9. Programación orientada a objetos

9.1 Clases y objetos

La programación orientada a objetos es un paradigma de programación que se basa, como su nombre indica, en la utilización de objetos. Estos objetos también se suelen llamar instancias.

Un objeto en términos de POO no se diferencia mucho de lo que conocemos como un objeto en la vida real. Pensemos por ejemplo en un coche. Nuestro coche sería un objeto concreto de la vida real, igual que el coche del vecino, o el coche de un compañero de trabajo, o un deportivo que vimos por la calle el fin de semana pasado... Todos esos coches son objetos concretos que podemos ver y tocar.

Tanto mi coche como el coche del vecino tienen algo en común, ambos son coches. En este caso mi coche y el coche del vecino serían **instancias** (objetos) y coche (a secas) sería una **clase**. La palabra coche define algo genérico, es una abstracción, no es un coche concreto sino que hace referencia a unos elementos que tienen una serie de propiedades como matrícula, marca, modelo, color, etc.; este conjunto de propiedades se denominan **atributos** o **variables de instancia**.

Clase

Concepto abstracto que denota una serie de cualidades, por ejemplo **coche**.

Instancia

Objeto palpable, que se deriva de la concreción de una clase, por ejemplo **mi coche**.

Atributos

Conjunto de características que comparten los objetos de una clase, por ejemplo para la clase **coche** tendríamos **matrícula**, **marca**, **modelo**, **color** y **número de plazas**.

En Java, los nombres de las clases se escriben con la primera letra en mayúscula mientras que los nombres de las instancias comienzan con una letra en minúscula. Por ejemplo, la clase coche se escribe en Java como `Coche` y el objeto “mi coche” se podría escribir como `miCoche`.

Definiremos cada clase en un fichero con el mismo nombre más la extensión `.java`, por tanto, la definición de la clase `Coche` debe estar contenida en un fichero con nombre `Coche.java`

Vamos a definir a continuación la clase `Libro` con los atributos `isbn`, `autor`, `titulo` y `numeroPaginas`.

```java
/**
 * Libro.java
 * Definición de la clase Libro
 * @author Luis José Sánchez
 */

public class Libro {

  // atributos
  String isbn;
  String titulo;
  String autor;
  int    numeroPaginas;
}
```

A continuación creamos varios objetos de esta clase.

```java
/**
 * PruebaLibro.java
 * Programa que prueba la clase Libro
 * @author Luis José Sánchez
 */

public class PruebaLibro {
  public static void main(String[] args) {

    Libro lib = new Libro();
    Libro miLibrito = new Libro();
    Libro quijote = new Libro();
  }
}
```

Hemos creado tres instancias de la clase libro: `lib`, `miLibrito` y `quijote`. Ya sabemos definir una clase indicando sus atributos y crear instancias.

Las **variables de instancia** (atributos) determinan las **cualidades** de los objetos.

9.2 Encapsulamiento y ocultación

Uno de los pilares en los que se basa la Programación Orientada a Objetos es el **encapsulamiento**. Básicamente, el **encapsulamiento** consiste en definir todas las

propiedades y el comportamiento de una clase dentro de esa clase; es decir, en la clase `Coche` estará definido todo lo concerniente a la clase `Coche` y en la clase `Libro` estará definido todo lo que tenga que ver con la clase `Libro`.

El **encapsulamiento** parece algo obvio, casi de perogrullo, pero hay que tenerlo siempre muy presente al programar utilizando clases y objetos. En alguna ocasión puede que estemos tentados a mezclar parte de una clase con otra clase distinta para resolver un problema puntual. No hay que caer en esa trampa. Se deben escribir los programas de forma que cada cosa esté en su sitio. Sobre todo al principio, cuando definimos nuestras primeras clases, debemos estar pendientes de que todo está definido donde corresponde.

La **ocultación** es una técnica que incorporan algunos lenguajes (entre ellos Java) que permite esconder los elementos que definen una clase, de tal forma que desde otra clase distinta no se pueden "ver las tripas" de la primera. La **ocultación** facilita, como veremos más adelante, el **encapsulamiento**.

9.3 Métodos

Un coche arranca, para, se aparca, hace sonar el claxon, se puede llevar a reparar... Un gato puede comer, dormir, maullar, ronronear...

Las acciones asociadas a una clase se llaman métodos. Estos métodos se definen dentro del cuerpo de la clase y se suelen colocar a continuación de los atributos.

Los **métodos** determinan el **comportamiento** de los objetos.

9.3.1 Creí haber visto un lindo gatito

Vamos a crear la clase `GatoSimple`. La llamamos así porque más adelante crearemos otra clase algo más elaborada que se llamará `Gato`. Para saber qué atributos debe tener esta clase hay que preguntarse qué características tienen los gatos. Todos los gatos son de un color determinado, pertenecen a una raza, tienen una edad, tienen un determinado sexo - son machos o hembras - y tienen un peso que se puede expresar en kilogramos. Éstos serán por tanto los atributos que tendrá la clase `GatoSimple`.

Para saber qué metodos debemos implementar hay que preguntarse qué acciones están asociadas a los gatos. Bien, pues los gatos maullan, ronronean, comen y si son machos se pelean entre ellos para disputarse el favor de las hembras. Esos serán los métodos que definamos en la clase.

```java
/**
 * GatoSimple.java
 * Definición de la clase GatoSimple
 * @author Luis José Sánchez
 */

public class GatoSimple {

  // atributos ////////////////////////////

  String color, raza, sexo;
  int    edad;
  double peso;

  // métodos //////////////////////////////

  // constructor
  GatoSimple (String s) {
    this.sexo = s;
  }

  // getter
  String getSexo() {
    return this.sexo;
  }

  /**
   * Hace que el gato maulle
   */
  void maulla() {
    System.out.println("Miauuuu");
  }

  /**
   * Hace que el gato ronronee
   */
  void ronronea() {
    System.out.println("mrrrrrr");
  }

  /**
   * Hace que el gato coma.
   * A los gatos les gusta el pescado, si le damos otra comida
   * la rechazará.
   *
```

```
    * @param comida la comida que se le ofrece al gato
    */
  void come(String comida) {
    if (comida.equals("pescado")) {
        System.out.println("Hmmmm, gracias");
    } else {
        System.out.println("Lo siento, yo solo como pescado");
    }
  }

  /**
    * Pone a pelear dos gatos.
    * Solo se van a pelear dos machos entre sí.
    *
    * @param contrincante es el gato contra el que pelear
    */
  void peleaCon(GatoSimple contrincante) {
    if (this.sexo.equals("hembra")) {
      System.out.println("no me gusta pelear");
    } else {
      if (contrincante.getSexo().equals("hembra")) {
          System.out.println("no peleo contra gatitas");
      } else {
          System.out.println("ven aquí que te vas a enterar");
      }
    }
  }
}
```

Constructor

El método constructor tiene siempre el mismo nombre que la clase y se utiliza normalmente para inicializar los atributos.

Los atributos de la clase `GatoSimple` - `color`, `raza`, `sexo`, `edad` y `peso` - se declaran igual que las variables que hemos venido usando hasta ahora, pero hay una gran diferencia entre estos atributos y las variables que aparecen en el `main` (programa principal). Una variable definida en el cuerpo del programa principal es única, sin embargo cada uno de los objetos que se crean en el programa principal tienen sus propios atributos; es decir, si en el programa principal se crean 20 objetos de la clase `GatoSimple`, cada uno tiene sus valores para los atributos `color`, `raza`, etc.

Fíjate en la cabecera del método `peleaCon`:

```
void peleaCon(GatoSimple contrincante)
```

Como puedes comprobar, es posible pasar un objeto como parámetro. A continuación se muestra un programa que prueba la clase `GatoSimple`. Te recomiendo seguir el programa línea a línea y observar atentamente la salida que produce.

```
/**
 * PruebaGatoSimple.java
 * Programa que prueba la clase GatoSimple
 * @author Luis José Sánchez
 */

public class PruebaGatoSimple {
  public static void main(String[] args) {

    GatoSimple garfield = new GatoSimple("macho");

    System.out.println("hola gatito");
    garfield.maulla();
    System.out.println("toma tarta");
    garfield.come("tarta selva negra");
    System.out.println("toma pescado, a ver si esto te gusta");
    garfield.come("pescado");

    GatoSimple tom = new GatoSimple("macho");

    System.out.println("Tom, toma sopita de verduras");
    tom.come("sopa de verduras");

    GatoSimple lisa = new GatoSimple("hembra");

    System.out.println("gatitos, a ver cómo maulláis");
    garfield.maulla();
    tom.maulla();
    lisa.maulla();

    garfield.peleaCon(lisa);
    lisa.peleaCon(tom);
    tom.peleaCon(garfield);
  }
}
```

Observa cómo al crear una instancia se llama al constructor que, como decíamos antes, tiene el mismo nombre de la clase y sirve para inicializar los atributos. En este caso se inicializa el atributo `sexo`. Más adelante veremos constructores que inicializan varios atributos e incluso definiremos distintos constructores en la misma clase.

```
GatoSimple garfield = new GatoSimple("macho");
```

Como puedes ver hay métodos que no toman ningún parámetro.

```
garfield.maulla();
```

Y hay otros métodos que deben tomar parámetros obligatoriamente.

```
garfield.come("tarta selva negra");
```

9.3.2 Métodos *getter* y *setter*

Vamos a crear la clase `Cubo`. Para saber qué atributos se deben definir, nos preguntamos qué características tienen los cubos - igual que hicimos con la clase `GatoSimple`. Todos los cubos tienen una determinada capacidad, un color, están hechos de un determinado material - plástico, latón, etc. - y puede que tengan asa o puede que no. Un cubo se fabrica con el propósito de contener líquido; por tanto otra característica es la cantidad de litros de líquido que contiene en un momento determinado. Por ahora, solo nos interesa saber la capacidad máxima y los litros que contiene el cubo en cada momento, así que esos serán los atributos que tendremos en cuenta.

```
/**
 * Cubo.java
 * Definición de la clase Cubo
 * @author Luis José Sánchez
 */

public class Cubo {

  // atributos ////////////////////////////

  int capacidad; // capacidad máxima en litros
  int contenido; // contenido actual en litros

  // métodos //////////////////////////////

  // constructor
  Cubo (int c) {
    this.capacidad = c;
  }

  // métodos getter
  int getCapacidad() {
    return this.capacidad;
```

```
}

int getContenido() {
  return this.contenido;
}

// método setter
void setContenido(int litros) {
  this.contenido = litros;
}

// otros métodos
void vacia() {
  this.contenido = 0;
}

/**
 * Llena el cubo al máximo de su capacidad.
 */
void llena() {
  this.contenido = this.capacidad;
}

/**
 * Pinta el cubo en la pantalla.
 * Se muestran los bordes del cubo con el carácter # y el
 * agua que contiene con el carácter ~.
 */
void pinta() {
  for (int nivel = this.capacidad; nivel > 0; nivel--) {
    if (this.contenido >= nivel) {
      System.out.println("#~~~~#");
    } else {
      System.out.println("#    #");
    }
  }
  System.out.println("######");
}

/**
 * Vuelca el contenido de un cubo sobre otro.
 * Antes de echar el agua se comprueba cuánto le cabe al
 * cubo destino.
 */
void vuelcaEn(Cubo destino) {
```

```
    int libres = destino.getCapacidad() - destino.getContenido();

    if (libres > 0) {
      if (this.contenido <= libres) {
        destino.setContenido(destino.getContenido() + this.contenido);
        this.vacia();
      } else {
        this.contenido -= libres;
        destino.llena();
      }
    }
  }
}
```

Observa estos métodos extraídos de la clase `Cubo`:

```
int getCapacidad() {
    return this.capacidad;
}

int getContenido() {
    return this.contenido;
}
```

Se trata de métodos muy simples, su cometido es devolver el valor de un atributo. Podrían tener cualquier nombre pero en Java es costumbre llamarlos con la palabra *get* (obtener) seguida del nombre del atributo.

Fijémonos ahora este otro método:

```
void setContenido(int litros) {
    this.contenido = litros;
}
```

Ahora estamos ante un *setter*. Este tipo de métodos tiene el cometido de establecer un valor para un determinado atributo. Como puedes ver, `setContenido` está formada por *set* (asignar) más el nombre del atributo.

Podrías preguntarte: ¿por qué se crea un método *getter* para extraer el valor de una variable y no se accede a la variable directamente?

Parece más lógico hacer esto

```
System.out.print(miCubo.capacidad);
```

que esto otro

```
System.out.print(miCubo.getCapacidad());
```

Sin embargo, en Java se opta casi siempre por esta última opción. Tiene su explicación y lo entenderás bien cuando estudies el apartado Ámbito/visibilidad de los elementos de una clase - public , protected y private

Probamos, con el siguiente ejemplo, la clase `Cubo` que acabamos de definir. Crea tus propios cubos, pasa agua de unos a otros y observa lo que se muestra por pantalla.

```
/**
 * PruebaCubo.java
 * Programa que prueba la clase Cubo
 * @author Luis José Sánchez
 */

public class PruebaCubo {
  public static void main(String[] args) {

    Cubo cubito = new Cubo(2);
    Cubo cubote = new Cubo(7);

    System.out.println("Cubito: \n");
    cubito.pinta();

    System.out.println("\nCubote: \n");
    cubote.pinta();

    System.out.println("\nLleno el cubito: \n");
    cubito.llena();
    cubito.pinta();

    System.out.println("\nEl cubote sigue vacío: \n");
    cubote.pinta();

    System.out.println("\nAhora vuelco lo que tiene el cubito en el cubote.\n");
    cubito.vuelcaEn(cubote);

    System.out.println("Cubito: \n");
    cubito.pinta();

    System.out.println("\nCubote: \n");
    cubote.pinta();

    System.out.println("\nAhora vuelco lo que tiene el cubote en el cubito.\n");
    cubote.vuelcaEn(cubito);
```

```
    System.out.println("Cubito: \n");
    cubito.pinta();

    System.out.println("\nCubote: \n");
    cubote.pinta();
  }
}
```

9.3.3 Método `toString`

La definición de la clase `Cubo` del apartado anterior contiene el método `pinta` que, como su nombre indica, permite pintar en pantalla un objeto de esa clase. Se podría definir el método `ficha` para mostrar información sobre objetos de la clase `Alumno` por ejemplo. También sería posible implementar el método `imprime` dentro de la clase Libro con el propósito de mostrar por pantalla los datos de un libro. Todos estos métodos - pinta, ficha e imprime - hacen básicamente lo mismo.

En Java existe una solución muy elegante para mostrar información sobre un objeto por pantalla. Si se quiere mostrar el contenido de la variable entera `x` se utiliza `System.out.print(x)` y si se quiere mostrar el valor de la variable de tipo cadena de caracteres `nombre` se escribe `System.out.print(nombre)`. De la misma manera, si se quiere mostrar el objeto `miPiramide` que pertenece a la clase `Piramide`, también se podría usar `System.out.print(miPiramide)`. Java sabe perfectamente cómo mostrar números y cadenas de caracteres pero no sabe *a priori* cómo se pintan pirámides. Para indicar a Java cómo debe pintar un objeto de la clase `Piramide` basta con implementar el método `toString` dentro de la clase.

Veamos un ejemplo muy sencillo de implementación de `toString`. Definiremos la clase `Cuadrado` con el atributo `lado`, el constructor y el método `toString`.

```
/**
 * Cuadrado.java
 * Definición de la clase Cuadrado
 * @author Luis José Sánchez
 */

public class Cuadrado {

  int lado;

  public Cuadrado(int l) {
    this.lado = l;
  }

  public String toString() {
```

```
    int i, espacios;
    String resultado = "";

    for (i = 0; i < this.lado; i++) {
      resultado += "□□";
    }
    resultado += "\n";

    for (i = 1; i < this.lado - 1; i++) {
      resultado += "□□";
      for (espacios = 1; espacios < this.lado - 1; espacios++) {
        resultado += "  ";
      }
      resultado += "□□\n";
    }

    for (i = 0; i < this.lado; i++) {
      resultado += "□□";
    }
    resultado += "\n";

    return resultado;
  }
}
```

Observa que el método `toString()` devuelve una cadena de caracteres. Esa cadena es precisamente la que mostrará por pantalla `System.out.print()`. En el programa que se muestra a continuación y que prueba la clase `Cuadrado` puedes comprobar que se pinta un cuadrado igual que si se tratara de cualquier otra variable.

```
/**
 * PruebaCuadrado.java
 * Programa que prueba la clase Cuadrado
 * @author Luis José Sánchez
 */

public class PruebaCuadrado {
  public static void main(String[] args) {

    Cuadrado miCuadradito = new Cuadrado(5);
    System.out.println(miCuadradito);
  }
}
```

9.4 Ámbito/visibilidad de los elementos de una clase - `public`, `protected` y `private`

Al definir los elementos de una clase, se pueden especificar sus ámbitos (*scope*) de visibilidad o accesibilidad. Un elemento `public` (público) es visible desde cualquier clase, un elemento `protected` (protegido) es visible desde la clase actual y desde todas sus subclases y, finalmente, un elemento `private` (privado) únicamente es visible dentro de la clase actual.

En el siguiente apartado veremos qué son las subclases. De momento fíjate en el ámbito del atributo y de los métodos en la definición de la clase `Animal`.

```
/**
 * Animal.java
 * Definición de la clase Animal
 * @author Luis José Sánchez
 */

public abstract class Animal {

  private Sexo sexo;

  public Animal () {
    sexo = Sexo.MACHO;
  }

  public Animal (Sexo s) {
    sexo = s;
  }

  public Sexo getSexo() {
    return sexo;
  }

  public String toString() {
    return "Sexo: " + this.sexo + "\n";
  }

  /**
   * Hace que el animal se eche a dormir.
   */
  public void duerme() {
    System.out.println("Zzzzzzz");
  }
}
```

Fíjate que el atributo `sexo` se ha definido como `private`.

```
private Sexo sexo;
```

Eso quiere decir que a ese atributo únicamente se tiene acceso dentro de la clase `Animal`. Sin embargo, todos los métodos se han definido `public`, lo que significa que se podrán utilizar desde cualquier otro programa, por ejemplo, como veremos más adelante, desde el programa `PurebaAnimal.java`.

Salvo casos puntuales se seguirá la regla de declarar `private` las variables de instancia y `public` los métodos.

El sexo de un animal solo puede ser macho, hembra o hermafrodita. Una forma de delimitar los valores que puede tomar un atributo es definir un tipo enumerado.

Tipo enumerado

Mediante `enum` se puede definir un tipo enumerado, de esta forma un atributo solo podrá tener uno de los posibles valores que se dan como opción. Los valores que se especifican en el tipo enumerado se suelen escribir con todas las letras en mayúscula.

```
/**
 * Sexo.java
 * Definición del tipo enumerado Sexo
 * @author Luis José Sánchez
 */

public enum Sexo {
    MACHO, HEMBRA, HERMAFRODITA
}
```

9.5 Herencia

La herencia es una de las características más importantes de la POO. Si definimos una serie de atributos y métodos para una clase, al crear una subclase, todos estos atributos y métodos siguen siendo válidos.

En el apartado anterior se define la clase `Animal`. Uno de los métodos de esta clase es `duerme`. A continuación podemos crear las clases `Gato` y `Perro` como subclases de `Animal`. De forma automática, se puede utilizar el método `duerme` con las instancias de las clases `Gato` y `Perro` ¿no es fantástico?

La clase `Ave` es subclase de `Animal` y la clase `Pinguino`, a su vez, sería subclase de `Ave` y por tanto hereda todos sus atributos y métodos.

Clase abstracta (`abstract`)

Una clase abstracta es aquella que no va a tener instancias de forma directa, aunque sí habrá instancias de las subclases (siempre que esas subclases no sean también abstractas). Por ejemplo, si se define la clase `Animal` como abstracta, no se podrán crear objetos de la clase `Animal`, es decir, no se podrá hacer `Animal mascota = new Animal()`, pero sí se podrán crear instancias de la clase `Gato`, `Ave` o `Pinguino` que son subclases de `Animal`.

Para crear en Java una subclase de otra clase existente se utiliza la palabra reservada `extends`. A continuación se muestra el código de las clases `Gato`, `Ave` y `Pinguino`, así como el programa que prueba estas clases creando instancias y aplicándoles métodos. Recuerda que la definición de la clase `Animal` se muestra en el apartado anterior.

```
/**
 * Gato.java
 * Definición de la clase Gato
 * @author Luis José Sánchez
 */

public class Gato extends Animal {

  private String raza;


  public Gato (Sexo s, String r) {
    super(s);
    raza = r;
  }

  public Gato (Sexo s) {
    super(s);
    raza = "siamés";
  }

  public Gato (String r) {
    super(Sexo.HEMBRA);
    raza = r;
  }

  public Gato () {
    super(Sexo.HEMBRA);
```

```
    raza = "siamés";
  }

  public String toString() {
    return super.toString()
            + "Raza: " + this.raza
            + "\n************************\n";
  }

  /**
   * Hace que el gato maulle.
   */
  public void maulla() {
    System.out.println("Miauuuu");
  }

  /**
   * Hace que el gato ronronee
   */
  public void ronronea() {
    System.out.println("mrrrrrr");
  }

  /**
   * Hace que el gato coma.
   * A los gatos les gusta el pescado, si le damos otra comida
   * la rechazará.
   *
   * @param comida la comida que se le ofrece al gato
   */
  public void come(String comida) {
    if (comida.equals("pescado")) {
      System.out.println("Hmmmm, gracias");
    } else {
      System.out.println("Lo siento, yo solo como pescado");
    }
  }

  /**
   * Pone a pelear dos gatos.
   * Solo se van a pelear dos machos entre sí.
   *
   * @param contrincante es el gato contra el que pelear
   */
  public void peleaCon(Gato contrincante) {
```

```
    if (this.getSexo() == Sexo.HEMBRA) {
      System.out.println("no me gusta pelear");
    } else {
      if (contrincante.getSexo() == Sexo.HEMBRA) {
        System.out.println("no peleo contra gatitas");
      } else {
        System.out.println("ven aquí que te vas a enterar");
      }
    }
  }
}
```

Observa que se definen nada menos que cuatro constructores en la clase `Gato`. Desde el programa principal se dilucida cuál de ellos se utiliza en función del número y tipo de parámetros que se pasa al método. Por ejemplo, si desde el programa principal se crea un gato de esta forma

```
Gato gati = new Gato();
```

entonces se llamaría al constructor definido como

```
public Gato () {
  super(Sexo.HEMBRA);
  raza = "siamés";
}
```

Por tanto `gati` sería una gata de raza siamés. Si, por el contrario, creamos `gati` de esta otra manera desde el programa principal

```
Gato gati = new Gato(Sexo.MACHO, "siberiano");
```

se llamaría al siguiente constructor

```
public Gato (Sexo s, String r) {
  super(s);
  raza = r;
}
```

y `gati` sería en este caso un gato macho de raza siberiano.

El método `super()` hace una llamada al método equivalente de la superclase. Fíjate que se utiliza tanto en el constructor como en el método `toString()`. Por ejemplo, al llamar a `super()` dentro del método `toString()` se está llamando al `toString()` que hay definido en la clase `Animal`, justo un nivel por encima de `Gato` en la jerarquía de clases.

A continuación tenemos la definición de la clase `Ave` que es una subclase de `Animal`.

```java
/**
 * Ave.java
 * Definición de la clase Ave
 * @author Luis José Sánchez
 */

public class Ave extends Animal {

  public Ave(Sexo s) {
    super(s);
  }

  public Ave() {
    super();
  }

  /**
   * Hace que el ave se limpie.
   */
  public void aseate() {
    System.out.println("Me estoy limpiando las plumas");
  }

  /**
   * Hace que el ave levante el vuelo.
   */
  public void vuela() {
    System.out.println("Estoy volando");
  }
}
```

9.5.1 Sobrecarga de métodos

Un método se puede **redefinir** (volver a definir con el mismo nombre) en una subclase. Por ejemplo, el método `vuela` que está definido en la clase `Ave` se vuelve a definir en la clase `Pinguino`. En estos casos, indicaremos nuestra intención de sobreescribir un método mediante la etiqueta `@Override`.

Si no escribimos esta etiqueta, la sobreescritura del método se realizará de todas formas ya que `@Override` indica simplemente una intención. Ahora imagina que quieres sobreescribir el método `come` de `Animal` declarando un `come` específico para los gatos en la clase `Gato`. Si escribes `@Override` y luego te equivocas en el nombre del método y escribes `comer`, entonces el compilador diría algo como: "¡Cuidado! algo no está bien, me has dicho que ibas a sobreescribir un método de la superclase y sin embargo `comer` no está definido".

A continuación tienes la definición de la clase `Pinguino`.

```java
/**
 * Pinguino.java
 * Definición de la clase Pinguino
 * @author Luis José Sánchez
 */

public class Pinguino extends Ave {

  public Pinguino() {
    super();
  }

  public Pinguino(Sexo s) {
    super(s);
  }

  /**
   * El pingüino se siente triste porque no puede volar.
   */
  @Override
  public void vuela() {
    System.out.println("No puedo volar");
  }
}
```

Con el siguiente programa se prueba la clase `Animal` y todas las subclases que derivan de ella. Observa cada línea y comprueba qué hace el programa.

```java
/**
 * PruebaAnimal.java
 * Programa que prueba la clase Animal y sus subclases
 * @author Luis José Sánchez
 */

public class PruebaAnimal {
  public static void main(String[] args) {

    Gato garfield = new Gato(Sexo.MACHO, "romano");
    Gato tom = new Gato(Sexo.MACHO);
    Gato lisa = new Gato(Sexo.HEMBRA);
    Gato silvestre = new Gato();

    System.out.println(garfield);
    System.out.println(tom);
    System.out.println(lisa);
```

```
    System.out.println(silvestre);

    Ave miLoro = new Ave();
    miLoro.aseate();
    miLoro.vuela();

    Pinguino pingu = new Pinguino(Sexo.HEMBRA);
    pingu.aseate();
    pingu.vuela();
  }
}
```

En el ejemplo anterior, los objetos `miLoro` y `pingu` actúan de manera **polimórfica** porque a ambos se les aplican los métodos `aseate` y `vuela`.

Polimorfismo

En Programación Orientada a Objetos, se llama **polimorfismo** a la capacidad que tienen los objetos de distinto tipo (de distintas clases) de responder al mismo método.

9.6 Atributos y métodos de clase (`static`)

Hasta el momento hemos definido atributos de instancia como `raza`, `sexo` o `color` y métodos de instancia como `maulla`, `come` o `vuela`. De tal modo que si en el programa se crean 20 gatos, cada uno de ellos tiene su propia raza y puede haber potencialmente 20 razas diferentes. También podría aplicar el método `maulla` a todos y cada uno de esos 20 gatos.

No obstante, en determinadas ocasiones, nos puede interesar tener atributos de clase (variables de clase) y métodos de clase. Cuando se define una variable de clase solo existe una copia del atributo para toda la clase y no una para cada objeto. Esto es útil cuando se quiere llevar la cuenta global de algún parámetro. Los métodos de clase se aplican a la clase y no a instancias concretas.

A continuación se muestra un ejemplo que contiene la variable de clase `kilometrajeTotal`. Si bien cada coche tiene un atributo `kilometraje` donde se van acumulando los kilómetros que va recorriendo, en la variable de clase `kilometrajeTotal` se lleva la cuenta de los kilómetros que han recorrido todos los coches que se han creado.

También se crea un método de clase llamado `getKilometrajeTotal` que simplemente es un *getter* para la variable de clase `kilometrajeTotal`.

```java
/**
 * Coche.java
 * Definición de la clase Coche
 * @author Luis José Sánchez
 */

public class Coche {

  // atributo de clase
  private static int kilometrajeTotal = 0;

  // método de clase
  public static int getKilometrajeTotal() {
    return kilometrajeTotal;
  }

  private String marca;
  private String modelo;
  private int kilometraje;

  public Coche(String ma, String mo) {
    marca = ma;
    modelo = mo;
    kilometraje = 0;
  }

  public int getKilometraje() {
    return kilometraje;
  }

  /**
   * Recorre una determinada distancia.
   *
   * @param km distancia a recorrer en kilómetros
   */
  public void recorre(int km) {
    kilometraje += km;
    kilometrajeTotal += km;
  }
}
```

Como ya hemos comentado, el atributo `kilometrajeTotal` almacena el número total de kilómetros que recorren todos los objetos de la clase `Coche`, es un único valor, por eso se declara como `static`. Por el contrario, el atributo `kilometraje` almacena los kilómetros recorridos por un objeto concreto y tendrá un valor distinto para cada uno de ellos.

Si en el programa principal se crean 20 objetos de la clase `Coche`, cada uno tendrá su propio `kilometraje`.

A continuación se muestra el programa que prueba la clase `Coche`.

```
/**
 * PruebaCoche.java
 * Programa que prueba la clase Coche
 * @author Luis José Sánchez
 */

public class PruebaCoche {
  public static void main(String[] args) {

    Coche cocheDeLuis = new Coche("Saab", "93");
    Coche cocheDeJuan = new Coche("Toyota", "Avensis");

    cocheDeLuis.recorre(30);
    cocheDeLuis.recorre(40);
    cocheDeLuis.recorre(220);
    cocheDeJuan.recorre(60);
    cocheDeJuan.recorre(150);
    cocheDeJuan.recorre(90);
    System.out.println("El coche de Luis ha recorrido " + cocheDeLuis.getKilometraje() + "\
Km");
    System.out.println("El coche de Juan ha recorrido " + cocheDeJuan.getKilometraje() + "\
Km");
    System.out.println("El kilometraje total ha sido de " + Coche.getKilometrajeTotal() + \
"Km");
  }
}
```

El método `getKilometrajeTotal()` se aplica a la clase `Coche` por tratarse de un método de clase (método `static`). Este método no se podría aplicar a una instancia, de la misma manera que un método que no sea `static` no se puede aplicar a la clase sino a los objetos.

9.7 Interfaces

Una **interfaz** contiene únicamente la cabecera de una serie de métodos (opcionalmente también puede contener constantes). Por tanto se encarga de especificar un comportamiento que luego tendrá que ser implementado. La **interfaz** no especifica el “cómo” ya que no contiene el cuerpo de los métodos, solo el “qué”.

Una **interfaz** puede ser útil en determinadas circunstancias. En principio, separa la definición de la implementación o, como decíamos antes, el “qué” del “cómo”.

Tendremos entonces la menos dos ficheros, la **interfaz** y la clase que implementa esa **interfaz**. Se puede dar el caso que un programador escriba la **interfaz** y luego se la pase a otro programador para que sea éste último quien la implemente.

Hay que destacar que cada **interfaz** puede tener varias implementaciones asociadas.

Para ilustrar el uso de interfaces utilizaremos algunas clases ya conocidas. La superclase que va a estar por encima de todas las demás será la clase `Animal` vista con anterioridad. El código de esta clase no varía, por lo tanto no lo vamos a reproducir aquí de nuevo.

Definimos la **interfaz** `Mascota`.

```
/**
 * Mascota.java
 * Definición de la interfaz Mascota
 *
 * @author Luis José Sánchez
 */
public interface Mascota {
  String getCodigo();
  void hazRuido();
  void come(String comida);
  void peleaCon(Animal contrincante);
}
```

Como puedes ver, únicamente se escriben las cabeceras de los métodos que debe tener la/s clase/s que implemente/n la **interfaz** `Mascota`.

Una de las implementaciones de `Mascota` será `Gato`.

```
/**
 * Gato.java
 * Definición de la clase Gato
 *
 * @author Luis José Sánchez
 */
public class Gato extends Animal implements Mascota {

  private String codigo;

  public Gato (Sexo s, String c) {
    super(s);
    this.codigo = c;
  }

  @Override
```

```java
public String getCodigo() {
  return this.codigo;
}

/**
 * Hace que el gato emita sonidos.
 */
@Override
public void hazRuido() {
  this.maulla();
  this.ronronea();
}

/**
 * Hace que el gato maulle.
 */
public void maulla() {
  System.out.println("Miauuuu");
}

/**
 * Hace que el gato ronronee
 */
public void ronronea() {
  System.out.println("mrrrrrr");
}

/**
 * Hace que el gato coma.
 * A los gatos les gusta el pescado, si le damos otra comida
 * la rechazará.
 *
 * @param comida la comida que se le ofrece al gato
 */
@Override
public void come(String comida) {

  if (comida.equals("pescado")) {
    super.come();
    System.out.println("Hmmmm, gracias");
  } else {
    System.out.println("Lo siento, yo solo como pescado");
  }
}
```

```
  /**
   * Pone a pelear al gato contra otro animal.
   * Solo se van a pelear dos machos entre sí.
   *
   * @param contrincante es el animal contra el que pelear
   */
  @Override
  public void peleaCon(Animal contrincante) {
    if (this.getSexo() == Sexo.HEMBRA) {
      System.out.println("no me gusta pelear");
    } else {
      if (contrincante.getSexo() == Sexo.HEMBRA) {
        System.out.println("no peleo contra hembras");
      } else {
        System.out.println("ven aquí que te vas a enterar");
      }
    }
  }
}
```

Mediante la siguiente línea:

```
public class Gato extends Animal implements Mascota {
```

estamos diciendo que `Gato` es una subclase de `Animal` y que, además, es una implementación de la **interfaz** `Mascota`. Fíjate que no es lo mismo la herencia que la implementación.

Observa que los métodos que se indicaban en `Mascota` únicamente con la cabecera ahora están implementados completamente en `Gato`. Además, `Gato` contiene otros métodos que no se indicaban en `Mascota` como `maulla` y `ronronea`.

Los métodos de `Gato` que implementan métodos especificados en `Mascota` deben tener la anotación `@Override`.

Como dijimos anteriormente, una **interfaz** puede tener varias implementaciones. A continuación se muestra `Perro`, otra implementación de `Mascota`.

```java
/**
 * Perro.java
 * Definición de la clase Perro
 *
 * @author Luis José Sánchez
 */
public class Perro extends Animal implements Mascota {

  private String codigo;

  public Perro (Sexo s, String c) {
    super(s);
    this.codigo = c;
  }

  @Override
  public String getCodigo() {
    return this.codigo;
  }

  /**
   * Hace que el Perro emita sonidos.
   */
  @Override
  public void hazRuido() {
    this.ladra();
  }

  /**
   * Hace que el Perro ladre.
   */
  public void ladra() {
    System.out.println("Guau guau");
  }

  /**
   * Hace que el Perro coma.
   * A los Perros les gusta la carne, si le damos otra comida la rechazará.
   *
   * @param comida la comida que se le ofrece al Perro
   */
  @Override
  public void come(String comida) {

    if (comida.equals("carne")) {
```

```
      super.come();
      System.out.println("Hmmmm, gracias");
    } else {
      System.out.println("Lo siento, yo solo como carne");
    }
  }

  /**
   * Pone a pelear el perro contra otro animal.
   * Solo se van a pelear si los dos son perros.
   *
   * @param contrincante es el animal contra el que pelear
   */
  @Override
  public void peleaCon(Animal contrincante) {
    if (contrincante.getClass().getSimpleName().equals("Perro")) {
      System.out.println("ven aquí que te vas a enterar");
    } else {
      System.out.println("no me gusta pelear");
    }
  }
}
```

Por último mostramos el programa que prueba `Mascota` y sus implementaciones `Gato` y `Perro`.

```
/**
 * PruebaMascota.java
 * Programa que prueba la interfaz Mascota
 *
 * @author Luis José Sánchez
 */
public class PruebaMascota {
  public static void main(String[] args) {

    Mascota garfield = new Gato(Sexo.MACHO, "34569");
    Mascota lisa = new Gato(Sexo.HEMBRA, "96059");
    Mascota kuki = new Perro(Sexo.HEMBRA, "234678");
    Mascota ayo = new Perro(Sexo.MACHO, "778950");

    System.out.println(garfield.getCodigo());
    System.out.println(lisa.getCodigo());
    System.out.println(kuki.getCodigo());
    System.out.println(ayo.getCodigo());
    garfield.come("pescado");
```

```
    lisa.come("hamburguesa");
    kuki.come("pescado");
    lisa.peleaCon((Gato)garfield);
    ayo.peleaCon((Perro)kuki);
  }
}
```

Observa que para crear una mascota que es un gato escribimos lo siguiente:

```
Mascota garfield = new Gato(Sexo.MACHO, "34569");
```

Una **interfaz** no se puede instanciar, por tanto la siguiente línea sería incorrecta:

```
Mascota garfield = new Mascota(Sexo.MACHO, "34569");
```

Interfaces

La interfaz indica “qué” hay que hacer y la implementación especifica “cómo” se hace.

Una interfaz puede tener varias implementaciones.

Una interfaz no se puede instanciar.

La implementación puede contener métodos adicionales cuyas cabeceras no están en su interfaz.

9.8 Arrays de objetos

Del mismo modo que se pueden crear arrays de números enteros, decimales o cadenas de caracteres, también es posible crear arrays de objetos.

Vamos a definir la clase `Alumno`para luego crear un array de objetos de esta clase.

```
/**
 * Alumno.java
 * Definición de la clase Alumno
 * @author Luis José Sánchez
 */

public class Alumno {
  private String nombre;
  private double notaMedia = 0.0;

  public String getNombre() {
```

```
    return nombre;
  }

  public void setNombre(String nombre) {
    this.nombre = nombre;
  }

  public double getNotaMedia() {
    return notaMedia;
  }

  public void setNotaMedia(double notaMedia) {
    this.notaMedia = notaMedia;
  }
}
```

A continuación se define un array de cinco alumnos que posteriormente se rellena. Por último se muestran los datos de los alumnos por pantalla.

Observa que la siguiente línea únicamente define la estructura del array pero no crea los objetos:

```
Alumno[] alum = new Alumno[5];
```

Cada objeto concreto se crea de forma individual mediante

```
alum[i] = new Alumno();
```

Aquí tienes el ejemplo completo.

```
/**
 * ArrayDeAlumnosPrincipal.java
 * Programa que prueba un array de la clase Alumno
 * @author Luis José Sánchez
 */

public class ArrayDeAlumnosPrincipal {
  public static void main(String[] args) {

    // Define la estructura, un array de 5 alumnos
    // pero no crea los objetos
    Alumno[] alum = new Alumno[5];

    // Pide los datos de los alumnos ////////////////////////////////
```

```
    System.out.println("A continuacion debera introducir el nombre y la nota media de 5 al\
umnos.");

    String nombreIntroducido;
    double notaIntroducida;

    for(int i = 0; i < 5; i++) {

      alum[i] = new Alumno();

      System.out.println("Alumno " + i);

      System.out.print("Nombre: ");
      nombreIntroducido = System.console().readLine();
      (alum[i]).setNombre(nombreIntroducido);

      System.out.print("Nota media: ");
      notaIntroducida = Double.parseDouble(System.console().readLine());
      alum[i].setNotaMedia(notaIntroducida);
    } // for i

    // Muestra los datos de los alumnos ////////////////////////////////

    System.out.println("Los datos introducidos son los siguientes:");

    double  sumaDeMedias = 0;

    for(int i = 0; i < 5; i++) {
      System.out.println("Alumno " + i);
      System.out.println("Nombre: " + alum[i].getNombre());
      System.out.println("Nota media: " + alum[i].getNotaMedia());
      System.out.println("---------------------------");

      sumaDeMedias += alum[i].getNotaMedia();
    } // for i

    System.out.println("La media global de la clase es " + sumaDeMedias / 5);
  }
}
```

Veamos ahora un ejemplo algo más complejo. Se trata de una gestión típica - alta, baja, listado y modificación - de una colección de discos. Este tipo de programas se suele denominar CRUD (*Create Read Update Delete*).

Primero se define la clase `Disco`.

```java
/**
 * Disco.java
 * Definición de la clase Disco
 * @author Luis José Sánchez
 */

public class Disco {
  private String codigo = "LIBRE";
  private String autor;
  private String titulo;
  private String genero;
  private int duracion; // duración total en minutos

  public String getCodigo() {
    return codigo;
  }

  public void setCodigo(String codigo) {
    this.codigo = codigo;
  }

  public String getAutor() {
    return autor;
  }

  public void setAutor(String autor) {
    this.autor = autor;
  }

  public String getGenero() {
    return genero;
  }

  public void setGenero(String genero) {
    this.genero = genero;
  }

  public String getTitulo() {
    return titulo;
  }

  public void setTitulo(String titulo) {
    this.titulo = titulo;
  }
```

```
  public int getDuracion() {
    return duracion;
  }

  public void setDuracion(int duracion) {
    this.duracion = duracion;
  }

  public String toString() {
    String cadena = "\n----------------------------------------";
        cadena += "\nCódigo: " + this.codigo;
        cadena += "\nAutor: " + this.autor;
        cadena += "\nTítulo: " + this.titulo;
        cadena += "\nGénero: " + this.genero;
        cadena += "\nDuración: " + this.duracion;
        cadena += "\n----------------------------------------";

    return cadena;
  }
}
```

A continuación se crea el programa principal. Cada elemento de la colección será un objeto de la clase `Disco`. Se trata, como verás, de una gestión muy simple pero totalmente funcional.

No se incluye en el manual porque ocuparía varias páginas. El código del programa se puede descargar desde el siguiente enlace: https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/blob/master/ejemplos/09_POO/ColeccionDeDiscos/ColeccionDeDiscosPrincipal.java