Programación Orientada a Objetos

Herencia, Polimorfismo y Vinculación Dinámica

Clases Abstractas e Interfaces

1. Herencia, Polimorfismo y Vinculación Dinámica

1.1. Conceptos Básicos

- La **herencia** representa una relación en la cual una *clase derivada* (*subclase*) **es una** *especialización* o *extensión* de una *clase base* (*superclase*). La herencia permite definir *jerarquías* de clases.
  - La subclase hereda tanto los *atributos* (variables) como los *métodos* definidos por la superclase.
  - La subclase puede *añadir* nuevos atributos y nuevos métodos.
  - la subclase puede *redefinir* el comportamiento de los métodos de la superclase.
- El **polimorfismo** permite que un objeto de una subclase pueda ser considerado y utilizado como si fuese un objeto de la superclase (principio de sustitución).
- En un contexto de polimorfismo, donde las subclases redefinen el comportamiento de los métodos de la superclase, la **vinculación dinámica** permite que los métodos invocados se seleccionen adecuadamente, en tiempo de ejecución, dependiendo del tipo dinámico de la variable (el tipo del objeto referenciado), y no del tipo estático de la variable.

La relación de herencia permite representar tanto el concepto de especialización como de extensión, e incluso ambos conceptos simultáneamente:

- Es posible que la subclase sea una *especialización* más *concreta* de la superclase, que es más *general* y *abstracta*. Por ejemplo un *Coche de Ocasión* es un *Coche* con algunas características propias que lo distinguen (especializan) respecto del resto de coches en general.
- También es posible que la subclase sea una *extensión* de la superclase que añada funcionalidad adicional. Por ejemplo una *Partícula* es un *Punto* con masa, que adicionalmente puede calcular la atracción entre dos partículas.
- Finalmente, también es posible que la subclase sea tanto una *especialización* como una *extensión* de la superclase. Por ejemplo un *Médico* es una *Persona* con unas características propias que lo especializan en su comportamiento, y con la funcionalidad adicional de poder diagnosticar enfermedades a los pacientes, de tal forma que puede ser utilizado en diferentes contextos como una especialización de *Persona* o como un *Médico* con la funcionalidad extendida.

Aunque el mecanismo de herencia permite a las clases derivadas añadir nuevos métodos, este documento está enfocado hacia la explicación de los conceptos de polimorfismo y redefinición de comportamiento, por lo que en este documento no se utilizará dicha característica también asociada a la herencia.

1.2. Aplicación Práctica de los Conceptos de Herencia

En java, una clase derivada sólo puede heredar de **una única** clase base, y se utiliza la palabra reservada **extends** para especificar la superclase de la que se hereda. La subclase hereda tanto los atributos como los métodos de la superclase, de tal forma que un objeto de la clase base se encuentra

empotrado dentro del objeto de la clase derivada, es decir, forma parte del mismo objeto. En caso de que no se especifique explícitamente la superclase, entonces la clase hereda implícitamente de la clase **Object**, que es la clase base de toda la jerarquía de clases en java.

Clase `Coche`

Un *coche* puede ser definido especificando su *modelo* y su *precio base*. Por lo tanto, definimos la clase `Coche` para representar este concepto.

```
public class Coche {
    private String modelo;
    private double precioBase;
    public Coche(String m, double p) {
        modelo = m;
        precioBase = p;
    }
    public String getModelo() {
        return modelo;
    }
    public double calcPrecio() {
        return precioBase;
    }
    @Override
    public String toString() {
        return "(" + getModelo() + ", " + calcPrecio() + ")";
    }
}
```

Clase `CocheOcasion`

Un *Coche de Ocasión*, además de las características propias de un *Coche*, también viene definido por un determinado *Porcentaje de Descuento* que se aplica al calcular el precio del mismo. Por lo tanto, se puede considerar que un *Coche de Ocasión* también **es un** *Coche*, pero con algunas características diferentes, relacionadas con el cálculo del precio del mismo. Así, la clase `CocheOcasion` hereda de la clase `Coche`, pero redefine su comportamiento para calcular el precio del coche.

```
public class CocheOcasion extends Coche {  // extends especifica la relación de herencia
    private double porcDescuento;
    public CocheOcasion(String m, double p, double d) {
        super(m, p);                        // invocación al constructor de la superclase
        porcDescuento = d;
    }
    @Override
    public double calcPrecio() {
        double p = super.calcPrecio();      // invocación al método de la superclase que se está redefiniendo
        return p - p * porcDescuento / 100.0;
    }
}
```

Nótese como el constructor de la *clase derivada* (*subclase*) invoca al constructor de la *clase base* (*superclase*) mediante la palabra reservada **super** y los parámetros adecuados (esta invocación debe ser la primera línea del constructor). De esta forma, cuando se construya un objeto de la clase derivada, también se construye el objeto de la clase base que se encuentra *empotrado* en éste. Si no se especifica explícitamente la llamada al constructor de la superclase, entonces se invoca implícitamente al *constructor sin argumentos* de la superclase (si no existe, se notificará el error).

Así mismo, la clase `CocheOcasion` redefine el comportamiento del método `calcPrecio` de la clase base, de tal forma que ahora se aplica un porcentaje de descuento cuando se calcula el precio del coche de ocasión.

Un método, que redefine el comportamiento del método de la superclase, puede invocar al método que está redefiniendo, invocando al método directamente sobre el *objeto* **super**. Nótese como el metodo

`calcPrecio` de la clase `CocheOcasion` invoca al mismo metodo `calcPrecio` de la clase `Coche` a través del objeto **super**.

La anotación `@Override` indica al compilador que el método pretende *redefinir* el comportamiento de un método de la clase base, de tal forma que si el compilador no encuentra ese método en la clase base, entonces avisa del error. Esta anotación es importante, porque en caso de que cometamos errores en la redefinición del método (tal como especificar el nombre del método o los parámetros distintos) y no se utilice esta anotación, entonces en vez de redefinir el método, se crearía un método nuevo, y el compilador no nos avisaría del error.

Clase `CocheImportado`

Un *Coche Importado*, además de las características propias de un *Coche*, también viene definido por unos determinados *Gastos de Homologación* que se aplican al calcular el precio del mismo. Por lo tanto, se puede considerar que un *Coche Importado* también **es un** *Coche*, pero considerando que el coche importado debe añadir también los *Gastos de Homolgación* al precio del coche. Así, la clase `CocheImportado` hereda de la clase `Coche`, pero redefine su comportamiento para calcular el precio del coche.

```
public class CocheImportado extends Coche {
    private double costeHomologacion;
    public CocheImportado(String m, double p, double h) {
        super(m, p);
        costeHomologacion = h;
    }
    @Override
    public double calcPrecio() {
        return super.calcPrecio() + costeHomologacion;
    }
}
```

1.3. Aplicación Práctica de los Conceptos de Polimorfismo y Vinculación Dinámica

Una vez que tenemos definida nuestra jerarquía de clases con los diferentes tipos de coches, podemos definir una clase que represente un *concesionario*, de tal forma que pueda contener múltiples coches de diversas clases. Para ello, el **polimorfismo** y la **vinculación dinámica** que proporciona la relación de herencia es fundamental, ya que permiten que el concesionario pueda referenciar a coches de diversas clases (todas ellas derivadas de la clase `Coche`), y permiten la invocación adecuada a los métodos redefinidos.

Clase `ConcesionarioSimple`

En esta sección se define la clase `ConcesionarioSimple`, que representa un determinado concesionario de coches. La **composición**, a diferencia de la herencia, es un mecanismo por el cual un determinado objeto contiene y **está compuesto por** otros objetos. En este caso, un objeto de la clase concesionario está compuesto, entre otros, por múltiples objetos pertenecientes a la jerarquía definida por la clase base `Coche`.

```
import java.util.Arrays;
public class ConcesionarioSimple {
    private static final int CAPACIDAD_INICIAL = 8;
    private int nCoches;
    private Coche[] coches;
    public ConcesionarioSimple() {
        nCoches = 0;
        coches = new Coche[CAPACIDAD_INICIAL];
    }
    public void anyadir(Coche c) {                  // Polimorfismo
        if (nCoches == coches.length) {
            coches = Arrays.copyOf(coches, 2*coches.length);
        }
```

```
        coches[nCoches] = c;                        // Polimorfismo
        ++nCoches;
    }
    private Coche buscarCoche(String m) {           // Método privado
        Coche c = null;
        int i = 0;
        while ((i < nCoches)&&( ! m.equals(coches[i].getModelo()))) {
            ++i;
        }
        if (i < nCoches) {
            c = coches[i];
        }
        return c;
    }
    public double calcPrecioFinal(String m) {
        double p = 0;
        Coche c = this.buscarCoche(m);              // Invoca al método privado para buscar el coche
        if (c != null) {
            p = c.calcPrecio();                     // Vinculación dinámica
        }
        return p;
    }
    @Override
    public String toString() {
        // También se puede realizar de forma más eficiente utilizando la clase StringBuilder
        String str = "";
        if (nCoches > 0) {
            str += " " + coches[0].toString();      // Vinculación dinámica indirecta
            for (int i = 1; i < nCoches; ++i) {
                str += ", " + coches[i].toString(); // Vinculación dinámica indirecta
            }
        }
        return "[" + str + " ]";
    }
}
```

El **polimorfismo** permite que el array de `Coche` contenga referencias tanto a objetos de la clase `Coche` como de sus clases derivadas. Así mismo, también permite que el objeto recibido como parámetro en el método `anyadir(Coche)` sea tanto de la clase `Coche` como de cualquiera de sus clases derivadas.

Por otra parte, para que la redefinición del comportamiento y el principio de sustitución funcionen consistentemente en un contexto de polimorfismo, es necesario que la invocación a los métodos se realice a través de mecanismos de **vinculación dinámica**. Así, cuando el método `calcPrecioFinal(String)` de la clase `ConcesionarioSimple` invoca al método `calcPrecio()` de la clase `Coche` sobre un determinado objeto, el método que realmente se invoca no está determinado por el **tipo estático** de la variable (en este caso `Coche`), sino que está determinado por su **tipo dinámico** (el tipo con el que se creó el objeto referenciado por la variable), y depende de la ejecución del programa.

La implementación del método `toString()` de la clase `ConcesionarioSimple` muestra otro ejemplo en el que la vinculación dinámica es relevante. Este método invoca al método `toString()` sobre una variable polimórfica de tipo estático `Coche`, que referencia a un objeto de clase `Coche` o derivado, y éste método `toString()` de la clase `Coche`, a su vez, invoca al método `calcPrecio()` sobre el mismo objeto polimórfico, de tal forma que el método `calcPrecio()` realmente invocado dependerá del tipo dinámico del objeto, gracias a la vinculación dinámica, ya que las clases derivadas de `Coche` redefinen este método.

Adicionalmente, los métodos privados (**private**), tal como `buscarCoche(String)` ayudan a descomponer la solución de un determinado método en partes más simples, facilitando además la reutilización del código. Así mismo, los métodos privados sólo son accesibles desde el ámbito interno de la propia clase, por lo que no se altera la definición pública de la clase.

El siguiente programa principal permite comprobar la funcionalidad de las clases anteriores:

```
public class Main1 {
    public static void main(String[] args) {
        ConcesionarioSimple concesionario = new ConcesionarioSimple();
        concesionario.anyadir(new CocheOcasion("Seat-Marbella", 9000, 20));
        concesionario.anyadir(new Coche("Seat-Leon", 15000));
        concesionario.anyadir(new CocheImportado("Porsche-911", 39000, 1000));

        System.out.println("Precio Seat-Marbella: "+concesionario.calcPrecioFinal("Seat-Marbella"));
        System.out.println("Precio Seat-Leon: "+concesionario.calcPrecioFinal("Seat-Leon"));
        System.out.println("Precio Porsche-911: "+concesionario.calcPrecioFinal("Porsche-911"));

        System.out.println("Concesionario: "+concesionario.toString());
    }
}
```

La ejecución del programa anterior muestra la siguiente salida:

```
Precio Seat-Marbella: 7200.0
Precio Seat-Leon: 15000.0
Precio Porsche-911: 40000.0
Concesionario: [ (Seat-Marbella, 7200.0), (Seat-Leon, 15000.0), (Porsche-911, 40000.0) ]
```

En la que se puede apreciar como, gracias a la vinculación dinámica, el cálculo del precio final del *Seat-Marbella* se realiza según el método `calcPrecio()` redefinido por la clase `CocheOcasión`, el cálculo del precio final del *Seat-Leon* se realiza según el método `calcPrecio()` definido por la clase base `Coche`, y el cálculo del precio final del *Porsche-911* se realiza según el método `calcPrecio()` redefinido por la clase `Cocheimportado`.

Así mismo, al mostrar todos los coches del concesionario, podemos apreciar como el precio final de cada coche se ha calculado correctamente según la defición de la clase a la que pertenece cada coche.

Tipo Estático y Tipo Dinámico de una Variable

El **tipo estático** de una variable se establece en el momento de la declaración de la variable, y no cambia, ya que viene establecido por la declaración en el programa Java, por eso se denomina estático. Por ejemplo:

```
Coche coche;
```

El tipo de un objeto se establece en el momento de su creación, y es el tipo que se especifica en la creación del objeto con el operador **new**. Por ejemplo:

```
new CocheOcasion("Seat-Marbella", 9000, 20);
```

El **tipo dinámico** de una variable es el tipo del objeto (establecido en el momento de la creación del objeto) al que referencia la variable en un determinado momento. Se denomina *dinámico* porque puede cambiar durante la ejecución del programa, ya que durante la ejecución del programa, una misma variable puede referenciar a diferentes objetos en diferentes momentos. Por ejemplo:

```
(1) Coche coche = new CocheOcasion("Seat-Marbella", 9000, 20);
    // ...
(2) coche = new Coche("Seat-Leon", 15000);
    // ...
(3) coche = new CocheImportado("Porsche-911", 39000, 1000);
    // ...
```

En este ejemplo anterior, la clase `Coche` es el tipo estático de la variable `coche`, ya que es el tipo con el que se declaró la variable. El tipo estático de la variable no cambia , ya que está especificado en la declaración de la misma.

Sin embargo, durante la ejecución del programa, en el momento de ejecutar la sentencia (1), el tipo dinámico de la variable `coche` es la clase `CocheOcasion`, ya que es el tipo con el que se ha creado el objeto al que referencia la variable `coche` en ese preciso momento.

Del mismo modo, durante la ejecución del programa, en el momento de ejecutar la sentencia (2), el tipo dinámico de la variable `coche` es la clase `Coche`, ya que es el tipo con el que se ha creado el objeto al que referencia la variable `coche` en ese preciso momento.

Finalmente, durante la ejecución del programa, en el momento de ejecutar la sentencia (3), el tipo dinámico de la variable `coche` es la clase `CocheImportado`, ya que es el tipo con el que se ha creado el objeto al que referencia la variable `coche` en ese preciso momento.

1.4. Métodos Protegidos y Redefinición del Comportamiento

A continuación mostramos ejemplos de la posibilidad de redefinir el comportamiento de los métodos definidos dentro de la jerarquía de clases. Así mismo, se muestra la importancia de los métodos protegidos, ya que facilitan la redefinición del comportamiento y la reutilización del código por parte de las clases derivadas.

Clase `ConcesionarioIva`

Por simplicidad en la explicación, en este ejemplo, en lugar de utilizar la clase `ConcesionarioSimple`, definida en las secciones anteriores, definimos una nueva clase denominada `ConcesionarioIva`, en la que el concesionario calcula el precio final de los coches anadiéndoles el IVA correspondiente.

```
import java.util.Arrays;
public class ConcesionarioIva {
    private static final int CAPACIDAD_INICIAL = 8;
    private int nCoches;
    private Coche[] coches;
    private double porcIva;
    public ConcesionarioIva(double iva) {
        nCoches = 0;
        coches = new Coche[CAPACIDAD_INICIAL];
        porcIva = iva;
    }
    public void anyadir(Coche c) {
        if (nCoches == coches.length) {
            coches = Arrays.copyOf(coches, 2*coches.length);
        }
        coches[nCoches] = c;
        ++nCoches;
    }
    private Coche buscarCoche(String m) {           // Método privado
        Coche c = null;
        int i = 0;
        while ((i < nCoches)&&( ! m.equals(coches[i].getModelo()))) {
            ++i;
        }
        if (i < nCoches) {
            c = coches[i];
        }
        return c;
    }
    private double calcPrecioIva(Coche c) {     // Método privado
        double p = c.calcPrecio();              // Invoca directamente al método calcPrecio sobre un coche
        return p + p * porcIva / 100.0;
    }
    public double calcPrecioFinal(String m) {
        double p = 0;
        Coche c = this.buscarCoche(m);          // Invoca al método privado para buscar el coche
        if (c != null) {
            p = this.calcPrecioIva(c);          // Invoca al método privado para calcular el precio
        }
        return p;
    }
    private String cocheToString(Coche c) {     // Método privado
```

```
        return "(" + c.getModelo() + ", " + this.calcPrecioIva(c) + ")";  // Invoca al método privado
    }
    @Override
    public String toString() {
        // También se puede realizar de forma más eficiente utilizando la clase StringBuilder
        String str = "";
        if (nCoches > 0) {
            str += " " + this.cocheToString(coches[0]);       // Invoca al método privado
            for (int i = 1; i < nCoches; ++i) {
                str += ", " + this.cocheToString(coches[i]);  // Invoca al método privado
            }
        }
        return "Iva("+porcIva+"%) [" + str + " ]";
    }
}
```

Esta definición de la clase `ConcesionarioIva` incorpora dos nuevos métodos respecto a la clase `ConcesionarioSimple`. El método privado `calcPrecioIva(Coche)` permite calcular el precio de un coche añadiendo el IVA correspondiente, y el método privado `cocheToString(Coche)` permite obtener la representación textual de un coche, incluyendo el precio con IVA.

Nótese que el método privado `calcPrecioIva(Coche)` permite calcular el precio de un coche, y será utilizado dentro de la clase `ConcesionarioIva` en todas las situaciones donde sea necesario calcular dicho precio (`calcPrecioFinal(String)`, `cocheToString(Coche)` y `toString()`).

Como se mostró en secciones anteriores, los métodos privados facilitan la definición de los métodos de la clase, la reutilización del código, y restringen su ámbito de acceso a la propia clase de la que forma parte.

El siguiente programa principal permite comprobar la funcionalidad de las clases anteriores:

```
public class Main2 {
    public static void main(String[] args) {
        ConcesionarioIva concesionario = new ConcesionarioIva(10);
        concesionario.anyadir(new CocheOcasion("Seat-Marbella", 9000, 20));
        concesionario.anyadir(new Coche("Seat-Leon", 15000));
        concesionario.anyadir(new CocheImportado("Porsche-911", 39000, 1000));

        System.out.println("Concesionario: "+concesionario.toString());
    }
}
```

La ejecución del programa anterior muestra la siguiente salida:

```
Concesionario: Iva(10.0%) [ (Seat-Marbella, 7920.0), (Seat-Leon, 16500.0), (Porsche-911, 44000.0) ]
```

Clase `ConcesionarioOferta`

Tomando como clase base la clase `ConcesionarioIva` definida anteriormente, queremos diseñar una nueva clase `ConcesionarioOferta` que derive (herede) de la clase `ConcesionarioIva`, pero que redefina su comportamiento para que calcule el precio final de un determinado coche considerando aplicar un descuento para el *modelo en oferta*.

Debido a que el método `calcPrecioIva(Coche)` de la clase `ConcesionarioIva` fue definido con ámbito de acceso **privado**, las clases derivadas no pueden ni invocarlo ni redefinirlo. Sin embargo, para que la clase `ConcesionarioOferta` pueda redefinir el comportamiento de la clase `ConcesionarioIva` cuando se calcula el precio de un coche, es necesario que la clase `ConcesionarioOferta` pueda invocar y tambien redefinir el método `calcPrecioIva(Coche)` de la clase `ConcesionarioIva`. Por ello, el método privado `calcPrecioIva(Coche)` de la clase `ConcesionarioIva` debería haber sido definido como **protegido**, ya que un método protegido permite su invocación desde las clases derivadas, así como además también permite su redefinición.

```
public class ConcesionarioIva {
    /* variables y métodos programados en la sección anterior */
```

```
    protected double calcPrecioIva(Coche c) {
        // Este método protegido permite a las clases derivadas tanto
        // invocar como redefinir el cálculo del precio con IVA del coche
        double p = c.calcPrecio();
        return p + p * porcIva / 100.0;
    }
}
```

Una vez que la clase `ConcesionarioIva` ha sido definida adecuadamente, se puede definir la clase derivada `ConcesionarioOferta` que redefina el comportamiento, cuando se calcule el precio del coche, para que permita aplicar un descuento a un determinado modelo.

```
public class ConcesionarioOferta extends ConcesionarioIva {
    private double valorDescuento;
    private String modeloOferta;
    public ConcesionarioOferta(double iva, double d, String m) {
        super(iva);                          // invocación al constructor de la clase base (superclase)
        valorDescuento = d;                  // almacena el valor del descuento
        modeloOferta = m;                    // almacena el valor del modelo en oferta
    }
    @Override
    protected double calcPrecioIva(Coche c) {       // método redefinido
        double p = super.calcPrecioIva(c); // invocación al método de la superclase que se está redefiniendo
        if ( modeloOferta.equals(c.getModelo()) ) {
            p = p - valorDescuento;
            if (p < 0) {
                p = 0;
            }
        }
        return p;
    }
    @Override
    public String toString() {
        return "Oferta(" + modeloOferta + ", " + valorDescuento+ ") " + super.toString();
    }
}
```

Para calcular el precio final de un determinado coche, el método protegido `calcPrecioIva(Coche)` ha sido redefinido por la clase derivada para que ahora aplique un determinado valor de descuento si el coche es del modelo en oferta, para ello, invoca al método `calcPrecioIva(Coche)` de la superclase para calcular el precio con IVA del coche, y posteriormente le aplica el descuento si el modelo del coche es igual al modelo en oferta. Nótese como se utiliza la palabra reservada **super** para invocar al mismo método que está siendo redefinido en la clase derivada.

★ Nótese que en la clase `ConcesionarioOferta` el descuento se aplica sobre el precio del coche que ya incluye el IVA. Se deja como ejercicio para el lector el análisis de las modificaciones que deberían ser necesarias tanto en la clase `ConcesionarioIva` como en la clase `ConcesionarioOferta` para que el descuento fuese aplicado antes de calcular el IVA, de tal forma que el IVA fuese aplicado sobre el precio del coche con el descuento ya aplicado.

El siguiente programa principal permite comprobar la funcionalidad de las clases anteriores:

```
public class Main3 {
    public static void main(String[] args) {
        ConcesionarioIva concesionario = new ConcesionarioOferta(10, 500, "Seat-Leon");
        concesionario.anyadir(new CocheOcasion("Seat-Marbella", 9000, 20));
        concesionario.anyadir(new Coche("Seat-Leon", 15000));
        concesionario.anyadir(new CocheImportado("Porsche-911", 39000, 1000));

        System.out.println("Concesionario: "+concesionario.toString());
    }
}
```

La ejecución del programa anterior muestra la siguiente salida (se han añadido saltos de línea para mejorar la legibilidad):

```
Concesionario: Oferta(Seat-Leon, 500.0) Iva(10.0%)
               [ (Seat-Marbella, 7920.0), (Seat-Leon, 16000.0), (Porsche-911, 44000.0) ]
```

2. Clases Abstractas

En el ejemplo anterior todas las clases implementan completamente el comportamiento de sus métodos, y las clases derivadas pueden redefinir el comportamiento de determinados métodos, como ocurre con el método `calcPrecio()`. Éste enfoque no es el único posible, sino que también se puede considerar la posibilidad de definir la clase `Coche` de forma más general y **abstracta**, de forma que represente a todos los coches, y por lo tanto, dejando el método `calcPrecio()` sin implementar (*abstracto*), de tal forma que las clases derivadas deberán proporcionar una implementación adecuada del mismo según su contexto y comportamiento. Nótese que si una subclase no proporciona la implementación de un método abstracto de la superclase, entonces la subclase también será considerada como *abstracta*.

Una clase abstracta tiene una implementación *incompleta*, por lo que no puede ser utilizada para crear objetos, ya que sólo es posible crear objetos de las clases derivadas cuya implementación esté *completa*.

A continuación se muestra una definición alternativa (*abstracta*) de la clase `Coche`, en la que el método `calcPrecio()` se deja sin implementar (*abstracto*). Nótese como se utiliza el modificador **abstract** para especificar que tanto la clase como el método son abstractos.

```
public abstract class Coche {   // Clase Abstracta
    private String modelo;
    private double precioBase;
    public Coche(String m, double p) {
        modelo = m;
        precioBase = p;
    }
    public String getModelo() {
        return modelo;
    }
    protected double getPrecioBase() {   // Método Protegido
        return precioBase;
    }

    public abstract double calcPrecio() ; // Método Abstracto, sin implementación. Nótese el punto y coma

    @Override
    public String toString() {
        return "(" + getModelo() + ", " + calcPrecio() + ")";
    }
}
```

Podemos apreciar como el método `calcPrecio()` es abstracto y no proporciona una implementación, por lo que las clases derivadas deberán proporcionar una implementación adecuada según su propio contexto y comportamiento.

Además, vemos como el método `toString()` invoca al método `calcPrecio()` que es abstracto (sin implementación), de tal forma que cuando se invoque sobre un determinado objeto concreto, la implementación proporcionada por la clase dinámica de dicho objeto será invocada adecuadamente gracias a la vinculación dinámica.

Nótese como, además, se ha añadido el método protegido (**protected**) `getPrecioBase()`, que sólo podrá ser invocado por las clases derivadas, y permite conocer el precio base de un determinado coche. Así, los métodos protegidos proporcionan mecanismos para que las clases derivadas puedan ser definidas adecuadamente, y acceder a información que de otro modo sería imposible de acceder (en caso de ser privada), pero se mantiene protegida, ya que no se hace pública.

Ahora, las clases derivadas pueden proporcionar una implementación adecuada al método `calcPrecio()`, dependiendo de su contexto y comportamiento. Para ello, además, deben invocar al método protegido `getPrecioBase()` que les permite conocer el precio base del coche.

Por ejemplo, en la clase `CocheNacional`, que representa a un coche básico, el precio del coche viene dado directamente por el precio base del mismo, mientras que en las clases `CocheOcasion` y `CocheImportado`, el precio se calcula a partir del precio base del mismo.

```
public class CocheNacional extends Coche {
    public CocheNacional(String m, double p) {
        super(m, p);
    }
    @Override
    public double calcPrecio() {
        return getPrecioBase();
    }
}
```

```
public class CocheOcasion extends Coche {
    private double porcDescuento;
    public CocheOcasion(String m, double p, double d) {
        super(m, p);
        porcDescuento = d;
    }
    @Override
    public double calcPrecio() {
        double p = getPrecioBase();
        return p - p * porcDescuento / 100.0;
    }
}
```

```
public class CocheImportado extends Coche {
    private double costeHomologacion;
    public CocheImportado(String m, double p, double h) {
        super(m, p);
        costeHomologacion = h;
    }
    @Override
    public double calcPrecio() {
        return getPrecioBase() + costeHomologacion;
    }
}
```

3. Interfaces

En la sección anterior se explicó cómo se puede definir una clase abstracta, que representa un concepto más general, a diferencia de las clases concretas que representan un concepto más particular. De hecho, una clase es abstracta porque representa un concepto tan abstracto y general para el cual la implementación de determinados métodos no puede ser especificada, ya que depende de casos más concretos y particulares, que serán definidos como clases derivadas más particulares que sí puedan especificar la implementación de dichos métodos abstractos.

Si llevamos la abstracción y generalización hasta el extremo de que la clase abstracta no defina ninguna variable de instancia, ni proporcione ninguna implementación de ningún método, entonces tendríamos la especificación de una clase abstracta que sólo especifica los métodos y sus parámetros, además de su semántica funcional. Posteriormente, varias clases podrían proporcionar implementaciones alternativas con distintas propiedades y características.

Esta situación de generalización abstracta extrema se puede definir en java utilizando **interfaces**. La interfaz especifica los métodos públicos y sus parámetros, y las clases que implementan la interfaz proporcionan la implementación de los métodos especificados, según contextos diferentes. Así, gracias al polimorfismo y a la vinculación dinámica, cualquier objeto que implemente una determinada interfaz podrá ser utilizado allí donde sea requerida dicha interfaz, es decir, cualquier variable o parámetro del tipo de la interfaz puede referenciar a cualquier objeto que implemente dicha interfaz.

En java, una interfaz se define mediante la palabra reservada **interface**, y permite especificar los

métodos que definen la interfaz, que por defecto son **públicos** y **abstractos**, sin implementación. Una interfaz tampoco tiene variables de instancia, ni constructores. De forma excepcional, puede haber métodos con una implementación *por defecto* (se debe especificar la palabra reservada **default**). Adicionalmente, también se pueden definir métodos de clase (`static`) y constantes de clase (`static final`).

Cuando una clase *implementa* una determinada interfaz, debe proporcionar la implementación de los métodos especificados por la interfaz, o en otro caso, ser una clase abstracta, de tal forma que las clases derivadas deberán proporcionar la implementación de esos métodos.

En java, como se ha dicho anteriormente, una clase sólo puede **heredar de una única** clase (utilizando la palabra **extends**), sin embargo, una clase puede **implementar a muchas** interfaces (utilizando la palabra **implements**). Por otra parte, una interfaz puede **heredar de muchas** interfaces (utilizando la palabra **extends**).

No se pueden crear explícitamente objetos de interfaces, de hecho, no definen constructores. Para ello, será necesario crear objetos de clases que implementen las interfaces requeridas.

3.1. Aplicación Práctica de las Interfaces. Abstracción Funcional (I)

En ocasiones, puede ser adecuado que un determinado método reciba como parámetro, entre otros, un objeto que implemente una determinada interfaz. Este objeto (parámetro) podrá ser de cualquier clase que implemente la interfaz y, por lo tanto, permite resolver el método de una forma flexible y adaptable a diferentes circunstancias, especificadas en el momento de la invocación al método, al proporcionar diferentes objetos que implementen la interfaz en función de las necesidades del contexto.

Por ejemplo, se puede definir la interfaz `Selector`, que permite decidir si un determinado coche debe ser seleccionado según algún criterio *abstracto* sin especificar.

```
public interface Selector {
    boolean esSeleccionable(Coche c);
}
```

Así, es posible definir diferentes clases que implementen la interfaz `Selector` y permitan seleccionar coches según diferentes criterios. Por ejemplo, la clase `SelectorPrecio` permite seleccionar coches si el precio del mismo se encuentra dentro de un determinado rango. Para ello implementa el método `esSeleccionable(Coche)` especificado en la interfaz, de tal forma que devolverá `true` si el precio del coche recibido como parámetro se encuentra dentro del rango especificado durante la construcción del objeto.

```
public class SelectorPrecio implements Selector {
    private double min;
    private double max;
    public SelectorPrecio(double min, double max) {
        this.min = min;
        this.max = max;
    }
    @Override
    public boolean esSeleccionable(Coche c) {
        return (min <= c.calcPrecio()) && (c.calcPrecio() < max);
    }
    @Override
    public String toString() {
        return "SelectorPrecio("+min+", "+max+")";
    }
}
```

Del mismo modo, la clase `SelectorModelo` permite seleccionar coches según los nombres de los modelos. Para ello implementa el método `esSeleccionable(Coche)` especificado en la interfaz, de tal forma que devolverá `true` si el modelo del coche recibido como parámetro se encuentra entre los modelos especificados durante la construcción del objeto:

```
import java.util.Arrays;
public class SelectorModelo implements Selector {
    private String[] modelos;
    public SelectorModelo(String[] m) {
        modelos = m;                    // modelos = Arrays.copyOf(m, m.length);
    }
    @Override
    public boolean esSeleccionable(Coche c) {
        int i = 0;
        while ((i < modelos.length)&&( ! modelos[i].equals(c.getModelo()))) {
            ++i;
        }
        return (i < modelos.length);
    }
    @Override
    public String toString() {
        return "SelectorModelo("+Arrays.toString(modelos)+")";
    }
}
```

Así, podríamos definir un nuevo método en la clase `ConcesionarioSimple`, definida en la sección 1.3, que permita seleccionar coches de una forma flexible, según diferentes criterios especificados por el objeto que se reciba como parámetro que implemente la interfaz `Selector`:

```
public class ConcesionarioSimple {
    /* variables y métodos programados en las secciones anteriores */
    public Coche[] seleccionar(Selector s) {
        Coche[] seleccion = new Coche[nCoches];
        int idx = 0;
        for (int i = 0; i < nCoches; ++i) {
            if (s.esSeleccionable(coches[i])) {
                seleccion[idx] = coches[i];
                ++idx;
            }
        }
        return Arrays.copyOf(seleccion, idx);
    }
}
```

Podemos apreciar como el método `seleccionar(Selector)` itera sobre todos los coches del concesionario, y si alguno es seleccionable, lo añade al array que contiene los coches seleccionados. Para comprobar si un determinado coche es seleccionable, se invoca al método `esSeleccionable(Coche)` sobre el objeto recibido como parámetro, con el coche como parámetro.

Nótese como el método `seleccionar(Selector)` de la clase `ConcesionarioSimple` no conoce el criterio de selección que será aplicado durante la ejecución del mismo en un determinado momento. El responsable de establecer dicho criterio es el que invoca a dicho método, utilizando, como parámetro en la invocación del mismo, a un determinado objeto que proporcione una implementación adecuada del criterio de selección deseado, como se puede ver en el siguiente ejemplo.

Así, podemos diseñar un programa principal que permita comprobar la funcionalidad de las clases e interfaces anteriores:

```
import java.util.Arrays;
public class Main4 {
    public static void main(String[] args) {
        ConcesionarioSimple concesionario = new ConcesionarioSimple();
        concesionario.anyadir(new CocheOcasion("Seat-Marbella", 9000, 20));
        concesionario.anyadir(new Coche("Seat-Leon", 15000));
        concesionario.anyadir(new CocheImportado("Porsche-911", 39000, 1000));

        System.out.println("Concesionario: "+concesionario.toString());

        Selector sel1 = new SelectorPrecio(7000, 16000);
        Selector sel2 = new SelectorModelo(new String[]{"Seat-Leon", "Porsche-911"});
```

```
        Coche[] s1 = concesionario.seleccionar(sel1);
        Coche[] s2 = concesionario.seleccionar(sel2);

        System.out.println("Seleccion-1: "+Arrays.toString(s1));
        System.out.println("Seleccion-2: "+Arrays.toString(s2));
    }
}
```

La ejecución del programa anterior muestra la siguiente salida:

```
Concesionario: [ (Seat-Marbella, 7200.0), (Seat-Leon, 15000.0), (Porsche-911, 40000.0) ]
Seleccion-1: [(Seat-Marbella, 7200.0), (Seat-Leon, 15000.0)]
Seleccion-2: [(Seat-Leon, 15000.0), (Porsche-911, 40000.0)]
```

3.2. Aplicación Práctica de las Interfaces. Abstracción Funcional (II)

Del mismo modo que en la sección anterior un método recibía como parámetro un objeto que implementaba una determinada interfaz, y utilizaba los métodos de la interfaz implementados por dicho objeto para resolver un problema, también es posible que una determinada clase defina alguna variable de instancia como una referencia a un objeto que implemente alguna determinada interfaz, de tal forma que se pueda invocar a los métodos especificados por la interfaz en todas aquellas situaciones donde sean necesarios.

Así, se puede definir la interfaz `CalcPrecio` que permita realizar el cálculo flexible del precio del coche de la siguiente forma:

```
public interface CalcPrecio {
    double calcularPrecio(Coche c);
}
```

Así, es posible definir diferentes clases que implementen la interfaz `CalcPrecio` y permitan calcular el precio de los coches según diferentes criterios. Por ejemplo, la clase `CalcPrecioIva` proporciona una implementación de la interfaz `CalcPrecio` que permite aplicar un determinado porcentaje de IVA, establecido durante la construcción del objeto, en el cálculo del precio del coche.

```
public class CalcPrecioIva implements CalcPrecio {
    private double porcIva;
    public CalcPrecioIva(double p) {
        porcIva = p;
    }
    @Override
    public double calcularPrecio(Coche c) {
        double p = c.calcPrecio();
        return p + p * porcIva / 100.0;
    }
    @Override
    public String toString() {
        return "Iva(" + porcIva + "%)";
    }
}
```

Del mismo modo, la clase `CalcPrecioOferta` permite aplicar un determinado porcentaje de descuento, establecido durante la construcción del objeto, en el cálculo del precio del coche, según un determinado *modelo en oferta*, también establecido durante la construcción del objeto.

```
public class CalcPrecioOferta implements CalcPrecio {
    private double porcDescuento;
    private String modeloOferta;
    public CalcPrecioOferta(double p, String m) {
        porcDescuento = p;
        modeloOferta = m;
    }
    @Override
    public double calcularPrecio(Coche c) {
```

```
        double p = c.calcPrecio();
        if (modeloOferta.equals(c.getModelo())) {
            p = p - p * porcDescuento / 100.0;
        }
        return p;
    }
    @Override
    public String toString() {
        return "Oferta(" + modeloOferta + ", " + porcDescuento+ "%)";
    }
}
```

Ahora podemos diseñar una nueva clase `ConcesionarioFlex`, que permita calcular el precio final de un determinado coche de forma flexible, dependiendo del comportamiento definido por un determinado objeto que implemente la interfaz `CalcPrecio`.

Tanto la relación de herencia como la implementación de interfaces proporcionan un soporte adecuado para el polimorfismo, y la vinculación dinámica. Por ello, aquellas variables y parámetros cuyo tipo estático es una interfaz pueden referenciar a cualquier objeto de una clase que implemente dicha interfaz, e invocar adecuadamente a los métodos especificados por la interfaz sobre dichos objetos.

```
import java.util.Arrays;
public class ConcesionarioFlex {
    private static final int CAPACIDAD_INICIAL = 8;
    private int nCoches;
    private Coche[] coches;
    private CalcPrecio calc;
    public ConcesionarioFlex(CalcPrecio calc) {
        nCoches = 0;
        coches = new Coche[CAPACIDAD_INICIAL];
        this.calc = calc;                    // se almacena una referencia al objeto que implementa la interfaz
    }
    public void anyadir(Coche c) {
        if (nCoches == coches.length) {
            coches = Arrays.copyOf(coches, 2*coches.length);
        }
        coches[nCoches] = c;
        ++nCoches;
    }
    private Coche buscarCoche(String m) {          // Método privado
        Coche c = null;
        int i = 0;
        while ((i < nCoches)&&( ! m.equals(coches[i].getModelo()))) {
            ++i;
        }
        if (i < nCoches) {
            c = coches[i];
        }
        return c;
    }
    protected double calcPrecioFlex(Coche c) {  // Método protegido
        return calc.calcularPrecio(c);          // Invoca al método sobre el objeto que implementa la interfaz
    }
    public double calcPrecioFinal(String m) {
        double p = 0;
        Coche c = this.buscarCoche(m);          // Invoca al método privado para buscar el coche
        if (c != null) {
            p = this.calcPrecioFlex(c);         // Invoca al método protegido para calcular el precio
        }
        return p;
    }
    private String cocheToString(Coche c) {     // Método privado
        return "(" + c.getModelo() + ", " + this.calcPrecioFlex(c) + ")";  // Invoca al método protegido
    }
    @Override
```

```
    public String toString() {
        // También se puede realizar de forma más eficiente utilizando la clase StringBuilder
        String str = "";
        if (nCoches > 0) {
            str += " " + this.cocheToString(coches[0]);       // Invoca al método privado
            for (int i = 1; i < nCoches; ++i) {
                str += ", " + this.cocheToString(coches[i]);  // Invoca al método privado
            }
        }
        return calc.toString() + " [" + str + " ]";
    }
}
```

Para calcular el precio final de un determinado coche, el método protegido `calcPrecioFlex(Coche)` invoca al método `calcularPrecio(Coche)` especificado por la interfaz `CalcPrecio` sobre el objeto que implementa la interfaz.

El siguiente programa principal permite comprobar la funcionalidad de las clases e interfaces anteriores:

```
public class Main5 {
    public static void pruebaIva(double porcIva) {
        CalcPrecio calcPrecio = new CalcPrecioIva(porcIva);
        ConcesionarioFlex concesionario = new ConcesionarioFlex(calcPrecio);
        concesionario.anyadir(new CocheOcasion("Seat-Marbella", 9000, 20));
        concesionario.anyadir(new Coche("Seat-Leon", 15000));
        concesionario.anyadir(new CocheImportado("Porsche-911", 39000, 1000));

        System.out.println("Concesionario: "+concesionario.toString());
    }
    public static void pruebaOferta(double porcOferta, String modeloOferta) {
        CalcPrecio calcPrecio = new CalcPrecioOferta(porcOferta, modeloOferta);
        ConcesionarioFlex concesionario = new ConcesionarioFlex(calcPrecio);
        concesionario.anyadir(new CocheOcasion("Seat-Marbella", 9000, 20));
        concesionario.anyadir(new Coche("Seat-Leon", 15000));
        concesionario.anyadir(new CocheImportado("Porsche-911", 39000, 1000));

        System.out.println("Concesionario: "+concesionario.toString());
    }
    public static void main(String[] args) {
        pruebaIva(10);
        pruebaOferta(20, "Seat-Leon");
    }
}
```

La ejecución del programa anterior muestra la siguiente salida:

```
Concesionario: Iva(10.0%) [ (Seat-Marbella, 7920.0), (Seat-Leon, 16500.0), (Porsche-911, 44000.0) ]
Concesionario: Oferta(Seat-Leon, 20.0%) [ (Seat-Marbella, 7200.0), (Seat-Leon, 12000.0), (Porsche-911, 40000.0) ]
```

3.3. Aplicación Práctica de las Interfaces. Generalización Abstracta de Clases

Las interfaces, así mismo, tambien permiten especificar conceptos abstractos como generalizaciones abstractas de clases, sin considerar la forma en que esa abstracción sea implementada. De hecho, varias implementaciones alternativas de la interfaz podrán dar lugar a múltiples definiciones concretas con características diferentes.

En este caso, el diseño se enfoca en la especificación de conceptos abstractos, de tal forma que la utilización de los conceptos abstractos viene determinada por las características de las distintas clases que proporcionan una implementación adecuada de los mismos.

Ejemplo

La siguiente interfaz `Conjunto` representa el concepto abstracto de un *conjunto de objetos*, que contiene objetos sin repetición (según el método `equals`), y especifica las siguientes operaciones sobre los elementos del conjunto:

- `clear()`: elimina todos los elementos del conjunto, quedando éste vacío.
- `size()`: devuelve el número de elementos que contiene el conjunto. El conjunto vacío se corresponde con un número de elementos igual a cero.
- `add(Object)`: si el objeto que se recibe como parámetro ya existe en el conjunto, según el método `equals`, entonces se reemplaza el objeto existente por el nuevo recibido como parámetro. En otro caso, añade el objeto recibido como parámetro al conjunto.
- `rem(Object)`: si el objeto que se recibe como parámetro existe en el conjunto, según el método `equals`, entonces elimina el elemento recibido como parámetro del conjunto.
- `pertenece(Object)`: si el objeto que se recibe como parámetro existe en el conjunto, según el método `equals`, entonces devuelve `true`. En otro caso devuelve `false`.

```
public interface Conjunto {
    void    clear();
    int     size();
    void    add(Object o);
    void    rem(Object o);
    boolean pertenece(Object o);
}
```

Dada la definición de la interfaz que especifica el concepto abstracto de conjunto de objetos, la siguiente clase `ConjuntoArray` proporciona una implementación de la interfaz `Conjunto`, y podrá ser utilizada allí donde se requiera un `Conjunto`.

```
import java.util.Arrays;
public class ConjuntoArray implements Conjunto {
    private static final int CAPACIDAD_INICIAL = 8;
    private int sz;
    private Object[] elementos;
    public ConjuntoArray() {
        sz = 0;
        elementos = new Object[CAPACIDAD_INICIAL];
    }
    @Override
    public void clear() {
        for (int i = 0; i < sz; ++i) {
            elementos[i] = null;       // facilita la recolección de memoria
        }
        sz = 0;
    }
    @Override
    public int size() {
        return sz;
    }
    private int search(Object o) {
        int i = 0;
        while ((i < sz)&&( ! o.equals(elementos[i]))) { // no considera el caso null
            ++i;
        }
        if (i >= sz) {
            i = -1;
        }
        return i;
    }
    @Override
    public void add(Object o) {
        int p = search(o);
        if (p >= 0) {
            elementos[p] = o;
        } else {
            if (sz == elementos.length) {
                elementos = Arrays.copyOf(elementos, 2*elementos.length);
            }
```

```
            elementos[sz] = o;
            ++sz;
        }
    }
    @Override
    public void rem(Object o) {
        int p = search(o);
        if (p >= 0) {
            --sz;
            if (p < sz) {
                elementos[p] = elementos[sz];
            }
            elementos[sz] = null;       // facilita la recolección de memoria
        }
    }
    @Override
    public boolean pertenece(Object o) {
        return search(o) >= 0;
    }
    @Override
    public String toString() {
        // También se puede realizar de forma más eficiente utilizando la clase StringBuilder
        String str = "";
        if (sz > 0) {
            str += " " + elementos[0].toString();
            for (int i = 1; i < sz; ++i) {
                str += ", " + elementos[i].toString();
            }
        }
        return "{" + str + " }";
    }
}
```

Finalmente, se puede utilizar la clase `ConjuntoArray` allí donde sea necesario que un determinado objeto que proporcione adecuadamente el concepto abstracto especificado por la interfaz `Conjunto`. Por ejemplo, para almacenar y comprobar una serie de números, como se muestra en el siguiente programa principal:

```
public class Main6 {
    public static void main(String[] args) {
        Conjunto conjunto = new ConjuntoArray();
        conjunto.add(1);
        conjunto.add(2);
        conjunto.add(3);
        conjunto.add(4);
        conjunto.add(3);
        System.out.println("Conjunto: "+conjunto.toString());
        conjunto.rem(1);
        conjunto.rem(3);
        conjunto.rem(0);
        System.out.println("Conjunto: "+conjunto.toString());
        if (conjunto.pertenece(3)) {
            System.out.println("Error");
        }
        conjunto.clear();
        System.out.println("Conjunto: "+conjunto.toString());
    }
}
```

La ejecución del programa anterior muestra la siguiente salida:

```
Conjunto: [ 1, 2, 3, 4 ]
Conjunto: [ 4, 2 ]
Conjunto: [ ]
```