

Especificadores de acceso

Java

Visible en	Mismo paquete		Otro paquete
	Clase	Otra	Otra
private	✓		
package	✓		
protected	✓	✓	
public	✓	✓	✓

Ruby

Visible en	Desde el propio objeto	Clase	Otra
private	✓		
protected	✓	✓	
public	✓	✓	✓

Especificadores de acceso (herencia)

Java

Visible en	Subclase	
	Mismo paquete	Otro paquete
private		
package	✓	
protected	✓	✓
public	✓	✓

Ruby

Visible en	El mismo objeto en la subclase (self)	Cualquier objeto de la subclase
private	✓	
protected	✓	✓
public	✓	✓

En Ruby, los métodos *protected* solo pueden llamarse desde un método de instancia. Además, pueden ser llamados únicamente desde una instancia de la clase que lo define, o de una subclase suya.

Diagramas de clases

Nombre ^{<i>multiplicidad</i>}
[visibilidad] nombreAtributo [:tipo[multiplicidad]][=valorInicial]
.
.
.
[visibilidad] nombreMétodo ([lista parámetros]):[tipo retorno]
.
.
.
+ Pública - Privada ~ Paquete # Protegida

Diagramas de secuencia

Los argumentos de los fragmentos de interacción descritos a continuación se indican dentro, en un recuadro.

alt	Se ejecuta si se cumple la condición
break	Se ejecuta si se cumple la condición y no se continúa
loop	Se realiza arg veces el fragmento
opt	Como alt pero con un solo fragmento

Diagramas de comunicación

Tipos de enlace (objetoX envía mensaje a objetoY):

Global	G El ámbito de objetoY es superior al de objetoX
Asociación	A Existe una relación fuerte y duradera
Parámetro	P objetoY es pasado como parámetro de objetoX
Local	L objetoY es referenciado en un método de objetoX
Self	S objetoX siempre se conoce a sí mismo

Estructuras de control:

[condición] Selectivas.

*[condición] Iterativas.

Relación *es-un*. Diferente a composición.

Métodos y atributos

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 modifica 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. Hay que poner el require_relative para heredar. Si se modifica un atributo de clase, se modifica en todo el árbol de herencia.
- ◊ 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.meti() Invoca el método 'meti' 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*.

Clases abstractas e interfaces

Clases abstractas. Palabra clave *abstract*. 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 interfaces. 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
```

```
try{ ...
a.metodo();
...
}catch (UnTipoDeExcepcion e) {
}catch (OtroTipoDeExcepcion o){
}finally {
}
void metodo() throws UnTipoDeExcepcion, OtroTipoDeExcepcion {
```

```
if (algotpasa)
throw new UnTipoDeExcepcion(mensaje_error1);
```

```
if (otracosapasa)
throw new OtroTipoDeExcepcion(mensaje_error2);
```

```
}
```

```
begin
...
a.metodo
...
```

```
rescue UnTipoDeExcepcion => e
rescue OtroTipoDeExcepcion => o
else
ensure
end
def metodo
```

```
...
if (algotpasa)
raise UnTipoDeExcepcion, 'mensaje_error1'
...
if (otracosapasa)
raise OtroTipoDeExcepcion, 'mensaje_error2'
...
end
```

Para comparar estado en **Java** se redefine *equals(obj)*

A la hora de redefinir hay que tener en cuenta como casos base, obj == null, obj == this, obj.getClass().getSimpleName().equals("Mi Clase").

En **Ruby**, ==, !=, .equal? y .eq!? comparan identidad por defecto. Para comparar estado, redefinir ==, ya que .equal? se usa internamente para determinar la identidad.

En **Java**: Método clone() definido en Object como funcionalidad general de todos los objetos, su significado por defecto es la copia superficial.

En **Ruby**: La funcionalidad de clone y dup es la misma, copia superficial de objetos, la diferencia es que clone copia el propio objeto teniendo en cuenta todo lo definido en él y dup copia el objeto teniendo en cuenta las propiedades definidas en la clase a la que pertenece. Si se desea realizar la copia profunda debe realizarla el programador redefiniendo clone o dup.