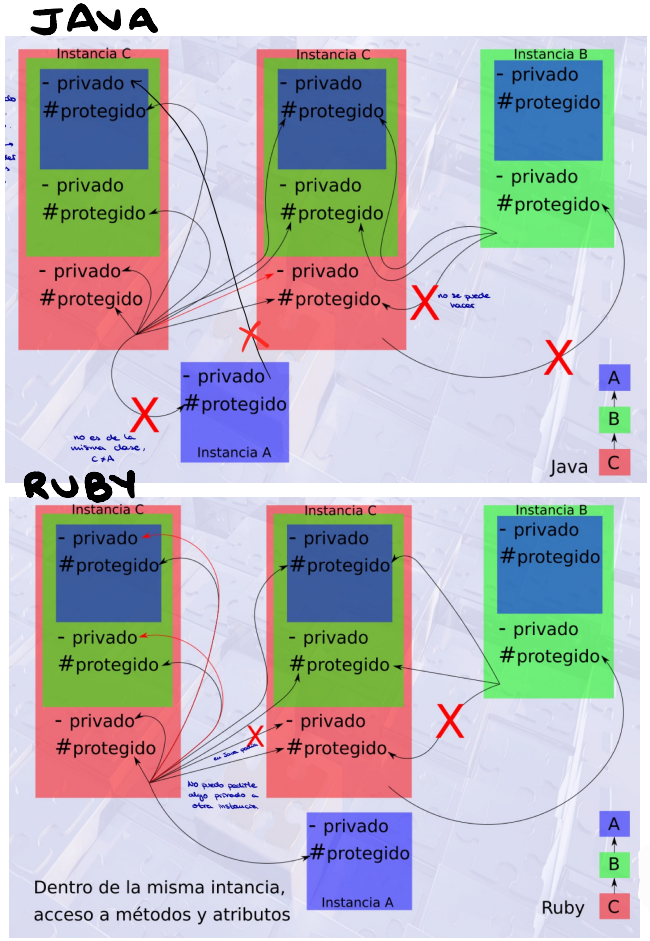

Herencia

Java

◇ Los atributos privados de instancia se 'heredan' (al crearse en el constructor), pero no son accesibles desde la subclase. Igual en Ruby.
◇ Los métodos de clase se heredan, pero quedan ligados a la clase donde se definen. Pueden sobreescribirse, pero lo que hacemos en realidad es ocultar el antiguo (NO @Override). Sin embargo, no se comportan de forma polimórfica. No se pueden redefinir métodos *final*.
◇ Si modificamos un atributo de clase, la modificación será visible desde la clase donde se modficia hacia abajo en el árbol de herencia, siempre y cuando en la clase donde modificamos definamos un nuevo método static para consultarlo. En otro caso, utilizará el consultor static de la clase padre, que está ligado a ella, por lo que no imprimirá el valor modificado. Los atributos de clase en realidad no se sobreescriben, sino que se ocultan.
◇ Los constructores se heredan siempre que no tengan argumentos. En otro caso, hay que definirlos explícitamente.
◇ Se pueden modificar métodos que se redefinen, en cuanto a tipo o número de parámetros. En realidad estaríamos sobrecargando el método, sin ocultar el de la clase padre. Se puede cambiar el tipo de retorno, siempre que sea una subclase del original. También se puede cambiar la visibilidad, siempre que sea menos restrictiva que la del original.

Ruby

◇ Los atributos de instancia de la clase no se heredan, cada clase tiene el suyo. Sin embargo, los métodos de clase sí se heredan. Se pueden sobreescribir. Si se modifica un atributo de clase, se modifica en todo el árbol de herencia, sin necesidad de instanciar la clase hija.
◇ Se hereda el constructor.
◇ Se pueden modificar los métodos que se sobreescriben, en cuanto al número o el tipo de parámetros. No existe la sobrecarga, por lo que al hacer eso, se 'oculta' el método antiguo.

Pseudovariable super

Permite invocar métodos de la clase padre. En Java, puede llamarse de dos formas:

super.met1()	Invoca el método 'met1' de la clase padre.
super(args)	Invoca el constructor de la clase padre con los argumentos *args*. Únicamente en la primera línea del constructor.

En Ruby, *super* solo puede llamar al método de la clase padre al que sobreescribe. Tiene tres variantes:

super	Invoca con los mismos argumentos.
super()	Invoca sin argumentos (¡OJO!).
super(args)	Invoca con los argumentos *args*.

Visibilidad

Java

◇**private** solo es accesible desde el código de la propia clase (ámbito de instancia o de clase). Desde instancias se puede acceder a elementos privados de la clase o de otras instancias distintas de la misma clase. **package** indica que es *público dentro del paquete* y privados fuera.
◇**protected** es público dentro del paquete (con independencia de la herencia o no que exista), y *accesible* desde *subclases de otros paquetes*. Para acceder a elementos protegidos de una **instancia distinta** (ambito clase/instancia), la instancia debe ser de la misma clase que la **propietaria** del código desde el que se realiza el acceso o de una subclase de de esta y además el elemento debe estar declarado en la clase propietaria del código desde el que se realiza el acceso o en una superclase de la misma.
Es decir, si se llama desde `Hija h1` en otro paquete, para poder acceder a un protected debe ser un `Hija h2` o un `Nieta n1`, no vale un `Padre p` aunque una hija sea un padre. Además lo que se quiere acceder debe estar al menos en la *clase hija o en una superclase* (no puedo acceder a un protegido de una subclase).

```
public class C extends B // Dentro de un método...
B b = new B(); b.protegidoB = 666; // Fallo, otro paquete,
     protegido, es de un padre
C c = new C(); c.protegidoA = 555;// Correcto
D d = new D(); d2.protegidoB = 555;// Correcto, otro paquete,
     protegido, es de una subclase
// que desde donde se intenta acceder
d2.protegidoD = 555;// Fallo, otro paquete, protegido, el
     atributo está declarado en una subclase
```

Ruby

◇ *Atributos* e *initialize* **siempre privados**, métodos públicos pero se puede cambiar. Un especificador afecta a **todo lo que viene después**.
◇ **private**, un método privado no puede ser usado como receptor de mensaje explícito, salvo `self`. Solo se puede usar un método privado de la propia instancia. Si $B < A$, desde *ámbito de instancia* (resp clase) de **B** se puede llamar a métodos de instancia (resp clase) *privados* de A. No se puede llamar a métodos privados de *clase (resp. instancia)* desde ámbito de *instancia (resp. clase)*. **En resumen**: se puede llamar a métodos privados de super clases sin mezclar los ámbitos.
◇ Los métodos **protected** son *privados*, pero pueden ser invocados con **receptor de mensaje explícito** (debe ser la misma o una subclase). *No existen* métodos protegidos *de clase*. La clase que invoca debe ser la misma o una subclase de **donde se declaró el método**.

Clases abstractas e interfaces

Clases abstractas. No instanciables. Tienen al menos un método sin implementar, también marcado como *abstract*.
Interfaces. No son clases. Palabra clave *interface*. Son una colección de métodos públicos (*default*, *static*, o implícitamente *abstract*) y de constantes (implícitamente *public*, *static* y *final*).
En ambos casos, si una clase hereda o implementa, debe definir todos los métodos que queden sin definir (excepto *default*). De lo contrario, esta clase debe marcarse como *abstract*. No se pueden instanciar. No existen en Ruby.
Se permite herencia múltiple entre inferfaces. Los métodos *default* pueden sobreescribirse. Los métodos *static* se pueden llamar desde otros métodos *static* o *default* de la interfaz, y también desde fuera. No se pueden sobreescribir (no se heredan).
Si una clase implementa una o varias interfaces (ej: In), y hay conflicto de nombres al llamar a un método de esta última, debe ponerse *In.super.metodo()*.

Polimorfismo

Es necesario *downcast* para que cuadre el tipo estático. También en llamada a métodos o al añadir a un ArrayList.

```
Hijo1 h1 = new Hijo1();
Padre p = new Hijo1();

p.doSomething(); // "hijo1" (tipo dinámico)
h1.doSomething(); // "hijo1"

// UPCAST: automático. Innecesario
((Padre) p).doSomething(); // "hijo1"
((Padre) h1).doSomething(); // "hijo1"

// DOWNCAST: Evita errores de compilación.
//p.doHijo1(); // No compila
((Hijo1) p).doHijo1(); // Compila, y funciona
p = new Hijo2();
//((Hijo1) p).doHijo1(); // Compila, pero runtime error.

Padre pp = new Padre();
//Hijo1 hh1 = pp; // Un Hijo1 no puede apuntar a Padre
//Hijo1 hh1 = (Hijo1) pp; // Compila, pero al ejecutar explota

pp = new Hijo1();
//Hijo1 hh1 = pp; // No compila, misma razón que antes
Hijo1 hh1 = (Hijo1) pp; // Compila, y ejecuta bien (downcast)
hh1.doSomething();

//Hijo2 h2 = new Hijo1();
Hijo1 hhh1;
//Hijo2 h2 = hhh1;
//Hijo2 h2 = (Hijo2) hhh1;
Padre ppp = new Hijo1();
//Hijo2 h2 = (Hijo2) ppp; // Compila, pero explota
```