INTRODUCCIÓN A LA PROGRAMACIÓN

Semana 6

ING. MARTIN POLIOTTO
Docente a cargo del módulo

Agosto 2020

DIPLOMATURA EN
NUEVAS TECNOLOGÍAS

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA SEU | UTN Facultad Regional Córdoba | Ministerio de Promoción del Empleo y de la Economía Familiar | Ministerio de Ciencia y Tecnología | CBA Entre Todos | Oficina de Montevideo

Semana 06

1. Herencia

1.1 Concepto

Las personas tendemos a asimilar nuevos conceptos basándonos en lo que ya conocemos. La **herencia** es una herramienta que permite **definir nuevas clases en base a otras clases**. La herencia es una relación entre clases, en la que una clase comparte la estructura y/o el comportamiento definidos en una (herencia simple) o más clases (herencia múltiple). La clase de la que otras heredan se denomina **superclase**, o **clase padre**. La clase que hereda de una o más clases se denomina **subclase** o **clase hija**.

La herencia permite que se puedan definir nuevas clases basadas de unas ya existentes **a fin de reutilizar el código**, generando así una jerarquía de clases dentro de una aplicación. Si una clase deriva de otra, esta hereda sus atributos y métodos y puede añadir nuevos atributos, métodos o redefinir los heredados.

No hay nada mejor en programación que poder usar el mismo código una y otra vez para hacer nuestro desarrollo más rápido y eficiente. El concepto de herencia ofrece mucho juego. Gracias a esto, lograremos un código mucho más limpio, estructurado y con menos líneas de código, lo que lo hace más legible.

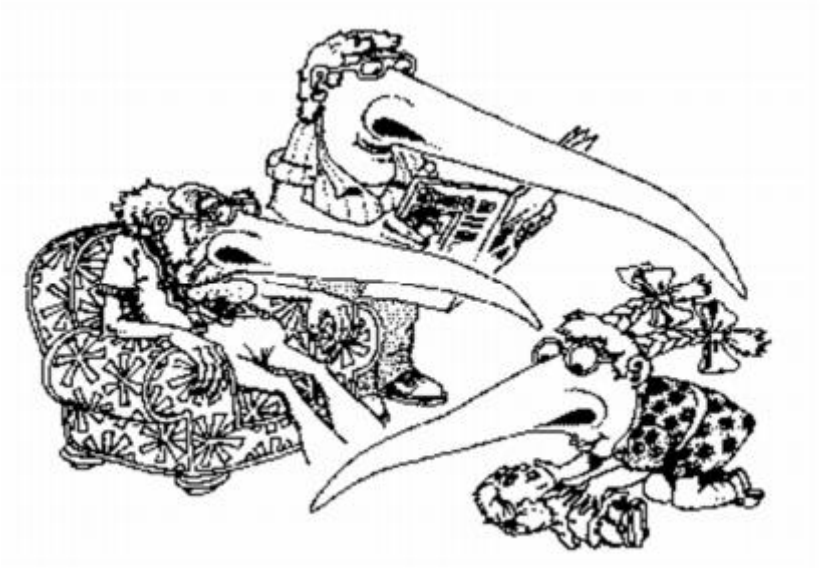


Fig. 5 - Herencia entre Clases

Algunas de las clases tendrán instancias, es decir se crearán objetos a partir de ellas, y otras no. Las clases que tienen instancias se llaman **concretas** y las clases sin instancias son **abstractas**. Una clase abstracta se crea con la idea de que sus subclases añadan elementos a su estructura y comportamiento, usualmente completando la implementación de sus métodos (habitualmente) incompletos, con la intención de favorecer la reutilización

1.2 Herencia en Java

No todos los lenguajes orientados a objetos soportan (o permiten) la herencia múltiple: este mecanismo es difícil de controlar a nivel de lenguaje y en general se acepta que la herencia múltiple lleva a diseños más complejos que los que se podrían obtener usando sólo herencia simple y algunos recursos adicionales como la implementación de **clases de interface**. El caso es que Java no soporta herencia múltiple, un ejemplo de un lenguaje que si soporta herencia múltiple es C++ (antecesor de nuestro Java).

Para indicar que una clase hereda de otra, se usa en Java la palabra reservada **extends** al declarar la clase derivada, nombrando luego de ella a la super clase de la misma. Esto hace que todo el contenido de la clase base esté presente también en la derivada, sin tener que volver a definirlo.

En sintaxis:

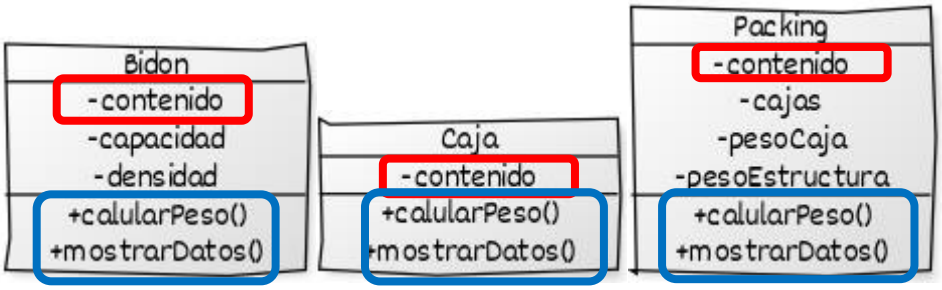
public class ClaseHija ***extends*** *ClaseBase{*

}

En Java, si al declarar una clase no se indica si la misma deriva de otra (o sea, no se escribe extends), entonces el lenguaje asume que esa clase hereda desde la clase **Object**, que es la base de toda la jerarquía de clases de Java (predefinidas o no: incluso las clases del programador derivan desde Object en Java). En otras palabras, si al definir, por ejemplo, una clase Vehiculo no se indica de quien hereda: *public class Vehiculo* entonces el compilador Java reemplaza la declaración anterior por la siguiente en el bytecode: *public class Persona extends Object*.

Este es el motivo por el cual el método **toString**( ) está presente en una clase aun cuando la misma no lo redefina: lo está heredando desde Object. La clase Object provee una serie de métodos que serán comunes a todo objeto. La clase Object provee los siguientes métodos (los nombramos sólo a los efectos documentales): toString() – finalize() – getClass() – clone() – equals() – hashCode() – wait() – notify() – notifyAll(). Y todos estos métodos están entonces presentes en toda clase Java predefinida o definida por el programador.

Supongamos que nos piden desarrollar una solución para una empresa de transporte de cargas que necesita organizar la carga de sus camiones. La empresa puede recibir packings, cajas sueltas y bidones que transportan líquido. El siguiente diagrama clases muestra lo que conocemos de estas cargas:



Como se puede observar, vemos que en las tres clases tenemos atributos y métodos que son iguales ya que los tres tienen los atributos *contenido*; y las tres tienen los métodos de *calcularPeso()* y *mostrarDatos()*.

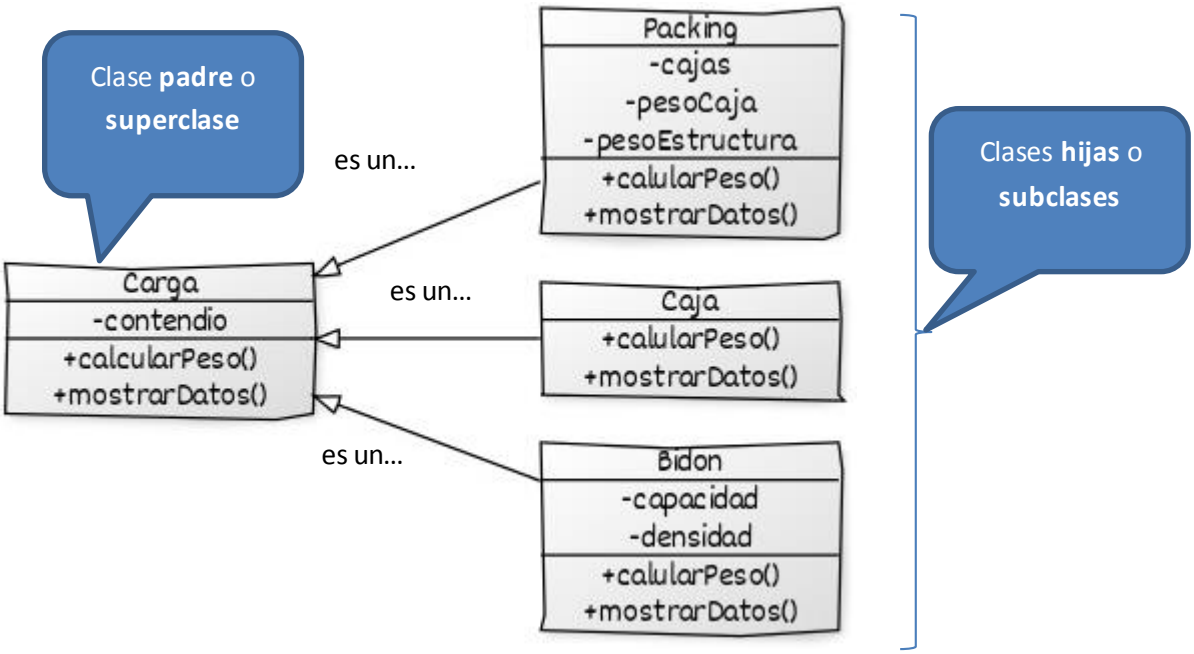
A nivel de código

```
public class Bidon{

private String contenido;

private int capacidad;

private int densidad;

//constructores, getters y
setters

public int calcularPeso(){

// código para calcular el peso

}

public String mostrarDatos(){

// código para devolver los
datos del bidón

}

}
```

```
public class Caja{

private String contenido;

//constructores, getters y
setters

public int calcularPeso(){

// código para calcular el peso

}

public String mostrarDatos(){

// código para devolver los
datos de la caja

}

}
```

```
public class Packing{

private String contenido;

private int cajas;

private int pesoCaja

private int pesoEstructura;

//constructores, getters y
setters

public int calcularPeso(){

// código para calcular el peso

}

public String mostrarDatos(){

// código para devolver los
datos del packing

}

}
```

Como se puede observar, a nivel de implementación, tenemos código repetido, tanto en los campos pero fundamentalmente en los comportamientos. Cuando detectamos en el análisis del dominio que dos o más **entidades** tienen atributos en común pero principalmente se comportan de manera similar, podemos pensar que existe una entidad de un nivel abstracción más alto en el que podamos generalizar las características comunes. En las clases más específicas no solo podremos reutilizar lo generalizado sino también especializarlo en las implementaciones particulares, resultando:



El criterio para colocar cada atributo y/o comportamiento en las clases es el siguiente:

- Los atributos comunes suben a la superclase **siempre y se quitan de las subclases**
- Los métodos también suben pero validamos que:
    - **si puede implementarse** en la clase padre, entonces **se elimina** de la clase hija
    - Si **puede implementarse parcialmente o bien no podemos desarrollar el método** en la superclase (porque faltan atributos necesarios), entonces **se escriben dichos métodos tanto en la clase base como en sus derivadas**.

En código:

```
public class Carga{

protected String contenido;

public Carga(){

}

public int calcularPeso(){

}

public String mostrarDatos(){

}

}
```

```
public class Bidon extends Carga{

private int capacidad;

private int densidad;

public Bidon(){

super();

}

public int calcularPeso(){

// código para calcular el peso

}

public String mostrarDatos(){

// código para devolver los datos del bidón

}

}
```

```
public class Caja extends Carga{

public Caja (){

super();

}

public int calcularPeso(){

// código para calcular el peso

}

public String mostrarDatos(){

// código para devolver los datos de la caja

}

}
```

```
public class Packing extends Carga{

private int cajas;

private int pesoCaja

private int pesoEstructura;

public Packing (){

super();

}

public int calcularPeso(){

}

public String mostrarDatos(){

}

}
```

Resultando código mucho más limpio, estructurado y con menos líneas de código, lo que lo hace *más legible y reutilizable*. Esto significa que ahora si queremos añadir más clases a nuestra aplicación como por ejemplo una clase BultoFragil lo podemos hacer de forma muy sencilla ya que en la clase padre (Carga) tenemos implementado parte de sus datos y de su comportamiento, y solo habrá que implementar los atributos y métodos propios de esa clase.

Ahora bien en el código que se ha escrito se resaltan en las palabras reservadas: **extends**, **protected** y **super**.

1.2.1 Extends, protected y super

- **extends**: Esta palabra reservada, indica a la clase hija cual va a ser su clase padre, es decir que por ejemplo en la clase Caja al poner *public class Caja extends Carga* le estamos indicando a la clase *Caja* que su clase padre es la clase *Carga*.
- **protected**: sirve para indicar un tipo de visibilidad de los atributos y métodos de la clase padre y significa que cuando un atributo es *protected* o protegido, es visible ese atributo o método desde cualquiera de las clases hijas (y no desde otra clase).
- **super**: sirve para referenciar los elementos de la clase Padre. En caso de usarse con paréntesis, permite reusar el/los constructores de la clase base.

1.2.2 Niveles de acceso

El cuadro que se muestra a continuación sintetiza la visibilidad de los elementos definidos (atributos o métodos) según su modificador de acceso.

Visibilidad	Public	Protected	Default	Private
Desde la misma Clase	SI	SI	SI	SI
Desde cualquier Clase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase fuera del mismo Paquete	SI	SI, a través de la herencia	NO	NO
Desde cualquier Clase fuera del Paquete	SI	NO	NO	NO

Recordar que el tipo **default** se indica simplemente con la ausencia del modificador de acceso. No es una palabra reservada como public, protected o private.

1.3 Clases abstractas y Polimorfismo

En muchos casos una clase se diseña pero no para ser instanciada, sino para permitir el agrupamiento de características comunes, facilitar le herencia, y posibilitar el **polimorfismo**. Por ejemplo, en la jerarquía de clases que representan cargas de la empresa de transporte, no se esperaría que en una aplicación se creen instancias de la clase Carga, simplemente porque dichas instancias en la práctica serían incompletas. En la realidad solo existen bidones, cajas o packings (clases concretas).

Se puede indicar al compilador Java que no permita instanciar clases que cumplan esos requisitos de abstracción, declarando a dichas clases como **abstractas**. La clase Carga podría serlo:

public abstract class Carga.

Las clases así declaradas se dicen clases abstractas y el intento de instanciarlas provoca un error de compilación. No se pueden crear objetos de estas clases, es decir:

Carga unaCarga = new Carga(); // no compila

Carga otraCarga = new Caja(); // si compila y es válido para //cualquier clase concreta que pertenezca a la jerarquía de Cargas.

Esta última línea implica que *otraCarga* será una variable **polimórfica**. Lo cual quiere decir que la misma variable *otraCarga* recibirá por ejemplo el mensaje calcularPeso() y en algunas ocasiones responderá como Caja pero en otras ocasiones podría responder como Packing o como Bidon dependiendo del tipo

específico de objeto que este referenciando en tiempo de ejecución.

En general:

- Es posible definir siempre: *ClaseBase objeto = new ClaseHija() (siempre que ClaseHija sea concreta);*
- También es posible definir como **abstract** un método de la una clase. Por ejemplo:

public abstract void unMetodo();

Notar que el método queda sin cuerpo (no se abren y cierran llaves {}) luego de la cabecera.

- Si una clase tiene al menos un método *abstract* entonces la clase pasa a ser abstracta.
- Las clases abstractas pueden tener tanto métodos implementados como abstractos
- Si una clase hereda de otra que en su definición existe un método abstracto, entonces es obligatorio que la clase hija redefina el método abstracto (lo implemente). De lo contrario pasaría a ser ella también una clase abstracta.

1.4 Arreglo de objetos de clases diferentes

El máximo nivel de polimorfismo en Java, se logra definiendo una referencia a **Object**: Como esa clase es la base de la jerarquía de todas las clases en Java, incluidas las del programador, una referencia a **Object** podrá apuntar a objetos de cualquier clase en un programa.

Lo siguiente es válido:

```
Object x = new Caja();
```

El polimorfismo permite definir estructuras de datos genéricas, esto es, estructuras que son capaces de contener objetos de distintos tipos, pero **siempre** de la misma jerarquía. Si por ejemplo definimos un arreglo de cargas de la forma:

```
Carga vec[] = new Carga[3];

vec[0] = new Caja();

vec[1] = new Packing();

vec[2] = new Bidon();
```

Con la definición del vector anterior se tiene una estructura que me permite almacenar objetos de diferentes clases, siempre que dichas clases sean hijas de Carga (estén en la misma jerarquía).

El procesamiento de un arreglo de referencias polimórficas puede hacerse sin mayores problemas. El siguiente ciclo, recorre el arreglo e invoca al método calcularPeso() para cada objeto carga. Como ese método está definido en la clase base (Carga) y cada derivada lo hereda o lo redefine, entonces cada invocación funciona sin

problemas y la JVM invoca siempre a la versión correcta del método:

```
int suma = 0;

for (int i = 0; i < vec.length;i++){

        suma += vec[i].calcularPeso(); // llamada polimórfica
```

Una última apreciación en relación a la redefinición de un método heredado:

- Redefinir un método es la acción que tiene lugar en una clase derivada, para volver a definir un método heredado desde la **superclase** pero tal que su especificación no es adecuada a la derivada. Para redefinir un método, la clase derivada debe volver a escribir su cabecera pero tal cual como está en la clase base, excluyendo eventualmente el calificador de acceso. Al redefinir un método, en la clase derivada valdrá entonces el redefinido, y no el heredado.
- Cuando se redefine un método es posible:
  - ✓ especializar el comportamiento heredado (caso del toString())
  - ✓ Refinar el método para dar un comportamiento diferente al heredado.