**HERENCIA**

Permite reutilizar código, definiendo nuevas clases a partir de otras ya existentes. Si una clase hereda de otra (la especializa), hereda todos sus métodos y atributos públicos o protegidos, pudiendo declarar más atributos y métodos, además de redefinir los heredados. Para aplicar la herencia se usa la palabra **extends** (la subclase hereda de la superclase, **extiende**).

*Ejemplo 01 Eclipse*

***Niveles de protección***

En la implementación de la clase subclase podemos utilizar todos los atributos públicos y protegidos de la clase superclase, pero nunca los privados. **Protected** indica que estos se comportan como públicos en presencia de herencia. Si no hay herencia, protected se comporta como atributo público dentro del paquete que contiene a la clase.



**Constructoras**

Para inicializar a un objeto de una subclase es necesario invocar a alguna constructora de su superclase. En caso de no poner la llamada explícita, se invocará por **defecto** a la constructora sin argumentos de la superclase. La invocación a la constructora de la superclase debe ser la primera instrucción de cualquier constructora de la subclase. Para invocar a la constructora sin argumentos se pone super(); y para invocar a una constructora con argumentos, super(a1, …, an);.

*Ejemplo 02 Eclipse*

**Colisión de nombres con herencia**

Si una subclase contiene un atributo con el mismo nombre que su superclase, entonces toda referencia a ese atributo en la implementación de la subclase se referirá al de la subclase.

*Ejemplo 03 Eclipse*

**Métodos**

Se pueden heredar y redefinir (sobreescritura de un método, se resuelve en **tiempo de ejecución** debido a la existencia del polimorfismo.). Al invocar a un método se busca si existe en la clase del objeto. Si existe, se ejecuta. Si no, se sube por la jerarquía de clases hasta encontrarle y se ejecuta. También es posible redefinir un método e invocar dentro de su cuerpo al equivalente en su superclase. Cuando se redefine un método es posible invocar al de la superclase utilizando el objeto super: super.m();. Utilizar super supone subir por la jerarquía de clases hasta encontrar el método con ese nombre y argumentos del mismo tipo.

*Ejemplo 04 Eclipse*

**Modificador final**

Al ponerle **final** a un método indica que no puede ser redefinido. Al ponerle final a una clase indica que no se puede heredar de dicha clase.

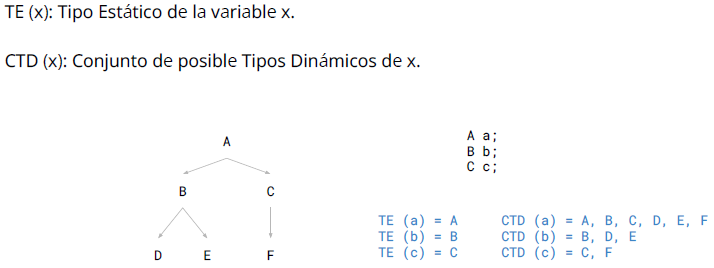
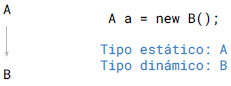
**Sobrecarga**

Dentro del mismo ámbito podemos tener métodos con el mismo nombre, tipo de retorno, mismo número de argumentos, pero de distinto tipo o número de parámetros. La sobrecarga se resuelve en **tiempo de compilación**.

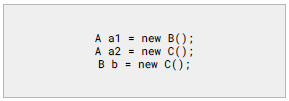
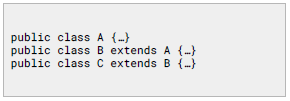
*Ejemplo 05 Eclipse*

**POLIMORFISMO**

Una variable polimórfica es una variable que se declara de un tipo (clase) pero en tiempo de ejecución puede contener valores de distinto tipo (subclase). Polimorfismo significa que hay un nombre con distintas definiciones. Implica que toda variable tiene un tipo estático (tipo asociado en la declaración) y otro dinámico (tipo del objeto contenido dentro de una variable).



Dada una jerarquía de clases podemos asignar a un elemento de una superclase cualquiera de sus subclases.



Cuando una clase B extiende a una clase A hay que entender que la clase A está contenida en la clase B (por eso B la extiende). Por ejemplo, si A = animales y B = perros, está claro que un perro siempre es un animal, pero un animal no siempre es un perro.

**Vinculación dinámica**

Al establecer una jerarquía de clases, algunas pueden sobreescribir métodos. En tiempo de ejecución se decide qué método se debe aplicar (vinculación dinámica) de acuerdo a las siguientes reglas:

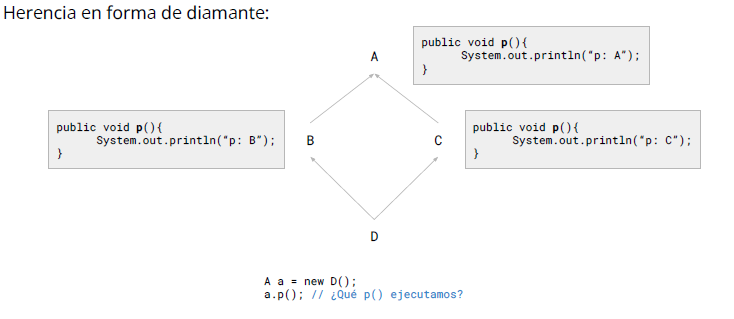


1. Se comprueba que p(…) está definido en A (tipo estático) o en alguna superclase de A. En otro caso, error.
2. Desde B subiendo por sus superclases, buscamos la primera implementación de p(….) y la aplicamos.

*Ejemplo 06 Eclipse*

**Herencia múltiple**

Una clase tiene más de una superclase. En Java está prohibida, aunque puede obtenerse un comportamiento similar utilizando interfaces. El principal problema radica en la herencia en forma de diamante.



**Casting**

Si B contiene algún método no declarado en A, ni en ninguna de sus posibles superclases, entonces hay que hacer casting. Por ejemplo, si B implementa un método m que no está implementado en B, y tenemos la asignación:

A a = new B();

Entonces para llamar con a al método m debemos usar la siguiente sintaxis:

((B) a).m(...);

((B) a) hace un casting para que el tipo del objeto A pase a ser B.

**Tipo de un objeto en ejecución**

Antes de hacer un casting se puede preguntar por el tipo del objeto en tiempo de ejecución. Podemos hacerlo de dos formas:

1. Utilizando getClass(), heredado de la clase Object. Esto devolverá la clase exacta de un objeto en tiempo de ejecución.
2. Utilizando instanceof. Esto comprobará si el objeto tiene un tipo compatible con la clase correspondiente.

