

# Taller de Lenguajes II

- **Especificadores de acceso: ¿para qué sirven? ¿cómo se usan?**
  - public
  - protected
  - private
  - acceso predeterminado o por omisión (package)
- **Relación entre especificadores de acceso y herencia**
- **Tipos enumerativos: ¿qué son? ¿para qué usarlos? ¿cómo se definen?**
- **El patrón singleton: ¿qué es? ¿para qué usarlo?**

# Especificadores de acceso

- En JAVA es posible **controlar el acceso** a **clases, interfaces**, y sus **miembros y constructores**, mediante **especificadores de acceso**.
- Los **especificadores de acceso** establecen **qué está disponible y qué no** para los programadores que utilizan dichas entidades.
- Los **especificadores de acceso** permiten implementar **diferentes niveles de ocultamiento de información**.

**¿A qué entidades es posible aplicar control de acceso?**

- Clases e interfaces de nivel superior
- Miembros de clases (métodos, propiedades) 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** en las **declaraciones de clases** se usan para determinar **qué clases están disponibles para usar**. Lo mismo ocurre con las declaraciones de **interfaces**.

Una **clase** declarada **public** es parte de la **API** que se **exporta** y está disponible mediante la cláusula **import**.

```
package gui;  
public class Control {  
    //TODO  
}  
  
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 del paquete gui?

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

La clase **Soporte** es **privada del paquete gui**: sólo está accesible en el 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 (propiedades y métodos) de una clase de acceso **package** tengan también acceso **package**.

# Especificadores de acceso en Métodos, Variables y Constructores

Los **especificadores de control de acceso** son reglas de control de acceso que restringen el uso de las **variables, métodos y constructores de una clase**. Permiten que el programador de la clase determine **qué está disponible** para el programador que usa dicha clase y **qué no**.

Los **especificadores de control de acceso de JAVA** son:

<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**. Separa la **interface pública** de la **implementación**, permite hacer cambios que no afectan al código del usuario de la clase.

# Especificadores de acceso en Métodos, Variables y Constructores

En JAVA los **especificadores de acceso** se ubican delante de la definición de cada **variable, método y constructor**. El **especificador** solamente **controla el acceso a dicha definición**.

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

**Cada especificador 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. 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 Métodos, Variables y Constructores

## public

- Las **variables, métodos y constructores** declarados **public** están disponibles para **TODOS**. Cualquier clase de cualquier paquete tiene acceso a ellos.
- Las **definiciones públicas** forman parte de la **interface pública**.
- Están disponibles para los programadores que usan 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.taller2.*;
public class Estructuras {

    public static void main(String[] args) {

        Cola cola1 = new Cola();

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

    }
}
```

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

✓ Es posible acceder a los elementos de la Cola.

✓ Es posible agregar elementos a la Cola.

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

# Especificadores de acceso en Métodos, Variables y Constructores **private**

- Las **variables, métodos y constructores** declarados **private** solamente son accesibles para la **clase que los contiene**.
- Las **definiciones privadas** están disponibles para **uso interno de la clase**, solo pueden usarse en métodos de dicha clase.
- Forman parte de la **implementación**.

```
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}
}
```

La clase Lista es pública y es parte de **ar.edu.unlp.taller2**

```
package pruebastaller;
import ar.edu.unlp.taller2.*;
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; ✗
    }
}
```

¿Cómo creamos instancias de Cola?  
¿A qué métodos se tiene acceso?

# Especificadores de acceso en Métodos, Variables y Constructores

## Control de acceso private y herencia

¿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) es **privado**: no está disponible en la clase ColaPrioridades

En Java la herencia se implementa a través de la invocación a 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 en Métodos, Variables y Constructores

## Privado del paquete (package)

- Las **variables, métodos y constructores** declarados **privados del paquete** son accesibles sólo por las clases pertenecientes al mismo paquete donde se declaran dichas definiciones.
- Forman parte de la **implementación**.

```
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 clases?, ¿se mantiene válido el acceso? y si a la clase `Estructuras` la ubicamos en un paquete diferente al de la clase `Cola`, ¿es válido el acceso?

# Especificadores de acceso en Métodos, Variables y Constructores

## Control de acceso package y herencia

¿Se pueden crear instancias de Auto en la clase Carrera? ¿Se puede crear Sedan como subclase Auto?

```
package taller2;
public class Auto {
    public String marca;
    Auto() {
        System.out.println("Constructor
de Auto");
    }
    void arrancar() {
        System.out.println("arrancar");
    }
}
```

```
public class Sedan extends Auto {}
```

```
import taller2.*;
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();
    }
}
```



# Especificadores de acceso en Métodos, Variables y Constructores

## Control de acceso package y herencia

¿Se pueden crear instancias de Auto en la clase Carrera? ¿Se puede crear Sedan como subclase Auto?

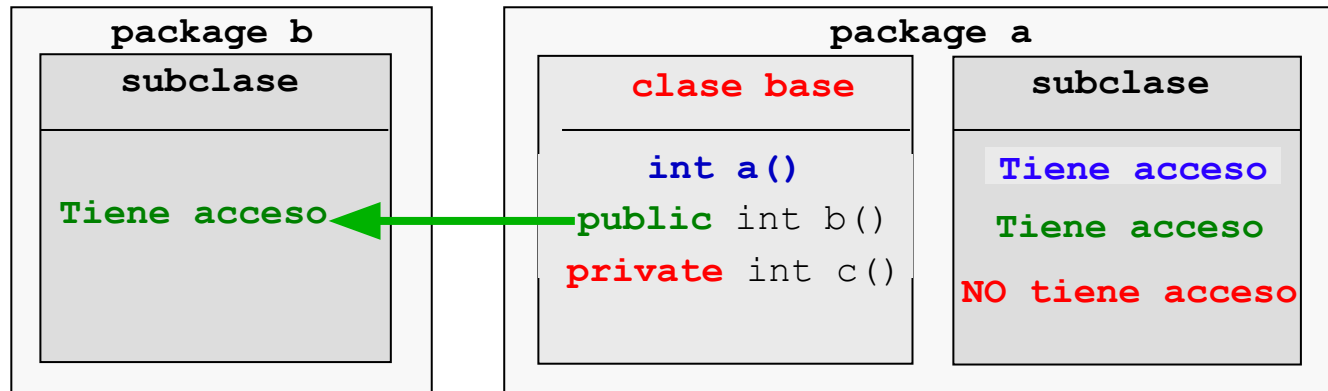
- Sólo es posible **crear objetos Auto** desde las clases del paquete **taller2**.
- **arrancar()** es un método **privado del paquete taller2**: no está disponible en otros paquetes y **no lo heredan subclases declaradas en otros paquetes**. Un auto que no se puede arrancar, ¿raro?
- El especificador **package** **impacta** sobre la **herencia** de la clase Auto siendo **sólo** posible **crear subclases de Auto en el paquete taller2**.
- El especificador **package** **inhibe el mecanismo de herencia** desde afuera del paquete. ¿Por qué?



# Especificadores de acceso en **protected**

El control de acceso **protected** está relacionado con la herencia:

- Las **subclases** definidas en **paquetes diferentes** que la **superclase** tienen acceso sólo a los miembros definidos **public** de la superclase.
- Las **subclases** definidas en el **mismo paquete** que la **superclase** tienen acceso a todos los miembros declarados **public** y **package**.



Entonces..... la subclase “no hereda” todos los métodos de la superclase.

¿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 todas las **subclases**, no importa el paquete al que pertenece la subclase. Además **protected** provee acceso **package**.

Las definiciones de acceso **protected** forman parte de la **interface pública**.

# Especificadores de acceso en Métodos, Variables y Constructores

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;
}
```

```
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) {
            ...
        }
    }
}
```

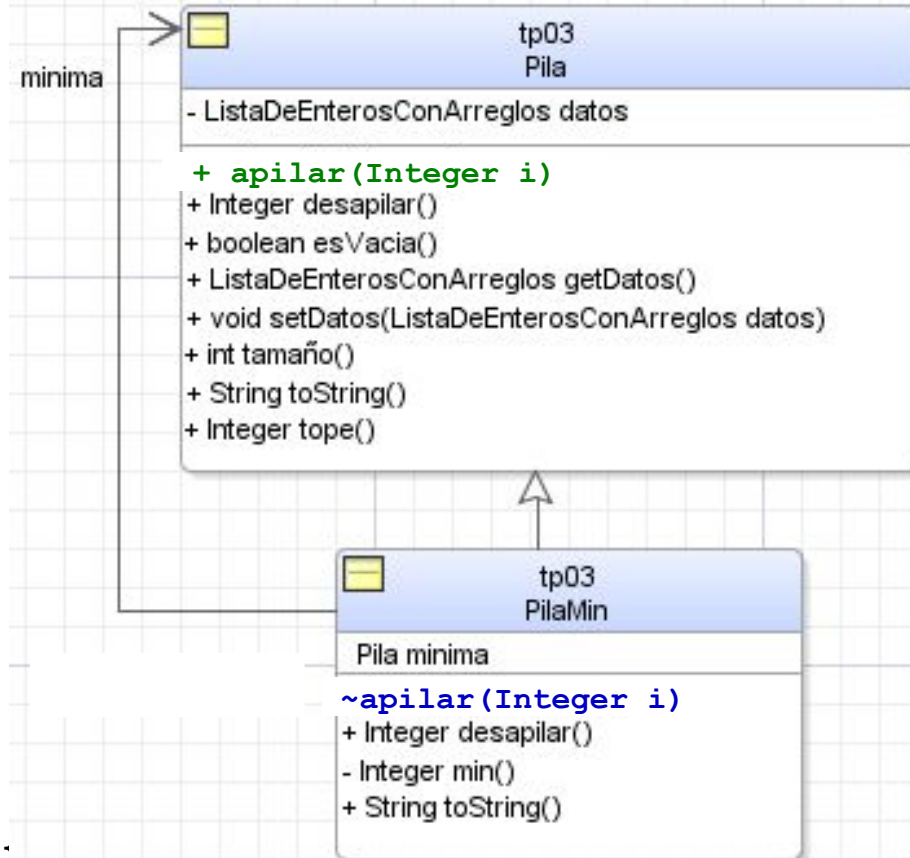
El método **getCurrent()** está definido en la clase **Lista**, entonces también **debería formar parte de la interface pública de cualquier subclase de Lista**. Pero como tiene acceso **package** y la clase **ListaPosicional** no está en el mismo paquete que la clase **Lista**, **getCurrent()** no está disponible en **ListaPosicional**.

Si **getCurrent()** lo definimos **protected** es accesible para cualquier subclase de **Lista** pero no es **public** !! Las subclases pueden usarlo y sobreescribirlo

# Especificadores de acceso y herencia

`apilar()` es visible cuando se aplica sobre objetos **Pila** y deja de serlo cuando se aplica sobre objetos de una subclase. Por lo tanto las subclases y las superclases dejan de tener una interface de comportamiento común.

```
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()` y `#apilar()` no son válidos.

# Especificadores de acceso

## Síntesis

	Especificador de Acceso	¿Qué acceso habilita?	¿Qué implicancias tiene?
Interface Pública			
Implementación			

# Tipos enumerativos: ¿qué son? ¿para qué usarlos? ¿cómo se definen?

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

**package taller2;**

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

- Los **valores** son **constantes públicas de clase** (public static final).
- Los **valores** de un **tipo enumerativo** se llaman **valores enumerados** y también **constantes enum**.
- A una **variable** de tipo **Estados** se le puede asignar uno de los **4 valores definidos o null**. Se hace referencia los valores del tipo **Estados** de la siguiente manera: **Estados.CONECTANDO**, **Estados.LEYENDO**, **Estados.LISTO** y **Estados.ERROR**.
- El **tipo enumerativo** es una **clase** y sus **valores** son **instancias** de dicha **clase**.
- Los **tipos enumerativos** garantizan **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 a 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 las clases **java.util.EnumMap** y **java.util.EnumSet**.

**java.util.EnumMap**: es una **implementación especializada de un Map** que requiere como clave un tipo Enumerativo.

**java.util.EnumSet**: es una **implementación especializada de un Set** que requiere valores de tipo Enumerativo.

Ambas **estructuras de datos** están **optimizadas para tipos Enumerativos**.

# Tipos enumerativos

## Ejemplos

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.

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

```
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");  
    }  
}
```

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

# Características de los Tipos Enum

Cuando se crea un tipo enumerativo el compilador crea una clase que es subclase de **java.lang.Enum**. No es posible extender la clase **Enum** para crear un tipo enumerativo propio. La única manera de crear un tipo enumerativo es usando la palabra clave **enum**.

```
public abstract class Enum<E> extends Enum<E>> extends Object implements Comparable<E>, Serializable
```

Constructors		
Modifier	Constructor	Description
protected	<code>Enum(String name, int ordinal)</code>	Sole constructor.

All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type		Method	
protected final <code>Object</code>		<code>clone()</code>	
final <code>int</code>		<u><code>compareTo(E o)</code></u>	
final <code>Optional&lt;Enum.EnumDesc&lt;E&gt;&gt;</code>		<code>describeConstable()</code>	
final <code>boolean</code>		<u><code>equals(Object other)</code></u>	
protected final <code>void</code>		<code>finalize()</code>	
final <code>Class&lt;E&gt;</code>		<code>getDeclaringClass()</code>	
final <code>int</code>		<code>hashCode()</code>	
final <code>String</code>		<u><code>name()</code></u>	
final <code>int</code>		<u><code>ordinal()</code></u>	
<code>String</code>		<u><code>toString()</code></u>	
static <code>&lt;T extends Enum&lt;T&gt;&gt;</code>		<code>valueOf(Class&lt;T&gt; enumClass, String name)</code>	

Los tipos enumerativos no tienen constructores públicos. Las únicas instancias son las declaradas por el tipo **enum**.

- Los tipos enumerativos implementan la interface **java.lang.Comparable** y **java.io.Serializable**.
- El método **compareTo()** establece un orden entre los valores enumerados de acuerdo al orden en que aparecen en la declaración del **enum**. Es **final**.
- Es seguro comparar valores enumerativos usando el operador **==** en lugar de el método **equals()** dado que el conjunto de valores posible es limitado. El método **equals()** internamente usa el operador **=** y además es **final**.
- El método **name()** devuelve un `String` con el nombre de la constante **enum**. Es **final**.
- El método **ordinal()** devuelve un entero con la posición del **enum** según está declarado. Es **final**.
- El método **toString()** puede **sobreescribirse**. Por defecto retorna el nombre de la instancia del enumerativo.

<sup>T</sup> No es posible extender un tipo enumerativo, son implícitamente **final**. El compilador define **final** a la clase que lo soporta.



# Tipos enumerativos

## Sobreescritura del método toString

```
package taller2;
```

```
public enum Señales {  
    VERDE, ROJO, AMARILLO;
```

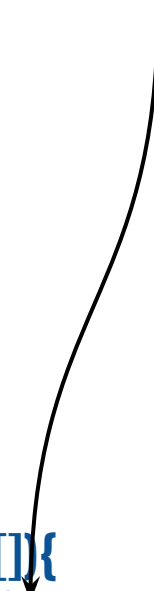
Cuando se crea un tipo **enum** el compilador automáticamente agrega el método **values()** que retorna un arreglo con todos los valores del enum.

```
    public String toString() {  
        String id = name(); Recupera el nombre de la instancia  
        String minuscula = id.substring(1).toLowerCase();  
        return id.charAt(0) + minuscula;  
    }  
}
```

Verde  
Rojo  
Amarillo

```
package taller2;  
import static java.lang.System.out;
```

```
public class PruebaSeñales {  
    public static void main (String args[]){  
        for (Señales s: Señales.values())  
            out.println(s);  
    }  
}
```



# Tipos enumerativos

## Con variables y métodos de instancia

```
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 **objeto Prefijo** se declara con **valores** para las variables de instancia **abrev** (abreviatura) y factor **multiplicador**

- Se debe proveer un constructor.
- El constructor tiene acceso privado o privado del paquete.
- El constructor automáticamente crea todas las instancias del tipo y no puede invocarse.
- El constructor es único, no hay sobrecarga de constructores.
- No se definen constructores públicos.

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

# Tipos enumerativos

## Con variables y métodos de instancia

```
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 el método **values()** que retorna un **arreglo con todos los valores del enum**.

# Patrones de diseño: ¿qué son?

## ¿para qué usarlos?

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**, son **plantillas** que se pueden 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.

Los patrones de diseño pueden clasificarse en las siguientes categorías:

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

# El Patrón 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**.

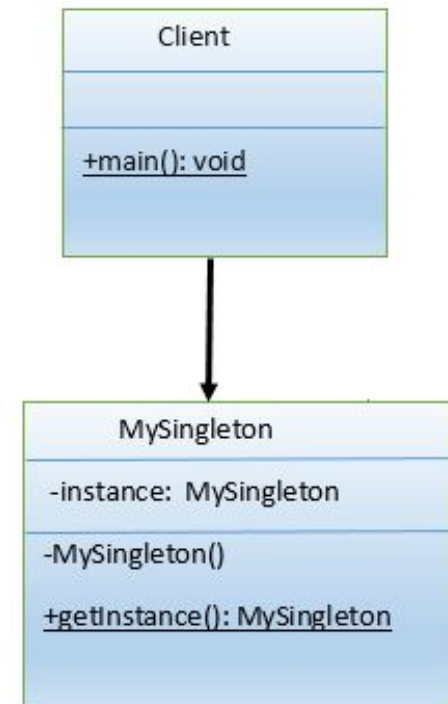
La **clase singleton** debe proveer un **acceso público a esa instancia de la clase**.

El patrón singleton se usa en muchas soluciones como drivers, pool de threads, logging, etc. **También en nuestro Proyecto de cátedra.**

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 dicha instancia.

**Existen diferentes alternativas  
para implementarlo**





# El Patrón Singleton en JAVA

## Inicialización temprana

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;
    }
}
```

# El Patrón Singleton en JAVA

## Inicialización temprana

Es similar al anterior, excepto que la instancia de la clase es creada en el bloque estático. También se crea al 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;
    }
}
```

# El Patrón Singleton en JAVA

## Inicialización lazy o perezosa

Se crea la instancia al invocar al 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;
    }
}
```

# El Patrón Singleton en JAVA

## Con tipos enumerativos

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

```
package patrones;

public enum EnumSingleton {
    INSTANCE;
}
```

```
package patrones;

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

La **instancia enum** podría **contener** variables de **instancia**, dependiendo el objeto que necesitemos representar.

# Proyecto “Billetera de Cripto”

**¿Dónde usarías enumerativos? y ¿singleton?**