

The Java logo is visible in the background, rendered in a faded, pixelated style. It consists of a blue sphere with a red and yellow swirl around it, and the word "JAVA" in a stylized font below it.

# Anotaciones & Reflection

JAVA

# Anotaciones

## ¿Qué son?

- A partir de la versión 5, JAVA incorpora un tipo de dato nuevo llamado anotaciones. Las anotaciones son un tipo especial de interface que se utiliza para “anotar” declaraciones.
- Las anotaciones son **metadatos** que proveen a nuestros programas de información extra que es testeada y verificada en compilación. **Son metadatos del código**. Proporcionan información para describir nuestro programa que no puede expresarse en código JAVA.
- Las anotaciones pueden aplicarse a paquetes, clases, interfaces, enumerativos, variables, parámetros, métodos, constructores, en general a diferentes elementos de nuestros programas.
- El objetivo de las anotaciones es facilitar la combinación de **metadatos** con código fuente (JAVA), en lugar de mantenerlos en archivos separados.
- Las anotaciones pueden ser leídas desde el código fuente, desde archivos .class y usando el mecanismo de *reflection*.
- El código anotado no es afectado directamente por sus anotaciones. Éstas proveen información para otros sistemas.
- El desarrollo basado en anotaciones alienta al estilo de programación declarativa, donde el programador dice lo qué debe hacerse y las herramientas lo hacen automáticamente (producen el código que lo hace).

# La anotación predefinida: @Override

Una anotación es una instancia de un tipo anotación y asocia metadatos con un elemento de la aplicación. Se expresa en el código fuente con el prefijo @.

El compilador JAVA soporta los siguientes tipos de anotaciones definidas en `java.lang`:

## @Override

```
public class Subclase extends Base {  
    @Override  
    public void m() {  
        // TODO Auto-generated method stub  
        super.m();  
    }  
}
```

```
public class Subclase extends Base {  
    @Override  
    public void m() {  
        // TODO Auto-generated method stub  
        super.m();  
    }  
    @Override  
    public void x() {  
        // TODO Auto-generated method stub  
    }  
}
```

Error de Compilación: The method x() of type Subclase must override or implement a super type method

**Declaramos que sobreescribimos el método m() definido en la superclase, Base.** El compilador examina la superclase (Base) y garantiza que m() está definido.

```
public class Base {  
  
    public void m() {  
        // TODO Auto-generated method stub  
    }  
}
```

Las anotaciones **@Override** son útiles para indicar que un método de una subclase sobreescribe un método de la superclase y no lo sobrecarga por ejemplo.

# @Override

La anotación `@Override` en la declaración de un método que sobrescribe una declaración de un supertipo (clase o interface), permite que el compilador nos ayude a evitar errores.

Las IDEs proveen chequeos automáticos de código conocidos como “inspección de código”. Si se habilita la “inspección de código”, el IDE generará *warnings* si un método sobrescribe un método de la superclase y no tiene la anotación `@Override`.

Si se usa la anotación `@Override` consistentemente, los *warnings de los chequeos del IDE* nos alertarán de sobrescrituras no intencionales. Estos *warnings* complementan los mensajes de error del compilador, los cuáles nos alertan de fallas al sobrescribir. Entre los chequeos del IDE y el compilador, se garantiza que estamos sobrescribiendo los métodos en el lugar que deseamos hacerlo.

# La anotación predefinida: **@Deprecated**

La anotación **@Deprecated** es útil para indicar que el elemento marcado dejará de usarse. El compilador advierte (o emite un error) cuando un elemento anotado como “deprecated” es accedido por un código que está en uso. Puede aplicarse a métodos, clases y propiedades.

## **@Deprecated**

```
public class Base {  
    @Deprecated  
    public void s(){  
        System.out.println("Hola");  
    }  
    public void m() {  
        System.out.println("Método m()");  
    }  
}
```

```
public class UseBase {  
  
    public static void main(String[] args) {  
        new Base().s();  
    }  
}
```

Indica que el elemento marcado con la anotación **@Deprecated** es **obsoleto** y se dejará de usar. El compilador genera una advertencia o un error (de acuerdo a la configuración del IDE) cada vez que un programa lo usa, el objetivo es desalentar su uso.

The method s() from type Base is deprecated

# La anotación predefinida: **@SuppressWarnings**

La anotación **@SuppressWarnings** es útil para eliminar advertencias del compilador a ciertas partes del programa. Las advertencias que pueden suprimirse varían entre las diferentes IDEs, los más comunes son: "deprecation", "unchecked", "unused".

## @ SuppressWarnings

```
import java.util.Date;
public class SuppressWarningsExample {
    @SuppressWarnings(value={"deprecation"})
    public static void main(String[] args) {
        Date date = new Date(2016, 9, 30);
        System.out.println("date = " + date);
    }
}
```

Si comentamos la anotación **SuppressWarnings**, obtendríamos este resultado en ejecución:

Warning(10,25): constructor Date(int, int, int) is deprecated

**Suprime determinadas advertencias de compilación** en el elemento anotado y sus subelementos.

Advertencias "unchecked" ocurren cuando se mezcla código que usa tipos genéricos con código que no lo usa.

Advertencias "unused" ocurren cuando no se usa un método o una variable.

Lista de advertencias que pueden suprimirse en Eclipse:

[http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.jdt.doc.user/tasks/task-suppress\\_warnings.htm](http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.jdt.doc.user/tasks/task-suppress_warnings.htm)

# Declarar Anotaciones

Declarar una anotación es similar a declarar una interface: el carácter @ precede a la palabra clave **interface** (@ = "AT" Annotation Type). Las anotaciones se compilan a archivos .class de la misma manera que las interfaces, clases y tipos enumerativos.

```
public @interface SolicitudDeMejora{  
    int id();  
    String resumen();  
    String ingeniero() default "[no asignado]";  
    String fecha() default "[no implementado]";  
}
```

## Definición de la ANOTACIÓN SolicitudDeMejora

El cuerpo de la declaración de las anotaciones contiene **declaraciones de elementos** que permiten especificar valores para las anotaciones (parámetros). Los programas o herramientas usarán los valores de los elementos o parámetros para procesar la anotación.

Una vez que definimos un tipo de anotación la podemos usar para anotar declaraciones:

```
@ SolicitudDeMejora(  
    id = 2868724,  
    resumen= "Habilitado para viajar en el Tiempo",  
    ingeniero = "Mr. Peabody",  
    fecha = "4/12/3008" )  
public static void viajarEnElTiempo(Date destino) { }
```

**El método viajarEnElTiempo() está  
anotado con la anotación  
SolicitudDeMejora**

- El código precedente no hace nada por sí mismo, el compilador verifica la existencia del tipo de anotación @SolicitudDeMejora.
- Las anotaciones consisten de un @ seguido por un tipo de anotación y una lista entre paréntesis de pares elemento-valor.
- Los valores de los elementos deben ser constantes predefinidas en compilación.

# Anotaciones con un único elemento

Para las anotaciones que tienen un único elemento podemos usar el nombre **value** y de esta manera cuando anotamos un elemento omitimos el nombre del elemento seguido del signo igual (=).

```
/*Asocia un copyright al elemento anotado*/  
public @interface Copyright {  
    String value();  
}
```

## Definición de la ANOTACIÓN Copyright

El elemento de nombre **value** es el único que no requiere el uso de la sintaxis elemento-valor para anotar declaraciones, alcanza con especificar el valor entre paréntesis:

```
@Copyright("2018 Sistema de Auditoria de Juegos de Azar")  
public class MaquinasDeJuego{ }
```

**La clase MaquinasDeJuego está anotada con la anotación Copyright**

Esta sintaxis abreviada sólo puede usarse cuando la anotación tiene un único elemento y de nombre **value**.

El nombre **value** puede aplicarse a cualquier elemento y de esta manera cuando anotamos una declaración omitimos el nombre del elemento seguido del signo igual (=).



# Anotaciones *Marker*

Las **Anotaciones *Markers*** NO contienen elementos, son útiles para marcar elementos de una aplicación con algún propósito.

## Definición de las anotaciones *markers* “InProgress” y “Test”:

```
/*Indica que el elemento anotado está sujeto a cambios, es una versión preliminar */  
public @interface InProgress { }
```

Declaración completa de la anotación @Test:

```
import java.lang.annotation.*;  
/**  
 * Indica que el método anotado es un método de testeo.  
 * Se usa sólo en métodos estáticos y sin argumentos.  
 */  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Test {  
}
```

↓  
**¿Puede el compilador garantizarlo?**

# Anotaciones *Marker*

## Uso de las anotaciones @InProgress y @Test:

```
@InProgress public class ViajeEnElTiempo{ }
```

```
public class Ejemplo {  
    @Test public static void m1() { }  
    public static void m2() { }  
    @Test public static void m3() { throw new RuntimeException("Boom");}  
    public static void m4() { }  
    @Test public void m5() { }  
    public static void m6() { }  
    @Test public static void m7() {throw new RuntimeException("Crash");}  
    public static void m8() { }  
}
```

Las anotaciones *marker* simplemente “marcan” el elemento anotado.

Si el programador escribe mal **Test** o la aplica a un elemento que no es la declaración de un método, el programa no compilará.

Se puede omitir el paréntesis cuando usamos una anotación *marker*.

La anotación **Test** no tienen efecto directo sobre la semántica de la clase **Ejemplo**, sólo sirve para proveer información a herramientas de testing.

# Anotaciones

## ¿Para qué se usan?

Las anotaciones no contienen ningún tipo de lógica y no afectan el código que anotan. Por ejemplo las anotaciones **Inprogress** y **Test** no tienen efecto directo sobre la semántica de las clases que las usan como **ViajeEnElTiempo** y **Ejemplo**.

Las anotaciones son usadas por los **procesadores o consumidores de anotaciones o parsers de anotaciones**, que son aplicaciones o sistemas que hacen uso del código anotado y ejecutan diferentes acciones dependiendo de la información suministrada.

Ejemplos de procesadores de anotaciones: la herramienta **JUnit**, que lee y analiza las clases de testeo anotadas y decide por ej en qué orden serán ejecutadas las unidades de testeo. **Hibernate** usa anotaciones para mapear objetos a tablas de la BD.

Las anotaciones pueden ser procesadas en compilación por herramientas de pre-compilación o en ejecución usando *reflection*.

Los procesadores de anotaciones usan *reflection* para leer y analizar el código anotado en ejecución.

Las anotaciones son compiladas a `.class` y recuperadas en ejecución y usadas por los procesadores de anotaciones que hacen uso de la información suministrada.

En el caso de la anotación **@Test** podríamos pensar en un **testeador** que procese los métodos anotados.

# Testeador: RunTest

```
package anotaciones;
import java.lang.reflect.*;
public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " falló: " + exc);
                } catch (Exception exc) {
                    System.out.println("INVALIDO @Test: " + m);
                }
            }
        }
        System.out.printf("Pasó: %d, Falló: %d\n", passed, tests - passed);
    }
}
```

## Reflection

Es la capacidad de inspeccionar, analizar y modificar las propiedades o el comportamiento de un objeto en ejecución.

¿Cuál es el resultado?

**java RunTest Ejemplo**

# Anotaciones & Reflection

La facilidad de reflection: el paquete `java.lang.reflect` ofrece acceso programático a información de las clases cargadas en ejecución. A partir de un objeto `Class`, es posible obtener constructores, métodos, variables, etc. Es posible crear instancias, invocar métodos, acceder a variables de instancia.

La API de Reflection fue extendida en JAVA 5 para soporte de lectura de código de anotaciones en ejecución.

Java 5 soporta una nueva interface, `java.lang.reflect.AnnotatedElement`, que es implementada por las clases: `Class`, `Constructor`, `Field`, `Method` y `Package`. Esta interface nos provee acceso a anotaciones retenidas en ejecución mediante los métodos: `getAnnotation()`, `getAnnotations()` y `isAnnotationPresent()`.

# La Anotación @ExceptionTest

```
package anotaciones;
import java.lang.annotation.*;
/**
 * Indica que el método anotado es un método de testeo
 * que se espera dispare la excepción consignada como parámetro
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
```

## ¿Cómo la usamos?

```
package anotaciones;
public class Ejemplo2 {
    @ExceptionTest(ArithmeticException.class)
    public static void m1() {
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() {
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { }
```

El tipo del elemento value() es un objeto Class para cualquier subclase de Exception

# Testeador de Excepciones: RunTest2

```
package anotaciones;
import java.lang.reflect.*;
public class RunTest2 {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(ExceptionTest.class)) {
                tests++;
                try {
                    m.invoke(null);
                    System.out.printf("Test %s Falló: no hay excepciones%n", m);
                } catch (InvocationTargetException wrappedEx) {
                    Throwable exc = wrappedEx.getCause();
                    Class<? extends Exception> excType = m.getAnnotation(ExceptionTest.class).value();
                    if (excType.isInstance(exc)) {
                        passed++;
                    } else {
                        System.out.printf( "Test %s Falló: se esperaba %s, ocurrió %s%n", m, excType.getName(), exc);
                    }
                } catch (Exception exc) {
                    System.out.println("INVALIDO @Test: " + m);
                }
            }
        }
        System.out.printf("Pasó: %d, Falló: %d%n", passed, tests - passed);
    }
}
```

¿Cuál es el resultado?

**java RunTest2 anotaciones.Ejemplo2**

# Anotaciones usando meta-anotaciones

La declaración de anotaciones requiere de **meta-anotaciones** que indican **cómo será usada la anotación**


Declaración de la anotación **@CasoDeUso**.

```
package anotaciones;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME) } Meta-anotaciones
public @interface CasoDeUso {

    public int id ( ) ;
    public String descripcion ( ) default "no hay descripción";

}
```



**@Target:** indica dónde se aplica la anotación (métodos, clases, variables de instancia, variables locales, paquetes, constructores, etc).

**@Retention:** indica dónde están disponibles las anotaciones (código fuente, archivos .class o en ejecución). Es el tiempo de vida de las anotaciones.

En nuestro ej. la anotación se aplica a métodos y el **RetentionPolicy.RUNTIME** indica que los declaraciones anotadas con **CasoDeUso** son retenidas por la JVM y se pueden leer vía *reflection* en ejecución.

- La declaración de los elementos permite declarar valores **predeterminados**.
- Las declaraciones de elementos no tienen parámetros ni cláusulas *throws*.
- Las anotaciones que no contienen elementos se llaman *markers*.

**Un programa o una herramienta usa los valores de los elementos para procesar las anotaciones.**



# Uso de la Anotación CasoDeUso

```
package anotaciones;  
import java.util.List;
```

## 3 métodos anotados

```
public class UtilitarioPassw {  
    @CasoDeUso(id = 47, descripcion = "Passw deben contener al menos un número")  
    public boolean validarPassw(String password) {  
        return (password.matches("\\w*\\d\\w*"));  
    }  
    @CasoDeUso(id = 48)  
    public String encriptarPass(String password) {  
        return new StringBuilder (password).reverse().toString() ;  
    }  
    @CasoDeUso(id = 49, descripcion = "Nuevas passw no pueden ser iguales a otras ya usadas ")  
    public boolean chequearPorNuevasPassw(List<String> prevPasswords, String password) {  
        return !prevPasswords.contains(password);  
    }  
}
```

Los valores de las anotaciones son expresados entre paréntesis como pares elemento-valor después de la declaración de @CasoDeUso.

Para la anotación del método encriptarPass() se omite el valor del elemento *descripcion*. Se usará el valor definido en @interface CasoDeUso

– Los métodos validarPassw(), encriptarPass() y chequearPorNuevasPassw() están anotados con @CasoDeUso.

– Las anotaciones se usan en combinación con otros modificadores como public, static, final. Por convención los preceden.

# RastreadorDeCasosDeUso

Rastreador de casos de uso de un proyecto:

- 1) los programadores anotan los métodos que cumplen los requerimientos de cada caso de uso.
- 2) el líder del proyecto usa el rastreador para conocer el grado de avance del proyecto contando la cantidad de casos de uso implementados.
- 3) los desarrolladores que mantienen el proyecto fácilmente pueden encontrar los casos de uso para actualizar o depurar reglas de negocio.

```
package anotaciones;
import java.lang.reflect.*;
import java.util.*;

public class RastreadorDeCasosDeUso {
    public static void rastrearCasosDeUso(List<Integer> casosDeUso, Class<?> cl) {
        for (Method m : cl.getDeclaredMethods() ) {
            CasoDeUso uc = m.getAnnotation(CasoDeUso.class) ;
            if (uc != null) {
                System.out.println("Caso de Uso encontrado:" + uc.id() + " " + uc.descripcion());
                casosDeUso.remove(new Integer(uc.id()));
            }
        }
        for (int i : casosDeUso)
            System.out.println("Advertencia: Falta el caso de uso-" + i);
    }

    public static void main(String[] args) {
        List<Integer> casosDeUso = new ArrayList<Integer>();
        Collections.addAll(casosDeUso, 47, 48, 49, 50);
        rastrearCasosDeUso(casosDeUso, anotaciones.UtilitarioPassw.class);
    }
}
```

Lista los casos de uso completados y localiza los faltantes

Métodos de la API Reflexion

¿Cuál es la salida?

Caso de Uso encontrado:47 Passw deben contener al menos un número

Caso de Uso encontrado:48 no hay descripción

Caso de Uso encontrado:49 Nuevas passw no pueden ser iguales a otras ya usadas

Advertencia: Falta el caso de uso-50



# Meta-Anotaciones

Java provee 4 meta-annotaciones. Las meta-annotaciones se usan para anotar anotaciones

<b>@Target</b>	<p>Indica a qué elementos se le aplica la anotación. Los valores son los definidos en el enumerativo <b>ElementType</b>:</p> <p><b>ElementType.ANNOTATION_TYPE</b>: se aplica solamente a anotaciones</p> <p><b>ElementType.TYPE</b>: se aplica a la declaración de clases, interfaces, anotaciones y enumerativos.</p> <p><b>ElementType.PACKAGE</b>: se aplica a la declaración de paquetes.</p> <p><b>ElementType.CONSTRUCTOR</b>: se aplica a un constructor</p> <p><b>ElementType.FIELD</b>: se aplica a la declaración de una propiedad (incluye constantes <b>enum</b>)</p> <p><b>ElementType.LOCAL_VARIABLE</b>: se aplica a la declaración de variables locales</p> <p><b>ElementType.METHOD</b>: se aplica a la declaración de métodos</p> <p><b>ElementType.PARAMETER</b>: se aplica a los parámetros de un método.</p>
<b>@Retention</b>	<p>Indica dónde y cuánto tiempo se mantiene la información de las anotaciones. Los valores son los definidos en el enumerativo <b>RetentionPolicy</b>:</p> <p><b>RetentionPolicy.SOURCE</b>: son retenidas en el código fuente y luego descartadas por el compilador.</p> <p><b>RetentionPolicy.CLASS</b>: son retenidas por el compilador en tiempo de compilación e ignoradas por la JVM. Es el valor de defecto.</p> <p><b>RetentionPolicy.RUNTIME</b>: son retenidas por la JVM en ejecución y pueden ser leídas mediante el mecanismo de reflexión.</p>
<b>@Documented</b>	<p>Indica que la anotación se incluye en la documentación generada por el <b>javadoc</b>. Por defecto, las anotaciones no se incluyen en el javadoc.</p>
<b>@Inherited</b>	<p>Indica que la anotación es heredada automáticamente por todas las subclases de la clase anotada. Por defecto, las anotaciones no son heredadas por las subclases.</p>

# Meta-Anotación: @Retention

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

En estos ejemplos el compilador (javac) es la herramienta que procesa la anotación. La anotaciones luego de ser procesadas son descartadas por el compilador. Aparecen sólo en el código fuente.

```
public class Subclass extends Base {
    @Override
    public void m() {
        // TODO Auto-generated method stub
        super.m();
    }
}
```

```
import java.util.Date;
public class SuppressWarningsExample {
    @SuppressWarnings(value={"deprecation"})
    public static void main(String[] args) {
        Date date = new Date(2008, 9, 30);
        System.out.println("date = " + date);
    }
}
```

# Meta-Anotación: @Documented

```
package anotaciones;
import java.lang.annotation.Documented;
@Documented
public @interface Preambulo {
    String autor();
    int version() default 1;
    String fechaUltimaRevision();
    String[] revisores();
}
```

```
package anotaciones;
import java.lang.annotation.Documented;
@Documented
public @interface InProgress { }
```

```
import java.util.EnumMap;
import anotaciones.InProgress;
import anotaciones.Preambulo;
@InProgress
@Preambulo (autor = "Claudia",
    fechaUltimaRevision = "17/11/2009",
    revisores = { "Jorge, Pablo" })
public class TestEnumHash {
    //Código de la clase TestEnumHash
}
```

Las anotaciones pueden ser usadas para documentar.

Para que la información especificada en **@Preambulo** y **@InProgress** aparezca en la documentación generada por el **javadoc**, es preciso anotar la definición de estas anotaciones con la anotación **@Documented**

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#)

Ejecutamos el javadoc con TestEnumHash.class y obtenemos el siguiente archivo html

```
@InProgress
@Preambulo(autor="Claudia",
           fechaUltimaRevision="17/11/2009",
           revisores="Jorge, Pablo")
public class TestEnumHash
extends java.lang.Object
```

Constructor Summary

[TestEnumHash](#) ()

# Meta-Anotaciones: @ Inherited

Por defecto las anotaciones no se heredan.

Si una anotación tiene la meta-anotación **@Inherited** entonces una clase anotada con dicha anotación causará que la anotación sea heredada por sus subclases.

```
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface InProgress { }
```

```
@InProgress
@TODO("Calcula el interés mensual")
public class CuentaBase{
    public void calcularIntereses(float amount, float rate) {
        // Sin terminar}
    }
```

```
public class CajaDeAhorro extends CuentaBase{
    //TODO
}
```

Las anotaciones meta-anotadas con **@Inherited** son heredadas por subclases de la clase anotada.

Las clases no heredan anotaciones de las interfaces que ellas implementan y los métodos no heredan anotaciones de los métodos que ellos sobrescriben

La anotación **@InProgress** se propaga en las subclases de **CuentaBase**.

# Los elementos de las Anotaciones

Los tipos permitidos para los elementos de las anotaciones son:

- Todos los tipos primitivos (int, long, byte, char, boolean, float, double)
- String
- Class
- Enumerativos
- Anotaciones
- Arreglos de cualquiera de los tipos mencionados

```
package anotaciones;
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulandoNull{
    public int id ( ) default -1;
    public String descripcion ( ) default "";
}
```

El compilador aplica ciertas restricciones sobre los valores predeterminados de los elementos de las anotaciones:

- Ningún elemento puede no especificar valores -> los elementos tienen valores predeterminados o valores provistos.
- Ninguno de los elementos (de tipo primitivo o no-primitivo) puede tomar el valor **null**.

Es importante tener esto en cuenta cuando escribimos un procesador de anotaciones y necesitamos detectar la ausencia o presencia de un elemento, ya que todos los elementos están presentes en una anotación.

Definiendo valores predeterminados como números negativos o strings vacíos nos permitirá simular la ausencia de elementos

# Ejemplos de Anotaciones más complejas

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
```

```
public @interface ExceptionTest2 {
    Class<? extends Exception>[] value();
}
```

¿Cómo las uso?

```
@ExceptionTest({ IndexOutOfBoundsException.class, NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<String>();
    // El método de testeo podrá disparar alguna de estas 2 excepciones
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}
```



# Ejemplos de Anotaciones más complejas

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface Revisiones {
    Revision[] value();
}

public @interface Revision {
    enum Concepto { EXCELENTE, SATISFACTORIO, INSATISFACTORIO };
    Concepto concepto();
    String revisor();
    String Comentario() default "";
}
```

## ¿Cómo las uso?

```
@Revisiones(
    {@Revision(concepto=Revision.Concepto.EXCELENTE, revisor="df"),

    @Revision (concepto=Revision.Concepto.INSATISFACTORIO, revisor="eg",
        comentario="Este método necesita la anotación @Override ")
    }
)
```

# Generación de Archivos Externos

Las anotaciones son especialmente útiles cuando trabajamos con frameworks Java que requieren de cierta información adicional que acompaña al código fuente. Tecnologías como web services, librerías de custom tags y herramientas mapeadoras objeto/relacional como **Hibernate** requieren de archivos descriptores XML que son externos al código Java. El programador después de escribir el código Java debe volver a especificar información como el nombre de la clase, del paquete, etc que ya existe en el código original. Trabajando con un archivo descriptor separado, tenemos que mantener 2 fuentes de información separadas sobre una clase y es frecuente que aparezcan problemas de sincronización entre ambas. Por otro lado el programador además de saber escribir código Java, debe saber cómo editar el archivo descriptor.

Consideremos el siguiente ejemplo: proveer un soporte básico de mapeo objeto-relacional para automatizar la creación de una tabla de la BD y guardarla en un JavaBean (clase Java). Usando anotaciones podemos mantener toda la información en el archivo fuente del JavaBean -> **necesitamos anotaciones para definir el nombre de la tabla asociada con el bean, las columnas y los tipos SQL que mapean con las propiedades del JAVABean**