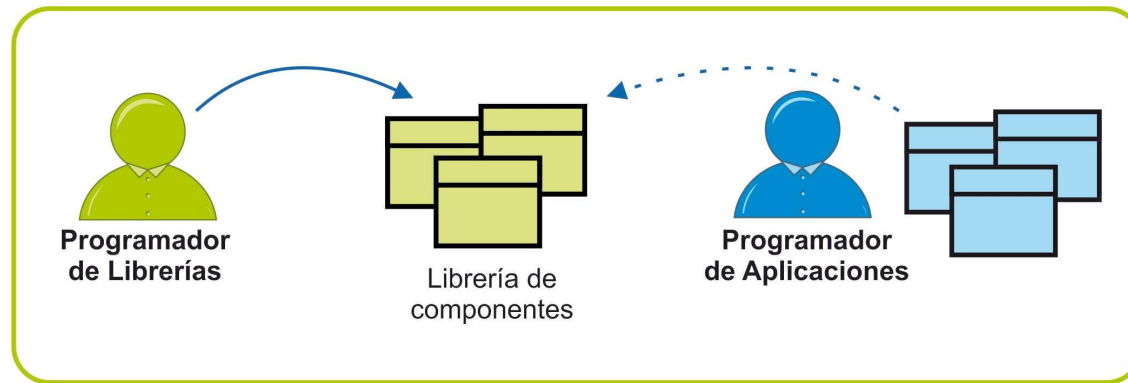


# Taller de Lenguajes II

Temas de hoy:

- **Librería de componentes: package**
  - Creación de librerías: paquete
  - Nombres únicos de clases
  - Archivos JAR (Java ARchive)
- **Especificadores de acceso**
  - public
  - protected
  - private
  - acceso predeterminado o por omisión (package)
- **Especificadores de acceso y herencia**
- **Tipos enumerativos**
- **Patrón singleton**

# ¿Qué podría pasar si se modifica una librería de clases que está siendo usada por otros programadores?



## El código podría romperse !!

El creador de la clase debe sentirse libre para mejorar el código y el programador cliente no debería escribir nuevamente su código si la librería se actualiza.

### ¿Cómo se asegura esto?

(1) Por convención: no quitar métodos existentes en la versión preva.

#### **Compatibilidad con versiones previas.**

(2) Usando especificadores de acceso para indicarle al programador cliente qué esté disponible y qué no lo está.

Antes de entrar en especificadores de acceso, falta responder una pregunta útil en este contexto: **¿cómo se crea una librería de clases en java?**

# Librería de componentes

## Paquetes JAVA

- En Java una librería de componentes (clases e interfaces) es un grupo de archivos `.class`, también llamado **paquete**.
- Para agrupar componentes en una librería o paquete, debemos anteponer la palabra clave **package** junto con el nombre del paquete al comienzo del archivo fuente de cada una de las componentes.

```
package graficos;  
public class Rectangulo {  
    //código JAVA  
}
```

Establece que la clase **Rectangulo** pertenece al paquete **graficos**

- Las clases e interfaces que se crean sin usar la sentencia **package** se ubican en un paquete sin nombre, llamado *default package*.

```
public class HolaMundo {  
    //código JAVA  
}
```

Establece que la clase **HolaMundo** pertenece al paquete **por defecto**.

¿Qué piensan es recomendable usar, paquetes propios o el *default package*? ¿por qué?

# Librería de componentes

## Paquetes JAVA

El nombre completo de la clase o nombre canónico contiene el nombre del paquete.

<code>graficos.Rectangulo</code>	→	Nombre completo de la clase <code>Rectangle</code>
<code>java.util.Arrays</code>	→	Nombre completo de la clase <code>Arrays</code>

Para usar la clase **Rectangulo** se debe usar la palabra clave `import` o **especificar el nombre completo** de la clase:

```
package ar.edu.unlp.taller2;
import graficos.Rectangulo;
// import graficos.*; otra forma

class Figuras {
    Rectangulo r = new Rectangulo();
}
```

```
package ar.edu.unlp.taller2;
class Figuras {
    graficos.Rectangle r;
    r = new graficos.Rectangle();
}
```

La sentencia `import` permite usar el **nombre corto de la clase** en todo el código fuente. Si no se usa el `import` se debe especificar el **nombre completo** de la clase.

# Librería de componentes

## Paquetes JAVA

### ¿Qué sucede si se crean 2 clases con el mismo nombre?

Supongamos que 2 programadores escriben una clase de nombre **Vector** en el paquete *default*, **se plantea un conflicto de nombres**.

Es necesario crear nombres únicos para cada clase para **evitar colisión de nombre**.

```
package util;  
public class Vector {  
    //código JAVA  
}
```

```
package taller2.estructuras;  
public class Vector {  
    //código JAVA  
}
```

### ¿Qué sucede si se importan dos librerías que incluyen el mismo nombre de clase?

```
import taller2.estructuras.*;    Ambas contienen la clase Vector  
import util.*;
```

```
Vector vec1 = new Vector();
```

```
taller2.estructuras.Vector vec2 = new taller2.estructuras.Vector();
```

**Colisión!** ¿A qué clase hace referencia?: el compilador no puede determinarlo, fuerza a escribir el nombre completo de la clase

ok!

# Librería de componentes

## Paquetes JAVA

- Las clases e interfaces que son parte de distribución estándar de JAVA están agrupadas en paquetes de acuerdo a su funcionalidad. Algunos paquetes son:  

<code>java.lang</code>	clases básicas para crear aplicaciones.
<code>java.util</code>	librería de utilitarios, colecciones.
<code>java.io</code>	manejo de entrada/salida.
<code>java.awt / javax.swing</code>	manejo de GUI (Graphic User Interface).
- Los únicos paquetes que se importan automáticamente es decir no requieren usar la sentencia `import` son el paquete `java.lang` y el **paquete actual** (paquete en el que estamos trabajando).

```
package taller2.estructuras;  
public class Vector {  
    //código JAVA  
}
```

Se pueden usar, sin importar  
todas las clases del paquete  
`taller2.estructuras`

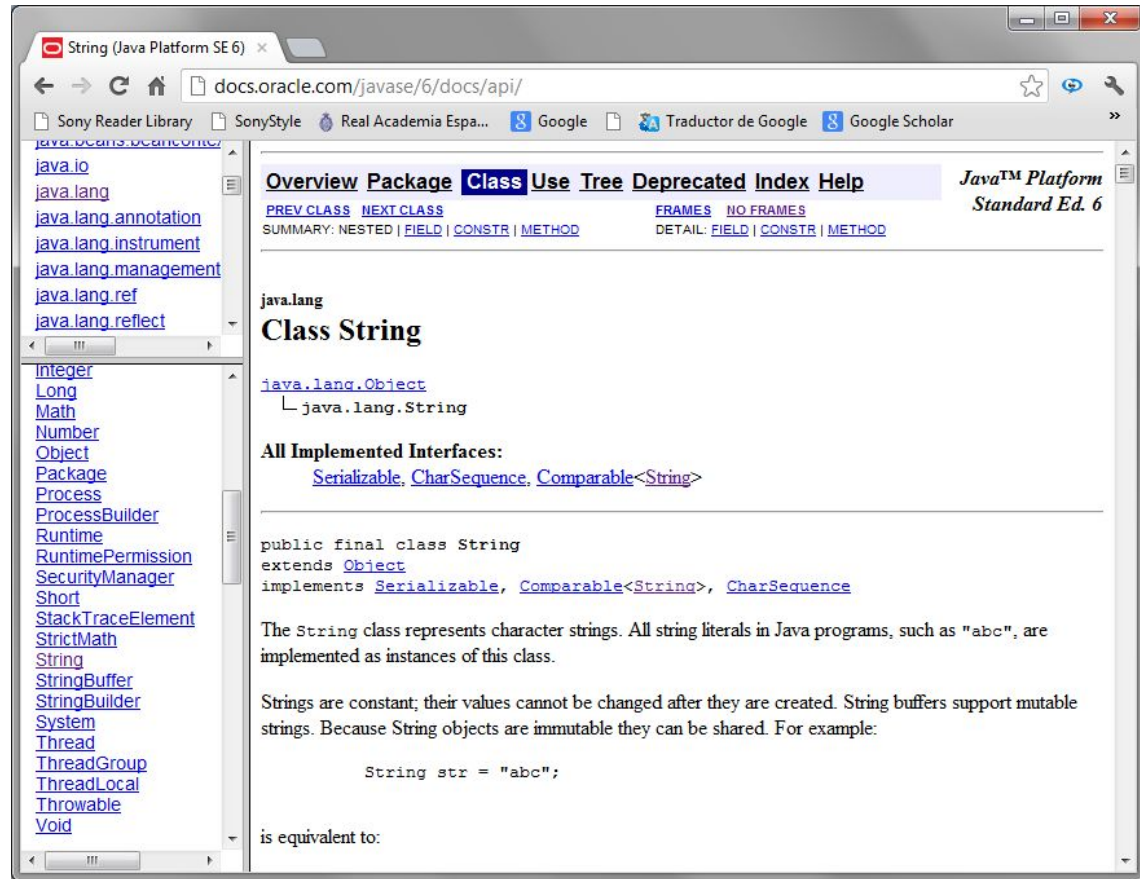
**Recomendación:** usar como primera parte del nombre del paquete, el nombre invertido del dominio de Internet y así evitar la colisión de nombres. Usar **minúscula para nombres de paquetes** e **inicial mayúscula para nombres de clases**.

**Ejemplos** `ar.edu.unlp.graficos`

# Paquetes en JAVA

## La API (Application Programming Interface)

- La API JAVA es una colección de clases y otras componentes de software compiladas (archivos .class) que proveen una amplia gama de funcionalidades como componentes de GUIs, I/O, manipulación de colecciones, etc.
- La API está agrupada en librerías de clases e interfaces relacionadas, llamadas paquetes.
- El programador puede combinar las componentes de la API JAVA con su código para crear una aplicación.



La documentación de la API esta disponible en: <https://docs.oracle.com/javase/10/docs/api/>

# Librería de componentes

## Paquetes JAVA

Un paquete normalmente está formado por varios archivos `.class`. Java se beneficia de la **estructura jerárquica de directorios del sistema operativo** y ubica todos los `.class` de un mismo paquete en un mismo directorio. De esta manera, se resuelve:

- el nombre único del paquete
- la búsqueda de los `.class` (que de otra forma estarían diseminados en el disco)

```
package ar.edu.unlp.utiles;  
public class Vector {  
    //código JAVA  
}
```

`\ar\edu\unlp\utiles\Vector.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 un miembro `static`, se procede de la siguiente manera:

- Busca en los **directorios estándares del JRE** y recupera todos los `.class` de la API de JAVA de la carpeta `/lib`
- Busca en el directorio actual (paquete de la clase que se está ejecutando)
- Recupera la variable de entorno **CLASSPATH**, que contiene la lista de directorios usados como raíces para buscar los archivos `.class`. Comenzando en la raíz, el intérprete toma el nombre del paquete (de las sentencias `import`) y reemplaza cada `“.”` por una barra `“\”` o `“/”` (según el SO) para generar un camino donde encontrar las clases a partir de las entradas del **CLASSPATH**.



# Paquetes en JAVA

## Nombres únicos

Consideremos el dominio `unlp.edu.ar` invertido y obtenemos un nombre de dominio único y global: `ar.edu.unlp`. Si creamos una librería `utiles` con las clases `Vector` y `List`, tendríamos:

```
package ar.edu.unlp.utiles;  
public class Vector {  
    //código JAVA  
}
```

```
package ar.edu.unlp.utiles;  
public class List {  
    //código JAVA  
}
```

Supongamos que a ambos archivos los guardamos en el directorio `c:\tallerjava\`.

```
C:\tallerjava\ar\edu\unlp\utiles\Vector.class  
C:\tallerjava\ar\edu\unlp\utiles>List.class
```

¿A partir de dónde comienza el intérprete JAVA a buscar el paquete `ar.edu.unlp`?

A partir de alguna de las entradas indicadas en la variable de entorno **CLASSPATH**:

```
CLASSPATH=.;c:\tallerjava;c:\java\librerias
```

Esta variable puede contener muchas entradas separadas por ";"

# Paquetes en JAVA

## Organización de archivos – Formato JAR

Es posible agrupar archivos `.class` pertenecientes a uno o más paquetes en un único archivo con extensión **jar (Java ARchive)**. El formato **JAR** usa el formato **zip**. Los archivos JAR son multi-plataforma, es estándar. Es posible incluir además de archivos `.class`, archivos de imágenes y audio, recursos en general, etc.

El JSE o JDK tiene una herramienta que permite crear archivos JAR, desde la línea de comando, es el utilitario **jar**.

Por ejemplo: si se ejecuta el comando `jar` desde el directorio donde están los archivos `.class` podríamos ponerlo así:

```
c:\tallerjava\ar\edu\unlp\utiles\jar cf utiles.jar *.class
```

- En este caso, en el **CLASSPATH** se especifica el nombre del archivo jar:

```
CLASSPATH=.; c:\utiles.jar ;c:\java\librerias
```

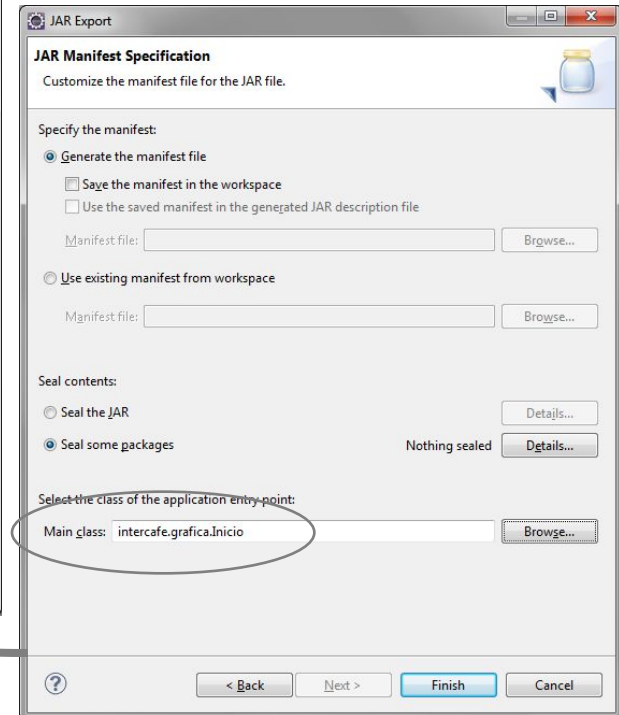
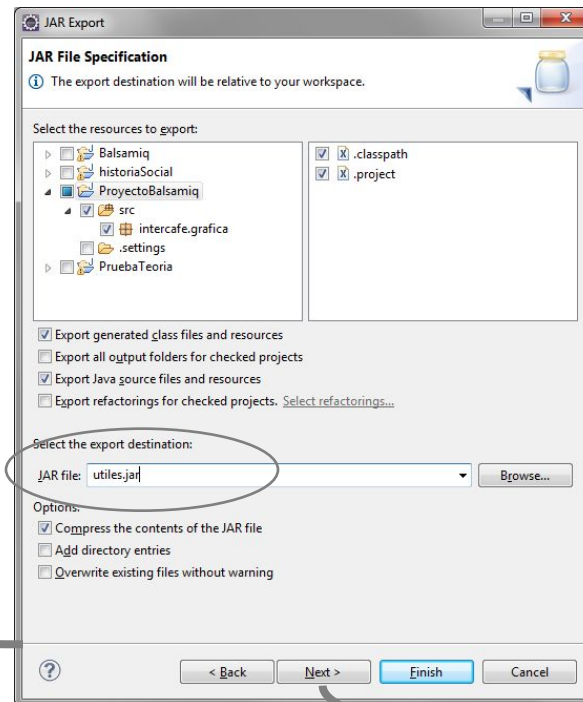
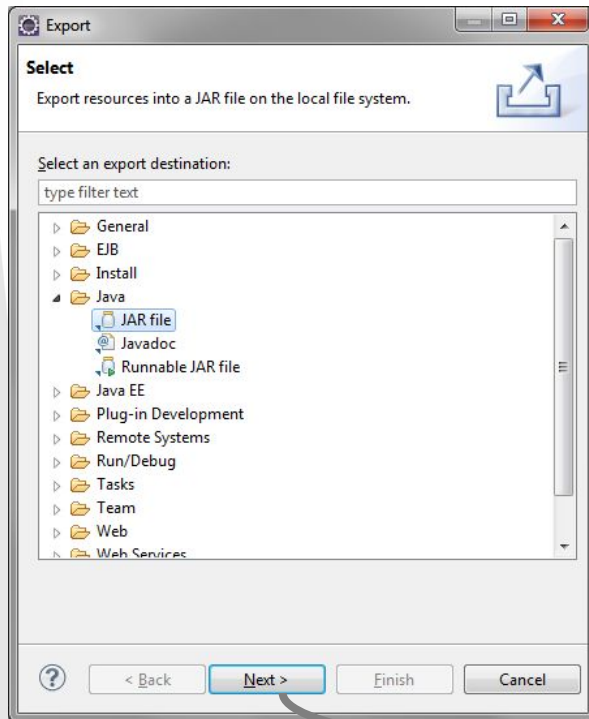
**Los archivos jar pueden ubicarse en cualquier lugar del disco**

- El intérprete JAVA se encarga de buscar, descomprimir, cargar e interpretar estos archivos.

# Paquetes en JAVA

## Organización de archivos – Formato JAR

El archivo JAR también puede construirse desde un proyecto Eclipse, con la opción **export**.



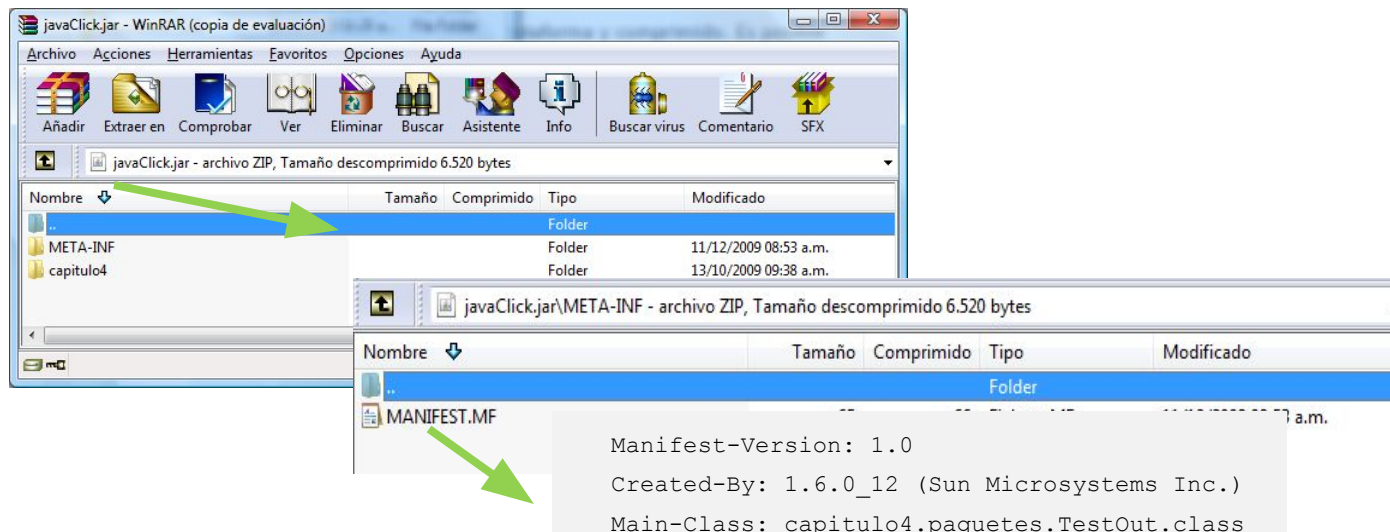
# Paquetes en JAVA

## El formato JAR

Los archivos JAR contienen todos los paquetes con sus archivos .class, los recursos de la aplicación y 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 clase que contiene método main).

Para especificar cuál será esta **clase “principal”**, el archivo MANIFEST.MF debe contener la **entrada Main-Class**.



# Especificadores de acceso

- Permiten al autor de una **librería de componentes** establecer **qué está disponible** para el usuario (programador cliente ) y **qué no**. Esto se logra usando alguno de los siguientes especificadores de acceso:

<b>public</b>	<b>protected</b>	<b>package</b> (no tiene palabra clave)	<b>private</b>
---------------	------------------	--	----------------

más libre (+)

más restrictivo (-)

- El **control de acceso** permite **ocultar la implementación**. Le permite a un programador limitar el acceso a las clases para posteriormente poder hacer cambios que no afecten al código del usuario de dicha clase.
- En Java, los **especificadores de acceso** se ubican delante de la definición de la clase, método y/o atributo de la clase.

```
package utiles;

public class Pila {
    private Lista items;
    public Pila(){
        items = new Lista();
    }
}
```

**Cada especificador  
controla el acceso a  
dicha definición**

# Especificadores de acceso

## ¿Qué pasa si a un miembro de una clase no se le define especificador de acceso?

- Tiene acceso por defecto, no tiene palabra clave y comúnmente se lo llama acceso **package** o *friendly*. Implica que tienen acceso a dicho miembro sólo las clases ubicadas en el mismo paquete que él.
- El acceso **package** le da sentido al agrupamiento de clases en paquetes.

```
package ar.edu.unlp.taller2;

public class Cola {
    Lista elementos;
    Cola() {
        elementos = new Lista();
    }
    Object pop() {
        return
        elementos.getFirst();
    }
    void push(Object o) {
        elementos.addLast(o);
    }
}
```

```
package ar.edu.unlp.taller2;
```

```
public class Estructuras {
```

```
    public static void main(String[] args) {
        Cola cola1 = new Cola();
        cola1.push(1);
        cola1.elementos=new Lista();
    }
}
```

✓ El acceso es válido porque pertenecen al mismo paquete

¿Qué pasa si elimino las líneas `package ar.edu.unlp.taller2` en ambas definiciones de las clases?, ¿se mantiene válido el acceso?

# Especificadores de acceso

## public

- El atributo o método declarado **public** está disponible para TODOS. Cualquier clase de cualquier paquete tiene acceso.
- Esto es útil para los programadores que hacen uso de la librería o paquete.

```
package ar.edu.unlp.taller2;
public class Cola {
    public Lista elementos;
    public Cola(){
        elementos = new Lista();
    }
    Object pop(){
        return
        elementos.getFirst();
    }
    public void push(Object obj){
        elementos.addLast(obj);
    }
}
```

```
package pruebastaller;
import ar.edu.unlp.taller.*;
public class Estructuras {
```

```
    public static void main(String[] args){
```

```
        Cola cola1 = new Cola();
```

✓ La clase es pública y el constructor es público, es posible crear objetos Cola

```
        cola1.elementos=new Lista()
        cola1.push(1);
    }
}
```

✓ También es posible acceder a sus miembros.

¿Es posible invocar cola1.pop() desde la clase Estructuras?




# Especificadores de acceso

## private

- El atributo, método o constructor declarado **private** solamente está accesible para la clase que lo contiene. Los miembros **private** están disponibles para su uso adentro de los métodos de dicha clase. Lo mismo ocurre con los atributos y constructores **private**.

```
package ar.edu.unlp.taller2;

public class Cola {
    private Lista elementos;
    private Cola(Lista e){
        this.elementos = e;
    }
    public static Cola getCola(Lista lis){
        // podría hacerse algún control
        return new Cola(lis);
    }
    public Object pop(){//Código JAVA}
    public void push(Object o)
        {//Código JAVA}
}
```

```
package pruebastaller;
import ar.edu.unlp.ayed.Cola;
public class Estrucutras {
    public static void main(String[] args){
        Lista lis = new Lista();
        Cola c1 = new Cola(lis); 
        Cola c2 = Cola.getCola(lis); 
        c2.elementos= lis; 
    }
}
```



# Especificadores de acceso

## private

¿Es posible definir una subclase de Cola?

```
package ar.edu.unlp.taller2;
public class Cola {
    private Lista elementos;
    private Cola(Lista e){
        elementos = e;
    }
    public static Cola getCola(Lista lis){
        // podría hacerse algun control
        return new Cola(lis);
    }
    public Object pop(){//Código JAVA}
    public void push(Object o)
        {//Código JAVA}
}
```

```
package ar.edu.unlp.taller2;
public class ColaPrioridades
    extends Cola {

    public ColaPrioridades(Lista l){
        super(l);
    }
    public Object pop(){//Código JAVA}
    public void push(Object o)
        {//Código JAVA}
}
```

**La clase ColaPrioridades no compila debido a que el constructor de la superclase Cola(Lista e) no está disponible en la clase ColaPrioridades**

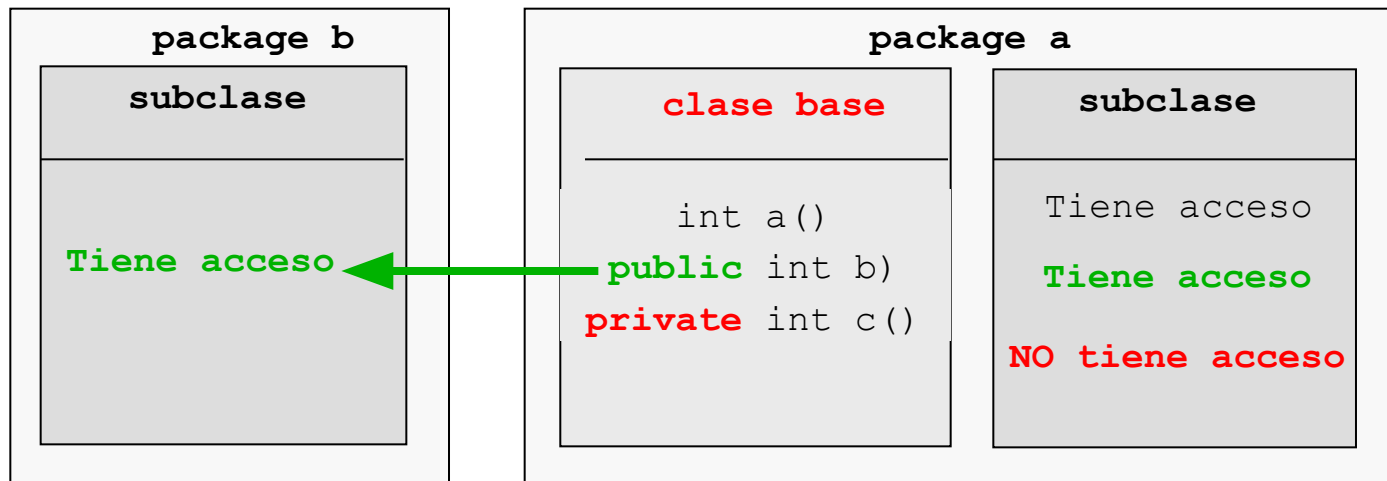
La herencia se implementa a través de la invocación de constructores de las superclases hasta alcanzar la clase Object. En este ejemplo el constructor de la clase ColaPrioridades intenta invocar al constructor de la clase Cola, el cual es inaccesible debido a que está definido como privado. **El especificar de acceso private impacta sobre la herencia.**

# Especificadores de acceso

## protected

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

- Si se crea una subclase en un paquete diferente que el de la superclase, la subclase tiene acceso sólo a los miembros definidos **public** de la superclase.
- Si la subclase pertenece al mismo paquete que la superclase, entonces la subclase tiene acceso a todos los miembros declarados **public** y **package**.



Es decir la subclase no hereda todos los métodos.

¿Es posible definir métodos que sean accesibles por las subclases y no por todos?

Si !! esto es **protected**. Un miembro **protected** puede ser accedido por las subclases definidas en cualquier paquete. Además **protected** provee acceso **package**

# Especificadores de acceso

## protected

```
package ar.edu.unlp.taller2;
public class Lista {
    private Nodo first;
    private Nodo current;
    public boolean add(String elto){
        Nodo nodo = new Nodo(elto);
        if (current == null)
            first = nodo;
        else {
            nodo.setNext(current.getNext());
            current.setNext(nodo);
        }
        current = nodo;
        return true;
    }
    protected Nodo getCurrent()
        return current;
    ...
}
```

Si `getCurrent()` es **protected** es accesible para cualquier subclase de `Lista` y no es **public**. !!

```
package misListas;
import ar.edu.unlp.taller2.*;

public class ListaPosicional extends Lista {

    public String get(int pos) {...}
    public boolean remove(int pos) {...}
    public boolean add(String elto, int pos) {
        Nodo nodo= new Nodo(elto);
        if (this.getCurrent()==null) {
            ...
        }
    }
}
```

El método `getCurrent()` está definido en la clase `Lista`, entonces también debería estar disponible en cualquier subclase de `Lista`.

Pero, si dicho método tiene acceso **package**, como la clase `ListaPosicional` no está en el mismo paquete que la clase `Lista`, `getCurrent()` no está disponible en `ListaPosicional`.

# Especificadores de acceso en clases

- En Java, los especificadores de acceso en clases se usan para determinar cuáles son las clases disponibles de una librería.
- Una clase declarada **public** está disponible para cualquier clase, mediante la cláusula **import**. Se pueden crear instancias de la clase (siempre y cuando exista algún constructor público).

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

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

**Cualquier clase que importa el paquete:**  
**import gui.\***  
**Tiene disponible ambas clases!!**

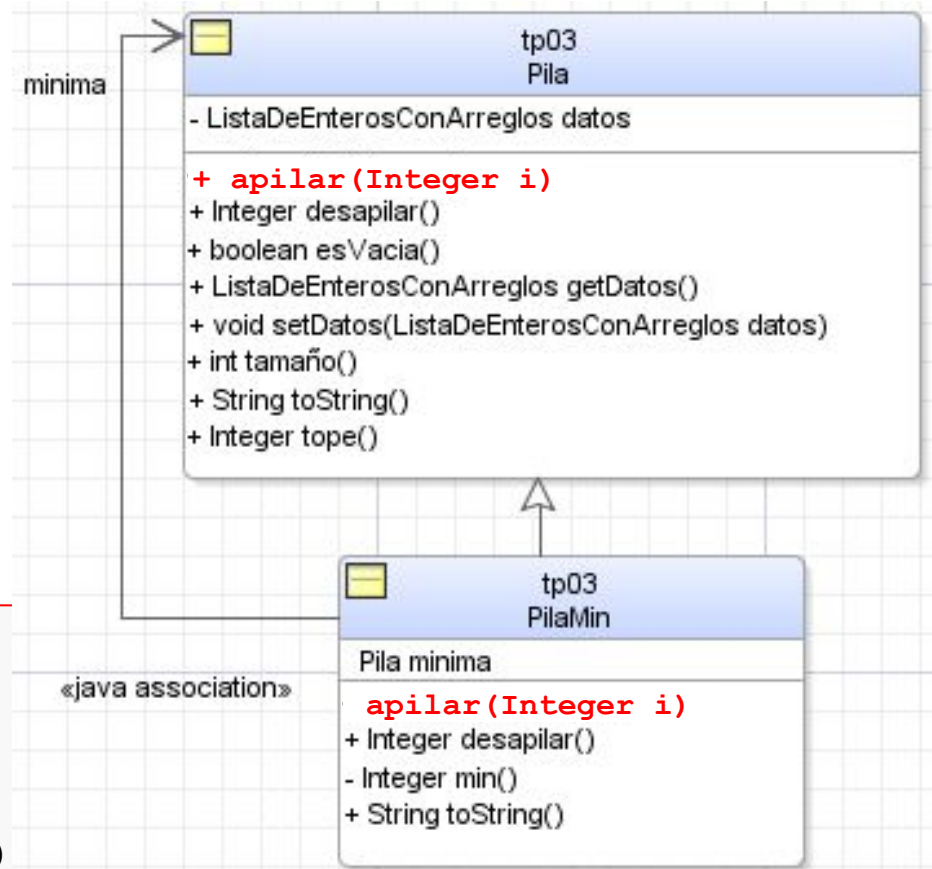
Supongamos que la clase **Soporte** la usan clases del paquete **gui**, pero no se quiere que esté accesible a clases pertenecientes a otros paquetes distintos de **gui**, ¿cómo se define?

Se la define de acceso **package** y de esta manera solamente puede usarla las clases del paquete **gui**. Es razonable que los miembros de una clase de acceso **package** tengan también acceso **package**.

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

 **La clase Soporte puede usarse solamente en el paquete **gui****

# Especificadores de acceso y herencia



```
package tp03.accesos;
import tp03.Pila;
import tp03.PilaMin;
public class PilasTest {
    public static void main(String[] args)
    {
        Pila[] pilas = { new Pila(), new PilaMin(), new Pila() };
        for (int i = 0; i < pilas.length; i++) {
            pilas[i].apilar(2*(i+5));
        }
    }
}
```

Los métodos sobrescritos no pueden tener un control acceso más restrictivo que el declarado en la superclase. En las subclases **apilar()**, **#apilar**, **-apilar()** no son válidos.

# Tipos enumerativos

## ¿Qué son?

- Los tipos enumerativos se incorporaron a la plataforma JAVA partir de JAVA 5.0. Constituyen una **categoría especial de clases**.
- Un tipo enumerativo es un tipo “referencial” que tiene asociado un **conjunto de valores finito y acotado**.
- La **palabra clave enum** se usa para definir un nuevo tipo enumerativo:

**package taller2;**

**public enum Estados {CONECTANDO, LEYENDO, LISTO, ERROR ;}**

- Los **valores** son **constantes públicas de clase** (**public static final**) y se hace referencia a ellas de la siguiente manera: **Estados.CONECTANDO**, **Estados.LEYENDO**. A una variable de tipo **Estados** se le puede asignar uno de los **4 valores definidos o null**. Los valores de un tipo enumerado se llaman **valores enumerados y también constantes enum**.
- El **tipo enumerativo es una clase** y sus **valores son instancias de dicha clase**. Garantiza seguridad de tipos. Es una diferencia fundamental con usar constantes de tipo primitivo. El compilador puede chequear si a un método se le pasa un objeto de tipo enum.
- Por convención los valores de **los tipos enumerativos se escriben en mayúsculas** como cualquier otra constante de clase.

# Tipos enumerativos

## Ejemplos

Los tipos enumerativos son tipos de datos y por lo tanto pueden usarse para declarar y asignar valores a variables simples, arreglos y colecciones de objetos. Se declaran y usan de manera similar que las clases e interfaces.

```
Estados servidor = ESTADOS.CONECTANDO;
```

```
Estados servidor[] = {Estados.CONECTANDO, Estados.LEYENDO, Estados.LISTO, Estados.ERROR};
```

```
public abstract class Servicio implements Runnable{  
    private String nombre;  
    private String descripcion;  
  
    // Estado  
    protected Estados estado = Estados.CONECTANDO;  
  
    //Código JAVA  
}
```

Java 5.0 incorpora la clase **java.util.EnumMap** que es una implementación especializada de un Map que requiere como clave un tipo Enumerativo y la clase **java.util.EnumSet** que requiere valores de tipo Enumerativo. Ambas estructuras de datos están optimizadas para tipos Enumerativos.

# Tipos enumerativos

## Ejemplos

A partir de Java 5.0 la sentencia **switch** soporta tipos enumerativos.

Si el tipo de la declaración de la expresión switch es un tipo enumerativo, las etiquetas de los case deben ser todas instancias sin calificación de dicho tipo. Es ilegal usar null como valor de una etiqueta case.

```
public void testSwitch(Estados unEstado) {  
  
    switch(unEstado) {  
    case CONECTANDO: {  
        System.out.println(unEstado);  
        break;  
    }  
    case LEYENDO: {  
        System.out.println(unEstado);  
        break;  
    }  
    case LISTO: {  
        System.out.println(unEstado);  
        break;  
    }  
    case ERROR:  
        throw new IOException("Error");  
    }  
}
```

Al tener un **conjunto finito de valores**, los tipos enumerados son **ideales** para usar con la sentencia **switch**.

```
package taller2;  
public enum Estados {  
    CONECTANDO, LEYENDO, LISTO, ERROR;  
}
```



# Tipos enumerativos

## Con variables y métodos de instancia

### Ejemplo

```
package taller2;
public enum Prefijo {
    MM("m",.001),
    CM("c",.01),
    DM("d",.1),
    DAM("D",10.0),
    HM("h",100.0),
    KM("k",1000.0) ;
    private String abrev;
    private double multiplicador;
    Prefijo(String abrev, double multiplicador) {
        this.abrev = abrev;
        this.multiplicador = multiplicador;
    }

    public String abrev() { return abrev; }
    public double multiplicador() { return multiplicador; }
}
```

Cada **constante** se declara con **valores** para la **abreviatura** y para el factor **multiplicador**

El **constructor** tiene **acceso privado** o **privado del paquete**.

**Automáticamente** crea todas las instancias del tipo y no puede ser invocado. El constructor es único, no hay sobrecarga de constructores.

**No tienen constructores públicos.**

**Se debe proveer un constructor.**

**Métodos** que **permiten recuperar** la **abreviatura** y el **factor multiplicador** de cada **Prefijo**

# Tipos enumerativos

## Con variables y métodos de instancia

### Ejemplo

```
package taller2;

public class TestPrefijo {
    public static void main(String[] args) {
        double longTablaM= Double.parseDouble(args[0]);

        for (Prefijo p : Prefijo.values() )
            System.out.println("La longitud de la tabla en "+ p+ " "
                               +longTablaM*p.multiplicador());
    }
}
```

**java TestPrefijo 15**

La longitud de la tabla en MM 0.015  
La longitud de la tabla en CM 0.15  
La longitud de la tabla en DM 1.5  
La longitud de la tabla en DAM 150.0  
La longitud de la tabla en HM 1500.0  
La longitud de la tabla en KM 15000.0

Cuando se crea un tipo **enum** el compilador automáticamente agrega algunas características útiles. Los tipos **enum** disponen por ejemplo del método **toString()** que por defecto **retorna el nombre de la instancia enum**. El método **toString()** se puede sobrescribir. Todos los tipos enumerativos soportan el método **values()** que retorna un arreglo con todos los valores del enum

# Patrones de diseño

Los patrones de diseño son una solución general reutilizable para problemas comunes. Son las mejores prácticas utilizadas por desarrolladores experimentados.

Los patrones no son códigos completos, pero pueden usarse como una plantilla que se puede aplicar a un problema. Son reutilizables, se pueden aplicar a un tipo similar de problema de diseño independientemente de cualquier dominio.

En otras palabras, podemos pensar en patrones como problemas recurrentes de diseño con sus soluciones. Un patrón usado en un contexto práctico puede ser reutilizable en otros contextos también.

Los patrones de diseño pueden clasificarse en las siguientes categorías:

- **Patrones de creación:** Singleton, Builder.
- Patrones estructurales: Adaptador, Decorador, etc.
- Patrones de comportamiento: Iterador, Estrategia, etc

# Patrón de Diseño Singleton

## Implementación en JAVA

El patrón de diseño Singleton restringe la instanciación de una clase y asegura que solamente una instancia de la clase exista en la máquina virtual JAVA.

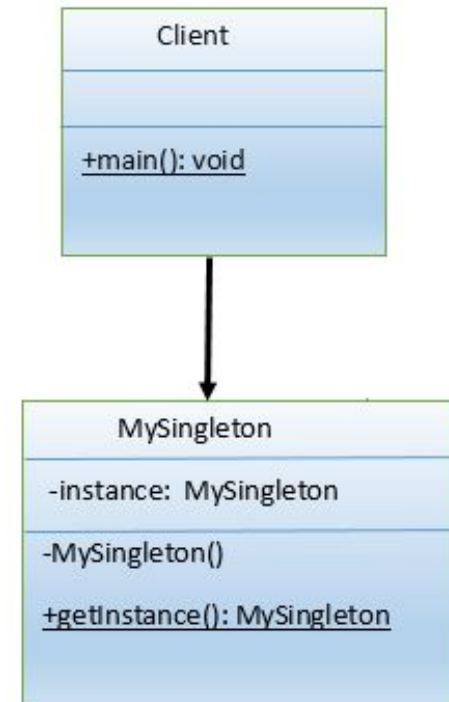
La **clase singleton** debe proveer un **acceso público a esa instancia de la clase**.

El patrón singleton se usa en muchas soluciones como logging, drivers, pool de threads, etc. y en la propia API Java, por ejemplo en la clase `java.lang.Runtime` o `java.awt.Desktop`.

Para implementar el **patrón singleton**, debemos:

- Crear un **constructor privado** para evitar la creación de objetos desde otras clases.
- Definir una **variable privada de clase** del tipo de la clase para **referenciar a la única instancia** de esa clase.
- Definir un **método público de clase** que retorne esa instancia.

Existen diferentes alternativas  
para implementarlo



# Patrón de Diseño Singleton

## Implementación en JAVA

La **instancia de la clase singleton** se crea en el momento de la **carga de la clase**, este es el método más sencillo para crear una clase singleton. **Inicialización temprana.**

```
package patrones;

public class EagerSingleton {
    private static EagerSingleton INSTANCE = new EagerSingleton();

    //constructor privado
    private EagerSingleton() {
    }

    public static EagerSingleton getInstanceEager(){
        return INSTANCE;
    }
}
```

# Patrón de Diseño Singleton

## Implementación en JAVA

La implementación de este patrón es similar al anterior, excepto que la instancia de la clase es creada en el bloque estático, cuando se carga la clase. **Inicialización temprana.**

```
package patrones;
public class BloqueEstaticoSingleton {
    private static BloqueEstaticoSingleton INSTANCE;

    // bloque de inicialización estático
    static {
        INSTANCE = new BloqueEstaticoSingleton();
    }
    private BloqueEstaticoSingleton() {}

    public static BloqueEstaticoSingleton getInstance() {
        return INSTANCE;
    }
}
```

# Patrón de Diseño Singleton

## Implementación en JAVA

Se crea la instancia en un método de clase de acceso público. **Inicialización *lazy* o *perezosa*.**

```
package patrones;

public class LazySingleton {
    private static LazySingleton INSTANCE;

    private LazySingleton(){}

    public static LazySingleton getInstance(){
        if (INSTANCE == null)
            INSTANCE = new LazySingleton();

        return INSTANCE;
    }
}
```

# Patrón de Diseño Singleton

## Implementación en JAVA

Implementación **usando tipos enumerativos** dado que garantiza la existencia de una instancia en la JVM.

```
package patrones;

public enum EnumSingleton {
    INSTANCE;
}
```

```
package patrones;

public enum EnumSingleton {
    INSTANCE;
    private EnumSingleton () {
        System.out.println("Constructor");
    }
}
```



# Patrón de Diseño Singleton

## Un caso de uso: Pool de conexiones a una Base de Datos

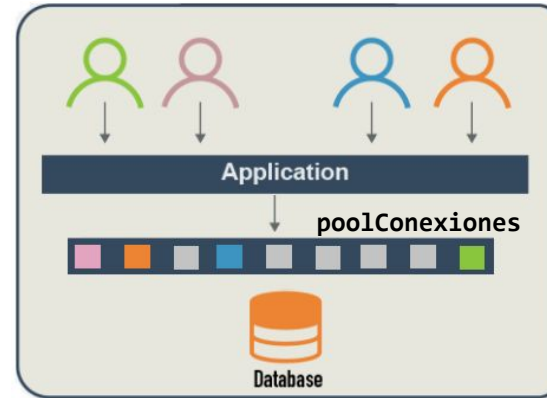
```
package patrones;
import java.sql.Connection;

public class PoolConexiones {
    private static PoolConexiones INSTANCE;
    private Connection pool[] = new Connection[10];

    private PoolConexiones(){
        // Se establecen las conexiones con la DB
        // y se guarda en la variable de instancia pool
    }

    public static PoolConexiones getInstance(){
        if (INSTANCE == null){
            INSTANCE = new PoolConexiones();
        }
        return INSTANCE;
    }

    public Connection getConnection() {
        // Buscar una conexión libre
        // int x=...
        return pool[x];
    }
}
```



```
package patrones;
import java.sql.Connection;
import java.sql.SQLException;

public class TestPoolConexiones {
    public static void main(String[] args) {
        Connection con = PoolConexiones.getInstance().getConnection();
        con.prepareStatement("select * from usuarios where usr=?");
    }
}
```

# Patrón de Diseño Singleton

## Un caso de uso: Pool de conexiones a una Base de Datos

### Con enumerativos:

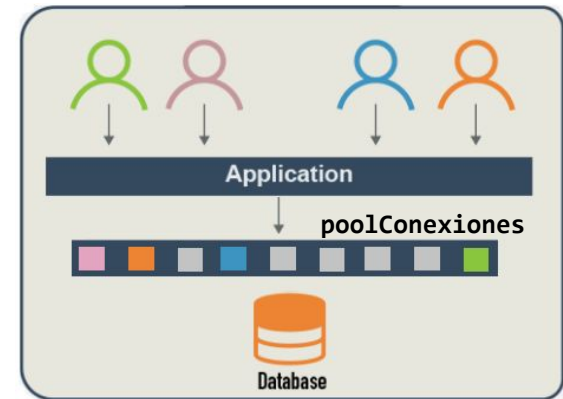
```
package patrones;
import java.sql.Connection;

public enum PoolConexionesEnum {
    INSTANCE;

    private Connection[] pool = new Connection[10];

    private PoolConexionesEnum() {
        // se establece las conexiones con la DB
    }

    public Connection getConnection() {
        // Buscar una conexión libre
        // int x=...;
        return pool[x];
    }
}
```



```
package patrones;

import java.sql.Connection;
import java.sql.SQLException;

public class TestPoolConexiones {
    public static void main(String[] args) {
        Connection con = PoolConexionesEnum.INSTANCE.getConnection();
        con.prepareStatement("select * from usuarios where usr=?");
    }
}
```