Paquetes Especificadores de Acceso

Paquetes y Espacios de Nombres

- Un **paquete** en JAVA **es una colección de componentes de software (clases, interfaces, tipos enumerativos, anotaciones) con nombre**. Los **paquetes** son útiles para **agrupar** componentes de software relacionados y definir un **espacio de nombres común a las entidades** contenidas en él.
- Las clases e interfaces esenciales (*core*) de la Plataforma JAVA están ubicadas en un paquete cuyo nombre comienza con **java** y luego una serie de subnombres, resultando en un nombre jerárquico.
  - Las clases e interfaces fundamentales de JAVA están ubicadas en el paquete **java.lang**.
  - Las clases e interfaces utilitarias están en el paquete **java.util**.
  - Las clases e interfaces usadas para realizar I/O están en el paquete **java.io**.
  - Las clases e interfaces usadas para establecer conexiones con hosts remotos, que usan protocolos de comunicación de redes, etc. están en el paquete **java.net**.
  - Las clases e interfaces usadas para realizar funciones matemáticas y trigonométricas están en el paquete **java.math**.
  - Las clases e interfaces usadas para construir interfaces gráficas de usuario están en **java.awt**.

Paquetes y Espacios de Nombres

- Los paquetes a su vez pueden contener subpaquetes, como por ej. el paquete **java.awt** contiene los subpaquetes **java.awt.event** y **java.awt.image**; el paquete **java.lang** contiene los subpaquetes **java.lang.reflect** y **java.util.regex**.
- Las extensiones de la plataforma JAVA que han sido estandarizadas por el JCP, tienen nombres de paquetes que comienzan con **javax**, por ej. **javax.swing** y **javax.sql.** Estas extensiones han sido incorporadas a la plataforma JAVA.
- La plataforma JAVA a su vez incluye varios paquetes estándares, como por ej. **org.w3c.dom** y **org.omg** que implementan estándares definidos por la W3C y la OMG.
- Todas las clases, interfaces, tipos enumerativos y anotaciones tienen un **nombre simple** o no calificado y un **nombre completo** o calificado. El simple es el que usamos para definir la clase y el completo o calificado es el que incluye como prefijo el nombre del paquete al que pertenece la clase. Por ej. la clase **String** es parte del paquete **java.lang,** su nombre simple es **String** y su nombre completo es **java.lang.String**.

Nombres Únicos de Paquetes

- Una de los propósitos más importantes de los paquetes es el de dividir el espacio de nombres global de JAVA y evitar colisiones de nombres entre clases.

Por ej. el nombre del paquete permite diferenciar la interface **java.util.List** de la clase **java.awt.List**.
Para que este mecanismo funcione correctamente, los nombres de los paquetes deben ser distintos.

- Un esquema de nombres de paquetes que garantiza nombres de paquetes globalmente únicos consiste en usar el nombre invertido del dominio de Internet como prefijo de todos los nombres de paquetes.

Por ej. si consideramos el nombre del dominio de Internet del LINTI, **linti.unlp.edu.ar**, los nombres de paquetes JAVA desarrollados en el LINTI comenzarán con **ar.edu.unlp.linti**. Luego, con algún criterio se debe decidir cómo particionar el nombre después de **ar.edu.unlp.linti**.
Tenemos que tener en cuenta que como el nombre del dominio de Internet es único, ningún otro desarrollador u organización que respete esta regla definirá un paquete con el mismo nombre.

ar.edu.unlp.linti.graficos
ar.gov.gba.ec.rrhh.sueldos
ar.com.afip.estadisticas.reportes

Nombres Únicos de Paquetes

Pautas para elegir nombres únicos de paquetes:

- Si somos desarrolladores de clases que serán usadas por otros programadores y que las combinarán con múltiples clases desconocidas por nosotros, es importante que los nombres de los paquetes sean globalmente únicos. Por ej. desarrolladores de compañías de software, de comunidades de software libre, etc.

- Si estamos desarrollando una aplicación JAVA, cuyas clases no están concebidas para ser usadas por otros programadores fuera de nuestro equipo de trabajo, podemos elegir un esquema de nombres de paquetes que se ajuste a nuestra conveniencia. En este caso conocemos el nombre completo del conjunto de clases que nuestra aplicación necesita para el *deployment* y no tendremos imprevistos por conflictos de nombres.

La palabra clave *package*

En JAVA las clases e interfaces típicamente se agrupan en paquetes usando como primer *token* del archivo fuente la palabra clave **package** seguido por el nombre del paquete.

```
package ar.com.laplataautos;
public class Vehiculo {
private String marca;
private String nroMotor;
public String getMarca(){
    return marca;
 }
public String getNroMotor(){
    return nroMotor;
}
}
```

Si se omite la palabra clave **package** en la definición de una clase, la misma se ubicará en el paquete predeterminado, conocido como **default package**. Este mecanismo resulta útil sólo para realizar pequeñas pruebas o testeos de algoritmos o lógica, sin embargo no es una buena práctica de programación.

Importar Tipos de Datos

Si para escribir una nueva clase usamos clases o interfaces existentes, el mecanismo predeterminado para incluir esos nombres consiste en usar el **nombre completo** de la clase o interface.

Por ej. si estamos escribiendo código que manipula un archivo vamos a necesitar usar la clase **File** que pertenece al paquete **java.io**, entonces debemos escribir **java.io.File** cada vez que usamos la clase.

¿Cuándo podemos usar el **nombre simple** de un tipo de datos?

Importación automática

- Si usamos clases e interfaces del paquete **java.lang**.
- Si usamos clases e interfaces que están definidas en el mismo paquete de la clase o interface que estamos escribiendo.
- Si incluimos tipos al espacio de nombres con la sentencia **import**.

Es posible explícitamente importar tipos de datos desde otros paquetes al espacio de nombres actual usando la sentencia **import**.

El **import** tiene 2 formas:

- Importación de un tipo:
- Importación de tipos por demanda:

(los tipos son importados a medida que se necesitan)

```
import ar.edu.unlp.linti.graficos.Rectangulo;
import java.io.File;
import java.io.PrintWriter;
```

```
import ar.edu.unlp.linti.graficos.*;
import java.io,*;
```

Es posible usar el nombre simple de las clases por ej. Rectangulo, File, PrintWriter

La sintaxis de importación de tipos por demanda no se aplica a subpaquetes.

Colisión de Nombres

La sentencia **import** puede generar conflictos.

Consideremos el paquete **java.util** y el paquete **java.awt**
Ambos paquetes contienen el tipo **List**: **java.util.List** es una interface muy importante y comúnmente usada; **java.awt.List** es una clase no tan frecuentemente usada.

```
package pruebas;
import java.util.List;
import java.awt.List;
public class ConflictoDeNombres {
    ......
}
```

Es ilegal importar en el mismo archivo JAVA ambos tipos!!!
Error de compilación!!!

```
package pruebas;
import java.util.*;
import java.awt.*;
public class ConflictoDeNombres {
    .....
}
```

Es legal si usamos importación por demanda pues tenemos la posibilidad de importar otros tipos, no sólo List.
Si necesitamos usar el tipo List tenemos que usar el nombre completo, el nombre simple da error de compilación!!!

```
package pruebas;
import java.awt.*;
import java.util.*;
public class ConflictoDeNombres {
    List l;
    ......
}
```

¿A qué List hace referencia?

Colisión de Nombres

¿Cómo lo resolvemos?

La interface **java.util.List** es usada más comúnmente que la clase **java.awt.List**, en este caso es útil combinar importación de un tipo e importación por demanda y así quitamos la ambigüedad cuando hacemos referencia al tipo **List**.

```
package pruebas;
import java.util.List;
public class ConflictoDeNombres {
  List l;
  java.awt.List  l2;
}
```

```
package pruebas;
import java.util.*;
public class ConflictoDeNombres {
  List<String> l=new ArrayList<String>();
  java.awt.List  l2=new java.awt.List(4);
}
```

Hacemos referencia a java.util.List

En estos casos a la clase **List** de **java.awt** la tenemos que usar con el nombre completo, **java.awt.List** y a la interface de **java.util** con el nombre corto.

Importar Miembros Estáticos

- A partir de JAVA 5.0 es posible importar miembros estáticos de clases e interfaces, usando la palabra clave **import static**.
- El **import static** tiene 2 formas:
  - Importación de un miembro estático
  - Importación de miembros estáticos por demanda

Consideremos la siguiente situación: necesitamos imprimir texto en pantalla usando la salida estándar, **System.out**. Usamos la **importación de un miembro estático:**

```
package pruebas;
import static java.lang.System.out;
public class ImportOUT {
   public static void main(String[] args) {

    out.print("hola");
   }
}
```

Nos evitamos escribir System.out.print( )

Podemos escribir expresiones concisas sin tener que usar Math como prefijo de cada método estático de la clase Math.

Consideremos la siguiente situación: hacemos uso exhaustivo de funciones trigonométricas y otras operaciones de la clase **Math**.Usamos **importación de miembros estáticos por demanda:**

```
package pruebas;
import static java.lang.Math.*;
import static java.lang.System.out;
public class ImportOUTBis {
  public static void main(String[] args) {
        out.println( sqrt(abs(sin(90))) );
        out.println("Valor de PI: "+PI);
  }
}
```

Importar Miembros Estáticos

La importación estática **"importa nombres"**, NO un miembro específico con dicho nombre. JAVA soporta sobrecarga de métodos y también permite que una clase defina atributos con el mismo nombre que un método -> al importar un miembro estático, podríamos importar más de un miembro (métodos y atributos).

Consideremos el siguiente ejemplo:

```
package pruebas;
import static java.util.Arrays.sort;
public class SobrecargaImportEstatico {
    public static void main(String args[]) {
    String varones[]={"Juan", "Pedro", "Luis", "Ernesto"};
    sort(varones);
    }
}
```

Importa el nombre sort, no uno de los 18 métodos sort() definidos en la clase Arrays

El compilador analizando el tipo, cantidad y orden de los argumentos determina cuál de los métodos **sort()** queremos usar.

Importar Miembros Estáticos

```
package pruebas;
import java.util.ArrayList;
import static java.util.Arrays.sort;
import static java.util.Collections.sort;
public class SobrecargaImportEstatico {
    public static void main(String args[]) {
    Integer numeros[]={30,2,44,18,3,23};
    String varones[]={"Juan", "Pedro", "Luis", "Ernesto"};
    ArrayList<String> mujeres=new ArrayList<String>();
    mujeres.add("Elena");
    mujeres.add("Pilar");
    mujeres.add("Juana");
    sort(varones);
    sort(numeros);
    sort(mujeres);
    }
}
```

Es legal importar métodos estáticos con el mismo nombre, pertenecientes a clases distintas, siempre que los métodos tengan diferentes firmas.

Este código no tiene errores de compilación porque los métodos **sort()** definidos en la clase **Collections** tienen firma diferente a todos los definidos en la clase **Arrays**. Cuando usamos el nombre **sort** el compilador determina cuál de los 20 posibles métodos importados deseamos invocar, analizando el tipo, orden y cantidad de argumentos.

Ubicación de los Paquetes

- Un paquete está formado por múltiples archivos **.class**.
- Java aprovecha la estructura jerárquica de directorios del SO y ubica todos los **.class** de un mismo paquete en un mismo directorio. De esta manera se resuelven:
  - El nombre único del paquete (no existen 2 *paths* con el mismo nombre)
  - La búsqueda de los **.class** y **.java** (de otra forma estarían diseminados en todo el filesystem)
- Los nombres de paquetes se resuelven en directorios del SO: en el nombre del paquete se codifica el *path* de la ubicación de los **.class**.

```
package ar.edu.unlp.linti.graficos;
import java.awt.*;
public class Rectangulo extends Graphics implementes Draggable {
   . . .
}
```

ar/edu/unlp/linti/graficos/Rectangulo.class

- Cuando el "intérprete" JAVA ejecuta un programa y necesita localizar dinámicamente un archivo **.class** (por ej. cuando se crea un objeto o se accede a una variable *static*) procede de la siguiente manera:
  - Busca en los **directorios estándares** (donde está instalado el JRE y en el actual)
  - Recupera la variable de entorno **CLASSPATH** que contiene la lista de directorios usados como raíces para buscar los archivos **.class**. **CLASSPATH=.;C:\cursoJava\lab;C:\laboratorio12**
  - Toma el nombre del paquete de la sentencia **import** y reemplaza cada "." por una barra "\" 0 "/" (según el SO) para generar un *path* a partir de las entradas del **CLASSPATH**.
- El compilador JAVA procede de manera similar al "intérprete".
- Las versiones actuales del JSE configuran automáticamente la variable de entono **CLASSPATH**.

Archivos JAR

- Es posible agrupar múltiples archivos **.class** en un único archivo comprimido:

Usando archivos JAR (Java ARchive).

- El formato JAR usa el formato ZIP. Los archivos JAR son multiplataforma, son totalmente portables.
- En los archivos JAR pueden incluirse además de archivos **.class** recursos como archivos de imágenes y audio, etc.
- El formato JAR mejora la descarga de aplicaciones de escritorio que residen en un servidor HTTP.
- La distribución estándar del JSE o JDK tiene una herramienta que permite crear archivos JAR desde la línea de comando, es el utilitario **jar**.
- El "intérprete" JAVA se encarga de buscar, descompactar, cargar y ejecutar los **.class** contenidos en el JAR.

Si usamos archivos JAR, en el CLASSPATH se codifica el nombre real del archivo JAR:

CLASSPATH=.;C:\cursoJava\lab;C:\laboratorio12;c:\java12\utiles.jar

Archivos JAR

- El uso de archivos empaquetados (jar) es la opción recomendada para la entrega de aplicaciones o librerías de componentes.
- Una aplicación empaquetada en un JAR es un archivo ejecutable JAVA que es posible lanzar directamente desde el sistema operativo.
- Los archivos JAR además de contener todos los paquetes con sus archivos .class y los recursos de la aplicación, contienen un archivo MANIFEST.MF ubicado en el camino META-INF/MANIFEST.MF, cuyo propósito es indicar cómo se usa el archivo JAR. Las aplicaciones de escritorio a diferencia de las librerías de componentes o utilitarias requieren que el archivo MANIFEST.MF contenga una entrada con el nombre de la clase que actuará como punto de entrada de la aplicación –la que define el método main()-.

```
Manifest-Version: 1.0
Created-By: 1.6.0_12 (Sun Microsystems Inc.)
Main-Class: capitulo4.paquetes.TestOut.class
```

Archivos JAVA
Restricciones

```
package ar.edu.unlp.linti.graficos;
public class Rectangulo extends Graphics implements Draggable {
.......
}
class Helper {
.....
}
```

Rectangulo.java

Compilador

\ar\edu\unlp\linti\graficos\Helper.class **\ar\edu\unlp\linti\graficos\Rectangulo.class**

Helper.class **Rectangulo.class**

- El **archivo fuente JAVA** se llama **unidad de compilación** y tiene extensión **.java**.
- Cada archivo fuente JAVA puede contener a lo sumo una sola clase o interface declarada **public**. Si contiene clases o interfaces adicionales, éstas deben declararse **no-public** o **package.**
- El nombre del archivo fuente JAVA debe coincidir, incluyendo mayúsculas y minúsculas, con el nombre de la clase o interface declarada **public**.
- Si el archivo fuente contiene múltiples clases e interfaces, el compilador crea diferentes archivos **.class**. Cada uno de ellos tiene el nombre de la clase o interface y extensión **.class**
- Es una buena práctica de programación definir una clase o interface por archivo fuente.

Especificadores de Acceso

JAVA dispone de facilidades para proveer **ocultamiento de información**: el control de acceso define la accesibilidad a clases, interfaces y a sus miembros, estableciendo **qué está disponible** y **qué no** para los programadores que utilizan estas entidades.

Los **especificadores de acceso** determinan la accesibilidad a clases, interfaces y sus miembros y proveen **diferentes niveles de ocultamiento**.

El control de acceso lo podemos utilizar, para:

- Clases e interfaces de nivel superior
- Miembros de clases (métodos, atributos) y constructores

Uno de los factores más importantes que distingue un módulo bien diseñado de uno pobremente diseñado es el nivel de ocultamiento de sus datos y de otros detalles de implementación. Un módulo bien diseñado oculta a los restantes módulos del sistema todos los detalles de implementación separando su interface pública de su implementación.

Desacoplamiento **Reusabilidad de Código**

Especificadores de Acceso en Clases

- En JAVA los especificadores de acceso de las declaraciones de clases se usan para determinar qué clases están disponibles para los programadores. Da lo mismo para interfaces y anotaciones.
- Una clase declarada **public** es parte de la API que se exporta y está disponible mediante la cláusula **import**.

```
package gui;
public class Control{
//.....
}
```

```
import gui.Control;
```

¿Qué pasa si en el paquete gui tengo una clase que se usa de soporte para la clase Control y para otras clases públicas del paquete gui?

La definimos de acceso **package** y de esta manera solamente puede usarse en el paquete gui. Es privada del paquete. Es razonable que los miembros de una clase de acceso *package* tengan también acceso **package**.

```
package gui;
class Soporte{
//.....
}
```

La clase Soporte es privada del paquete gui: sólo puede usarse en el paquete gui

Especificadores de Acceso en Miembros

- Son **reglas de control de acceso** que restringen el uso de los miembros (atributos y métodos) de una clase. Permiten que el programador de la clase determine **qué está disponible** para el usuario programador y **qué no**.
- Los **especificadores de control de acceso** son:

- El **control de acceso** permite **ocultar la implementación**. Separa la **interface** de la **implementación**, permite hacer cambios que no afectan al código del usuario de la clase.
- En JAVA los **especificadores de acceso** se ubican adelante de la definición de cada atributo, método y constructor de una clase. El especificador solamente controla el acceso a dicha definición.

¿Qué pasa si a un miembro de una clase no le definimos especificador de acceso?

Tiene acceso por defecto, no tiene palabra clave y comúnmente se lo llama **acceso package o *friendly o privado del paquete***. Implica que tienen acceso a dicho miembro solamente las clases ubicadas en el mismo paquete que él y para las clases declaradas en otro paquete, es un miembro privado.

El acceso **package** le da sentido a agrupar clases en un paquete.

Especificadores de Acceso en Miembros

public

El atributo, método o constructor declarado **public** está disponible para **TODOS.**

```
package labo12;
public class Auto{
  public String marca;
  public Auto() {
    System.out.println("Constructor de Auto");
  }
  void arrancar(){
    System.out.println("arrancar");
  }
}
```

```
import labo12.*;
public class Carrera{
  public Carrera() {
     System.out.println("Constructor de Carrera");
  }
  public static void main(String[] args){
     Auto a=new Auto();
     System.out.println("Marca: "+ a.marca);
     a.arrancar();
  }
}
```

✓ La clase y el constructor son públicos-> es posible crear objetos Auto

✓ **marca** es un atributo público

✗ **arrancar()** es un método privado del paquete labo12

Especificadores de Acceso en Miembros

Privado del paquete (package)

Los miembros y constructores declarados **privados del paquete** son accesibles sólo desde clases pertenecientes al mismo paquete donde se declaran estos miembros.

```
package labo12;
public class Auto{
 public String marca;
Auto() {
   System.out.println("Constructor de Auto");
}
void arrancar(){
     System.out.println("arrancar");
}
}
```

```
import labo12.*;
public class Carrera{
public Carrera() {
   System.out.println("Constructor de Carrera");
}
public static void main(String[] args){
    Auto a=new Auto();
   System.out.println("Marca: "+ a.marca);
   a.arrancar();
}
}
```

✗ no se pueden crear objetos Auto afuera de labo12.

✗ arrancar() es un método privado del paquete labo12 ->no está disponible en otros paquetes y no lo heredan subclases declaradas en otros paquetes.

✓ Sólo es posible crear objetos **`Auto`** desde las clases del paquete **labo12**.

✓ Impacta sobre la herencia de la clase **`Auto`** siendo sólo posible crear subclases de **`Auto`** en el paquete **labo12**, inhibiendo el mecanismo de herencia desde afuera de este paquete. ¿Por qué?

Especificadores de Acceso en Miembros

private

El atributo, método o constructor declarado **private** solamente está accesible para la clase que lo contiene. Está disponible para usar adentro de los métodos de dicha clase.

```
public class Postre {
private Postre() {
   System.out.println("Constructor de Auto");
}
static Postre getPostre(){
    return new Postre();
}
}
```

```
public class Helado{
public static void main(String[] args){
    Postre p=new Postre();
    Postre z= Postre. getPostre();
}
}
```

El constructor es privado> NO es posible crear objetos Postre afuera de la clase Postre

```
public class Auto {
private String marca="sin marca";
private boolean tieneCombustible(){ return true;}
public String getMarca(){ return marca;}
public void arrancar(){
  if (tieneCombustible()) return true
     else return false;
  }
 }
}
```

Los **métodos privados** funcionan como utilitarios propios de la clase, sólo pueden invocarse desde otros métodos de la clase donde se declaran, no puedan reemplazarse a través del mecanismo de herencia y las modificaciones en su código no "rompen" el código que hace uso de la clase.

Las **variables privadas** forman parte de la implementación de la clase, son de uso exclusivo de la misma, sólo pueden manipularse directamente en los métodos de la clase y pueden modificarse sin perjudicar el código que hace uso de la clase.

Una buena práctica es declarar **private** todo lo posible.

Especificadores de Acceso en Miembros

protected

La palabra **protected** está relacionada con la herencia:

- Si se define una subclase en un paquete diferente que el de la superclase, a los únicos miembros de la clase base que tendrá acceso la subclase son a los definidos **public**.
- Si se define una subclase en el mismo paquete que la superclase, la subclase tiene acceso a todos los miembros declarados **public** y **package**.
- El autor de la clase base podría determinar que ciertos miembros pueden ser accedidos por las clases derivadas, pero no por todo el mundo. Esto es **protected**.
- Además el acceso **protected** provee acceso **package**: las clases declaradas en el mismo paquete que el miembro **protected** tienen acceso a dicho miembro.

```
package labo12;
public class Auto{
public Auto() {
   System.out.println("Constructor de Auto");
```

```
 protected void arrancar(){
        System.out.println("arrancar");
 }
```

```
}
```

Ahora **arrancar()** sigue teniendo acceso **package** en el paquete labo12, pero además es accesible para cualquier subclase de Auto, "lo heredan". No es **public**

```
import labo12.*;
public class Sedan extends Auto{
 public Sedan() {
  System.out.println("Constructor de Sedan");
 }
  public static void main(String[] args){
    Sedan x=new Sedan();
     x.arrancar();
  }
}
```

El método **arrancar()** está definido en la clase Auto, entonces lo heredan las subclases de Auto. ¿Qué ocurrió?
El método **arrancar() tiene acceso privado del paquete** y la clase Sedan no está en el mismo paquete que la clase Auto, entonces **arrancar() no está disponible en Sedan**.

Especificadores de Acceso en Miembros

protected

Analizaremos el acceso **protected** aplicado a variables.

```
package labo12;
public class Auto{
protected String marca;
protected String nroMotor;
public Auto() {
  System.out.println("Constructor de Auto");
}
protected void arrancar(){
     System.out.println("arrancar");
}
}
```

```
import labo12.*;
public class Sedan extends Auto{
private String identificador;
 public Sedan() {
   System.out.println("Constructor de Sedan");
  }
  public void setIdentificador(Auto v){
     this.identificador=this.marca+v.nroMotor;
    }
}
```

✓ Es válido el acceso a la variable protegida **marca** a través de la instancia **this** , ya que el acceso protegido habilita la herencia

✗ La variable **nroMotor** no está accesible porque **Auto** está en un paquete diferente a **Sedan**.

El acceso protegido privilegia la relación de herencia, las subclases heredan todas las variables y los métodos protegidos de sus superclases, independientemente del paquete donde estén definidas las clases y las subclases. Además el acceso protegido también es acceso privado del paquete.
Un miembro declarado protegido es parte de la API que se exporta, debe mantenerse siempre, es un compromiso público de un detalle de implementación.

Control de Acceso y Herencia

La especificación de JAVA establece que una subclase hereda de sus superclases todos los atributos y métodos de instancia accesibles:

- Si la subclase está definida en el mismo paquete que la superclase, hereda todos los atributos y métodos de instancia no-privados.
- Si la subclase está definida en un paquete diferente que su superclase, hereda solamente los atributos y métodos de instancia protegidos y públicos.
- Los atributos y métodos de instancia privados nunca son heredados. Lo mismo ocurre con los atributos y métodos de clase privados.

Los constructores no se heredan, se encadenan.

Podría ser confuso: "una subclase NO HEREDA los atributos y métodos de su superclase inaccesibles para ella" -> NO IMPLICA que cuando se crea una instancia de la subclase, no se aloque memoria para los atributos privados o inaccesibles definidos por la superclase.
Toda las instancias de una subclase incluyen una instancia COMPLETA de la superclase, incluyendo los miembros inaccesibles.
Como los miembros inaccesibles no puden usarse en la subclase, decimos NO SE HEREDAN.