



Ministerio de Producción
Presidencia de la Nación

Ministerio de Educación y Deportes

Subsecretaría de Servicios Tecnológicos y Productivos



Programa
111
mil
VOS PODÉS
SER UNO.

Sobrescritura y
super

Repaso de Herencia

Una clase hereda de otra, estableciendo una relación de **padre-hijo** o, “es un tipo de”.

Por ejemplo, un colectivo “es un tipo de” vehículo, pero con alguna funcionalidad especial. Sin embargo, mucha funcionalidad del vehículo se puede reusar (arrancar, frenar, doblar, etc.)

Sin embargo, algunas **funcionalidades de las clases hijo pueden hacerse de forma diferente**, por ej., en el método “subir” del colectivo, se cobra un pasaje y sino, el pasajero se debe bajar.

Sobrescribir un método en Java

¿Para qué?

- a. **Para cambiar la funcionalidad** de un método que está implementado en una clase de la cual heredamos.
- b. **Porque no tenemos alternativa.** Si el método de la clase “padre” **no está definido** (está declarado como **abstracto**) y la **clase actual NO ES abstracta**, lo debemos definir (más sobre esto en las siguientes clases)

Ejemplo: Productos del supermercado

Una supermercado tiene un **conjunto de productos**.

Una clase Compra se encarga de almacenar cuántos productos compra cada persona y sus cantidades.


Dicha clase **Compra calcula el total de la compra** como `producto.getPrecio() * cantidadComprada`.



La clase Compra

Por cada compra,
obtengo el precio del
producto y su cantidad.
Luego los multiplico y
sumo al total.

```
public class Compra {  
  
    List<ElementoCompra> productosYCantidad;  
  
    public Integer calcularTotal() {  
        Integer total = 0;  
        for (ElementoCompra elemento : productosYCantidad) {  
            Producto p = elemento.getProducto();  
            Integer cantidad = elemento.getCantidad();  
            total += p.getPrecio() * cantidad;  
        }  
        return total;  
    }  
}
```



El par de Producto y su cantidad

```
public class ElementoCompra {  
  
    Producto producto;  
    Integer cantidad;  
  
    public Integer getCantidad() {  
        return cantidad;  
    }  
  
    public Producto getProducto() {  
        return producto;  
    }  
}
```

El Producto

```
public class Producto {  
    String nombre;  
    int precio;  
  
    public int getPrecio() {  
        return precio;  
    }  
}
```

Analicemos la solución

¡La solución está bien!

Pero, ¿y si el cliente nos había dicho que algunos productos **tienen un descuento de acuerdo al día de compra?**

Opción 1: Clases diferentes

```
public class Producto {  
    String nombre;  
    int precio;  
  
    public int getPrecio() {  
        return precio;  
    }  
}
```

```
public class ProductoConDescuento {  
    String nombre;  
    int precio;  
    int descuento;  
  
    public int getPrecio() {  
        return precio - descuento;  
    }  
}
```

Problemas con la opción 1

Tenemos que tratar las clases de forma diferente, lo cual significa que en la clase Compra vamos a tener un listado de Productos y ProductoConDescuento, **separadas**.

Hay cosas repetidas entre Productos y, si bien la forma de calcular el precio es simple en el ejemplo, podría ser más compleja (incluyendo impuestos, el margen de ganancia del dueño, etc.).

En resumen, estamos **repitiendo** la forma de obtener el precio. Noten que también se repite la forma de identificar el producto (el nombre).

Opción 2: Usar herencia

Es fácil notar que un `ProductoConDescuento` **“es un tipo de”** `Producto`.

De esta manera, podemos reusar parte de la funcionalidad de `Producto`.

Y sobrescribir lo que deseamos extender

En este caso, en `ProductoConDescuento` queremos aplicar un valor de descuento (en este ejemplo, simplemente restando).

¿Por qué no hacemos un método nuevo? Lo que queremos evitar **preguntar si se trata de un `Producto` o un `ProductoConDescuento`.**

```
public class Producto {  
    String nombre;  
    int precio;  
  
    public int getPrecio() {  
        return precio;  
    }  
}
```

No es la misma variable, una es de Producto y la otra es de ProductoConDescuento. En este caso, si creamos un ProductoConDescuento, estamos “desperdiciando” la variable precio de Producto.

```
public class ProductoConDescuento extends Producto {  
    int precio;  
    int descuento;  
  
    public int getPrecio() {  
        return precio - descuento;  
    }  
}
```

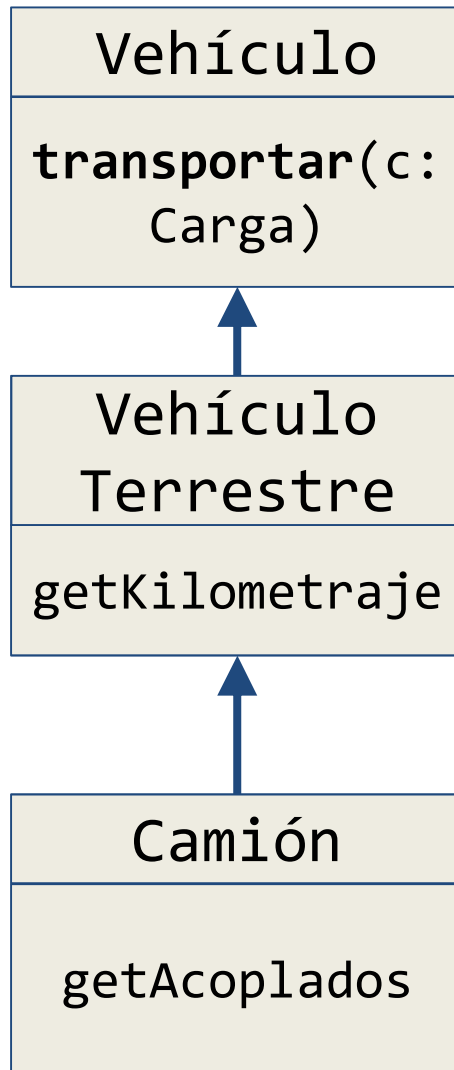
Usar un método de la clase padre

Para re-usar una funcionalidad de “arriba” en la jerarquía se utiliza la palabra reservada **super** (de super clase).

De forma similar a **this**, que referencia al objeto actual, **super** es una referencia a funcionalidad/atributos que pertenecen a la clase inmediatamente superior (aunque también estamos tratando con el mismo objeto).

```
public class ProductoConDescuento extends Producto{  
    int descuento;  
  
    public int getPrecio() {  
        return super.getPrecio() - descuento;  
    }  
}
```

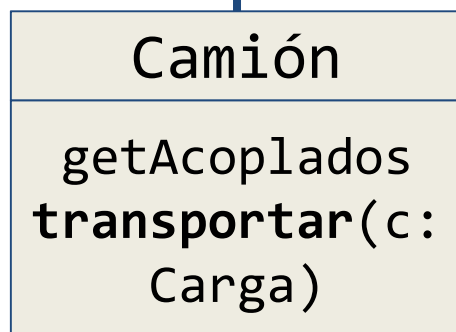
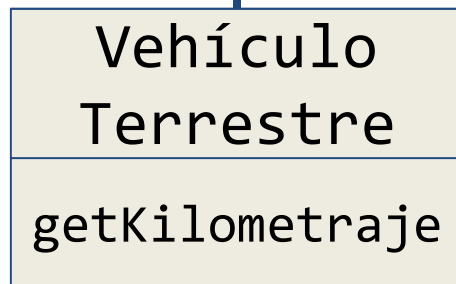
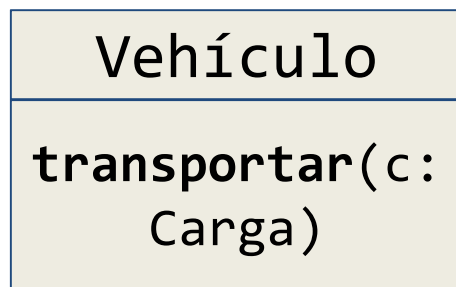
¿Qué sucede si tenemos más de un nivel de herencia?



```
Camión c = new Camión();  
// ¿Qué sección de código ejecuta  
// esta línea?  
c.transportar(carga);
```

En este caso, como **transportar** NO está definido en **Camión**, se busca en **VehículoTerrestre**. Como tampoco está allí, se busca en **Vehículo**.

Sobreescribimos transportar en Camión

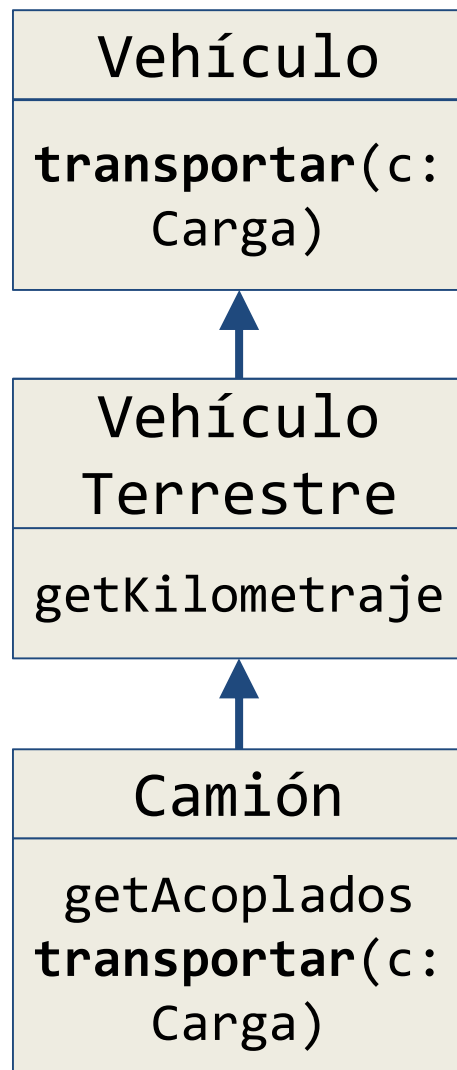


```
Camión c = new Camión();  
// ¿Y ahora, qué sección de código  
// ejecuta esta línea?  
c.transportar(carga);
```

En este caso, debido a que **transportar** está definido en **Camión**, se utiliza ese código.

No necesariamente **transportar** de **Vehículo** queda inutilizado: Puede ser que en algún punto del código, el **Camión** haga **super.transportar()**.

Ejemplo utilizando super en el código del Camión



```
//En Camión
public void transportar(Carga c) {
    this.acoplado.add(c);
    super.transportar(c);
}
```