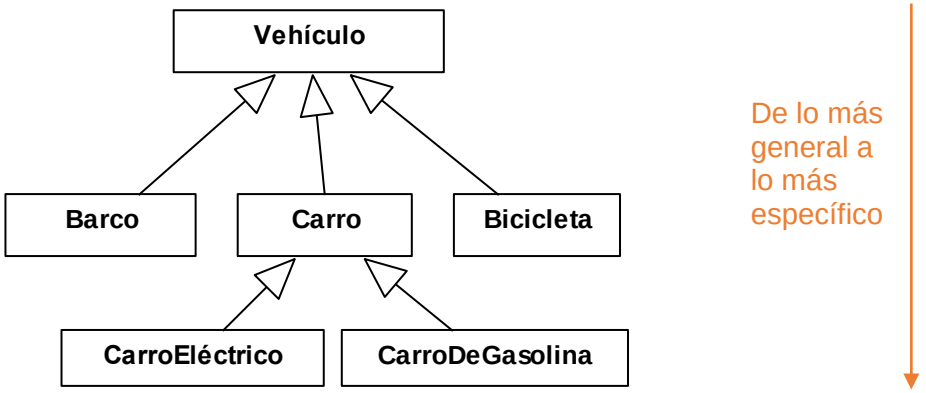
Herencia

Material de clase elaborado por Sandra Victoria Hurtado Gil

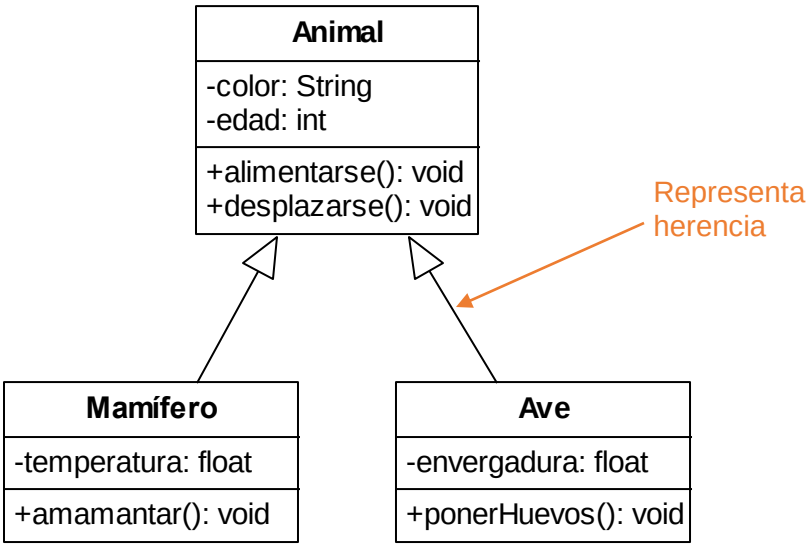
En Orientación a Objetos uno de los mecanismos que ayudan a modelar un sistema es la **herencia**. Mediante este mecanismo es posible identificar clases que son especializaciones o subtipos de otras.

Por ejemplo, los barcos, los carros y las bicicletas son tipos o especializaciones de vehículos. Además, los carros pueden ser de varios tipos: eléctricos o de gasolina.
Esto se representa en un diagrama de clases de la siguiente forma:



Es decir, un barco es un tipo de vehículo, lo mismo que un carro y una bicicleta: todos son vehículos. A su vez, un carro eléctrico es un carro, y también es un vehículo, y lo mismo un carro de gasolina.

Al definir una jerarquía de clases, las clases que son subtipos o especializaciones tienen todo lo que está definido en las clases generales ("**lo heredan**"), y además pueden tener atributos o métodos que sean solo de ellas. Por ejemplo:



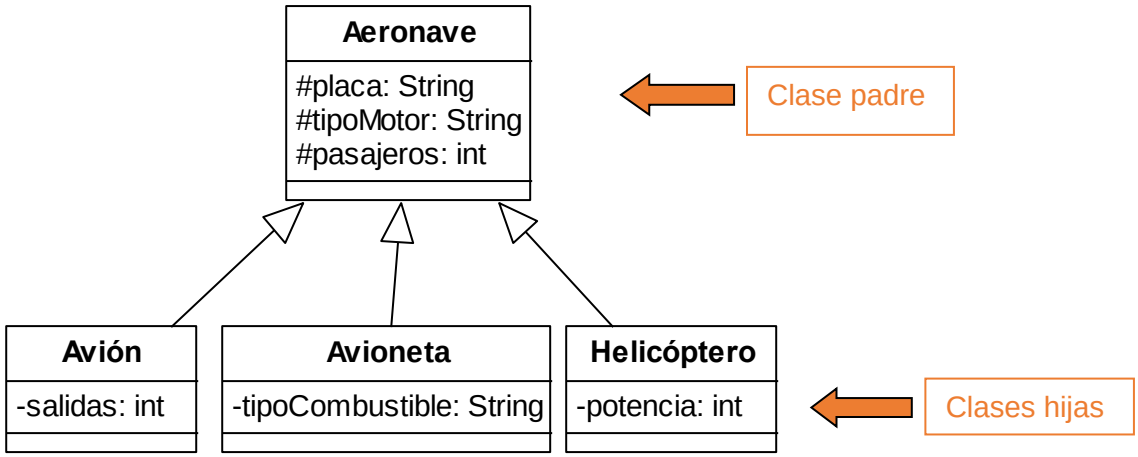
Los mamíferos **SON** animales, y por lo tanto tienen todas las características y el comportamiento de un animal, pero además tiene características y comportamientos propios que no tienen otros tipos de animales.
Los mamíferos, por lo tanto, tendrán: color, edad y temperatura, y pueden: alimentarse, desplazarse y amamantar.
Las aves también SON animales, y tienen como características: color, edad y envergadura[1]. Para las aves no es necesario conocer la temperatura. En cuanto al comportamiento pueden alimentarse y desplazarse, como todos los animales, pero no amamantan a sus crías. En su lugar las aves ponen huevos.

Al definir una herencia en programación Orientada a Objetos también se simplifica la elaboración del código, porque las **clases hijas** (los subtipos) no tienen que volver a definir todo lo que está en la **clase padre** (la más general).

Por ejemplo, en una empresa de aviación tienen diferentes tipos de aeronaves: los aviones, las avionetas y los helicópteros. Todas las aeronaves vienen en diferentes colores y tienen establecidos el tipo de motor y el número de pasajeros. Para los aviones es importante definir el número de salidas que tienen y para los helicópteros se debe saber cuál es su potencia máxima en condiciones de carga completa. En el caso de las avionetas es importante definir el tipo de combustible que usan: gasolina o energía solar, pues se está popularizando el uso de energías alternativas.

Para el anterior enunciado el diagrama de clases correspondiente, mostrando solo los atributos, es:

[1] Distancia entre las puntas de las alas



El código de las clases Aeronave y Avión es:

```
/**
 * Medio de transporte aéreo, que posee una empresa de aviación
 * @version 1.0
 */
public class Aeronave {
	protected String placa;
	protected String tipoMotor;
	protected int pasajeros;
}
```

```
/**
 * Un tipo de aeronave de gran tamaño, con varias salidas,
 * por lo general usado para vuelos comerciales
 * @version 1.0
 */
public class Avion extends Aeronave {
	private int salidas;
}
```

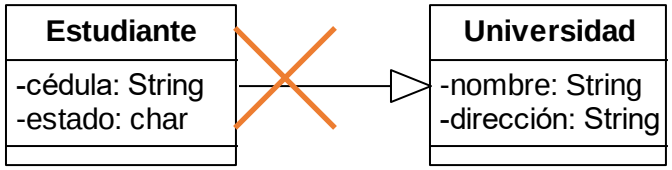
Es importante tener presente que las clases hijas **heredan** todo lo que está definido en la clase padre. En el ejemplo anterior:

- La clase Avión tiene cuatro atributos: tres que hereda (“placa”, “tipoMotor” y “pasajeros”) y uno propio (“salidas”). No es necesario que defina de nuevo los atributos de la clase padre, los incluye automáticamente cuando indica que es su hija.
- La clase Avioneta también tiene cuatro atributos: los tres que hereda de la clase Aeronave y su atributo propio: “tipoCombustible”.

Es importante tener en cuenta que la herencia no es solo para simplificar la elaboración del código, pues tiene implicaciones más importantes: una clase hija no solo debe compartir atributos y métodos de la clase padre, sino que debe tener el mismo significado. De esta manera, donde se pueda tener un objeto de la clase padre, se podrá tener un objeto de la clase hija.

Se puede entender que cada clase hija "ES UN TIPO DE" la clase padre, lo cual ayuda a verificar si la herencia se ha identificado apropiadamente. Por ejemplo: un cuadrado **es un tipo de** figura geométrica, un mamífero **es un tipo de** animal, un colectivo **es un tipo de** vehículo.

Por lo tanto, en el siguiente diagrama no es correcta la relación de herencia entre las clases:



Aunque el estudiante tenga nombre y dirección, no se debe usar la herencia para "definir" estos atributos, puesto que un estudiante **no es un tipo de** universidad.

En este caso, en la clase Estudiante se deben definir todos los atributos, incluyendo nombre y dirección. Entre Estudiante y Universidad se puede tener otra relación, por ejemplo, una asociación, pero no una herencia.

Implementación en Java

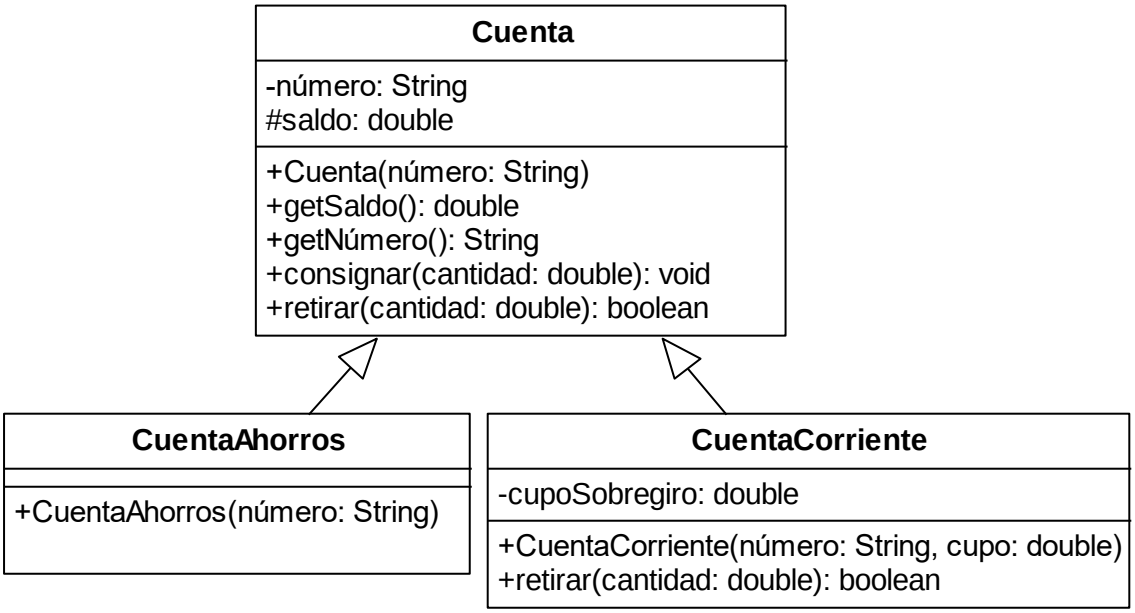
Para definir la herencia en Java, se tiene la palabra ***extends*** en las clases hijas, para indicar de cual clase heredan. La sintaxis es:

```
public class ClaseHija extends ClasePadre
{
      //código de la clase
}
```

Cada clase debe tener su propio constructor. **Los constructores son los únicos métodos que no se heredan**. Sin embargo, en las hijas es posible usar el constructor de la clase padre, lo cual se verá en la siguiente sección.

Por ejemplo, en un banco existen cuentas corrientes y cuentas de ahorro. Aunque ambos tipos de cuentas tienen características y servicios comunes, cada una se comporta de manera diferente: En las cuentas de ahorro no se pueden tener saldos negativos, mientras que en las cuentas corrientes sí se pueden tener saldos negativos, hasta cierta cantidad o cupo (lo que se

llama sobregiro). Esto hace que la forma de retirar dinero de cada cuenta sea diferente, porque se deben hacer diferentes validaciones. El diagrama de clases que muestra esto, es:



El código para las clases Cuenta y sus hijas:

```java
/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @version 3.5
 */
public class Cuenta {
	private String numero;
	protected double saldo;

	public Cuenta(String numero) {
		this.numero = numero;
		this.saldo = 0;
	}

	public double getSaldo() {
		return this.saldo;
	}

	public String getNumero() {
		return this.numero;
	}

	/**
	 * Consigna o adiciona una cantidad de dinero en la cuenta,
	 * lo cual incrementa el saldo.
	 * @param   cantidad    la cantidad de dinero que se desea
	 *                       consignar, en pesos
	 */
	public void consignar(double cantidad) {
		saldo = saldo + cantidad;
	}

	/**
	 * Retira o saca una cantidad de dinero de la cuenta,
	 * si hay saldo suficiente.
	 * @param  cantidad    la cantidad que se desea retirar, en pesos.
	 * @return  si se pudo hacer el retiro porque tenía dinero
	 *          suficiente o no (true o false)
	 */
	public boolean retirar(double cantidad)  {
		if (getSaldo() >= cantidad) {
			saldo = saldo - cantidad;
			return true;
		}
		else {
			return false;
		}
	}
} //fin clase Cuenta
```

```java
/**
 * Cuenta corriente en una entidad financiera,
 * en la cual se puede retirar incluso si no hay saldo,
 * hasta el monto dado por el cupo de sobregiro.
 * @version 1.0
 */
public class CuentaCorriente extends Cuenta {
	private double cupoSobregiro;

	public CuentaCorriente(String numero, double cupo)  {
		super(numero);
		cupoSobregiro = cupo;
	}

	/**
	 * Retira o saca una cantidad de dinero de la cuenta,
	 * incluso quedando negativo, hasta el cupo de sobregiro.
	 * @param  cantidad  la cantidad que se desea retirar, en pesos.
	 * @return  si se pudo hacer el retiro porque tenía dinero
	 *          suficiente o no (true o false)
	 */
	public boolean retirar(double cantidad) {
		if ((getSaldo() + cupoSobregiro) >= cantidad) {
			saldo = saldo - cantidad;
			return true;
		}
		else {
			return false;
		}
	}
} //Fin clase CuentaCorriente
```

Llama al constructor de la clase padre

Usa uno de los métodos que hereda

```java
/**
 * Cuenta de ahorros en una entidad financiera,
 * donde solo se puede retirar dinero hasta el saldo.
 * @version 1.0
 */
public class CuentaAhorros extends Cuenta  {

	public CuentaAhorros(String numero) {
		super(numero);
	}
} //Fin clase CuentaAhorros
```

Llama al constructor de la clase padre

Es importante notar que, aunque las clases hijas heredan todos los atributos de la clase padre, no los pueden usar directamente si tienen visibilidad privada (*private*). En ese caso deben usarlo mediante los métodos. Por ese motivo no se le puede asignar un valor directamente al atributo "número" en las clases hijas, y es necesario usar el constructor de la clase padre.

Palabra reservada *super*

En el constructor de la clase hija se reciben los valores de los atributos, e internamente, para asignar el valor a los atributos que heredó, hace uso del constructor de la clase papá. Esto se hace con la palabra reservada ***super***, pasándole los argumentos correspondientes.

El uso de *super* para llamar al constructor de la clase papá, debe ser la primera instrucción dentro del método constructor de la clase hija. No se pueden tener otras instrucciones antes.

La palabra reservada *super* es similar a la palabra *this*, pero se hace referencia a los atributos o métodos definidos en la clase padre. Por ejemplo, en el método "retirar" de la cuenta corriente se puede tener algo como:

```
public boolean retirar(double cantidad) {
      if ((super.getSaldo() + cupoSobregiro) >= cantidad) {
            super.saldo = super.saldo - cantidad;
            return true;
      }
      else {
            return false;
      }
}
```

Uso de objetos con herencia

La creación y el uso de los objetos de clases hijas es igual a la creación y uso de objetos de cualquier otra clase. Como ejemplo, se mostrará un código (en otra clase) donde se crean una cuenta corriente y una cuenta de ahorros; en cada una se consignan $500.000 y luego se intenta retirar $600.000 de cada una, teniendo en cuenta que la cuenta corriente tendrá un cupo de sobregiro de $200.000.

```
/**
 * Clase donde se crean unas cuentas bancarias (corriente y de ahorros),
 * se consigna dinero en ellas y luego se retiran
 * @version 1.0
 */
public class PruebaCuentas {
      public static void main(String[] args) {
            CuentaAhorros ahorros = new CuentaAhorros("123-a");
            CuentaCorriente corriente =
                                    new CuentaCorriente("456-c", 200000);

            ahorros.consignar(500000);
            corriente.consignar(500000);

            boolean pudoAhorro = ahorros.retirar(600000);
            boolean pudoCorriente = corriente.retirar(600000);

            if (pudoAhorro) {
                  System.out.println("Pudo retirar - ahorros");
            }
            else {
                  System.out.println("No pudo retirar - ahorros");
            }

            if (pudoCorriente) {
                  System.out.println("Pudo retirar - corriente");
            }
            else {
                  System.out.println("No pudo retirar - corriente ");
            }
      }
} // fin clase PruebaCuentas
```

La salida al ejecutar este código es:

```
No pudo retirar -cuenta de ahorros
Pudo retirar -cuenta corriente
```

Al observar el código anterior se puede notar que hay partes del código que son iguales para los dos tipos de objetos (ahorros y corriente), porque ambos son cuentas bancarias. De hecho, cuando se tiene herencia es posible definir variables de la clase que referencien objetos de las clases hijas.

Por ejemplo, en el siguiente fragmento de código, se crea primero un objeto CuentaCorriente y luego un objeto CuentaAhorros, pero ambos usando una sola variable, de tipo Cuenta. Esto es posible porque tanto las cuentas corrientes como las cuentas de ahorro son cuentas.

```
Cuenta bancaria;
```

Variable de la clase padre

```
// Primero se crea una cuenta corriente
bancaria = new CuentaCorriente("890-c",350000);
bancaria.consignar(450000);
System.out.println("Cuenta con saldo: $"+bancaria.getSaldo());

// Luego se crea una cuenta de ahorros
bancaria = new CuentaAhorros("654-a");
bancaria.consignar(450000);
System.out.println("Cuenta con saldo: $"+bancaria.getSaldo());
```

Es importante resaltar que esto solo es posible con clases hijas, no es posible definir una variable de una clase y asignarle un objeto de otra clase cuando no hay herencia entre ellas.
Por ejemplo, lo siguiente no es válido:

```
// Esto no es válido:
CuentaCorriente cuentaDinero = new CuentaAhorros("730-a");  //ERROR
```

En este caso, una cuenta de ahorros NO es una cuenta corriente.

Cuando se utiliza una variable de la clase padre para usar los objetos de las clases hijas hay una limitante: por medio de esa variable solo es posible usar los métodos que estén definidos en la clase papá, no los que estén definidos en la clase hija[2].

[2] Existen varias formas de superar esta limitante, pero eso no corresponde al alcance de este material.