGPS {
7
8      public static void main(String[] args) {
9          CocheConGPS coche = new CocheConGPS();
10
11         /* metodos en CocheConGPS */
12         coche.cambiarCoordenadas(0.01, 0.02);
13         coche.printInfoPosicion();
14
15         /* metodos en Coche */
16         coche.vender("X.X.X");
17         coche.matricular("PMM-000");
18         coche.printInfo();
19     }
20 }
```

no están declarados en `CocheConGPS`, sino en `Coche`, pero, a pesar de eso, se escriben igual que si esos métodos hubiesen sido declarados en `CocheConGPS`.

En programación, se dice que una clase como `CocheConGPS` es una **subclase** o *clase derivada* de `Coche`, de la cual **hereda** todas sus variables y métodos. Asímismo, se dice que `Coche` es la **superclase** de la clase derivada. Gracias a este mecanismo de **herencia**, los objetos de una clase son capaces de comportarse como lo harían los objetos de la superclase correspondiente. De hecho, al definir una clase como subclase de otra, los métodos de la superclase pueden usarse también para definir sus métodos. Por ejemplo, podría ser más razonable definir el método `cambiarCoordenadas` de manera que se incremente también el cuentakilómetros de un coche:

```
private double longitud = 0;

public void cambiarCoordenadas(double deltaLatitud,
                               double deltaLongitud)
{
    /*
     * actualizar cuentaKilometros con la distancia recorrida
     * al cambiar de coordenadas.
     * Por simplicidad, el algoritmo a utilizar se sustituye por
     * lo siguiente, que solo es una buena aproximacion
     * en lo que se refiere a la latitud:
     * un grado de longitud en el ecuador es mas largo
     * que cerca de los Polos!
     */
    double distancia =
        Math.abs(latitud()-deltaLatitud) * 110 +   /* aproximado */
        Math.abs(longitud()-deltaLongitud) * 110;  /* incorrecto */
```

```
        /* llamada al metodo de la superclase */
        recorrer(distancia);

        latitud = latitud+deltaLatitud;
```

7.1.1. Redefinición de métodos: overriding

Cuando se define una clase como subclase de otra puede ser interesante cambiar el comportamiento asociado con algunos de los métodos de la superclase. Por ejemplo, para definir un nuevo tipo de coches, representando taxis, parece conveniente definir una nueva clase por herencia a partir de la clase `Coche`:

class Taxi **extends** Coche {...

Ahora bien, quizás no sea razonable que los taxis se comportan **exactamente** igual que los coches, en lo que al comportamiento heredado de la clase `Coche` se refiere. Por ejemplo, porque cuando un taxi hace un recorrido baja la bandera al empezar el recorrido y la vuelve a subir cuando el recorrido se termina. Esto sugiere que la clase `Taxi` debiera definirse de manera que el método `recorrer` tenga en cuenta ese aspecto.

Programa 30 Métodos heredados: redefinición

```
1  package ehu.student.herencia;
2
3  import ehu.student.Coche;
4
5  /**
6   * Una clase definida mediante herencia
7   * con redefinicion de metodos heredados
8   */
9  public class Taxi extends Coche
10 {
11     /* heredado de Coche */
12     public void recorrer(double kms){
13         System.out.println("Taxi@: inicia carrera");
14         printInfo();
15
16         /* metodo "recorrer" heredado de Coche */
17         super.recorrer(kms);
18
19         System.out.println("Taxi@: fin de carrera");
20     }
21 }
```

El programa 30 muestra una definición de la clase `Taxi` conforme a lo dicho. Por ser una subclase de `Coche`, la clase `Taxi` hereda un método con la cabecera siguiente:

```
public void recorrer(double kms)
```

Por otra parte, el programa incluye también un método con esa misma cabecera; es decir, que coinciden el nombre y los parámetros declarados. En situaciones como ésta se dice que la clase `Taxi` `redefine` (*overrides*) el método `recorrer`. En consecuencia, cuando un objeto `Taxi` reciba el mensaje correspondiente, no se ejecutará el método heredado sino el incluido en la propia clase. Obsérvese, no obstante, que en la nueva versión de `recorrer` se usa también el método heredado, aunque la instrucción de llamada correspondiente tiene un formato algo peculiar:

super.`recorrer`**(** ... **);**

7.2. Polimorfismo

La relación *hereda-de* no es solamente un mecanismo para la definición de nuevas clases, sino que tiene también influye en la relación entre objetos y clases. Pensar, como hemos hecho hasta ahora, que cada uno de los objetos creados por un programa pertenece a una única clase, es una visión algo limitada de las cosas. En P.O.O.se considera que

todas las instancias de una clase son también instancias de su superclase

(en *java* todas las clases de una aplicación son subclases de alguna otra; los detalles se dejan para después).

Volviendo a la figura de la página 7, eso quiere decir que los taxis son, a todos los efectos, turismos, y que tanto taxis como autobuses son también vehículos. Una consecuencia de esto es que las variables de un programa pueden designar objetos de diferentes tipos durante la ejecución del programa. Esto ocurrirá, por ejemplo, al ejecutarse el programa siguiente:

```
1  package ehu.student.herencia;
2
3  import ehu.student.Coche;
4
5  /*
6   * Ejemplo de asignacion a la misma variable
7   * de objetos de diferentes clases.
8   */
9  public class EjemploVariablePolimorfica {
10
11     public static void main(String[] args) {
12         Coche coche = null;
13
14         /*
15          * A coche se le pueden asignar coches
16          * de diferentes tipos!!!
17          */
18         coche = new Coche();
19         coche = new CocheConGPS();
20         coche = new Taxi();
21
22         /* metodos en Coche */
23         coche.vender("X.X.X");
24         coche.matricular("PMM-000");
```

```
25          coche.printInfo();
26      }
27  }
```

Como se ve en ese programa, a una variable de tipo referencia como `coche` se le puede asignar, tanto un objeto de la clase indicada al declarar la variable (es decir, `Coche`), como de cualquiera de sus subclases:

`Taxi ... CocheConGPS`

Lo que ocurre, en definitiva, es que todos los objetos creados por ese programa son instancias de `Coche`: las instancias de `Taxi` y `CocheConGPS` son instancias de `Coche`, ya que ambas clases heredan de `Coche`.

Algo parecido ocurre también al definir el resultado de un método. En el programa siguiente, cada vez que una `FactoriaDeCoches` recibe el mensaje `fabricarCocheNuevo` devuelve un coche de nueva creación que unas veces es un `Coche`, otras veces es un `CocheConGPS`, y otras un `Taxi`. En cualquier caso, siempre es una instancia, directa o indirectamente, de `Coche`, conforme al tipo de resultado declarado por el método `fabricarCocheNuevo`.

```
1   package ehu.student.herencia;
2
3   import ehu.student.Coche;
4
5   /*
6    * Una clase para construir coches de diferentes tipos.
7    */
8   public class FactoriaDeCoches {
9
10      private int n = 0;
11
12      /**
13       * Devuelve un nuevo coche.
14       */
15      public Coche fabricarCocheNuevo()
16      {
17          Coche elCoche = null;
18
19          if (n == 0) {
20              elCoche = new Coche();
21          } else if (n == 1){
22              CocheConGPS cocheConGPS = new CocheConGPS();
23              elCoche = cocheConGPS;
24          } else {
25              Taxi taxi = new Taxi();
26              elCoche = taxi;
27          }
28          n = (n+1) % 3;
29          return elCoche;
30      }
31  }
```

7.2.1. Ligadura dinámica

Programa 31 Ligadura dinámica

```
1  package ehu.student.herencia;
2
3  import ehu.student.Coche;
4
5  /**
6   * Ejemplo de polimorfismo y ligadura dinamica.
7   */
8  public class EjemploPolimorfismo {
9
10     public static void main(String[] args) {
11         FactoriaDeCoches factoria = new FactoriaDeCoches();
12
13         for (int i = 0; i < 10; i++){
14             Coche c = factoria.fabricarCocheNuevo();
15             /* c puede ser de diferentes clases */
16
17             /* Qué método se ejecuta aquí? */
18             c.vender("X.X.X" + i);
19             c.matricular("PMM-" + i);
20
21             /* Qué método se ejecuta aquí? */
22             c.recorrer(i);
23         }
24         System.out.println("PE" == ("PE" + ""));
25     }
26 }
```

Aparentemente, el programa 31 es muy sencillo. Se crea un número de instancias de `Coche` y se usan las funciones que poseen para establecer quién es el propietario, cuál es el código de matrícula, etc. Seguramente, estará claro que los mensajes enviados en las líneas 18 y 19, dan lugar a la ejecución de los métodos `vender` y `matricular` de la clase `Coche`. Ahora bien, refiriéndonos a la asignación en la línea 14, hay que recordar que el tipo de coche creado por la `FactoriaDeCoches` es variable. En algunas iteraciones, el coche creado será una instancia de `Coche` o de `CocheConGPS`, mientras que en otras será una instancia de `Taxi`. En los dos primeros casos, el mensaje enviado en esa línea dará lugar a la ejecución del método `recorrer` de la clase `Coche`. Sin embargo, como la clase `Taxi` tiene redefinido el método `recorrer`, cuando el coche creado sea una instancia de `Taxi`, se ejecutará el método de dicha clase. Como puede verse, la repetición de una misma instrucción puede dar lugar a la ejecución de diferentes piezas de código en cada ocasión. Y solo en el momento mismo de ejecutar esa instrucción se decide cuál es el método elegido. En P.O.O., se usa el término **ligadura dinámica** o *dynamic binding* para referirse a ese mecanismo que pospone hasta el momento de ejecutar una llamada, la elección del método a ejecutar.

7.3. La clase `Object`

En *java* y otros lenguajes de programación, todas las clases de una aplicación son subclases de alguna otra. Cuando una clase no indica explícitamente cuál es su superclase, esa superclase es una clase primitiva de *java* llamada

`java.lang.Object`

Así pues, podemos definir una clase como se suele hacer muchas veces:

class Coche {...

o bien indicar explícitamente que la superclase es `Object`:

class Coche **extends** Object {...

7.3.1. Overriding y colecciones

La clase `Object` no es de mucha utilidad en sí misma. Sin embargo tiene métodos con las cabeceras siguientes:

public boolean `equals`**(**`Object o`**)**
public int `hashCode`**()**
public String `toString`**()**

y puesto que

todas las instancias de una clase son también instancias de `Object`

esos métodos son heredados también en cualquier otra clase que se defina.

Los dos primeros juegan un papel muy importante en el comportamiento de los diferentes tipos de colecciones existentes. Como su propio nombre indica, el método `equals` sirve para saber si un objeto es igual a otro, y podría usarse así:

```
Object uno = new Object();
Object otro = new Object();
boolean test = uno.equals(otro); // es false

Coche uno = new Coche();
Coche otro = new Coche();
boolean certeza = uno.equals(uno); // es true
boolean duda = uno.equals(otro); // es false; ¿sería mas razonable true?
```

Conforme a la implementación de `equals` en `Object`, un objeto únicamente es igual a sí mismo. Solamente en aquéllas clases en las cuales ese método esté adecuadamente redefinido se obtendrá un resultado más acorde con lo intuitivamente esperado. La importancia de ese método para el buen funcionamiento de los diferentes tipos de colecciones es obvia. Por ejemplo, una lista decide si contiene o no a un objeto dado, examinando si ese objeto es igual a alguno de sus elementos.

Por lo que se refiere al método `hashCode`, aunque es usado muy raramente, juega un papel muy importante en el rendimiento de algunos tipos de aplicaciones y conjuntos (aunque no se

ha mencionado, además de las clases `TreeSet<T>` y `TreeMap<K, V>`, hay otras más con funciones parecidas). Omitiendo detalles, una clase debe redefinir este método de tal manera que el código devuelto por dos objetos sea el mismo si el resultado del método **equals** es `true`.

El programa 32 define una subclase de `CartaBarajaInmutable` para corregir algunas deficiencias de esta clase. En particular, la nueva clase hará posible construir listas de cartas con un comportamiento más acorde con lo intuitivamente esperable. Por ejemplo, la ejecución de las instrucciones siguientes:

```
CartaBaraja unAsOros = new CartaBaraja(0, 0);
CartaBaraja otroAsOros = new CartaBaraja(0, 0);

ArrayList<CartaBaraja> lista = new ArrayList<CartaBaraja>();
lista.add(unAsOros);
lista.add(unAsCopas);

boolean estaAsOrosEnLista = lista.contains(otroAsOros);
```

se completará asignando el valor `true` a `estaAsOrosEnLista`. Para ello es necesario redefinir los métodos citados arriba. Obsérvese que en la definición de `equals` se examina si el objeto recibido como parámetro es una instancia de `CartaBaraja`. Caso de ser así, es necesario hacer una conversión de tipos, ya que si las líneas 35 y 36 se sustituyesen por algo parecido:

```
r = (getSuit() == o.getSuit()) &&
    (getFigure() == o.getFigure());
```

el texto sería rechazado por el compilador.

Programa 32 Redefiniendo `equals` y `hashCode`

```
1  package ehu.student.herencia;
2
3  import ehu.student.CartaBarajaInmutable;
4
5  /**
6   * Ejemplo de redefinicion de metodos heredados de Object
7   *
8   * Necesarios para el uso de colecciones.
9   */
10 public class CartaBaraja extends CartaBarajaInmutable
11 {
12     /**
13      * Construye un nuevo naipe.
14      */
15     public CartaBaraja(int cdgSuit, int cdgFigure)
16     {
17
18         /* ejecutar constructora de la superclase */
19         super(cdgSuit, cdgFigure);
20
21     }
22
23     /**
24      * Devuelve true si el naipe es igual al objeto o.
25      */
26     public boolean equals(Object o)
27     {
28         boolean r = false;
29         /* o puede ser de esta clase...*/
30         if (o instanceof CartaBaraja)
31             {   /* si lo es... */
32                 /* CASTING se asigna o a una variable de este tipo */
33                 CartaBaraja carta = (CartaBaraja)o;
34                 /* ahora ya se pueden usar los metodos de ese objeto */
35                 r = (getSuit() == carta.getSuit()) &&
36                     (getFigure() == carta.getFigure());
37             }
38         return r;
39     }
40
41     /**
42      * Devuelve un entero: el codigo "hash" del objeto
43      *
44      * El codigo hash de dos objetos iguales debe ser igual.
45      */
46     public int hashCode(){
47         int hc = (getSuit() * 4) + getFigure();
48         return hc;
49     }
50 }
```