Técnico en

DESARROLLO DE SOFTWARE

Programación Orientada a Objetos I

Programación Orientada a Objetos I

Unidad II

1. Herencia

Nos permite definir nuevas clases a partir de otras ya existentes. Las clases "hijas" van a heredar los atributos y métodos de la clase padre. La característica de las clases *hijas* es que no solo tendrán la funcionalidad y propiedades de la clase padre, sino que pueden agregarle características particulares propias de la nueva clase. Es decir, que la clase hija puede tener nuevos atributos y métodos de la clase padre y/o modificar el comportamiento de los métodos ya existentes en la clase padre.

Ejemplo

Tenemos a los animales, los cuales tienen muchas propiedades, entre las cuales podemos mencionar que tienen un color (o mezcla de colores), poseen una dieta específica, deberían de tener un peso ideal, tienen un tiempo estimado de vida, una edad actual, entre otros. Puede que tengan más características, pero por cuestiones de este ejemplo utilizaremos solo estas propiedades. Además tienen distintas acciones que llevan a cabo en general independientemente de la forma en que los clasifiquemos, algunas de las actividades son que emiten sonidos, comen, se movilizan, forma de alimentarse mientras son bebés o incluso el proceso de gestación.

Los animales pueden clasificarse de distintas formas, en este caso los clasificaremos por el tipo de medio en el que viven o interactúan. La clasificación es la siguiente:

- Marinos
- Aéreos
- Terrestres

También podrían existir más clasificaciones, pero por cuestiones de ejemplo colocaremos estas.

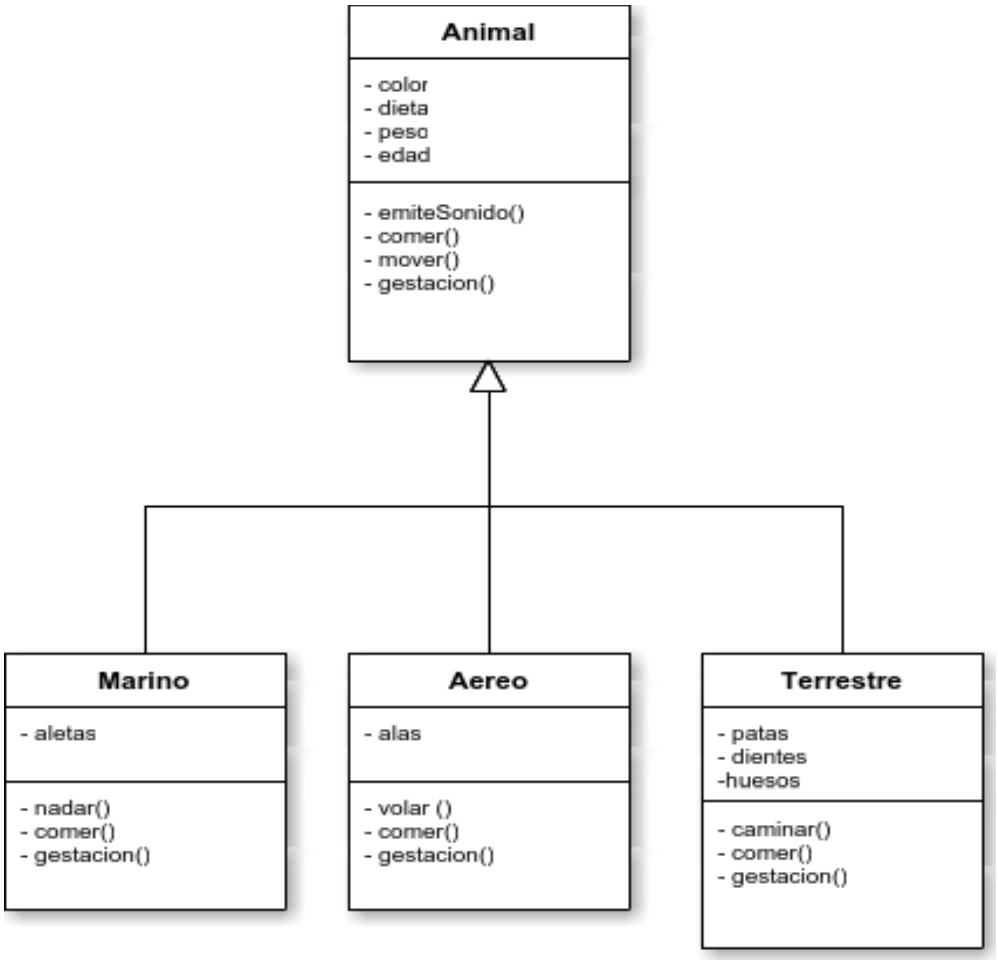
Para nuestro ejemplo, los distintos tipos de animales tienen las mismas propiedades que un animal en general (color, dieta, peso, edad).

Los animales marinos heredan en general de los animales las mismas características y puede que tengan más, para el caso de los animales agregaremos otras características tales como las aletas, y vamos a agregar acciones que no todos los animales realizan tales como: nadar, la forma en que se alimentan y la forma en que se reproducen o proceso de gestación.

Por otro lado, los animales aéreos se caracterizan porque tienen dos alas que les permite movilizarse, por lo que agregaremos una acción de volar, la forma en que se alimentan y la forma en que se reproducen o de gestación.

Por último tenemos a los animales terrestres, los cuales en su mayoría tienen patas, dientes, huesos y se movilizan con sus pies, tienen una forma de alimentarse y reproducirse de forma distinta a la de las otras dos clasificaciones.

A continuación mostraremos el diagrama de clases para que ustedes entiendan cómo funciona la herencia en este ejemplo:



Con esto podemos indicar que todas las clases hijas heredan los atributos y métodos de la clase padre, pero las clases hijas tendrán atributos más específicos y métodos más especializados que los de su clase padre.

2. Representación en Java

Para poder indicar que nuestra clase actual es hija de otra clase, debemos colocar la palabra reservada ***extends*** y luego el nombre de la clase de donde hereda los atributos

y métodos. Al momento de colocar la sentencia "public class Poodle extends Perro" estamos indicando que la clase Poodle tiene definido automáticamente, y sin necesidad de colocar manualmente ningún otro dato más, todos los métodos y atributos de la clase padre.

La palabra reservada ***super*** nos sirve para mandar a llamar a los atributos de nuestra clase padre. Si colocamos la palabra super.nombre entonces mandamos a llamar al atributo "nombre" de nuestra clase padre. Si colocamos la palabra super.getNombre() entonces estamos mandando a llamar al método getNombre() de nuestra clase padre y si mandamos a llamar a la palabra super() entonces estamos mandando a llamar al constructor de nuestra clase padre. Debemos de recordar que si los métodos o constructores tienen parámetros, entonces debemos de enviarlos al momento de mandar a llamar ya sea al constructor o al método.

A la clase padre se le llama ***superclase*** y a la clase hijo se le llama ***subclase***.

Ejemplo:

Crearemos la clase *Animal*, la cual está definida de la siguiente forma:

```
/*
*  Clase Animal
*/
public class Animal {
  String nombre, color, raza;
```

```
  public Animal(String  ingNombre, String ingColor, String ingRaza) {
    nombre = ingNombre;
    color = ingColor;
    raza = ingRaza;
  }
  public void setNombre(String name) {
        nombre = name;
  }
  public String getNombre() {
        return nombre;
  }
}
```

Esta clase tiene los parámetros nombre, color y raza los cuales son cadenas de caracteres. Además tiene dos métodos los cuales son ingresar y devolver el nombre del animal. De la clase Animal sacamos dos clases hijas: Perro y Gato.

El objeto Perro tiene todas las características y métodos del objeto Animal, pero además tiene dos acciones nuevas: ladra y aúlla. A continuación la clase Perro:

```
/*
*  Clase Perro
*/
public class Perro extends Animal {
  public Perro(String ingNombre, String ingColor, String ingRaza) {
     super(ingNombre, ingColor, ingRaza);
  }
        public void ladrar() {
                System.out.println("Woof");
        }
        public void aullar() {
                System.out.println("Auuuuu");
        }
}
```

Además tenemos al objeto Gato el cual también tiene todas las características y métodos del objeto Animal, pero además tiene dos acciones nuevas: maullar y ronronear.

```
/*
*  Clase Gato
*/
public class Gato extends Animal{
```

```
public Gato(String ingNombre, String ingColor, String ingRaza) {
   super(ingNombre, ingColor, ingRaza);
 }
        public void maullar() {
                System.out.println("Miau");
        }
        public void ronronear() {
                System.out.println("Rrrrrr");
        }
}
```

El objeto Perro no maúlla ni ronronea, por lo que estos dos métodos no pueden estar en la clase Perro, y es por eso que se deben detallar en la clase Gato. Lo mismo es con la clase Gato, el objeto Gato no ladra ni aúlla y por lo mismo estos dos métodos no pueden estar en la clase Animal ni en la clase Gato, por lo que están en la clase Perro.

Como se puede ver en el ejemplo, los constructores de las clases Perro y Gato mandan a llamar al constructor del objeto Animal con los parámetros que los tres tienen. Si la clase Perro o Gato tuviera un constructor con distintos parámetros (más o menos parámetros), podría mandar a llamar al constructor de la clase Animal siempre y cuando tuviera todos los parámetros que el constructor de la clase Animal solicita.

A continuación vamos a crear una clase para probar las tres clases que creamos con anterioridad. Tomando en cuenta que nuestra clase Humano únicamente tiene el método main y va a tener a dos animales: un Perro y un Gato:

```
/*
*  Clase Humano
*/
public class Humano {
  public static void main(String[] args) {
        Perro canelo = new Perro("Canelo", "cafe", "French Poodle");
        Gato penny = new Gato("Penny", "negro", "Mestiza");
        System.out.println(canelo.getNombre());
        canelo.ladrar();
        canelo.aullar();
        System.out.println("----------------------------");
        System.out.println(penny.getNombre());
        penny.maullar();
        penny.ronronear();
        System.out.println("----------------------------");
  }
}
```

Al ejecutarlo nos mostrará lo siguiente:

```
Canelo
Guau
Auuuuu
```

```
Penny

Miau

rrrrrr

----------------------------
```

3. Polimorfismo

En la programación orientada a objetos, las subclases de una clase pueden definir comportamientos propios y compartir cierta funcionalidad de sus clases padre. Para poder mostrar el ***Polimorfirsmo*** en Java, vamos a utilizar la clase ***Animal*** y sus clases hijas ***Perro*** y ***Gato***.

Ejemplo

Tomando en clase nuestra clase Animal, la cual tiene dos métodos: darNombre() y emitirSonido().

```
/*
*  Clase Animal
*/
public class Animal {
  String nombre, raza;
  public Animal(String  ingNombre, String ingRaza) {
     nombre = ingNombre;
     raza = ingRaza;
```

```
 }
  public String darNombre() {
        return nombre;
 }
  public void emitirSonido() {
        System.out.println("Sonido");
  }
}
```

Vamos a crear dos clases nuevas que van a heredar de nuestra clase ***Animal***, las cuales llamaremos ***Perro*** y ***Gato***. Estas también tendrán los métodos darNombre() y emitirSonido() los cuales son heredados de su clase padre ***Animal***.

Si queremos definir una acción distinta en los métodos emitirSonido() de las clases hijas, lo podemos hacer mediante la modificación de dichos métodos en sus respectivas clases (***Perro*** y ***Gato***):

```
/*
*  Clase Perro
*/
public class Perro extends Animal {
  public Perro(String ingNombre, String ingRaza) {
     super(ingNombre, ingRaza);
  }
```

```
  public void emitirSonido() {
        System.out.println("Woof");
  }
}
```

```
/*
*  Clase Gato
*/
public class Gato extends Animal {
  public Gato(String ingNombre, String ingRaza) {
     super(ingNombre, ingRaza);
  }
   public void emitirSonido() {
        System.out.println("Miau");
   }
}
```

A continuación, vamos a crear una clase para probar las dos clases hijas que creamos con anterioridad. Tomando en cuenta que nuestra clase Humano únicamente tiene el método main y va a tener a dos animales: un Perro y un Gato.

```
/*
*  Clase Humano
*/
public class Humano {
  public static void main(String[] args) {
        Perro canelo = new Perro("Maggie", "Beagle");
        Gato penny = new Gato("Penny", "Angora");
        System.out.println(canelo.darNombre());
        canelo.emitirSonido();
        System.out.println("----------------------------");
        System.out.println(penny.darNombre());
        penny.emitirSonido();
        System.out.println("----------------------------");
  }
}
```

Al ejecutarlo nos mostrará lo siguiente:

```
Maggie
Woof
----------------------------
Penny
Miau
----------------------------
```

4. Encapsulamiento

Como vimos en la unidad anterior, es una propiedad que ayuda a los atributos o propiedades y los métodos o funciones a mantenerse juntos en una única entidad que definen el comportamiento del objeto. El encapsulamiento nos permite definir el nivel de acceso de nuestras propiedades y métodos. También nos permite evitar que cualquiera pueda modificar los atributos del objeto de forma directa, a menos que lo lleve a cabo mediante métodos.

Estas propiedades son:

Public

Los atributos del objeto se pueden modificar en cualquier clase, en cualquier programa.

Ejemplo

Tomando en cuenta nuestra clase Carro:

```
/*
*  Clase Carro
*/


public class Carro {
        String marca, linea, color;
        public Carro(String nuevaMarca, String nuevaLinea, String nuevoColor)
{
```

```
            marca = nuevaMarca;
            linea = nuevaLinea;
            color = nuevoColor;
        }
        public void cambiarColor(String nuevoColor) {
            color = nuevoColor;
        }
        public String obtenerColor() {
            return color;
        }
        public void imprimirColor() {
            System.out.println(color);
        }
}
```

Al mandar a llamar al objeto Carro en nuestro método main que se encuentra en nuestra clase MainCarro:

```
/*
*  Clase MainCarro
*/
public class MainCarro {
        public static void main(String[] args) {
            Carro automovil = new Carro("Mazda", "323", "Verde");
```

```
            automovil.imprimirColor();
            System.out.println(automovil.color);
            automovil.cambiarColor("azul");
            automovil.imprimirColor();
            System.out.println(automovil.color);
        }
}
```

Nos damos cuenta que obtenemos el mismo resultado al mandar a llamar al método imprimirColor() que ejecutar System.out.println del objeto automóvil mandando a llamar a su propiedad color de la forma automovil.color. Esto se da porque nuestras propiedades de la clase Carro son públicas, si éstas fueran privadas ya no se daría el caso.

Protected

Los atributos del objeto se pueden modificar desde cualquier clase del mismo paquete, pero no en otras clases de otros paquetes. Un atributo o método protected es visto por la clase y por todas sus subclase, pero no por otras clases que no pertenezcan al paquete. Estaremos viendo más a detalle este tipo cuando aprendamos a implementar paquetes en el próximo curso.

Private

Los atributos del objeto se pueden obtener y modificar únicamente en su clase donde se encuentran definidos. Un atributo private no es visible para ninguna clase, ni siquiera para las subclases.

Ejemplo

Tomando en cuenta nuestra clase Carro:

```
/*
*  Clase Carro
*/
public class Carro {
        private String marca, linea, color;
        public Carro(String nuevaMarca, String nuevaLinea, String nuevoColor)
{
                marca = nuevaMarca;
                linea = nuevaLinea;
                color = nuevoColor;
        }
        public void cambiarColor(String nuevoColor) {
                color = nuevoColor;
        }
        public String obtenerColor() {
```

```
            return color;
        }
        public void imprimirColor() {
            System.out.println(color);
        }
}
```

Notar que los atributos son *private*. Al mandar a llamar al mismo objeto Carro en nuestro método main que se encuentra en nuestra clase MainCarro:

```
/*
*  Clase MainCarro
*/
public class MainCarro {
        public static void main(String[] args) {
            Carro automovil = new Carro("Mazda", "323", "Verde");
            automovil.imprimirColor();
            System.out.println(automovil.color);
            automovil.cambiarColor("azul");
            automovil.imprimirColor();
            System.out.println(automovil.color);
        }
}
```

Nos devuelve un error:

```
Verde

Exception in thread "main" java.lang.IllegalAccessError: tried to access field Carro.color from class MainCarro

        at MainCarro.main(MainCarro.java:10)
```

Esto es porque los atributos son *private*, por lo que en este caso ya solo deberíamos de utilizar los métodos para obtener la información y ya no como lo hicimos en el caso de los atributos *public*. Nuestra clase MainCarro debería de quedar de la siguiente forma:

```
/*
*  Clase MainCarro
*/
public class MainCarro {
        public static void main(String[] args) {
                Carro automovil = new Carro("Mazda", "323", "Verde");
                automovil.imprimirColor();
                automovil.cambiarColor("azul");
                automovil.imprimirColor();
        }
}
```

5. Manejo de control de excepciones

Este tema lo trataremos específicamente para Java. Dado que Java es un lenguaje de programación compilado, en este lenguaje pueden existir errores en tiempo de

compilación. También pueden existir errores en tiempo de ejecución; a estos últimos los llamaremos *excepciones* o *exceptions*.

Ejemplo de error en tiempo de compilación

```
javac MainCarro.java
MainCarro.java:13: error: ';' expected
                System.out.println(automovil.marca)
                                                   ^
1 error
```

Ejemplo de error en tiempo de ejecución

Al ejecutar la siguiente instrucción en nuestra clase MainCarro:

```
System.out.println(Double.parseDouble(automovil.marca));
```

Obtenemos el siguiente error porque lo que tenemos almacenado en automovil.marca no es un dato tipo Double sino que String:

```
java MainCarro
Verde
azul
Mazda
Exception in thread "main" java.lang.NumberFormatException: For input string: "Mazda"
        at
```

```
sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
        at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
        at java.lang.Double.parseDouble(Double.java:538)
        at MainCarro.main(MainCarro.java:13)
```

Cada vez que ocurre un error en tiempo de ejecución, se levanta una excepción, muestra el error y finaliza la ejecución del programa.

Java nos provee una forma de manejar las excepciones en tiempo de ejecución para evitar que éstas finalicen la ejecución del programa.

Ejemplo

```
/*
*  Clase MainCarro
*/
public class MainCarro {
        public static void main(String[] args) {
                Carro automovil = new Carro("Mazda", "323", "Verde");
                automovil.imprimirColor();
                automovil.cambiarColor("azul");
                automovil.imprimirColor();
                automovil.imprimirMarca();
                try {
```

```
        System.out.println(Double.parseDouble(automovil.marca));
                } catch (NumberFormatException nfe) {
                        System.out.println("No es un numero de tipo Double");
                }
        }
}
```

En este caso colocamos en el catch la excepción NumberFormatException, ya que al momento de ejecutar nuestra clase sin el try...catch nos daba el error “Exception in thread "main" java.lang.**NumberFormatException**”, por lo que el error que queremos manejar es el de NumberFormatException. Así como este puede haber muchas excepciones más y se puede manejar más de una a la vez.

Si nosotros queremos manejar nuestras propias excepciones, es necesario que primero definamos nuestras excepciones para poderlas manejar. A continuación un ejemplo para manejar la división entre 0 con una excepción creada por nosotros.

Ejemplo

```
class DivisionByZeroException extends Exception {
        String message;

        public DivisionByZeroException() {
                message = "El denominador no puede ser cero";
        }
```

```
}

public class Division {

        public double dividir(double numerador, double denominador) {

           try {

                if (denominador == 0) {

                        throw new DivisionByZeroException();

                }

           } catch (DivisionByZeroException error1) {

                System.out.println(error1);

           }


           return (numerador/denominador);

        }

        public static void main(String[] args) {

                Division numeroDividido = new Division();

                double numerador = 1;

                double denominador = 100;


        System.out.println(numeroDividido.dividir(numerador,denominador));

                numerador = 8;

                denominador = 4;


        System.out.println(numeroDividido.dividir(numerador,denominador));
```

```
            numerador = 1;
            denominador = 0;

        System.out.println(numeroDividido.dividir(numerador,denominador));
        }
}
```

La clase DivisionByZeroException tiene como padre a la clase Exception la cual hereda los atributos y métodos de Exception, pero nosotros vamos a colocar nuestro mensaje personalizado al momento en que ocurra esta excepción. Para mandar a llamar nuestro manejo de excepción utilizaremos “throw new NombreExcepcion(parametros);” dentro del try y en el catch mandamos a llamar a nuestro tipo de excepción. En el manejo del catch se colocó una impresión a pantalla con el error que nuestra excepción devuelve al ser generada.

Referencias

- API de Java: https://docs.oracle.com/javase/7/docs/api/
- Java in a Nutshell - Benjamin J. Evans & David Flanagan

Descargo de responsabilidad

La información contenida en este documento descargable en formato PDF o PPT es un reflejo del material virtual presentado en la versión online del curso. Por lo tanto, su contenido, gráficos, links de consulta, acotaciones y comentarios son responsabilidad exclusiva de su(s) respectivo(s) autor(es) por lo que su contenido no compromete al área de e-Learning del Departamento GES o al programa académico al que pertenece.

El área de e-Learning no asume ninguna responsabilidad por la actualidad, exactitud, obligaciones de derechos de autor, integridad o calidad de los contenidos proporcionados y se aclara que la utilización de este descargable se encuentra limitada de manera expresa para los propósitos educacionales del curso.