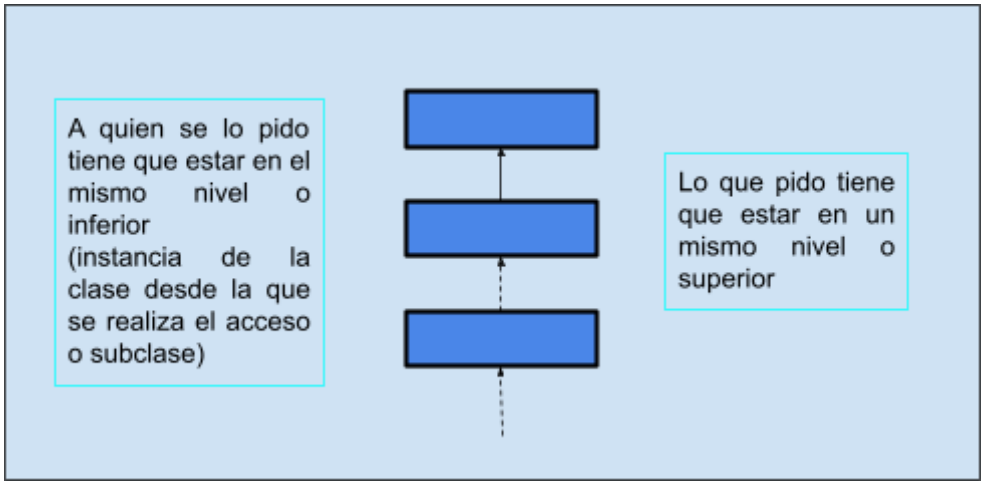
RESUMEN PDOO SEGUNDO PARCIAL

Visibilidad:

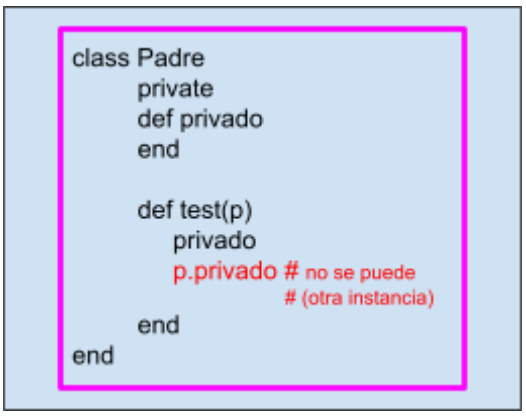
- Especificadores de acceso**(private, public, protected, paquete)** para restringir acceso a atributos y métodos ocultando detalles. Usar el nivel más restrictivo ante la duda.
- **Java** (cada elemento tiene que tener un especificador)**:**
    - **private:** accesible desde ćodigo de la clase (desde amb. instancia acceder a privados de la misma clase; puedo acceder a los atr. de otra instancia de la clase desde la implícita sin necesidad de consultores (ej. instancia parámetro de un método (inst. o clase).
    - **paquete:** no se pone nada para indicarlo. Públicos dentro del paquete y privados fuera (en ruby no existe la visibilidad de paquete).
    - **protected:** públicos dentro del mismo paquete (visibilidad de paquete). Accesibles desde subclases de otros paquetes; puedo acceder a un protected de una superclase independientemente del paquete. Se usaŕa protected según las subclases estén en el mismo paquete o no. Para poder acceder a elementos protegidos de una instancia distinta esa instancia tiene que ser de la misma clase que la propietaria del código desde el que se hace el acceso o de una subclase de la misma (la instancia tiene que ser-un yo)(sí estuviera en el mismo paquete daría igual). Elemento accedido declarado en la propietaria o en una superclase (elemento visible por mí).

  Para acceder a atributos de instancia en ámbito de clase necesito una instancia a la que pedírselo. Tampoco puedo hacer this.elemento_al_que_acceder

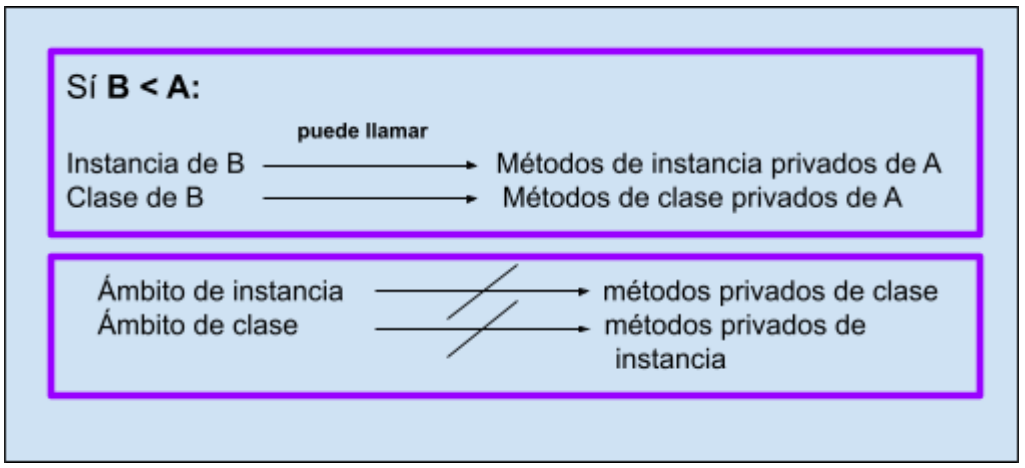


- **Ruby:**
  - Atributos todos privados (no se puede cambiar), métodos públicos(modificar con especificadores de acceso).
  - Initialize siempre privado (no cambiar).
  - Cuando se usa un especificador de acceso,afecta a todos los elementos declarados a continuación.
  - **private:**
    - Un método privado no puede ser accedido por un receptor explícito (sí hago **private** el método **met**, y fuera de la clase declaro una instancia de la clase en la que está el método met; **objeto = new Class objeto.met** **// error porque se está accediendo a met por un receptor explícito (objeto))**

    - Solo se puede utilizar un método privado de la propia instancia. P.ej:



    - Sí **B < A** desde un ámbito de instancia de B se puede llamar a métodos de instancia privados de A. Desde un ámbito de clase de B se puede llamar a método de clase privados de A.
    - No se puede acceder a métodos privados de clase desde amb.instancia y no se puede acceder a métodos privados de instancia desde amb.clase.



Para hacer un método de clase privado: **private_class_method :metodo**
Para hace un método de instancia privado: **private :metodo**

- **protected:**
  - Sí pueden ser invocados con un receptor de mensaje explícito.
  - La clase del código que invoca debe ser la misma o una subclase de la clase donde se declaró el método al que intentamos acceder.
  - ∄ métodos protegidos de clase.

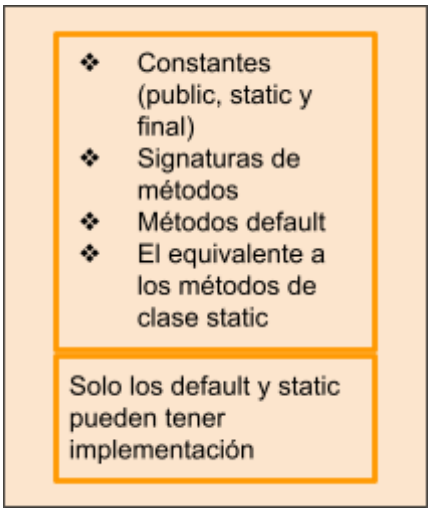
- **Se suele olvidar:** Tanto las clases como las instancias son objetos. En la práctica, la clase GameUniverse y las instancias de GameUniverse no son instancias de la misma clase. Estas últimas son objetos de GameUniverse pero la propia clase GameUniverse es un objeto de lo que denominamos "metaclase".
- Los **atributos de clase** (@@atributo_clase) sí pueden ser accedidos directamente desde el ámbito de instancia;visibles tanto desde el ámbito de clase como de instancia.
- Desde el ámbito de instancia podemos acceder a atributos de clase pero no a atributos de instancia de la clase. Por su parte, desde un método de clase no podemos acceder directamente a elementos /métodos de instancia.
- Sí **B < A** entonces, **B** hereda todos los atributos de clase, pero **no** los de instancia de la clase.

Clases abstractas:

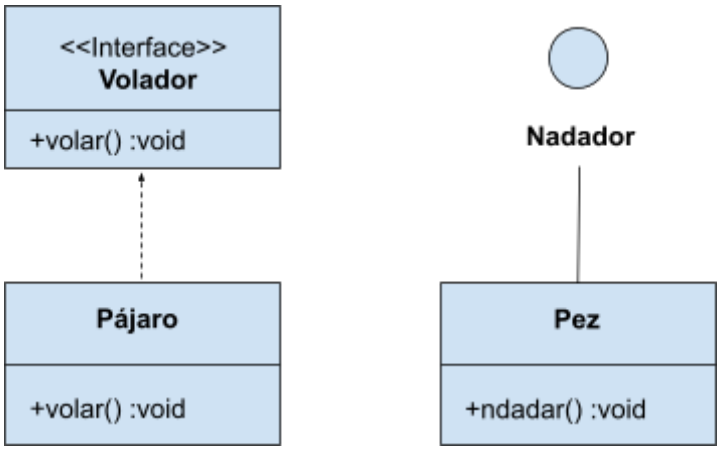
- Modelan entidades que representan de forma genérica a otras funcionalidades que sí concretan el funcionamiento desconocido.
- Ruby no soporta las clases abstractas.
- En Java:
    - Se declaran como: <**especificador_de_acceso > abstract class <nombre_clase>**
    - No proporcionan en general implementación a sus métodos (solo se ponen las cabeceras (aunque luego habrá métodos que sean generales a todas las instancias que representa la clase abstracta que sí se deberá implementar). Un método abstracto se declara de la siguiente manera:
      <visibilidad> abstract <tipo> <nombre_método>
- No se pueden instanciar pero sí declarar una variable de tipo la clase abstracta.
- Obligan a sus subclases a implementar una serie de métodos (los que precisamente declaran y no implementan), aunque sí no implementan alguno de los métodos abstractos serán también clases abstractas.
- Proporcionan métodos y atributos comunes a esas subclases, sin perjuicio de que las subclases añadan atributos y/o métodos o redefinan métodos heredados.
- Definen un tipo de dato común a todas sus subclases.
- Como en Ruby no se soportan las clases abstractas, en el caso de necesitar una clase que aglutine atributos y/o métodos comunes de sus clases derivadas (esa clase no se podrá instanciar, pero sus derivadas sí), se hace el método **new privado** en la clase **padre** y se vuelve a hacer **público** el método **new** en las **clases derivadas**.
- En cuanto a su representación **UML:** nombre de la clase en cursiva y/o también para evitar despistes, se pone **<<abstract>>** encima del nombre de la clase. Los métodos abstractos también se representan en cursiva.
- Hay que poner **@Override** encima del método abstracto que se va a redefinir en la subclase de una clase abstracta (sí no se pone, dicha subclase también será abstracta).

Interfaces:

- No soportadas por ruby
- Define un protocolo de comportamiento y permite reutilizar la especificación de dicho comportamiento. Será una clase la que implemente dicho comportamiento (lo que se denomina **realización** de la interfaz). Define en sí un contrato que cumplen las clases que realizan dicha interfaz.
- Define un tipo: se pueden declarar variables de ese tipo y dichas variables podrán referenciar instancias de clases que realicen dicha interfaz.
- Una clase puede realizar varias interfaces.
- Una interfaz puede heredar de una o más interfaces.
- ¿Qué puede tener solamente una interfaz?



- Las interfaces no pueden ser instanciadas, sólo realizadas por clases o extendidas por otras clases.
- Se pueden redefinir los 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, forzando así a sus subclases no abstractas a implementarlos.
- Representación UML:

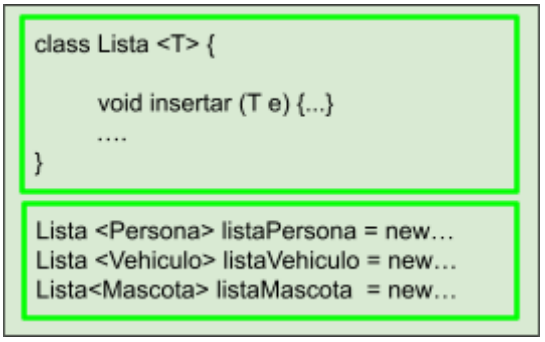


- Se declaran así:
  <visibilidad> interface <nmbre_interfaz>
- Sí una clase va a llevar a cabo la realización de una interfaz, se pone: **<visibilidad><nombre_clase>implements <nombre_interfaz> , …** (puede realizar más de una interfaz).

Ante la duda: usar una interfaz frente a una clase abstracta; esta última sólo cuando queremos atributos de instancias comunes.

Clases parametrizables:

- Son clases definidas en función de un tipo de dato (clase) en las que se generalizan un conjunto de operaciones válidas para diferentes tipos de datos.
- Pseudocódigo de una clase parametrizable que modela una lista de objetos cualquiera:



- En su representación UML, se pone:

  ——
  | T | en la esquina superior derecha de la cajita de la clase
  —-

- **Java:**
  - Se implementan mediante tipos genéricos.
  - Permite pasar tipos como parámetros a clases e interfaces.
  - Se puede forzar que el tipo suministrado a una clase parametrizable tenga que ser subclase de otro (T extends ClaseBase) o que tenga que realizar una interfaz (T **extends** Interfaz). **Nótese que cuando una clase realiza una interfaz se pone implements, pero AQUÍ se usa extends.**
  - La implementación de un método de una clase parametrizable puede requerir que T disponga de un determinado método. En ese caso, el método requerido formará parte de una interfaz. Al declarar la clase parametrizable se indicará que el tipo que sustituya al parámetro debe realizar dicha interfaz.
  - Las interfaces también pueden hacerse paramétricas.

Polimorfismo:

- Capacidad de un identificador de referenciar objetos de diferentes tipos. En lenguajes sin declaración de variable(ruby por ejemplo: cualquier variable puede referenciar cualquier tipo de objeto) se da de forma natural y sin limitaciones. En lenguajes con declaración de variables(java p.ej), existen limitaciones al respecto.
- **Principio de sustitución de Liskov:**
  - Sí B es un subtipo de A, se pueden utilizar instancias de B donde se esperan instancias de A:
    - Ejemplo:

Sí Estudiante es una subclase de Persona se pueden utilizar instancias de Estudiante donde se puedan utilizar instancias de Persona (**relación es-un).**

- **Tipo estático:** tipo del que se declara la variable.
- **Tipo dinámico:** clase a la que pertenece el objeto referenciado en un momento determinado por una variable.
- **Ligadura estática:** El enlace del código a ejecutar asociado a una llamada a un método se hace en tiempo de compilación. La decisión de dónde ir a buscar el método que se está llamando se toma en tiempo de ejecución. En tiempo de compilación solo se ve sí existe alguna posibilidad para lo que se llama o no.
- **Ligadura dinámica:** El tipo dinámico determina el código que se ejecutará asociado a la llamada a un método (cobra sentido el polimorfismo).
- **El tipo estático limita:** lo que puede referenciar una variable (instancias de la clase del tipo estático o de sus subclases), los métodos que pueden ser invocados (los disponibles en las instancias de la clase del tipo estático).
- **Casts:** Se le indica al compilador que considere, temporalmente que el tipo de una variable es otro. No transforma el objeto referenciado ni su comportamiento.
    - **Downcasting:** Se le indica al compilador que considere, temporalmente, que el tipo de la variable es una subclase del tipo con que se declaró. Permite invocar métodos que sí existen en el tipo del cast pero que no están en el tipo estático de la variable.
    - **Upcasting:** Se indica al compilador que considere temporalmente que el tipo de la variable es superclase del tipo con que se declaró.ç
- Deben evitarse, a toda costa, las comprobaciones explícitas de tipos.
- 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. Sí no se encuentra, se busca en la clase padre, así sucesivamente hasta encontrarlo o hasta que no existan ascendientes. Esto sigue siendo cierto para métodos invocados desde otros métodos.
- Tener en cuenta el polimorfismo cuando varias clases tienen el mismo método pero con distintas implementaciones, cuando existe relación de herencia entre dichas clases, o cuando no se sabe a priori a qué objeto concreto se le va a enviar el mensaje asociado a dicho objeto.

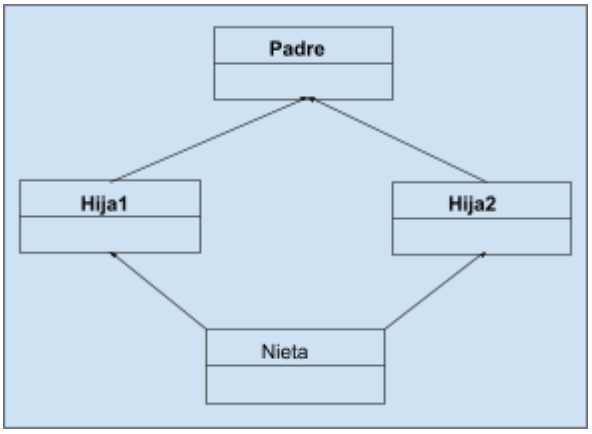
Herencia en el Ámbito de Clase:

- **Java:**
    - No permite la redefinición de métodos de clase al mismo nivel que de instancia. 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.
    - No se permite usar **super.metodoClase** a nivel de clase.

    - En ámbito de clase no funciona el polimorfismo como lo hemos visto (a nivel de instancia). Podemos decir, entre comillas, que “no hay herencia a nivel de clase”. Aquí es el tipo estático quien determina dónde buscamos el método al que llamamos (!= polimorfismo, donde era el tipo dinámico quien lo determinaba).
- **Ruby:**
    - Las clases son first class citizens y en el ámbito de clase todo funciona como es de esperar.
    - Hemos de evitar los atributos de clase salvo que tengamos claro que no habŕa herencia.
    - Los atributos de instancia de la clase SIEMPRE se buscan en la propia clase (nunca ancestros) porque no puedo acceder a los de otra clase (sí estoy buscando un atributo instancia de la clase, lo busca en la clase actual, y sí no lo encuentra, no lo coge de su Padre).

Herencia múltiple

- Se produce cuando una clase es descendiente de más de una superclase. Permite representar problemas donde un objeto tiene propiedades según criterios distintos. Java y Ruby no tienen herencia múltiple.
- Problemas comunes:
    - Colisión de nombres de métodos y/o atributos:
        - Cuando hay cabeceras iguales pero dos implementaciones diferentes → solución sencilla: hay ambigüedad en este método, solucionada en la implementación de la clase donde se genera el problema.
- Para que tenga sentido debe haber una relación es-un de la clase descendiente con todos sus ascendientes.
- Problema del diamante: provoca duplicidad en los elementos heredados



Nota:

Nieta: | Padre | Hija1 | Padre | Hija2 | Nieta |

No hay problemas con los métodos (sólo están repetidos). Pero sí hay problemas con los atributos, que pueden ser diferentes en el P de H1 que en el

de H2.

Para resolver el problema del diamante en el cual la clase Nieta tenía dos versiones de sus atributos, se usa **virtual** en la cabecera de las clases padre de Nieta:

class B: virtual public A{...}

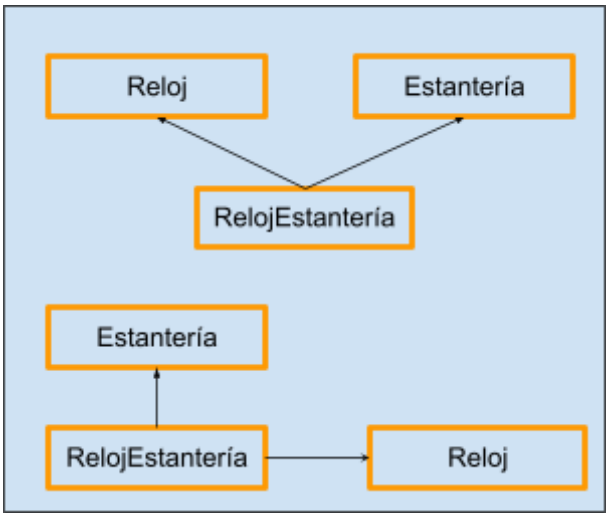
class C: virtual public A{...}

Y en el constructor de la clase Nieta, se ha de llamar al constructor de la clase abuelo o bisabuelo: **A**, lo que resulta ser una dependencia muy fea, pero que soluciona el problema de los atributos.

Quería entonces la representación de antes de la siguiente forma:

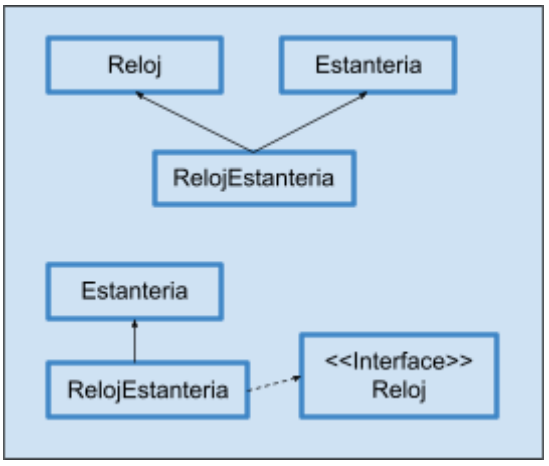
Padre | Hija1| Hija2 |Nieta|

- **Alternativas:**
  - **Composición:** sustituir una o varias relaciones de herencia por composición.



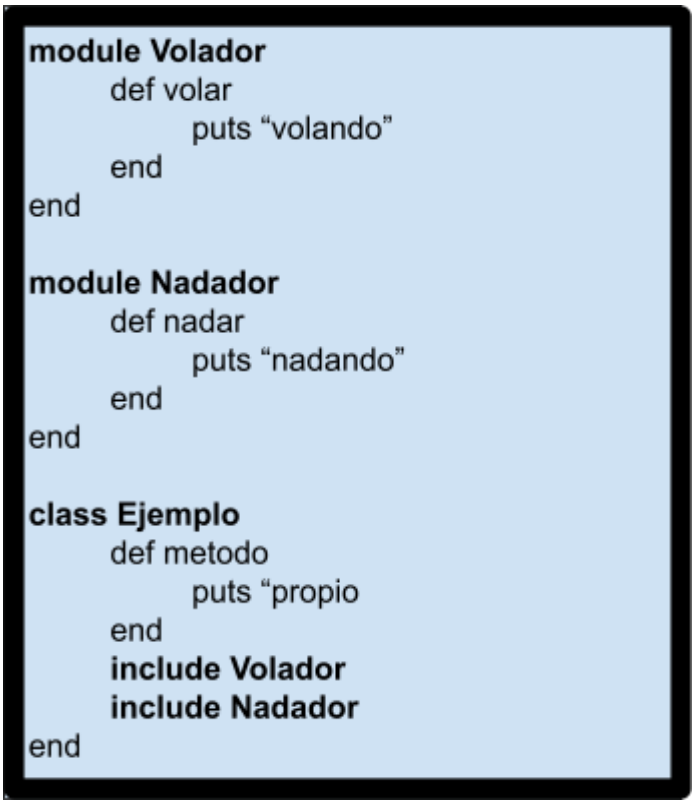
La clase RelojEstanteria hereda de Estantería y establece una relación de asociación con la clase Reloj; hereda todos los métodos de Estantería y declara atributos de Reloj y define sus métodos

  - **Interfaces Java**: se pueden realizar varias interfaces Java y heredar de una superclase.



La clase RelojEstanteria hereda de Estanteria y realiza la interfaz Reloj (se tienen que implementar todos los métodos de Reloj (a nivel de ahorro de código no conseguimos nada en comparación con el código anterior).

- **Mixins de Ruby:** permiten incluir código proveniente de varios módulos como parte de una clase



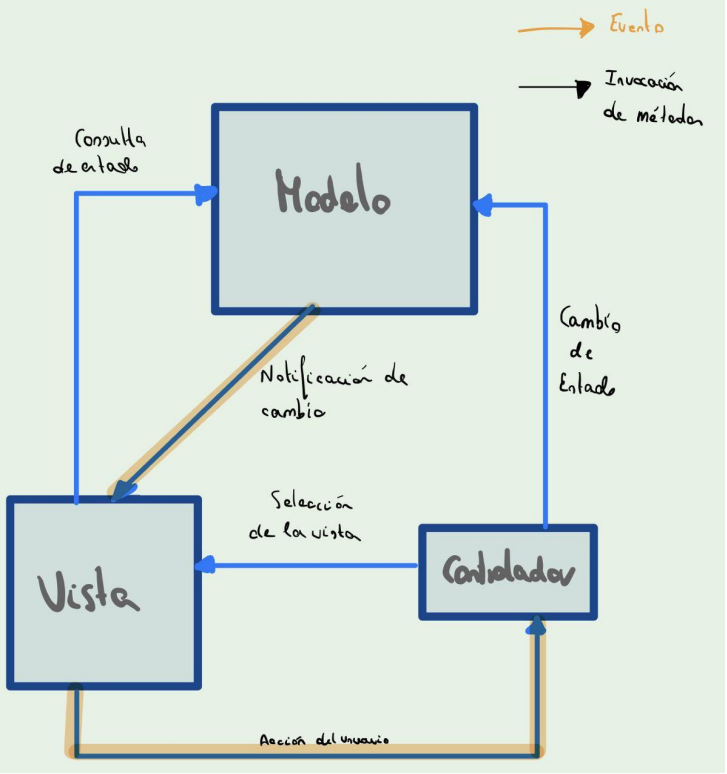
```
module Volador
    def volar
        puts "volando"
    end
end

module Nadador
    def nadar
        puts "nadando"
    end
end

class Ejemplo
    def metodo
        puts "propio
    end
    include Volador
    include Nadador
end
```

Se incluye el código de los módulos Nadador y Volador dentro de la clase Ejemplo. Modificar alguno de los dos módulos afecta directamente al comportamiento de la clase Ejemplo.

Modelos Vista Controlador:

- **Patrón de diseño:**
  - Un patrón de diseño describe un problema que ocurre numerables veces en nuestro entorno, describiendo además el núcleo de una solución a ese problema de forma que sea reutilizable, permite aprovechar soluciones previas probadas y validadas a problemas conocidos.
- **Modelos Vista Controlador(MVC):**
  - **Modelo:** Clases que representan la lógica del problema
  - **Vista:** Representación visual de los datos del modelo para mostrarlos al usuario (la interacción del usuario se produce con elementos de la vista.
  - **Controlador:** Actúa de intermediario entre la vista y el modelo.

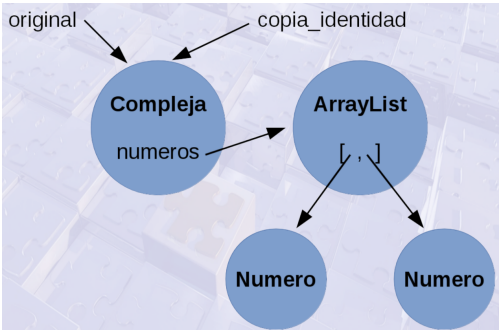
- Para un mismo modelo se pueden tener diferentes vistas.
- La información fluye en ambas direcciones.
- **Responsabilidad del controlador:** Ante una orden del usuario, siempre a través de la vista que implique cambios en el modelo, es el controlador quien la ejecuta actuando sobre el modelo. Cuando se producen cambios en el modelo, estos cambios deben verse reflejados en las vistas correspondientes (el controlador puede actuar de intermediario en este proceso).La mayoría de las veces, una acción del usuario implica un recorrido de ida y vuelta → ←
- Hay dos esquemas alternativos del modelo vista controlador:
  - Vista ⇔ Controlador ⇔ Modelo (sí comunicación entre modelo y vista)
  - Esquema alternativo:



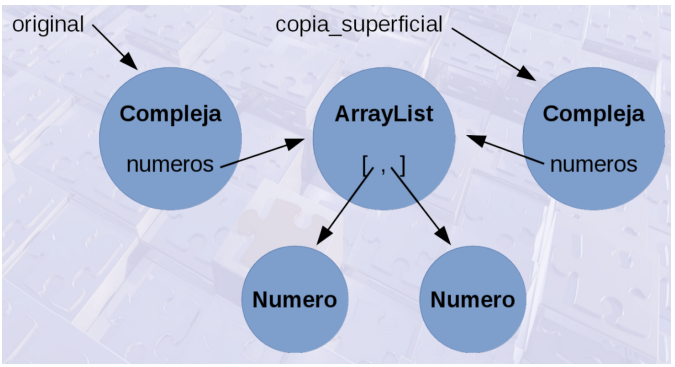
Copia de Objetos:

- Una operación muy habitual es la **Asignación**. No es una operación trivial, de hecho existen distintos casos: copia de identidad, copia de esto. Todo esto puede quedar “oculto” bajo el operador de asignación.
- **Profundidad de la copia:** hasta qué nivel se van a realizar copias de estado en vez de identidad.

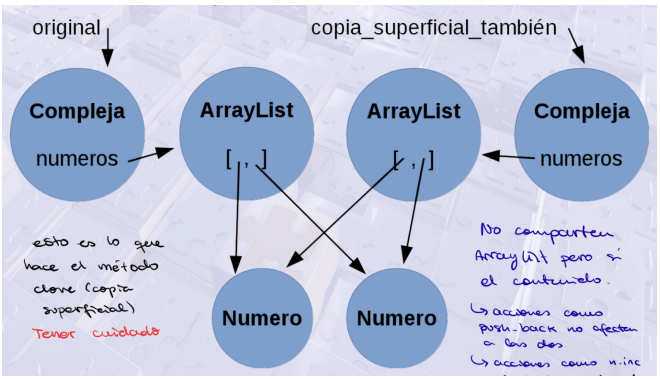
- **Inmutabilidad de los objetos:** un objeto es inmutable sí no dispone de métodos que modifiquen su estado. Hay que tener cuidado con los objetos que referencian a otros sí estos no son inmutables.
  Tenemos el mismo problema de siempre
  a —-> O <—- b (cambiar a ⇔cambiar b)
- **Copia defensiva:** alude a devolver una copia de estado en vez de devolver una copia de identidad.
  - **Objetivo:** evitar que el estado de un objeto se modifique sin usar los métodos que la clase designa para ello.
  - **Requisito:** realizar copias profundas y no solo copias superficiales.
  - **Cuándo:** los consultores deben usar este recurso con los objetos mutables que devuelvan.
- **Tipos de copia:**
  - **Copia de identidad:**



  - **Copia superficial de primer nivel:**

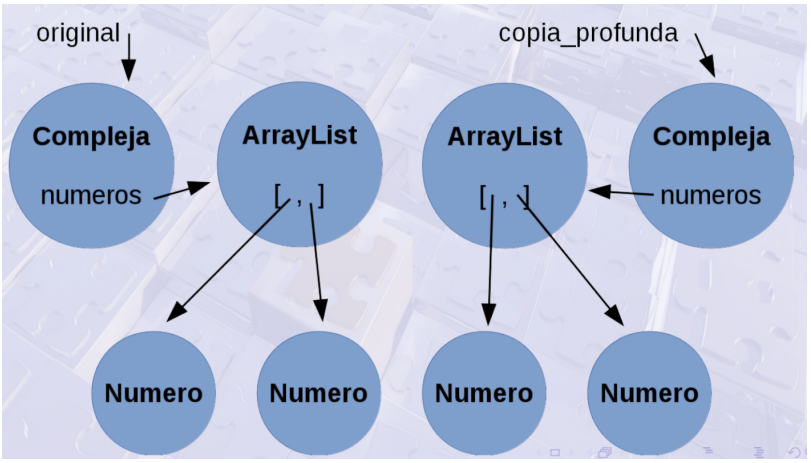


  - **Copia superficial de segundo nivel:**



El método **clone** (devuelve una copia de estado) hace esto mismo (copia superficial).

- **Copia profunda:** Todos los objetos **mutables** se han **duplicado.**



Hay que llegar al nivel requerido en cada caso. Para copia profunda, redefino (con @Override) el clone en todas las clases que necesite (la cabecera de dichas clases ha de ser: **class <Nombre> implements Cloneable)** y tengo mi problema resuelto. La cabecera del método clone redefinido es:

```
@Override
public <Nombre_Clase> clone () throws CloneNotSupportedException{
        return (Nombre_Clase) super.clone();  // basta con super solo
}
```

Por defecto, **clone()** devuelve Object por eso se necesita hacer el cast.

- **Clone y la interfaz Cloneable:**
  - Al utilizar este mecanismo se redefine **protected Object clone () throws CloneNotSupportedException** de la forma que se ha indicado anteriormente; se aumenta visibilidad (protected por public) ,y devolvemos algo más específico. El método creado debe crear una copia base (con super.clone) y crear copias de los atributos no inmutables (esto se puede conseguir usando el método clone sobre esos atributos.
- **Reflexiones sobre clone y la interfaz Cloneable:**
  - **Documentación oficial:**
    Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expressions are true: → (en clases distintas la profundidad de copia puede ser distinta)

    1 x. clone () != x
    2 x . clone ( ) . getClass ( ) == x . getClass ( )
    3 x. clone ( ) . equals (x)

4 / / These are not absolute requirements !!!!

By convention, the object returned by this method should (pero no tiene por qué serlo) be independent of this object (which is being cloned)

Sí vamos a usar el clone de una clase que no conocemos hay que ver hasta qué nivel llega (cómo funciona).

- La interfaz Cloneable no define ningún método. Esto rompe el significado habitual de Java (lo cual ha sido bastante criticado por autores):
  class Raro implements Copiable {} no produce errores de ningún tipo (firmar un contrato vacío (no en blanco, vacío)).
- En la clase Object se realiza la comprobación de sí la clase que originó la llamada a clone implementa la interfaz. Sí no es así → excepción.
- Los atributos **final no** son **compatibles** con el uso de **clone.**
- En el método clone no intervienen los constructores en todo el proceso.
- Clone es un método de instancia.

- **Constructor copia:**
  - Es posible copiar objetos creando un constructor que acepte como parámetro objetos de la misma clase, el cual se encargaría de hacer la copia profunda.

- **Ruby y clone:**
  - En Ruby el método clone de la clase Object también realiza copia superficial. Sí se desea realizar una copia profunda, la debe hacer el programador. En definitiva, clone no es distinto en Ruby.

- **Copia por serialización:**
  - En Ruby, se puede recurrir a la serialización, deserialización para crear un copia profunda:
    b = Marshal.load (Marshal.dump(a)) // clone a toda profundidad
    La clase Marshal copia los objetos .
  - En este proceso, el objeto se convierte a una secuencia de bits y después se construye a partir de esta secuencia. Esta técnica también es aplicable a Java y en ambos casos es poco eficiente. Por otra parte, se advierte que el uso del método load en Ruby, puede llevar a la ejecución de código remoto.
  - Luego, no es recomendable por razones de seguridad.

- **Copia de objetos:**
  - ¿En toda las asignaciones de parámetros mutables y en todas las devoluciones de atributos mutables debemos realizar copias defensivas? NO tiene porqué, pues depende de dónde vengan los parámetros a asignar, a quién se le dé el atributo, del software que se esté desarrollando, etc. Hay que estudiar cada caso y decidir.

Reflexión:

- Capacidad de un programa para manipularse a sí mismo y comprender sus propias estructuras en tiempo de ejecución.
- **Mecanismos:**
    - **Introspección:** Habilidad del programa para observar y razonar sobre su mismo estado (objetos y clases) en tiempo de ejecución.
    - **Modificación:** Habilidad del programa para cambiar su estado (objetos y clases) durante la ejecución. Normalmente sólo soportado por lenguajes interpretados.
    - **Java:**
        - Debido a la estructura de metaclases desarrollada por Java, el nivel de reflexión que se permite es de introspección.
        - Toda la funcionalidad para ello está definida en la clase Class de Java.
    - **Ruby:**
        - Debido a la estructura de metaclases desarrollada por Ruby, el nivel de reflexión que se permite es de introspección y de modificación.
        - En ejecución se puede consultar y modificar una clase y consultar y consultar y modificar la estructura y funcionalidad de un objeto haciéndolo distinto de los demás de la misma clase.

Nota: consultar los ejemplos de las diapositivas en todo aquello que no se comprenda adecuadamente durante la lectura del resumen. Ilustran bastante bien los conceptos y favorecen el aprendizaje.