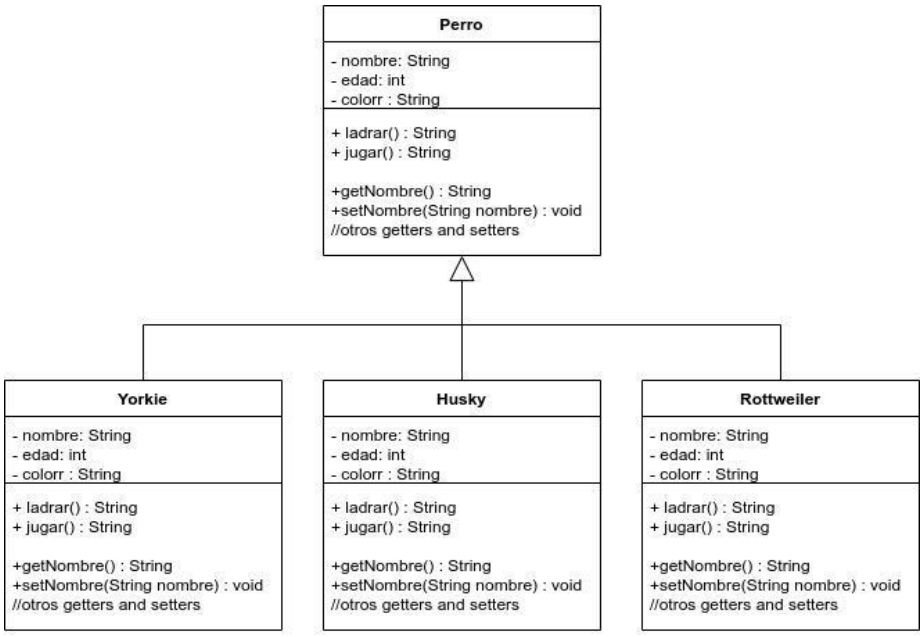
Sobreescritura. Casteo

Sobreescritura. Alterando el comportamiento heredado

¿Recuerdan el escenario de nuestros amigos peludos? Teníamos una jerarquía como esta:



Si nos centramos nuevamente en el Yorkie, cuando lo oímos ladrar, escucharemos que ladra de forma muy distinta a un Rottweiler o un Husky. ¿Cómo podemos mostrar esa diferencia dentro de la jerarquía de perros que tenemos aquí? De alguna manera, cuando el Yorkie vaya a ejecutar ese comportamiento heredado de ladrar, tendrá que aplicarle alguna modificación. Para lograrlo, utilizaremos un mecanismo llamado **sobreescritura de métodos**.

¿Recuerdan que el Yorkie, el Rottweiler o el Husky podían ladrar, incluso, si no habíamos escrito código para ello? Eso se debe a que heredaban todos los métodos no privados de Perro.

Hacer:

```
public class Yorkie extends Perro {

}
```

Era suficiente como para poder hacer:

```
public static void main(String [] a) {

  Yorkie y = new Yorkie();
  y.ladrar(); // GUAU

  //también
  Perro p = new Yorkie();
  p.ladrar(); // GUAU

}
```

Para aplicar sobreescritura, se debe volver a escribir el método ladrar, respetando la firma, es decir, escribir el mismo método con el mismo nombre, respetando el tipo, orden y cantidad de parámetros suministrados. ¿Logran ver la diferencia con la sobrecarga de métodos?

De esta forma, "pisamos" el comportamiento heredado y lo reemplazamos por otro. El Yorkie tendrá un ladrido más agudo que el de cualquier perro. Pongamos eso en código:

```
public class Yorkie extends Perro {
   public String ladrar() {
      return "WIF WIF";// ladra como los ladran los Yorkies
   }
}
```

Entonces, si hiciéramos en un main:

```
public static void main(String [] a) {
  Yorkie y = new Yorkie();
  System.out.println(y.ladrar());// < WIF WIF
}
```

Aquí, la clase Yorkie sí define (o mejor dicho, redefine) un método "ladrar". Como vimos con los métodos heredados, el compilador analiza el método `ladrar()` llamado sobre el Yorkie que está en la variable `y`.

Entonces el compilador "piensa": "A ver. El método ladrar, sin parámetros, ¿está definido en Yorkie?":

1. Sí, entonces la sintaxis es correcta.

2. No, Yorkie hereda de Perro. ¿Está definido en Perro? a) Sí, entonces la sintaxis es correcta; b) No, entonces no existe el método ladrar(). La sintaxis no es correcta.

Luego, es el turno de la ejecución del programa. Al momento de la ejecución, nuevamente, el runtime analiza: ¿el método `ladrar` sin parámetros está definido en Yorkie?

1. Sí, la ejecuto.
2. No, debo ejecutar el código que corresponde a la operación ladrar que se hereda de Perro.

Sobrescritura y reutilización de código

Husky también es un Perro. Los Husky siempre nos recuerdan un poco a los lobos porque tienen la costumbre de aullar. Muchas veces, al momento de ladrar, de acuerdo con su estado de ánimo, "estiran" un poco el ladrido y comienzan a tener con nosotros esas "conversaciones" tan simpáticas que vemos en internet.

La pregunta es: ¿podemos replicar ese comportamiento en nuestro código? ¡Por supuesto! Sin embargo, debemos tener en cuenta que esta es la forma que tienen de ladrar los Husky. **No es un comportamiento distinto,** es decir, no constituye un nuevo método en la clase Husky. Por el contrario, es el mismo ladrido de los Perros, pero "un poco distinto". Entonces, podríamos sobrescribir el método en Husky y lograr:

```
public class Husky extends Perro {
   public String ladrar() {
      return "GUAU GUAUUOOUU";// ladra como los ladran los Huskies
   }
}
```

¿Pero, que hay ahí? Es como el ladrido de todos los perros, con un agregado pequeño al final. Podríamos "pisar" el método heredado, pero, a la vez, podríamos **reutilizar** aquello que se hereda. Entonces, ¿qué tal si hacemos lo siguiente?

```
public class Husky extends Perro {
   public String ladrar() {
      String ladrido = super.ladrar();
      return ladrido + "GUAUUOOUU";
   }
}
```

El Rottweiler suele ser algo más agresivo, así que, cuando ladre, podemos hacer que gruña un poco también.

```
public class Rottweiler extends Perro {
   public String ladrar() {
```

```
        return super.ladrar() + " GRRRR"
    }
}
```

Entonces, si hiciéramos un main:

```
public static void main(String [] a) {
  Yorkie y = new Yorkie();
  Husky h = new Husky();
  Rottweiler r = new Rottweiler();

  System.out.println(y.ladrar());// < WIF WIF
  System.out.println(h.ladrar());// < GUAU GUAUUOOUU
  System.out.println(y.ladrar());// < GUAU GRRRR

}
```

Uso de la keyword super

La palabra "super" tiene esa denominación porque hacer referencia a la "superclase", es decir, una clase padre respecto de la clase a la cual estamos escribiendo el código. Visto desde el punto de vista de la clase padre, esta tendrá clases "hijas" o "subclases".

Recuerden que, para llamar a un método de la propia clase, podemos referenciarlo por el nombre o podemos hacerlo con la palabra "this". Veamos un ejemplo:

```
public class Persona {
 private String nombre;

 public String decirNombre() {
    //retorna el valor al atributo nombre del objeto
    return nombre; //es el nombre de este objeto

    // retorna el valor al atributo nombre del objeto
    //return this.nombre; // this hace referencia a este objeto
   }

   public String saludar() {
    //hace referencia al método decirNombre del objeto actual
    return "Hola, me llamo " + decirNombre();

    // hace referencia al método decirNombre del este objeto
    //return "Hola, me llamo " + this.decirNombre();

   }

}
```

Muchas veces, usábamos al this para desambiguar cuando recibíamos parámetros con el mismo nombre de un atributo. Así nos asegurábamos de que el compilador entendiera cuándo nos referíamos al parámetro y cuándo al atributo.

Con el super, pasa algo similar. Nosotros habíamos visto a la palabra clave "super()" con paréntesis, acompañada de cero o más parámetros como una manera de acceder al constructor de la clase padre. Aquí, en cambio, estamos usando a "super, sin paréntesis, para **acceder al comportamiento de la clase padre.** El uso de "super" como mecanismo de llamada a métodos es muy importante, ya que, de no contar con ella, si hiciéramos:

```
public class Rottweiler extends Perro {
   public String ladrar() {
      return ladrar() + " GRRRR" // loop infinito!
   }
}
```

El método ladrar() se llamaría a sí mismo infinitamente. En cambio, con el super explicitamos que queremos llamar al ladrar() que heredamos de la superclase, queremos **desambiguar**. Entonces, ¿cómo deberíamos corregir el código anterior?

```
public class Rottweiler extends Perro {
   public String ladrar() {
      return super.ladrar() + " GRRRR";
   }
}
```

¿Qué ocurrió aquí? **Nuevamente reutilizamos código**. El Rottweiler retorna "GUAU GRRRR" cuando ladra, pero el código del ladrido que devuelve "GUAU" lo escribimos solo una vez, en un solo lugar, en la clase Perro. Como siempre, esto es solo un ejemplo. Sin embargo, es importante tenerlo en cuenta porque pensemos que, cuando reutilizamos, podríamos llegar a reutilizar muchas líneas de código o algoritmos muy complejos. Reutilizar código es siempre una buena idea.

Cabe aclarar que, si tenemos una jerarquía de 3 o más escalones (por ejemplo: Husky, Perro, Canino) **no se pueden encadenar las llamadas a super**. Es decir, no se puede hacer super.super.unMetodo()) sino que siempre se hace de a un escalón por clase. Husky llamará a super.unMetodo() que en Perro hará super.unMetodo() que en Canino hará su trabajo.

Conversión de tipos referencia

Anteriormente, hemos analizado el casteo como la conversión entre tipos de datos primitivos. Con el casteo se puede aumentar la precisión de un tipo de dato numérico o reducirla eventualmente y perder información. Es decir, hasta este punto nos habíamos limitado a hacer cosas de tipo:

```
int i = 1300;

long l = (long) i;

double pi = 3.14159265359;
System.out.println(pi);//   3.14159265359

float piFl = (float)pi;
System.out.println(piFl);// 3.1415927
```

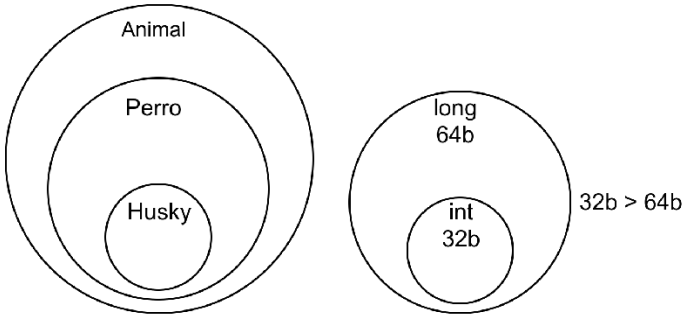
Sin embargo, llegamos a este punto y ahora conocemos el concepto de "herencia". Una subclase hereda de otra superclase sus atributos y operaciones. Pusimos especial énfasis en que nos importan más las operaciones de las clases, es decir, lo que hacen. Por ejemplo, la Clase Husky hereda todas los atributos y las operaciones de una clase Perro. En este caso, Perro heredará de una clase Animal, y esta última, a su vez, heredaría todo lo que los `Animal`es hacen: "comer()" y "dormir()". Entonces, a un Husky se lo puede considerar un Perro (de hecho, en la realidad lo es), y a un Perro se lo puede considerar un Animal (de hecho, en la realidad lo es). Por carácter transitivo, entonces, un Husky es una Animal.

Upcasting

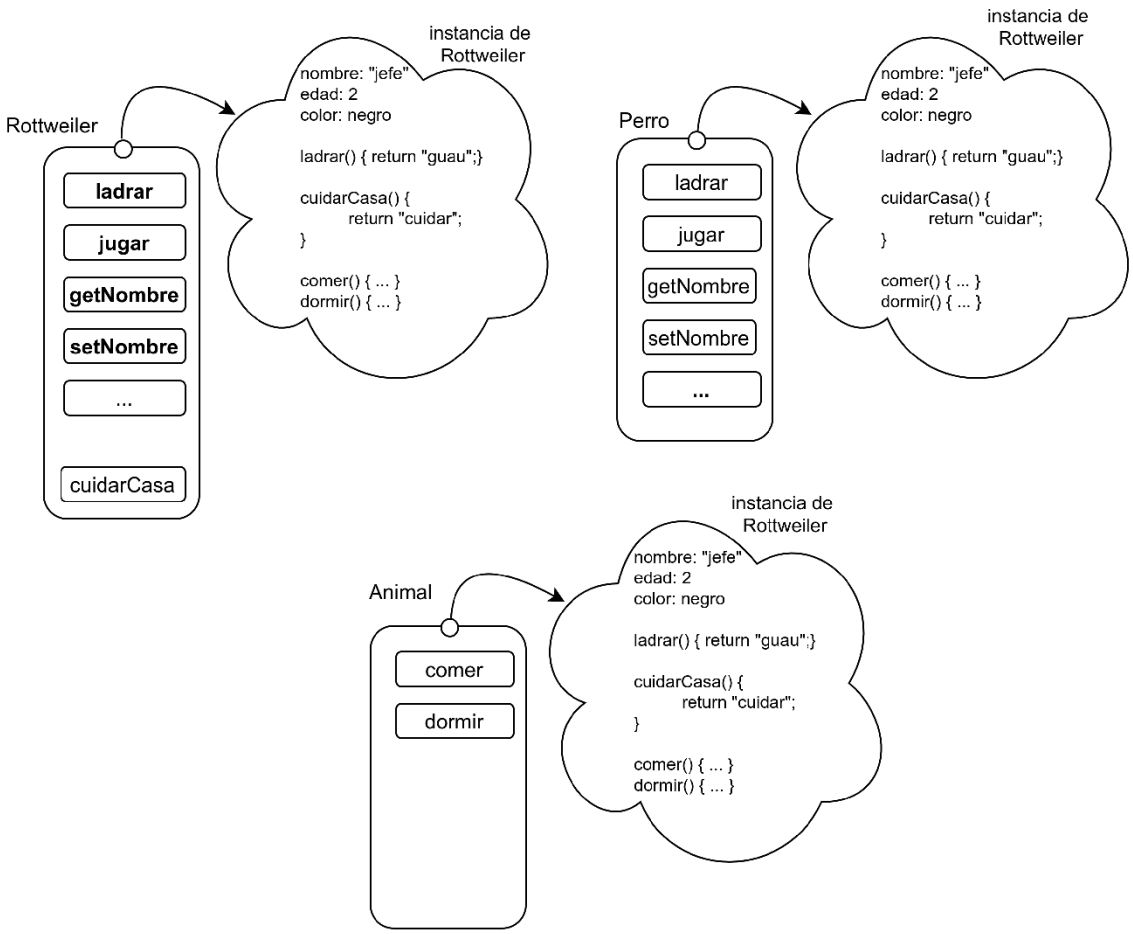
La conversión de tipos "hacia arriba" se hace de manera implícita y salta a la vista si seguimos lo que mencionamos más arriba:

```
Husky h = new Husky();//'h' aquí guarda un objeto Husky
Husky h2 = new Husky();//'h2' aquí guarda un objeto Husky

Perro p = new Perro();//'p' aquí guarda un objeto Perro

//entonces, según lo dicho es razonable poder hacer:
Perro x = h; //porque todo Perro es Animal.

//podemos hacerlo explícitamente también:
Perro y = (Perro)h2;

//y más aún
Rottweiler r new Rottweiler();
Animal a = r;
```

La operación de casteo se hace de la misma forma que con los tipos de datos primitivos, es decir, utilizando los paréntesis. Asimismo, la lógica es similar: estamos pasando de un Perro a un Animal, es como si fuéramos de algo más específico a algo más general; de la misma forma que pasamos de un int a un long, que puede almacenar más bits.



Es importante notar que, por más de que ‘x’ e ‘y’ sean Huskys, el tipo de la variable es Perro, por lo cual solo tendremos acceso a las características y las operaciones de Persona. ¿Recuerdan la analogía del control remoto? El control remoto tiene “los botones” para llamar a cada operación del objeto (se omitieron algunas operaciones para brevedad).



A medida que casteamos “hacia arriba” en la jerarquía, no nos vamos hacia lo “más general”, vamos “perdiendo botones” para controlar a nuestro objeto, aunque nuestro objeto no se ve alterado**.** Esto nos puede servir en el caso de que, con un mismo control remoto, decidamos controlar a más de un animal. Por ejemplo, podemos usar el control remoto de Animal para controlar a Perros, Gatos, Aves, etc. Entonces, si bien “perdemos botones”, ganamos mucho en flexibilidad.

Downcasting

Se trata de la conversión de tipos “hacia abajo”, de lo más genérico hacia lo más específico. De la superclase hacia la subclase.

Vimos que utilizar del lado izquierdo del igual a un tipo de dato más general nos da flexibilidad. Si queremos, podríamos utilizar un control remoto más genérico para controlar todo tipo de animales, por ejemplo. La contra que presenta esto es que “perdemos botones” para llamar a las operaciones específicas de cada animal. Rottweiler tiene

“cuidarCasa()”. Si usamos un control remoto de Animal, le podremos decir a nuestro Rottweiler que es hora de comer, pero no le podremos decir que cuide la casa. Con el downcasting ganamos nuevamente acceso a esos “botones” y podremos llamar a las operaciones específicas del objeto que estemos apuntando con nuestro control remoto.

Analicemos ahora el downcasting detenidamente. Si escribiera el siguiente código, no nos debería resultar novedoso:

```
Perro p = new Perro(); //(*)
```

Algo de esto habíamos visto también:

```
Perro p2 = new Rottweiler()
```

Y esto es correcto, ya que, por la estructura jerárquica que describimos al principio, el compilador es capaz de interpretar esto como correcto. Sin embargo, el camino inverso no necesariamente es correcto. Por ejemplo, si tenemos:

```
Rottweiler r = new Perro();
```

Esto se debe a que todo `Rottweiler` es-un `Perro`, pero no todo `Perro` es necesariamente un `Rottweiler`.

Ahora bien, supongamos que hacemos lo siguiente:

```
Perro p3 = new Rottweiler();//(**)
```

Según lo que vimos, no tiene nada de malo. En este punto, sabemos que el compilador entiende esto como correcto. Analizando detenidamente el código, podemos ver que hay una conversión de tipos implícita, es decir, hay un *cast* implícito. Asimismo, si hacemos:

```
Rottweiler rott = p3;
```

El compilador es capaz de entender que no necesariamente todo Perro es un Rottweiler (alguno puede serlo, pero no todos), por lo que arrojará un error de compilación.

Pero… ¡un momento” Si pusimos en p3 a un Rottweiler, ¿por qué no es correcto? Recordemos que el compilador analiza la sintaxis de nuestro código, no lo ejecuta (es común que nosotros tendamos a “correr el código con la vista").

Entonces, para solucionar el problema, podemos “prometerle” al compilador que, efectivamente, cuando se corra el código, en p3 almacenaremos una instancia de Rottweiler. Eso lo podemos hacer mediante un **downcast explícito**:

```
Rottweiler rottie = (Rottweiler)p3;
```

Instanceof

Pero… ¡cuidado! ¿Qué pasa si con el downcast anterior le estoy "mintiendo" al compilador, es decir, le estoy prometiendo algo que no puedo cumplir?

```
//acá le prometo al compilador, y le estoy cumpliendo
Rottweiler rottie = (Rottweiler)p3; //(ver **)

//acá estoy "mintiendo" porque en p (ver *) guardé una persona
Rottweiler someDog = (Rottweiler)p;
```

En este caso, no obtendremos un error en momento de compilación, porque le estamos prometiendo algo al compilador al momento de escribir el código que este interpreta correctamente (el compilador nos cree, sin más). Sin embargo, al en momento de la ejecución del código, obtendremos un error de casteo: una `ClassCastException`.

Es importante recordar que el compilador analiza la sintaxis del código, no lo ejecuta. Lo que ocurre durante la ejecución es distinto. En este caso, es válido hacer un casteo, pero ya dijimos que no necesariamente todos las Perros son Rottweilers (en el caso de arriba, p3 es un perro que sí es Rottweiler, pero p es un Perro que no lo es, es un simple perro mestizo a la espera de que lo adopten, por ejemplo).

Protección ante casteos incorrectos

Antes de castear podemos asegurarnos de que puede hacerse correctamente. Java incorpora un operador (no es un método ni una función) denominado **instanceof**. El operador instanceof sirve para averiguar si una variable efectivamente está almacenando una instancia de una clase particular. Demostramos, a continuación, el correcto uso de instanceof. Recordemos que es un operador:

```
miVarible instanceof MiClase;
```

Si la variable está almacenando un objeto de tipo MiClase, es decir, una instancia de la clase MiClase, entonces la operación devolverá un true. De lo contrario, un false.

Veamos un ejemplo con los perros:

```
if(p instanceof Rottweiler) {
  Rottweiler exampleRottweiler = (Rottweiler)p;
}
```

De esta manera, la operación de casteo se puede hacer de forma segura. Es decir, el casteo siempre es posible al momento de escribir el código. Sin embargo, cuando el código se ejecuta de acuerdo con la instancia que estemos almacenando en la variable, la operación puede fallar. Con el operador instanceof, me puedo asegurar de que no haya problemas en tiempo de ejecución.

Te dejamos los siguientes interrogantes para que reflexiones sobre lo visto en la lectura

- ¿Cómo te sientes con el tipado fuerte que establece Java?
- ¿Crees que el casteo en Java es un mecanismo sencillo o piensas que debería simplificarse?