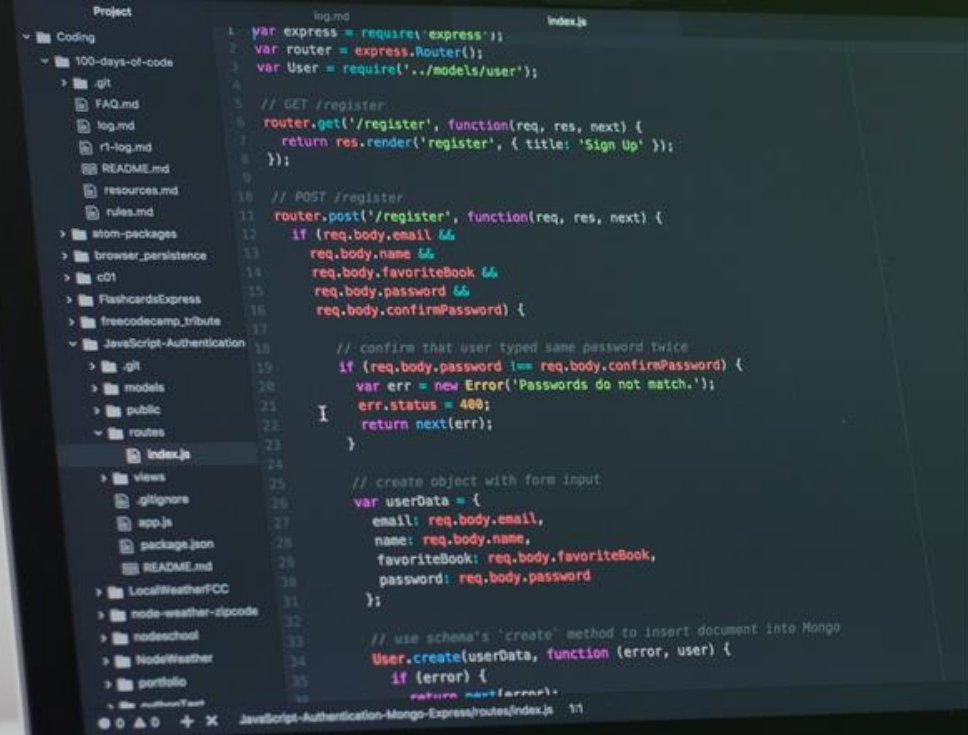




El futuro digital
es de todos

MinTIC

RELACIONES ENTRE CLASES



Universidad
Industrial de
Santander

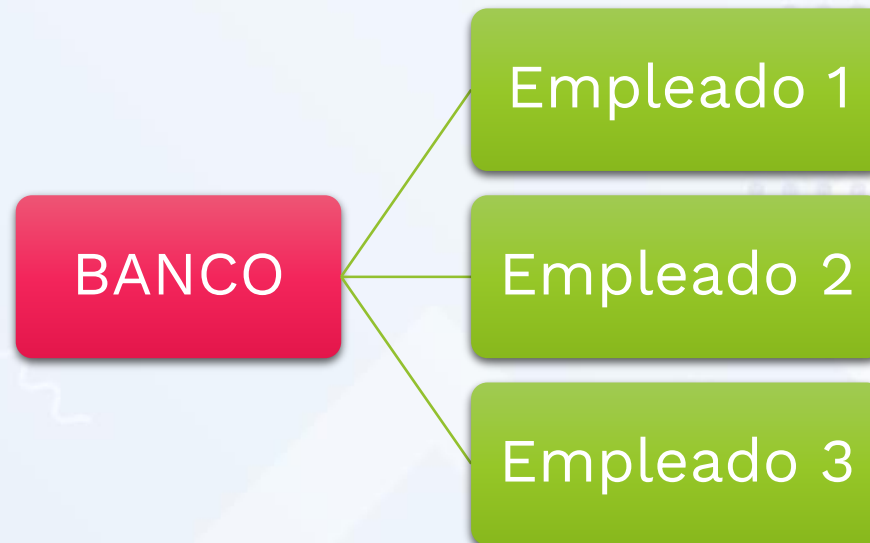


Misión
TIC 2022

3.1. Asociación, agregación y composición

3.1.1. Asociación

La asociación es la primera relación entre clases que veremos. Es la relación primaria que se tiene entre diferentes clases, es decir, la relación de asociación es la base para las demás relaciones entre clases. Una forma sencilla de describir la asociación sería la siguiente: Es la conexión o relación entre, al menos, dos clases separadas y diferentes, la cual se da a través de sus objetos. En la siguiente imagen vemos la relación que tiene, por ejemplo, un banco y sus empleados.



Esta relación es la más abstracta, aunque la más sencilla de entender, porque, en últimas, cuando nos referimos a la asociación estamos haciendo referencia a una relación entre diferentes clases. Esta relación puede darse de dos formas similares, pero con características específicas: la agregación y composición. Esto significa que la asociación engloba dichas relaciones específicas, como vemos en la siguiente imagen:



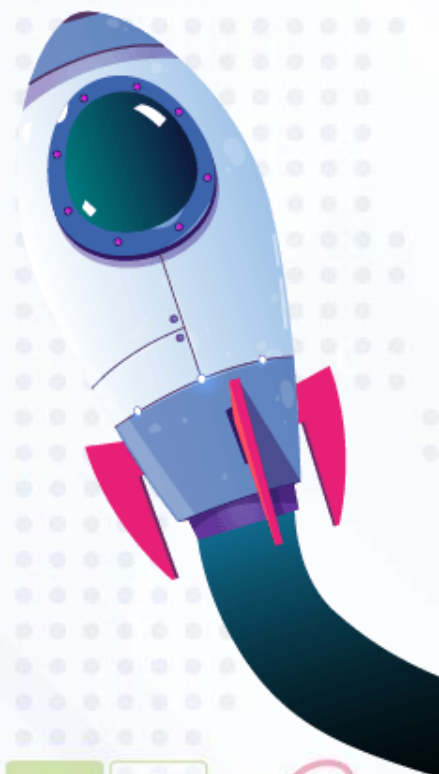
Más adelante, veremos cómo representamos la asociación en UML de forma detallada. Antes de pasar a ver ejemplos en código, es ideal primero entender en qué consisten las sub-relaciones presentadas.

3.1.2. Agregación

La agregación es la relación entre diferentes clases que se da con más frecuencia , esta tiene un concepto sencillo, consiste en, como dice su nombre, agregar objetos a una estructura que nos almacene objetos del mismo tipo, por ejemplo, un arreglo (Array).

Puede sonar un poco simple y confuso al mismo tiempo, pero veremos con los siguientes ejemplos lo fácil que es entender este concepto. Habíamos hablado en la sección anterior sobre la relación de asociación entre el banco y sus empleados, ¿recuerdas? En este caso, vamos a ser más específicos con la relación que ocurre en este caso.

Cuando nosotros pensamos en el banco como una entidad, sabemos que hay empleados trabajando para esta, es decir, hay una relación entre estos dos tipos. La forma en la que se representa es que nosotros agregamos empleados al banco, no al revés. El objeto instanciado del banco almacenará la información de sus empleados en una estructura de almacenamiento de datos, por ejemplo, un arreglo, de tal forma que, a través del banco, nosotros conozcamos cuántos y cuáles son los empleados que trabajan ahí.



Ahora sí veamos código. Lo primero que veremos será la clase Empleado, la cual generaliza a grandes rasgos, las características básicas de los empleados de un banco.

```
public class Empleado {  
    private String nombre;  
    private String apellido;  
    // etc  
    public Empleado(String nombre, String apellido) {  
        this.nombre = nombre;  
        this.apellido = apellido;  
    }  
}
```

Ahora, el código del banco. Vemos que el banco tiene un arreglo llamado empleados, de tipo Empleado. Esto quiere decir que ahí almacenaremos los empleados que trabajen para el banco.

```
public class Banco {  
    private String nombre;  
    private Empleado empleados[];  
    // etc  
    public Banco(String nombre, Empleado empleados[]) {  
        this.nombre = nombre;  
        this.empleados = empleados;  
    }  
  
    public void agregarEmpleado(Empleado nuevoEmpleado, int index) {  
        this.empleados[index] = nuevoEmpleado;  
        System.out.println(";Empleado agregado!");  
    }  
}
```

Finalmente, veamos cómo funciona esta relación de agregación en ejecución.

```
public class Main {  
    public static void main(String[] args) throws Exception {  
        Empleado emp1 = new Empleado("Jorge", "Macías");  
        Empleado emp2 = new Empleado("Ángela", "Cucalón");  
        Empleado emp3 = new Empleado("Irma", "Caceres");  
        // Array que almacena hasta 200 empleados  
        Empleado listaEmpleados[] = new Empleado[200];  
        listaEmpleados[0] = emp1;  
        listaEmpleados[1] = emp2;  
        Banco banco = new Banco("Bancolombia", listaEmpleados);  
        banco.agregarEmpleado(emp3, 2);  
    }  
}
```



3.1.2.1. Representación en UML de la agregación

La forma en la que la agregación se representa en un diagrama UML es la siguiente:



Notemos que usamos un pequeño rombo en el extremo que apunta hacia la clase a la cual agregamos. En este punto es importante resaltar una característica crucial de la agregación: Los objetos que nosotros agregamos pueden existir sin la necesidad de que exista la asociación entre ellos y el otro tipo de objeto, es decir, y haciendo uso de un ejemplo, los empleados pueden existir o vivir perfectamente sin necesidad de que exista el banco. Otra forma de entender este concepto es que, por ejemplo, si el banco deja de existir, los empleados pueden seguir existiendo.

3.1.3. Composición

La relación de composición es muy similar a la de agregación, de hecho, consisten en lo mismo, no obstante, tienen una diferencia crucial: a diferencia de la agregación, en la relación de composición, los objetos agregados **SÍ** dejan de existir si el objeto que los almacena deja de existir, porque ya no tendría sentido lógico mantenerlos.

Lo anterior puede sonar un poco confuso, pero veremos que no es tan complejo o extraño como parece. Por ejemplo, pensemos en un automóvil, sabemos que el automóvil está **compuesto** por varias piezas mecánicas, como el motor, las ruedas, entre otras. Tomemos las ruedas como objetos que pertenecen a nuestro automóvil. Si nuestro automóvil dejará de existir, **no** tendría mucho sentido para nosotros conservar todas las ruedas que este tenía, esto mismo se podría decir del motor, parabrisas, etc.

La relación de composición, como quizá puede que estés pensando, es muy dependiente del contexto. Esto es totalmente cierto, puede que haya ocasiones en las que a nosotros sí nos interese conservar los objetos que hacían parte de otro; en dado caso, la relación ya no sería de composición, sino de agregación.

En resumen, la relación de composición es una forma de la relación de agregación, pero donde los objetos que **nosotros agregamos, están destinados a dejar de existir** si su objeto padre deja de existir. A esa clase de objetos que van a cumplir ese propósito los llamaríamos componentes. Lo difícil de entender de esta relación es la forma en la que lo implementamos en el código, ya que tiene una forma especial de hacerlo. Esto es debido a que, al ser similar a la agregación, podríamos estar cometiendo el error de estar implementando esta última en lugar de una composición. Es por esto que la relación de composición debe seguir cierto patrón para que realmente la estemos aplicando.

A continuación, veremos una forma de hacerlo (aunque no es la única).

La estrategia que usaremos para aplicar esta relación será la siguiente: nosotros añadiremos e interactuaremos con los componentes a través del objeto que es compuesto. De esta forma, estaremos previniendo la manipulación directa de los componentes, además, es un poco más sencillo de entender. Empezaremos con una clase sencilla llamada Rueda. Esta clase tendrá las características generales de una rueda para automóviles:

```
public class Rueda {  
    private int radio;  
    private int ancho;  
    // etc  
    public Rueda(int radio, int ancho) {  
        this.radio = radio;  
        this.ancho = ancho;  
    }  
}
```

Ahora, veremos la clase Automóvil. Vemos que en el constructor estamos pidiendo que se nos indique el número de ejes que tendrá el automóvil, esto quiere decir que, si instanciamos un objeto Automóvil con cuatro ejes, tendrá cuatro ruedas.

```
public class Automovil {  
    private String modelo;  
    private Rueda ruedas[];  
    private int radioRuedas;  
    private int anchoRuedas;  
    private double cilindraje;  
  
    public Automovil(String modelo, int nRuedas, int radioRuedas, int anchoRuedas, double cilindraje){  
        this.modelo = modelo;  
        this.radioRuedas = radioRuedas;  
        this.anchoRuedas = anchoRuedas;  
        this.cilindraje = cilindraje;  
        this.ruedas = new Rueda[nRuedas];  
        for (int i = 0; i < nRuedas; i++){  
            this.ruedas[i] = new Rueda(this.radioRuedas, this.anchoRuedas);  
        }  
    }  
  
    public int getNumeroRuedas() {  
        return this.ruedas.length;  
    }  
}
```



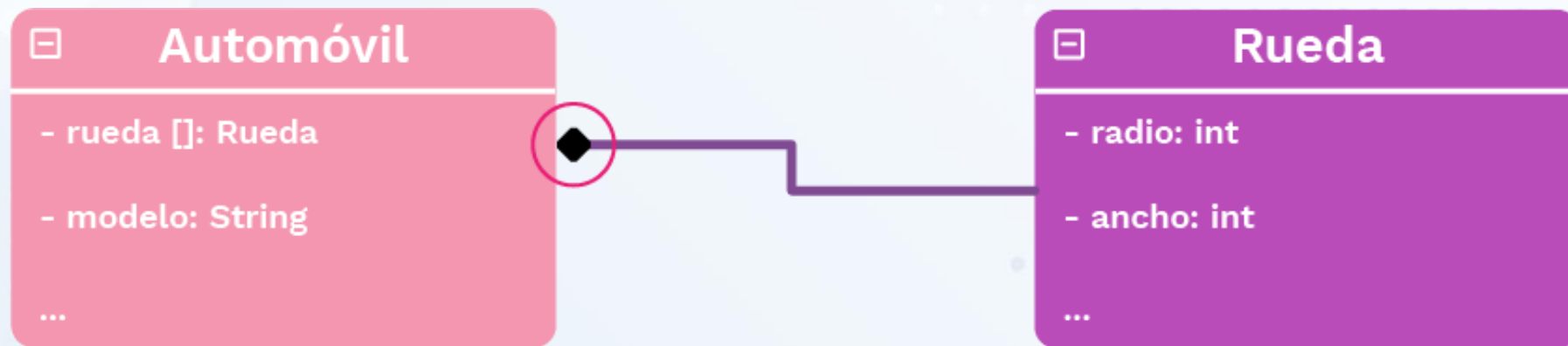
Notemos que la clase Automóvil tiene algunos métodos para interactuar con las ruedas, por ejemplo, el que se llama `getNumeroRuedas()`, el cual nos devuelve el número de ruedas que tiene nuestro objeto de tipo automóvil instaladas.

Finalmente, veamos el código en ejecución:

```
public class Main {  
    public static void main(String[] args) throws Exception {  
        Automovil carro = new Automovil("BMW Z3", 4, 8, 225, 2793);  
        System.out.println(carro.getNumeroRuedas());  
    }  
}
```


3.1.3.1. Representación en UML de la composición

La forma en la que se representa la relación de composición en UML es similar a la de agregación, con la salvedad de que el rombo está coloreado (generalmente de negro).



3.2. Cardinalidad

Ahora que conocemos las relaciones de asociación entre clases, podemos hablar de un concepto muy importante a la hora de modelar y trabajar en la programación orientada a objetos: la cardinalidad, concepto sencillo que consiste en indicarnos un rango de valores que un objeto, dada una relación de asociación, contendrá de otro tipo de objeto. En otras palabras, la cardinalidad, o también llamada multiplicidad, indica cuántas instancias de una clase pueden surgir, fruto de la relación con otra clase.

La cardinalidad se representa de 3 formas base diferentes, dependiendo del caso:

- n
- n .. m
- n .. *

Veamos qué hace cada uno de estos casos.

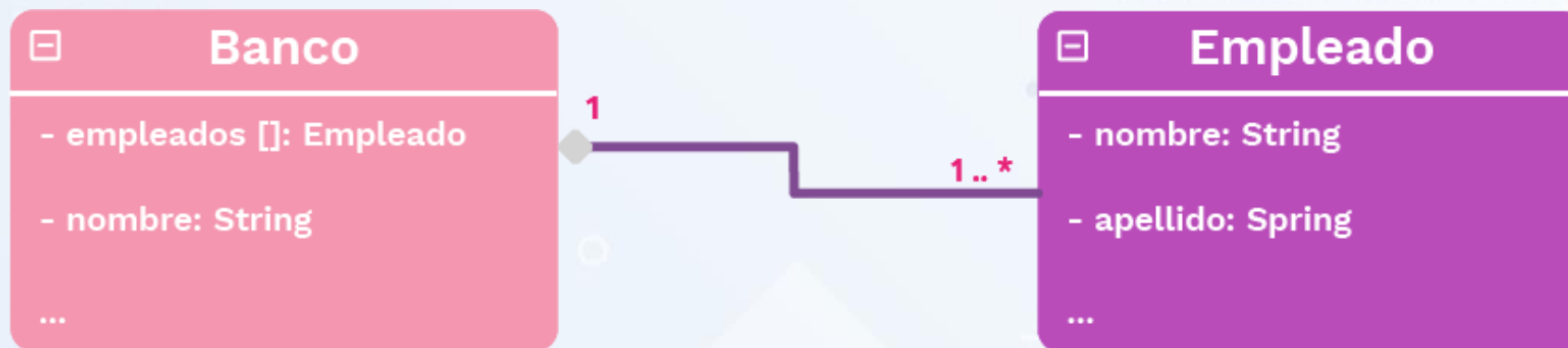
El primer caso (n), nos sirve para indicar, mediante un número entero cuyo mínimo es cero, el número máximo de instancias que se puede tener en ese extremo de la relación.

El segundo caso ($n \dots m$) funciona de forma similar al anterior, la diferencia radica en que el primer valor, la n , nos indica el mínimo posible, mientras que la m , el máximo. Es decir, este caso nos sirve para establecer un rango de valores.

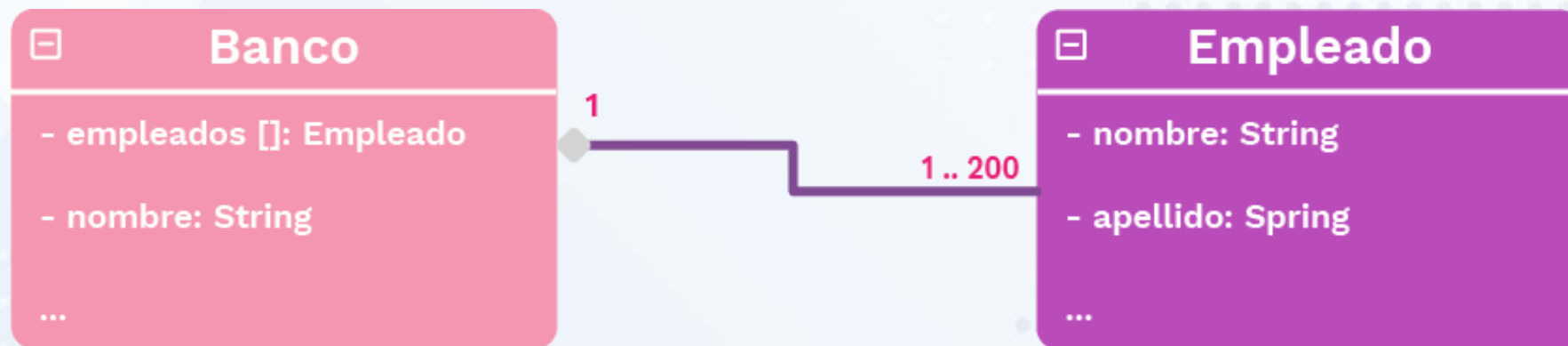
El tercer caso ($n \dots *$) nos sirve para establecer un mínimo de valores, con un máximo desconocido, es decir, que no tiene límite.

Estos casos son más sencillos de entender con ejemplos:

Recordemos la relación de asociación que teníamos entre el banco y los empleados. Nosotros mediante la cardinalidad ahora podemos, a través del diagrama UML, especificar la cantidad de instancias posibles que hay a cada lado de la relación. Como se puede ver en la siguiente imagen, estamos especificando que para un (1) banco podemos tener como mínimo un (1) empleado y no tenemos un máximo:

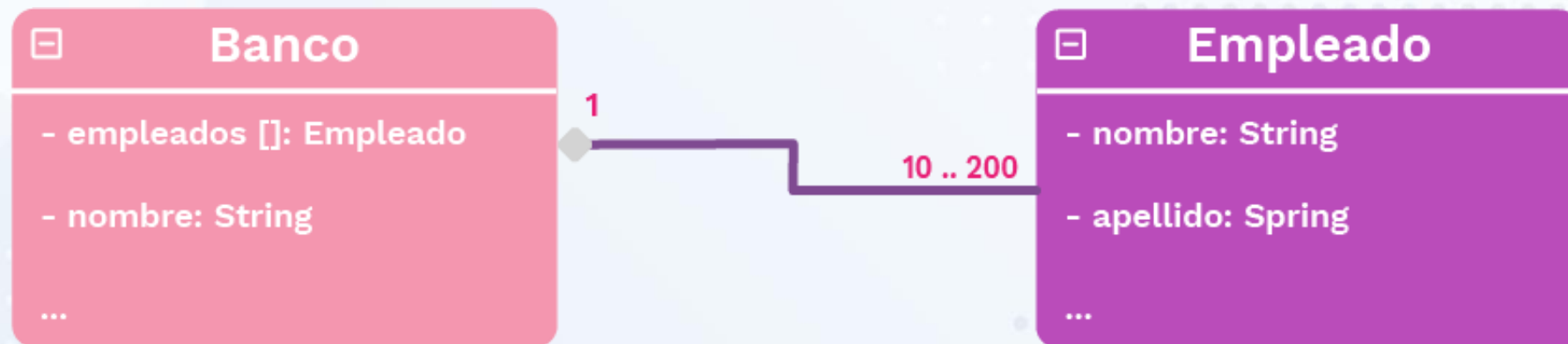


Siguiendo este orden de ideas, también podríamos decir que, como máximo, un banco pueda tener 200 empleados:



Notemos que, si pusiéramos cero como número mínimo de empleados asociados a un banco, estaríamos, mediante el diagrama UML, dando a entender que nuestro banco puede funcionar sin empleados, caso que no tiene mucho sentido, ¿verdad?

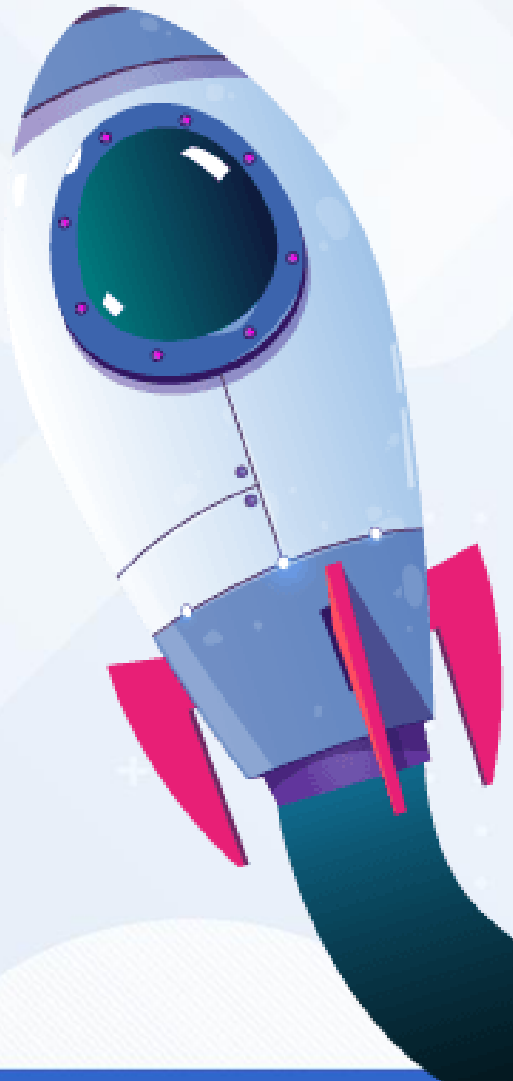
A continuación, otro ejemplo, donde como mínimo decimos que un banco debe de tener mínimo 10 empleados y máximo 200 para funcionar:



3.3. Herencia

La herencia es la piedra angular de la programación orientada a objetos. Este es un tipo de relación entre clases que, a diferencia de la asociación (y sus tipos), no trabaja estrictamente con clases diferentes. De hecho, trabaja con algo que llamaremos “clases hijas” y “clases padre” o, “subclases” y “superclases”.

Primero que todo, seguramente ustedes estarán pensando “bueno sí, bonita terminología y todo, pero... ¿Qué es herencia?” Ya vamos a ello. La herencia se puede definir como el poder o capacidad de crear clases que, de forma automática, tomen las características (atributos y métodos) de otra clase que exista previamente, donde nosotros, basados en una lógica, elegimos con un fin. A la nueva clase creada le podremos añadir nuevos atributos y métodos propios.

A stylized illustration of a rocket launching. The rocket is white with blue and red accents. It has a large blue circular window at the top. Red flames are coming out of the bottom, and a thick black smoke trail is following it. The background is light blue with white geometric shapes and a grid of dots.

Quizá se estarán preguntando “Interesante, ¿para qué sirve o qué diferencia tiene de yo crear otra clase para posteriormente copiar y pegar el código que necesito dentro de ella?”. Bueno, ya veremos las enormes ventajas que conlleva implementar la herencia y entender bien el concepto.

La principal ventaja que nos ofrece la herencia es la **reutilización del código**. Con la herencia, nosotros podemos evitar por completo el mal hábito de copiar y pegar código de otras clases, junto con los problemas que esto conlleva. Elaboremos más sobre esto con un ejemplo.

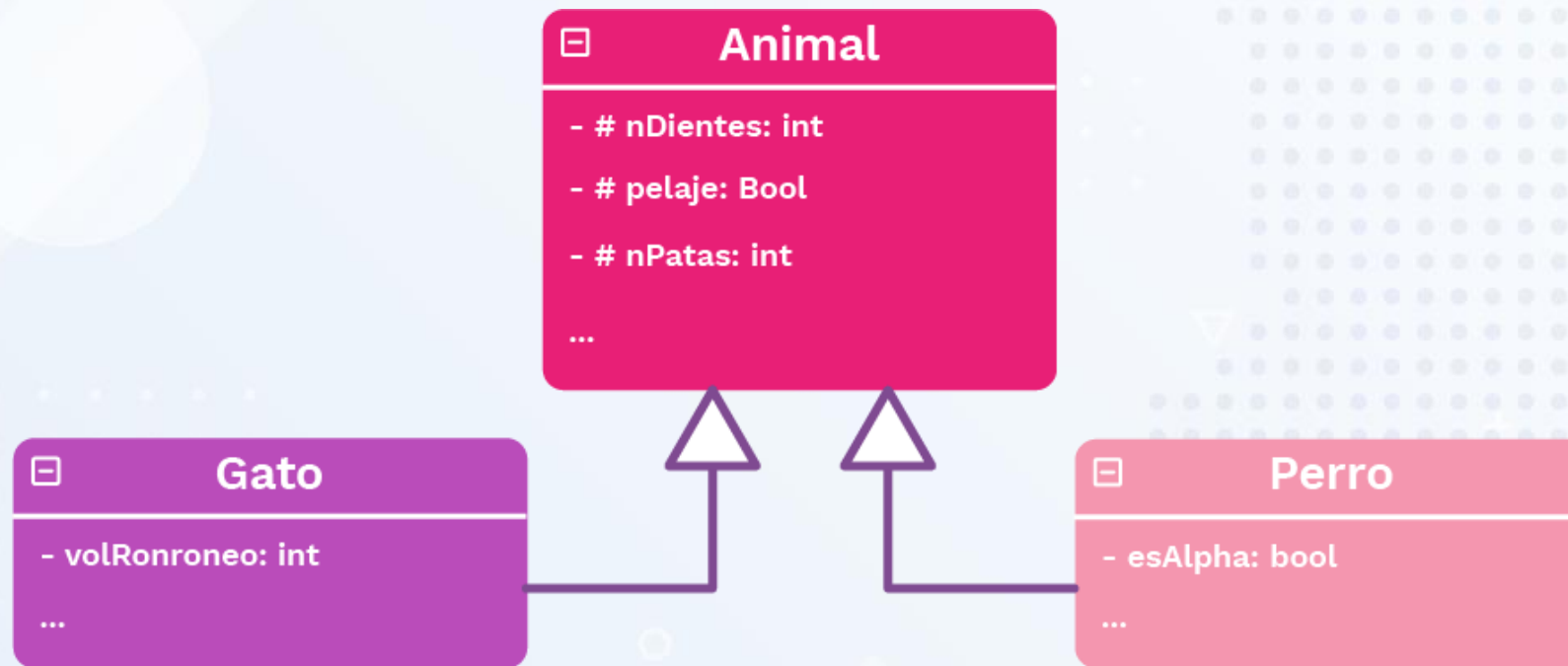
Supongamos que tenemos una clase llamada Gato, la cual contiene atributos y métodos que modelan a un gato. Bien, ahora queremos hacer una clase que modele a un perro. Inmediatamente, veremos que existen algunas características en común entre la clase Gato y la clase Perro, por ejemplo, el número de patas o que ambos tienen pelaje. Nosotros inocentemente copiamos todo ese código de la clase Gato y lo pegamos dentro de la clase Perro. Nos quedarían dos clases que contienen parte del código compartido, junto con algunas diferencias individuales, como podemos ver en la siguiente imagen:



Poco tiempo después, nos dimos cuenta de que había un pequeño error dentro del código de la clase Gato, que, debido a que copiamos el código, ahora también está presente dentro de la clase Perro. Al momento de corregirlo, tendremos que hacerlo tanto en la clase Gato como en la clase Perro, eso teniendo en cuenta que hicimos eso con dos clases, ¡Imagínate si hubiesen sido diez, veinte, o incluso cien!

Ahora que vimos un potencial problema, veamos cómo la herencia nos proporciona una forma limpia de reutilizar el código, evitando copiar y pegar. Usando el mismo ejemplo del perro y el gato, nosotros podemos identificar sus características en común (atributos y métodos) y extraerlos para ser puestos en una clase aparte llamada, por ejemplo, Animal.

De esta forma, nosotros estaríamos haciendo una pequeña abstracción del código, centralizando las características en común entre el perro, el gato y cualquier otro animal que queramos (y que, evidentemente, comparta las mismas características en común). La forma en la que reutilizaríamos el código, sería mediante la herencia de las características de la clase "padre o superclase" Animal, donde cada clase "hija o subclase", tendría únicamente las características individuales de cada animal, como se ve en la siguiente imagen:



Ahora, mediante la herencia, si detectáramos un error o problema en las características compartidas, solo tendríamos que corregirlo una sola vez, en la clase Animal. Bastante útil, ¿verdad?

En Java, la forma en la que nosotros podemos implementar la herencia es bastante sencilla. Basta con escribir el código de la superclase una sola vez, luego, cada subclase solo tendrá que añadir una pequeña palabra reservada, "**extends**", seguida del nombre de la clase a heredar. A continuación, se mostrará el código de la clase Animal, Gato y Perro. Clase animal:

```
public class Animal {  
    protected int nDientes;  
    protected boolean pelaje;  
    protected int nPatas;  
    // etc  
    // Métodos generales  
    public int getNDientes() {  
        return this.nDientes;  
    }  
    // ...  
}
```

Clase gato:

```
public class Gato extends Animal {  
    protected int volRonroneo;  
  
    public Gato(int nDientes, boolean pelaje, int nPatas, int volRonroneo) {  
        // Atributos heredados  
        this.nDientes = nDientes;  
        this.pelaje = pelaje;  
        this.nPatas = nPatas;  
        // Atributos particulares  
        this.volRonroneo = volRonroneo;  
    }  
  
    // Se heredan todos los métodos que no sean privados  
    // El siguiente es un método particular, solo presente en la clase Gato  
    public int getVolRonroneo() {  
        return this.volRonroneo;  
    }  
}
```


Clase Perro:

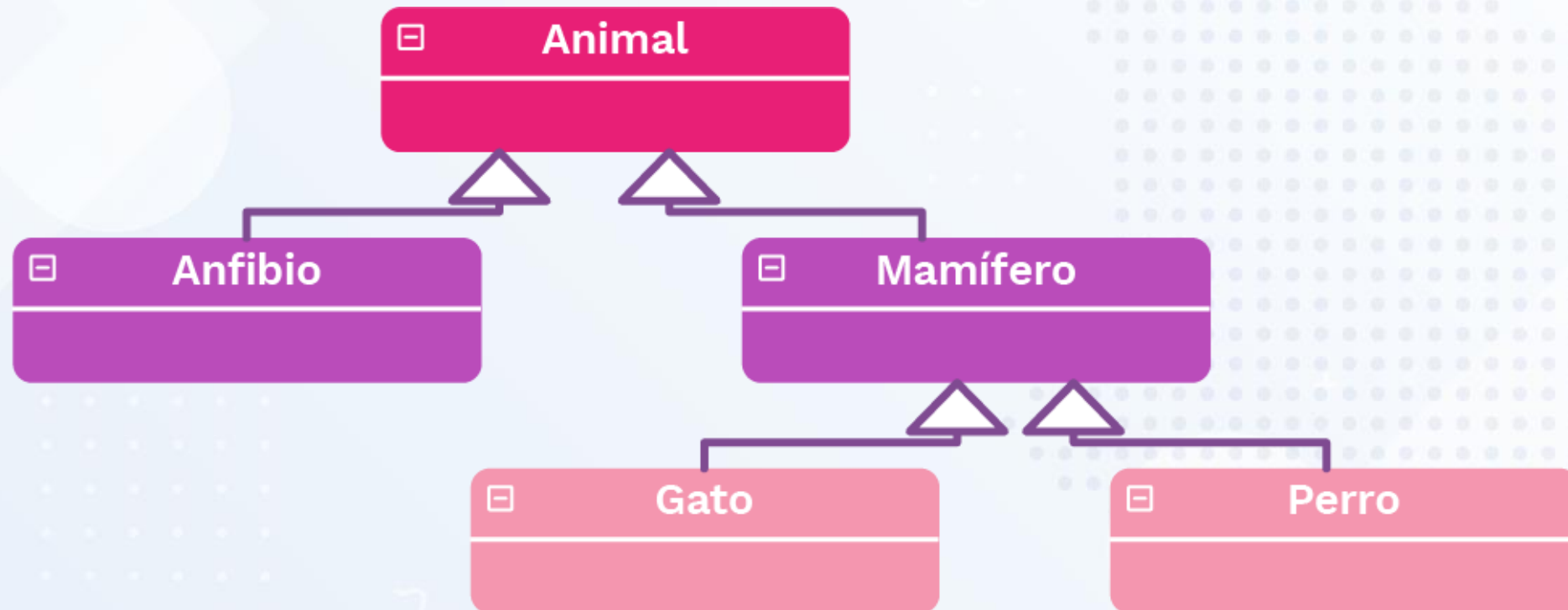
```
public class Perro extends Animal {  
    protected boolean esAlpha;  
  
    public Perro(int nDientes, boolean pelaje, int nPatas, boolean esAlpha) {  
        // Atributos heredados  
        this.nDientes = nDientes;  
        this.pelaje = pelaje;  
        this.nPatas = nPatas;  
        // Atributos particulares  
        this.esAlpha = esAlpha;  
    }  
  
    // Se heredan todos los métodos que no sean privados  
    // El siguiente es un método particular, solo presente en la clase Perro  
    public boolean esAlpha() {  
        return this.esAlpha;  
    }  
}
```

Nótese que no tenemos que reescribir el código de la clase Animal en las subclases, debido a que Java automáticamente lo incluye. Nosotros nos tendremos que preocupar por implementar el código que es exclusivo para la subclase.

Podemos, mediante el siguiente código, probar la ejecución de un método heredado:

```
public class Main {  
    public static void main(String[] args) throws Exception {  
        Gato gatito = new Gato(30, true, 4, 60);  
        Perro perrito = new Perro(42, true, 4, false);  
        // Un método heredado en ejecución:  
        System.out.println(gatito.getNDientes());  
    }  
}
```

Ahora que sabemos en qué consiste la herencia, nosotros podemos hacer todas las generalizaciones que queramos o que consideremos lógicas de hacer, con el fin de hacer un código más limpio y reutilizable. Por ejemplo, si vemos necesario, podríamos abstraer aún más las clases como vemos en la siguiente imagen:



La herencia también tiene otras utilidades derivadas de ella, como veremos en los siguientes temas.

3.3.1. Representación en UML de la herencia

Como quizá ya nos dimos cuenta, la forma en la que se representa la herencia en UML es a través del uso de una flecha triangular, vacía o en blanco por dentro, en dirección hacia la superclase, como se puede ver a continuación.



Algunas veces, dependiendo del consenso, también se le agrega "<<extends>>" o "extends" en el cuerpo de la flecha, como se muestra en la siguiente imagen:



3.4. Polimorfismo

Veremos que el polimorfismo está estrechamente ligado con la herencia, debido a que, en la mayoría de las ocasiones, no podemos hacer uso de esta relación entre clases sin haber hecho uso anteriormente de la herencia.

Este es, quizá, el concepto más difícil de comprender de todo este eje temático, por lo tanto, abordaremos este tema mediante el uso de ejemplos que faciliten entender el concepto de herencia.

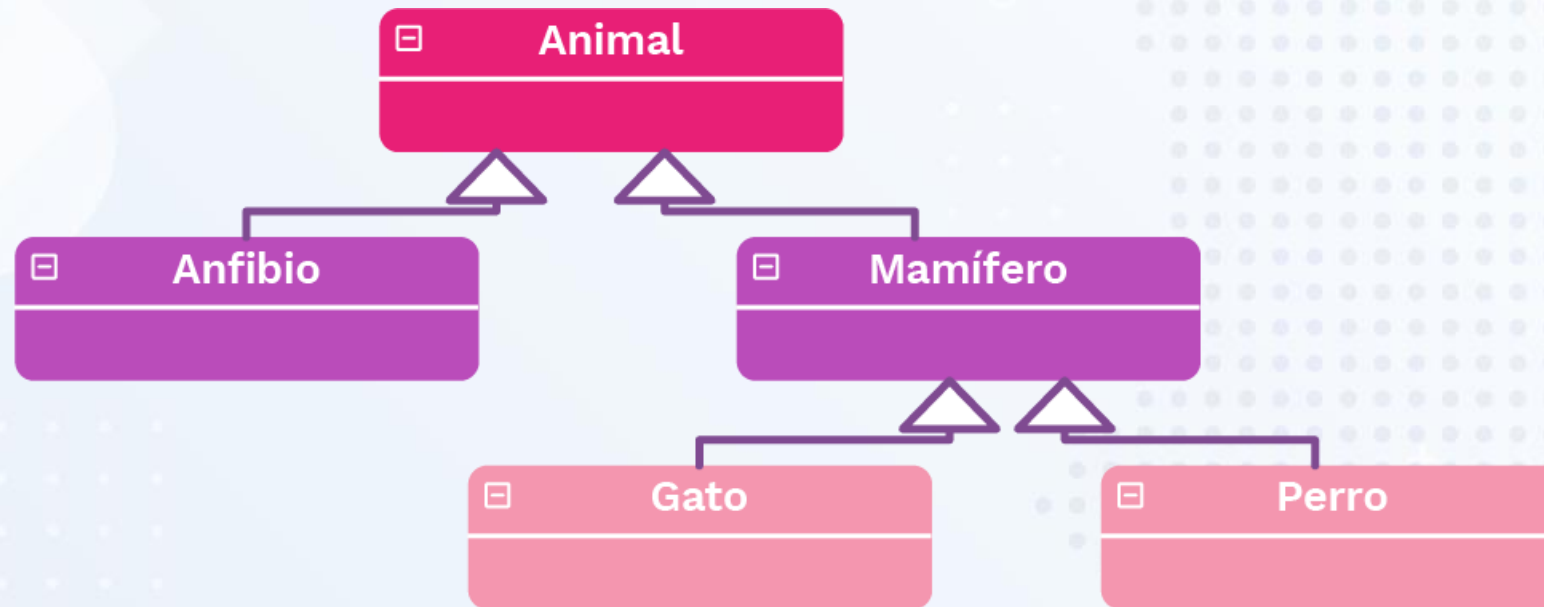
El concepto de polimorfismo consiste en que, al momento de usar un tipo de dato en cualquier estructura de datos (matriz, vector, ...), variable, etcétera; podamos definir un conjunto restringido de tipos válidos que están dados a partir de las clases que hereden de una superclase o de la superclase en sí misma.

Si te pareció súper confuso, no te preocupes, que iremos entendiendo este concepto a medida que avancemos en el tema. Para abordar poco a poco este concepto, primero recordemos cómo funciona Java (o cualquier otro lenguaje de programación de tipado estático), respecto a los tipos de datos de las variables.

Sabemos que, si nosotros decimos que una variable va a ser de tipo, por ejemplo, Gato, dicha variable solo podrá contener objetos de tipo Gato. Lo mismo ocurre si declaramos una variable con tipo Perro, String, int, o cualquier otro tipo de dato, siempre tendremos la restricción de que lo que vamos a manejar con esa variable es del tipo que se definió y no podemos cambiarlo. También sabemos que eso no solo aplica con variables, sino también con las funciones, parámetros o argumentos, arreglos, etcétera.

Entonces, teniendo en cuenta lo que la restricción anteriormente mencionada implica, ¿cómo haríamos para tener, por ejemplo, una matriz que almacena objetos de animales DIFERENTES, sin tener que definir una matriz para cada animal particular?

Ahí es donde entra en juego el polimorfismo. Usemos el mismo ejemplo anterior, supongamos que queremos hacer una matriz que nos permita incluir cualquier tipo de objeto que sea un animal. Para esto, recordemos el diagrama UML del tema anterior:

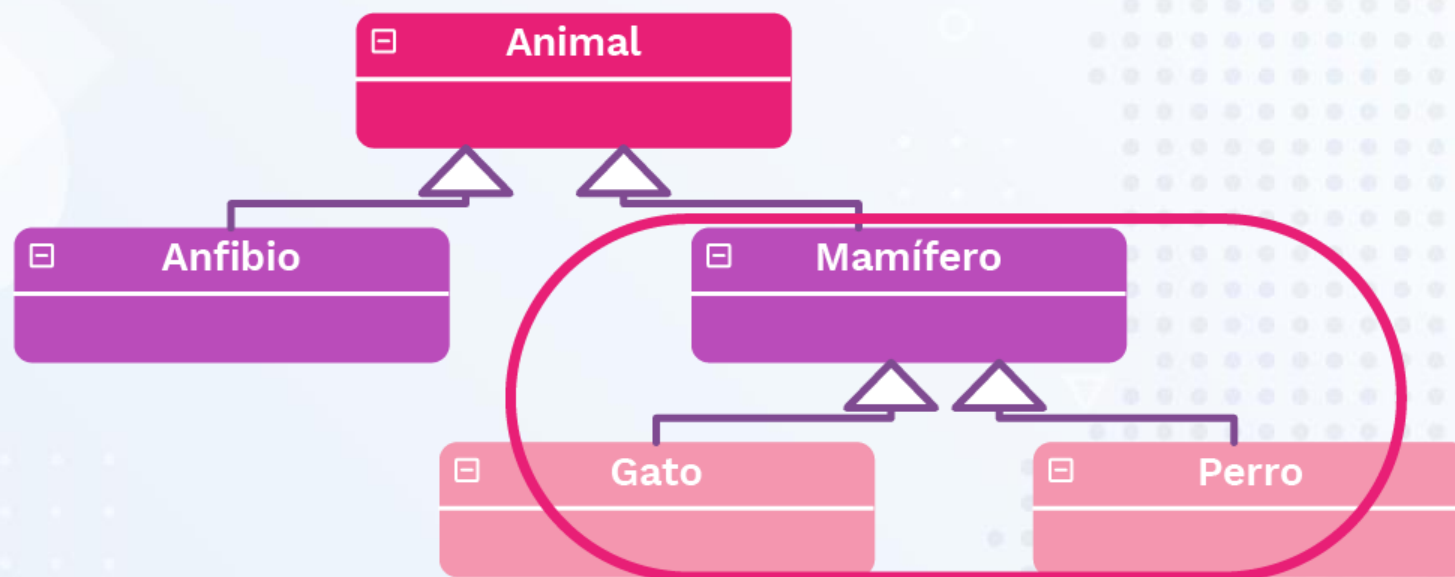


Evidentemente, sería extremadamente tedioso y poco práctico hacer una matriz más pequeña por cada tipo de animal para resolver el problema planteado, ¿verdad?

Pues, permítame contarte que podemos resolver este inconveniente de una forma mucho más sencilla y lógica. Nosotros definiremos el tipo de nuestra matriz como Animal, es decir, le diremos al lenguaje de programación que aceptaremos en nuestra matriz, objetos de tipo animal. Ahora viene la parte interesante, una vez definido el tipo de dato de nuestra matriz, nosotros podremos agregar elementos de objetos que sean de tipo Animal, Anfibio, Mamífero, Perro, Gato, o cualquiera que herede de la superclase Animal. Permitiéndonos así, efectivamente, almacenar objetos de cualquier tipo de animal. Esto es polimorfismo en su máxima expresión.

¿Sorprendido? Eso no es todo. Nosotros podemos aplicar este mismo principio para tipos de datos que retorne una función, o cualquier otra aplicación donde los tipos de datos se requieran.

También, siguiendo la lógica que planteamos, podríamos, en lugar de usar como tipo de dato Animal, usar como tipo de dato Mamífero, restringiendo los posibles tipos de animales aceptados al conjunto de objetos que sean de tipo Mamífero o hereden de este. Esto quiere decir que ya NO aceptaríamos objetos de tipo Anfibio ni objetos de, directamente, tipo Animal.



Ahora que ya tenemos una idea práctica del concepto, leemos la definición nuevamente y verás que ya entiendes:

“El concepto de polimorfismo consiste en que, al momento de usar un tipo de dato en cualquier estructura de datos (matriz, vector, ...), variable, etcétera; podamos definir un conjunto restringido de tipos válidos que están dados a partir de las clases que hereden de una superclase o de la superclase en sí misma.”

3.5. Sobreescritura de métodos

A diferencia del polimorfismo, el concepto de este tema es relativamente sencillo de entender.

La sobreescritura de métodos está estrechamente ligada a la herencia y se usa, generalmente, en conjunto con el polimorfismo. Este concepto consiste en algo muy simple: Sobrescribir un método de una superclase en una subclase. Lo entenderemos completamente con el siguiente ejemplo:

Nuevamente, haremos uso de nuestra superclase Animal y sus clases hijas. Supongamos que en la clase animal tenemos un método que se aplica para todos los animales (por eso estaría en el tope de nuestra abstracción), podría ser el método respirar(). Todos los animales que conocemos deben respirar, pero no todos los animales lo hacen de la misma forma. Por ejemplo, algunos anfibios pueden respirar a través de su piel, branquias o pulmones, a diferencia de los mamíferos que tienen una respiración totalmente pulmonar.

Teniendo en cuenta esa información, nosotros podríamos hacer varios métodos en nuestra clase animal, cada método para que sirva para cada subclase, pero eso iría en contra de la abstracción o generalización que hemos venido llevando a cabo. Para esto, podríamos definir el método respirar() como un procedimiento básico o genérico para cualquier animal, luego, nosotros en cada subclase, podemos sobrescribir el método respirar con una implementación más apropiada, teniendo en cuenta las características particulares necesarias.

Para realizar el proceso de sobreescritura basta con volver a escribir el método tal cual está definido en la clase padre, pero cambiaríamos el cuerpo del método conforme a nuestras necesidades.

La representación en código sería la siguiente:
Clase Animal:

```
public class Animal {  
    protected int nDientes;  
    protected boolean pelaje;  
    protected int nPatas;  
    protected double peso;  
    protected int nivelO2;  
    protected int nivelCO2;  
    // etc  
    // Métodos generales  
    public void respirar() {  
        this.nivelO2 = 100;  
        this.nivelCO2 = 0;  
    }  
    // ...  
}
```

Clase Mamífero:

```
public class Mamifero extends Animal {
    // Recordemos que los otros atributos son heredados de la superclase
    // Atributos particulares de mamíferos...
    protected int nFosasNasales;
    // etc

    public void respirar() {
        int tiempo = 0;
        // Unidad de aire * número de fosas nasales
        int diff = 1*nFosasNasales;
        // Inhalar
        while (this.nivelO2 < 100) {
            this.nivelO2 += diff/this.peso;
            if (this.nivelO2 > 100) {
                this.nivelO2 = 100;
            }
            tiempo += 1;
        }
        // Exhalar
        while (this.nivelCO2 > 0) {
            this.nivelCO2 -= diff;
            if (this.nivelCO2 < 0) {
                this.nivelCO2 = 0;
            }
            tiempo += 1;
        }
        System.out.println("Tiempo tomado: " + tiempo);
    }
}
```

Nótese que la declaración del método es igual, pero el contenido de este es diferente.

3.5.1. Representación en UML de la sobreescritura de métodos

En UML es sencillo representar la sobreescritura, basta con volver a escribir el método tal cual se muestra en la superclase, en la clase hija, como se muestra a continuación:



Nótese que para esto estamos sacando provecho al mostrar en el diagrama una relación de herencia, donde todo el contenido de la clase Animal, está siendo heredada por las clases hijas, por lo tanto, no es necesario volver a escribir todo lo que contiene la clase padre. Es por esto por lo que, si nosotros vemos un método de una subclase llamándose igual que el de una de sus clases padre, sabemos que hace referencia a una sobreescritura del método.

3.6. Modelado estructural

¡Que no te asuste o intimide el nombre de esta temática! Lo que veremos a continuación será una recopilación ejecutiva sobre los diagramas de clases y objetos, respecto a la estructura del software, que hemos venido viendo representados con UML.

3.6.1. Objetos

Empecemos a aprender la programación orientada a objetos desde los objetos en sí mismos. Recordemos que la forma en la que se representan estos en UML es mediante un rectángulo con el nombre, dos puntos, y su tipo de dato. En la siguiente imagen veremos la forma en la que se muestra un objeto genérico (es decir, la forma en la que se representa en UML), un objeto llamado firulais que es instancia de la clase Perro y un objeto de la clase Gato anónimo, respectivamente.

nombreObjeto: clase

firulais: Perro

: Gato

3.6.2. Clases

Recordemos nuestro recorrido, luego de que aprendimos qué era un objeto, aprendimos a generarlos a través de las clases, como si de un molde para hacer galletas se tratase. En UML, las clases, como hemos visto en algunos ejemplos de este y el anterior eje temático, se representan de la siguiente forma:



Nótese el símbolo que precede a los nombres de los atributos y métodos, estos representan los modificadores de acceso, es decir, la visibilidad, de estos. Recordemos que cada uno de esos símbolos representan un modificador diferente:

(+): Público. El método o atributo será accesible desde cualquier lugar.

(-): Privado. El método o atributo será accesible solamente dentro del contenido de la clase que lo contenga.

(#): Protegido. El método o atributo será accesible solamente dentro del contenido de la clase que lo contenga y sus clases hijas (si se llega a realizar una herencia sobre esta).

(~) Por defecto. El método o atributo será accesible solamente por las demás clases que estén en el mismo paquete.

También es importante recordar que, si dentro de la clase hay una separación después de un atributo, significa que lo siguiente que se está mostrando son los métodos pertenecientes a esa clase.

Todos los atributos y métodos siguen la misma lógica respecto a la forma en la que se representan sus tipos de datos: Primero va el nombre y luego, después de poner dos puntos (:), el nombre del tipo de dato. Sabemos que, en los métodos, especificar el tipo de dato nos indica precisamente el tipo del/los dato(s) que retorna dicho método.

Finalmente, si vemos que un atributo tiene un "=" después de definir su tipo, hace referencia a el valor por defecto que este atributo tiene.

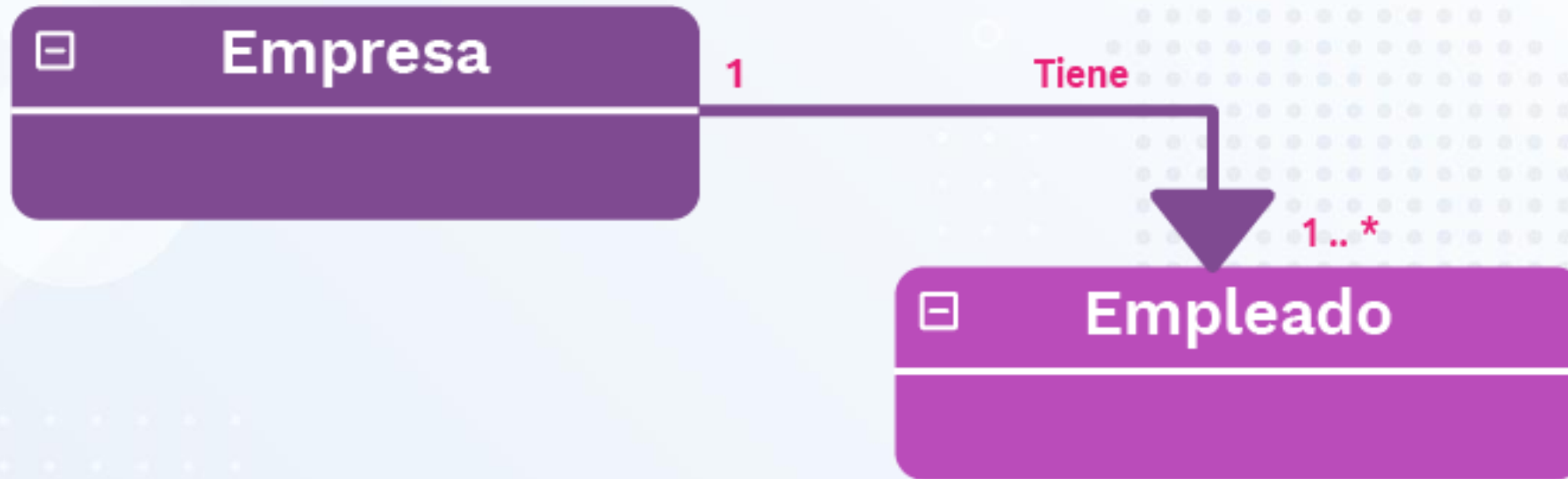
3.6.3. Asociación

La relación de asociación, como vimos anteriormente, representa de forma general una relación entre clases diferentes. En UML, la asociación se representa dependiendo de la direccionalidad de la relación, es decir, la relación tiene sentido si la vemos en la dirección en la que indica el enlace (la flecha). Generalmente, las relaciones de asociación son bidireccionales por lo que podemos usar, para mostrar el enlace que une las clases, una línea simple. Veamos dos ejemplos respectivos:

Relación de asociación bidireccional:



Relación de asociación direccionada:

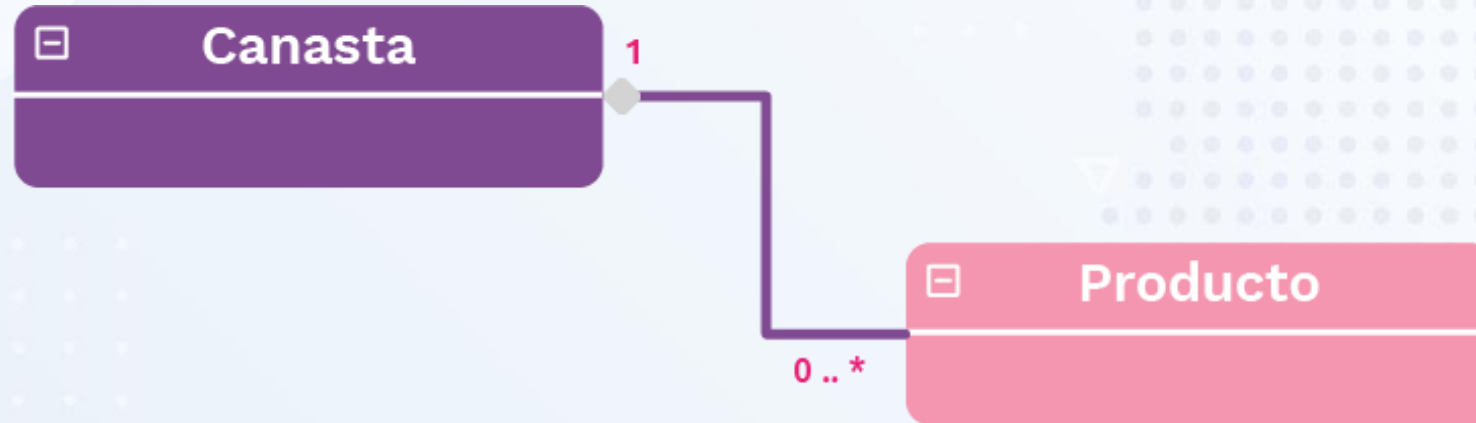


Vemos que, en esta relación, hay un nombre, "Tiene". Siguiendo la dirección de la relación y su cardinalidad, se interpretaría "Una empresa tiene de uno a muchos empleados".

Cabe aclarar que **NO** es necesario especificar el nombre de la relación en todos los casos, independientemente de si es bidireccional o no.

3.6.4. Agregación

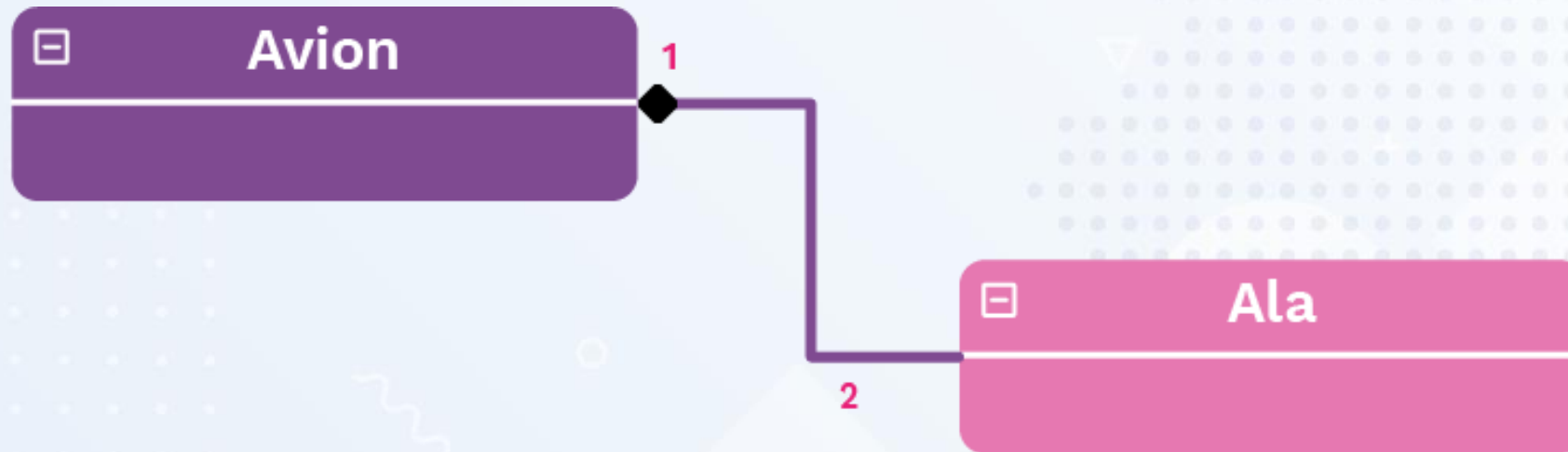
En la relación de agregación, a diferencia de la de asociación, la direccionalidad juega un factor importante, debido a que es de suma relevancia aclarar qué clase se le agrega a cuál. Recordemos que, la forma en la que se representa la agregación en UML es mediante el uso de un rombo con relleno vacío o de color blanco en el extremo del enlace de la clase, a la cual se le agrega la clase del otro extremo de la relación, tal como se puede ver en la imagen expuesta a continuación:



La cardinalidad **NO** es obligatoria; no obstante, en relaciones de agregación y composición es altamente recomendado definirla.

3.6.5. Composición

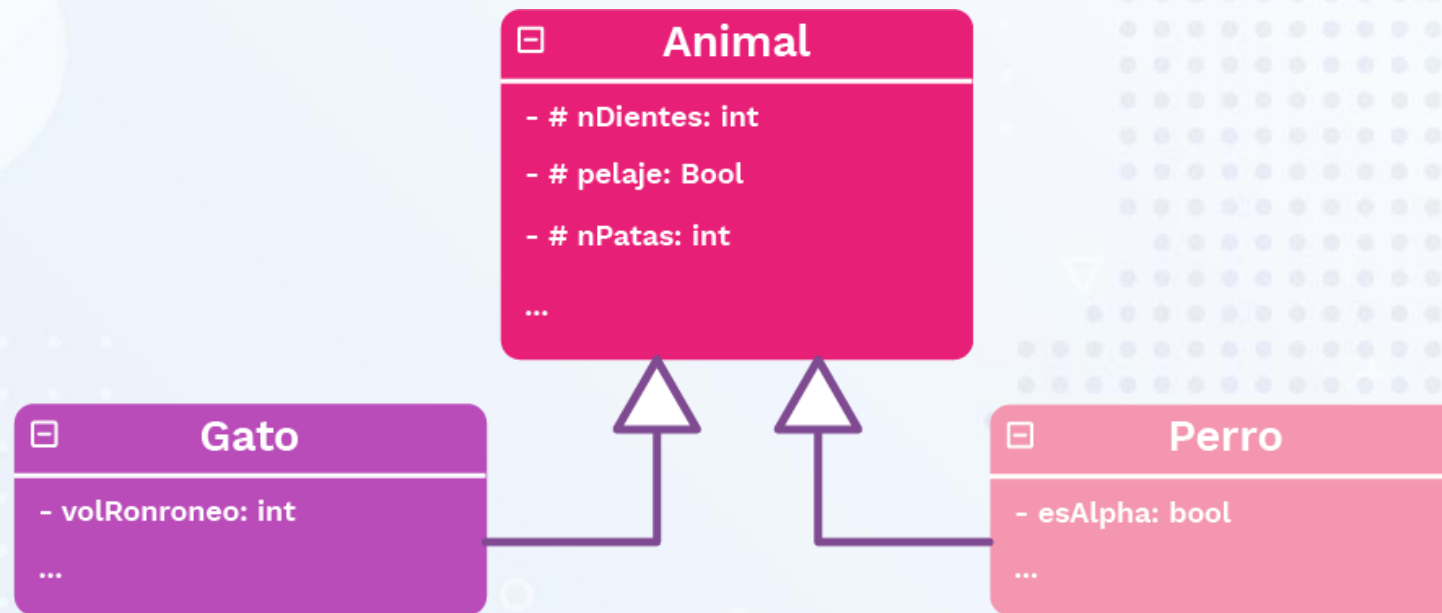
La relación de composición, como ya hemos aprendido, es muy similar a la de agregación, la diferencia es conceptual. No obstante, en UML, esta relación se representa de forma distinta. En lugar de usar un rombo con un color de relleno vacío o blanco, se usa un rombo con color negro (o, en su defecto, un color oscuro).



3.6.6. Herencia

La herencia, como habíamos visto en el tema respectivo, se representa en UML fácilmente a través del uso de una flecha triangular, vacía o en blanco por dentro, en dirección hacia la superclase. Este tipo de relación, por su naturaleza, no necesita especificaciones de cardinalidad.

En la siguiente imagen, podemos observar un ejemplo de herencia.



Nótese el uso de atributos protegidos en la superclase, con el fin de que conserven el principio de encapsulación, pero, al mismo tiempo, sus clases hijas hereden dichos atributos.

3.6.7. Lo aprendido en conjunto

A continuación, veremos un ejemplo de modelado estructural de un software mediante su diagrama de clases.

