```java
/**
 * 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.

```java
/**
 * 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`.

```
/**
 * 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");
  }
}
```

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`.

```
/**
 * 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.

```
/**
 * 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`.

```
/**
 * 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`.

```java
/**
 * 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
  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`.

```
/**
 * 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.