**HERENCIA**

**java**

-super permite acceder a cualquier metodo de la clase padre
-permite referenciar al constructor de la clase padre , debe aparecer en la primera linea
-No se pueden redefinir metodos final ni private
- usar @override para redefinir
-Se puede cambiar el tipo deaccesibilidad a una menos restrictiva de protected a public por ejemplo
- El valor retornado puede ser una subclase del valor del metodo ancestro
-No permite la redefinición de métodos de clase al mismo nivel
que de instancia. Se cambia el metodo pero no se redefine no se puede usar super.metododeclase()
-Aunque pueden existir métodos de clase con el mismo nombre en
una jerarquía de clases, no se obtienen los mismos resultados
que a nivel de instancia

**ruby**

-solo permite acceder al mismo metodo que se esta redefiniendo
-si se usa sin argumentos se pasan los mismos que en el redefinido
- Siempre se redefine , no hay sobrecarga
-Si no se usa super en initialize de la clase hija los atributos de instancia de la clase padre no existiran eb hija y dara error al llamar a metodos que los usen . Si no usan esos datos y por ejemplo solo hace puts los puts saldran vacios .

**VISIBILIDAD**

**java**

-private:solo es accesible desde codigo de la propia clase ya sea desde el ambito de instancia o clase , desde el ambito de instancia se puede acceder a elementos privados de la misma clase , Se puede acceder a elementos privados de otra instancia si es de la misma clase (puede recibirse como param).
-de paquete:no poner ningun especificador significa visibilidad de paquete , estos elementos son publicos dentro del paquete y privados respecto al exterior del paquete.
-protected:Son publicos dentro del mismo paquete con independencia de la relaccion de herencia que exista, tambien son accesibles desde subclases de otros paquetes , Dentro de una misma instancia se puede acceder a elementos definidos en cualquiera de sus superclases con independencia del paquete en el que se encuentren,
para poder acceder a elementos protegidos de una instancia distinta : tiene que ser de la misma clase que la propietaria del codigo desde el que se realiza el acceso o de una subclase de la misma , tiene que estar declarado en la clase propietaria del codigo desde el que se accede o en una superclase de la misma .
Si las clases involucradas estan en el mismo paquete los elementos protegidos son accesibles siempre
-public : se utiliza en cualquier lado , son solo utilizables dentro del paquete en el que se definen.espe

| 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 |

**ruby**

los atributos son siempre privados
los metodos son por defecto publicos pero se puede modificar si se especifica
initialize siempre es privado no se puede cambiar
- Solo se puede usar un metodo privado en la propia instancia
- si b hereda de a Desde el ambito de instancia de b se puede llamar a metodos de instancia privados en a lo mismo ocurre con los de clase
- No se puede acceder a métodos privados de clase desde el ámbito de instancia
-No se puede acceder a métodos privados de instancia desde el
ámbito de clase.
-Hay que tener en cuenta que una clase y una instancia de esa clase no pertenecen a la misma clase. (Esto influye en protected)

---

En Java se puede acceder a atributos y metodos privados desde una instancia, ambito de clase o ambito de instancia (dentro del codigo de la clase donde se declara o implementa) mientras que en ruby no se puede hacer nada de eso.
Para dar visibilidad en Ruby se coloca el especificador y todos los metodos que vayan delante tendran la visibilidad especificada. Tambien se puede poner especificador :nombre_metodo_instancia para los de instancia o especificador_class_method :nombre_metodo_clase para los de clase.

**CLASES ABSTRACTAS**

-usa la palabra abstract (public abstrac int hola;)
-No se puede instanciar una clase abstracta
- Se puede declarar una variable usando la clase abstracta como tipo estático
- Obligan a sus subclases a implementar una serie de métodos ,Si no implementan algún método, también serán abstractas

**java**

-permite clases abstractas sin metodos abstractos
-obligatorio @override , si no se redefine el metodo la clase que hereda sera abstracta

**ruby**

-clases no instanciables , se hace privado el metodo new de la clase padre y se hace publico en las clases derivadas

**INTERFACES**

- Define un contrato/disenio que cumplen todas las clases que realizan la interfaz
-Cada interfaz define un tipo y se pueden declarar variables de ese tipo

**java**

-usar @override
-Una clase puede realizar varias interfaces
-Una interfaz puede heredar de una o más interfaces
-una interfaz solo puede tener : constantes,signaturas de metodos,metodos tipo default , el equivalente a los metodos de clase static.
-los tipo default y static pueden tener asociada una implementacion
-No pueden ser instanciadas, solo realizadas por clases o extendidas por otras interfaces,pero se pueden declarar variables con el tipo de la interfaz y esas variables referenciar a una instancia de una clase que implemente a dicha interfaz(Interfaz int = new Class() y Class implements Interfaz).
-Se pueden redefinir los métodos default en interfaces que heredan y en clases que realizan esa interfaz
-Una clase puede heredar de una clase y además realizar varias interfaces
Una clase abstracta puede indicar que realiza una interfaz sin implementar alguno de sus métodos. (fuerza a hacerlo a sus descendientes no abstractos)
-Una clase puramente abstracta se parece a una interfaz pero Java no permite herencia múltiple para eso existe las interfaces.

**CLASES PARAMETRIZABLES**

**java**

-Permite pasar tipos como parámetros a clases e interfaces
-Se usan tipos al declarar atributos , tipo devuelto por un metodo,tipo de un marametro en un metodo.
-se puede forzar a que el tipo suministrado a una clase parametrizable sea :
-Tenga que ser subclase de otro *class Clase <t extends ClaseBase>*
-tenga que realizar una interfaz *class Calse < T extends Interfaz >*

**POLIMORFISMOS**

-Si B es un subtipo de A se pueden utilizar instancias de B donde se esperan instancias de A
- Tipo estatico : tipo del que se declara la clase
-Tipo Dinamico : clase al que pertenece el objeto referenciado en un momento determinado por la variable
-Ligadura estatica : El enlace del código a ejecutar asociado a una llamada a un método se hace en tiempo de compilación
-Ligadura dinamica:El tipo dinámico determina el código que se ejecutará asociado a la llamada de un método
-El tipo estatico limita lo que puede referenciar una variable (instancias de su mismo tipo o de subclases )
- El tipo estatico limita los metodos que pueden ser invocados (Los disponibles en las instancias de la clase del tipo estatico) ejemplo :

*class profesor extends persona{*
*public int metodoquenotieneclasepersona(){return 0;}*
*}*
*main {*
*Persona p =new Profesor();*
*p.metodoquenotieneclasepersona(); // error de compilacion , persona no tiene el metodo*
*p = new Object() // Error de compilacion object no es subclase de persona*
*}*

*CASTINGS*

-Indican al compilador que considere temporalmente que el tipo de una variable es otro solo para lainstruccion en la que aparece.

-*Downcast :*El compilador considera que el tipo de la variable es una subclase del tipo con el que se declaro (permite invocar metodos de la subclase)
-Upcast: El compilador considera el tipo de la variable es superclase del tipo con el que se declaro (innecesario cpd)

-Las operaciones de castingo no realizan ninguna transformacion en el objeto ni cambian su comportamiento.
Ejemplo

---

*((Persona) profesor).hablar; // habla como un profesor*
*Persona p2 =profesor;*
*p2.hablar(); //habla como profesor*

---

*Persona p = new Persona ( ) ;*
*Profesor profe = ( Profesor ) p ; / / Error*
*profe = ( ( Profesor ) new Persona ( ) ) ; / / Error*
*( ( Profesor ) ( ( Object ) new Profesor ( ) ) ) . impartirClase ( ) ; / / OK*

*-------Ejemplo clases hermanas alumno y profesor que heredan de persona --------*
*Alumno a1 = new Profesor ( ) ;/ / Error de compilación . Tipos incompatibles*
*Alumno a2 = ( Alumno ) new Profesor ( ) ;/ / Error de compilación . Tipos incompatibles*
*Alumno a3 = ( ( Alumno ) ( ( Object ) new Profesor ( ) ) ) ;/ /Error en tiempo de ejecución Profesor canot castto alun*
*------FIN--------*

*DETALLES :*

-Con ligadura dinámica, siempre se comienza buscando el código asociado al método invocado en la clase que coincide con el tipo dinámico de la referencia (java).
-Si no se encuentra se busca en la clase padre
-Así sucesivamente hasta encontrarlo o hasta que no existan ascendientes

**VISIBILIDAD RUBY**

```ruby
class Padre
    protected
    def protegido #Se puede acceder a un metodo protegido desde una subclase
        privado #Error si hicieramos instancia.privado
        puts "Metodo protegido"
    end
    private
    def privado
        protegido #Correcto tanto si es instancia.protegido como protegido
        puts "Metodo privado"
    end
end
class Hija < Padre
    def usoprivado(padre)
        privado #Metodo privado
        padre.protegido #Metodo protegido | protected admite receptor de mensaje
        self.privado #Metodo privado Correcto a partir Ruby 2.7
        padre.privado #Error: No se puede acceder a un metodo privado de otra clase
    end
end
h = Hija.new
p = Padre.new
h.privado #Error: No se puede acceder a un metodo privado desde fuera de la clase
h.protegido #Error: No se puede acceder a un metodo protegido de otra clase
```

**COPIA DE OBJETOS**

Cuidado con las referencias y lo que se copia y devuelve.
Copia de identidad dos referencias apuntan al mismo objeto, sin duplicar el objeto
Copia de estado superficial crea un nuevo objeto pero copia solo las referencias a los objetos internos, no los objetos en sí.
Copia de estado profunda duplica completamente el objeto y todos los objetos internos, creando una réplica independiente del original.
Copia defensiva: Devolver una copia de estado en lugar de identidad. Dependerá el nivel de profundidad de la complejidad de los objetos.
clone() Crea y devuelve una copia(en profundidad) del objeto. (Se intenta)
Interfaz Cloneable No define ningun metodo (extraño)
Constructor de copia...