Anotaciones & Reflection

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 utilizan 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**. Ofrecen información descriptiva del programa y que no puede expresarse en código JAVA.
- Las **anotaciones** pueden aplicarse a **paquetes, clases, interfaces, tipos 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 **qué debe hacerse** y las **herramientas lo hacen automáticamente** (producen el código que lo hace).

Anotaciones predefinidas: @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();
   }
}
```

Declaramos que se sobreescribe 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
   }
}
```

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

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

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

@Override

@Override es una anotación para el **compilador**.

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

Las **IDEs proveen**, además, **chequeos automáticos** de código conocidos como **"inspección de código"**. Si se habilita la "inspección de código", el **IDE genera *warnings*** si **un método sobreescribe 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 **sobreescrituras no intencionales.**

Los ***warnings* de los IDEs complementan** los **mensajes de error del compilador**: se garantiza que estamos sobreescribiendo los métodos en el lugar que deseamos hacerlo.

Anotaciones predefinidas: @Deprecated

La anotación **@Deprecated** es útil para indicar que el elemento marcado será reemplazado por otro en futuras versiones. 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** está **desaprobado** y se dejará de usar en futuras versiones.

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

Anotaciones predefinidas: @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, las 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**, obtendremos el siguiente mensaje de advertencia del compilador:

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

Se pueden suprimir varias advertencias:

@SuppressWarnings({"unchecked", "deprecation"})

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

@FunctionalInterface

La anotación **@FunctionalInterface** asegura que la **interface funcional define un solo método abstracto**. En el caso de que haya más de un método abstracto, el compilador emitirá el siguiente mensaje de error: **"Invalid @FunctionalInterface annotation"**.

@FunctionalInterface

```
package anotaciones;
@FunctionalInterface
public interface Square {
      int calculate(int x);
      int otro();
}
```

La **compilación falla** y el compilador emitirá el siguiente mensaje de error:

Invalid '@FunctionalInterface' annotation; Square is not a functional interface

Es una **indicación clara para los programadores** y para las **herramientas de desarrollo** que la **interface** está diseñada para ser utilizada como una **interface funcional**. Ayuda a documentar la intención del diseñador de la interfaz.

Se puede usar más fácilmente con expresiones lambda y referencias a métodos.

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 ingeniera() 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** o **parámetros** que permiten especificar valores para las anotaciones.
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",
ingeniera = “Elisa Bachofen",
fecha = "6/10/2024" )
public static void viajarEnElTiempo(Date destino) { }
```

El método viajarEnElTiempo() está anotado con la anotación SolicitudDeMejora

¿Saben quién fue Elisa Bachofen?

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 con un **único elemento** se puede 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("2022 Sistema de Auditoría 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 define 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 { }
```

La anotación **@Test** la usaremos para realizar **testeos** que se ejecutan **automáticamente**. Cuando un método falla se disparan excepciones.

```
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 tiene 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 ejemplo 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 y luego descartadas, o **en ejecución usando *reflection***.

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

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

En el caso de la anotación **@Test** podríamos pensar en un **testeador "de juguete"** que **analiza** una clase y **ejecuta** todos los métodos anotados con **@Test**. Lo vamos a llamar **RunTests**.

Testeador: RunTests

```
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);}}
```

¿Qué es Reflection?

Es la capacidad de **inspeccionar**, **analizar** y **modificar** código en **ejecución**.

Indica a RunTests qué método ejecutar

¿Cuál es el resultado?

java RunTests Ejemplo

Anotaciones & Reflection

La facilidad de reflection: el paquete **java.lang.reflect** ofrece **acceso** programático a información de las **clases cargadas en ejecución**.

El **punto de entrada** de reflection es el objeto **Class**: contiene métodos para recuperar información de 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**:

- la interface **java.lang.reflect.AnnotatedElement** implementada por las clases: Class, Constructor, Field, Method y Package, 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();
}
```

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

Las **pruebas que tienen éxito** sean las que arrojan **una excepción particular**, por eso esta anotación tiene parámetros.

```
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() { }
}
```

Testeador de Excepciones: RunTests2

```
package anotaciones;
import java.lang.reflect.*;
public class RunTests2 {
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);
    }
    }
```

¿La excepción disparada es la esperada?

Se obtiene el **valor** del **parámetro** de la anotación **ExceptionTest**

¿Cuál es el resultado?

java RunTest2 anotaciones.Ejemplo2

@ExpceptionTest extendido

Consideremos que **las pruebas que pasan** son las que **disparan** al **menos una excepción de un conjunto de excepciones.**

¿Cómo haríamos?

```
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();
}
```

El tipo del elemento **value() es un arreglo de objetos Class** para cualquier **subclase de Exception**

```
package anotaciones;
public class Ejemplo3 {
@ExceptionTest({ IndexOutOfBoundsException.class, NullPointerException.class})
public static void doublyBad() {
        List<String> list = new ArrayList<>();
        list.add(5, null);
}
```

Las 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)
public @interface CasoDeUso {
public int id ( ) ;
public String descripcion ( ) default "no hay descripción";
}
```

Metaanotaciones

@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. **Dónde se usan.**
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 @CasoDeUso

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

  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 usadas ")
   public boolean chequearPorNuevasPassw(List<String> prevPasswords, String password){
         return prevPasswords.contains(password);
    }
   }
```

3 métodos anotados

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

Lista los casos de uso completados y localiza los faltantes

```
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<>();
      Collections.addAll(casosDeUso, 47, 48, 49, 50);
      rastrearCasosDeUso(casosDeUso, anotaciones.UtilitarioPassw.class);
}

}
```

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.

¿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-anotaciones. Las meta-anotaciones se usan para anotar anotaciones

@Target	Indica a qué elementos se le aplica la anotación. Los valores son los definidos en el enumerativo **ElementType**: **ElementType.ANNOTATION_TYPE:** se aplica solamente a anotaciones **ElementType.TYPE:** se aplica a la declaración de clases, interfaces, anotaciones y enumerativos. **ElementType.PACKAGE:** se aplica a la declaración de paquetes. **ElementType.CONSTRUCTOR:** se aplica a un constructor **ElementType.FIELD:** se aplica a la declaración de un propiedad (incluye constantes **enum**) **ElementType.LOCAL_VARIABLE:** se aplica a la declaración de variables locales **ElementType.METHOD:** se aplica a la declaración de métodos **ElementType.PARAMETER:** se aplica a los parámetros de un método.
@Retention	Indica dónde y cuánto tiempo se mantiene la anotación. Los valores son los definidos en el enumerativo **RetentionPolicy**: **RetentionPolicy. SOURCE**: son retenidas en el código fuente y descartadas por el compilador. No están en los bycodes. Ej.: **@Override**, **@SuppressWarnings** **RetentionPolicy. CLASS**: son retenidas en compilación e ignoradas por la JVM. Se desechan durante la carga de la clase. Es el valor de defecto. Útil para procesar bytecodes. **RetentionPolicy. RUNTIME**: son retenidas por la JVM en ejecución y pueden ser leídas mediante el mecanismo de reflexión. No son descartadas.
@Documented	Indica que la anotación se incluye en la documentación generada por el **javadoc**. Por defecto, las anotaciones no se incluyen en el javadoc.
@Inherited	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.

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 Subclase 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();
}
```

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 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 = “6/10/2021",          revisores
= { "Isa, Diego" })
public class TestEnumHash {
    //Código de la clase TestEnumHash
}
```


MODULE PACKAGE CLASS USE TREE INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class TestEnumHash

java.lang.Object
anotaciones.TestEnumHash

```
@InProgress
@Preambulo(autor="Claudia",
           fechaUltimaRevision="6/10/2021",
           revisores={"Isa","Diego"})
public class TestEnumHash
extends Object
```

Constructor Summary

Constructors

Constructor	Description
TestEnumHash()	

Method Summary

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

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

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 sobreescriben

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**, dado que todos los elementos están presentes en una anotación.

Definir valores predeterminados como números negativos o strings vacíos nos permitirá **simular** la **ausencia de elementos.**

Ejemplos de Anotaciones 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.add(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 framewoks 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. Trabajar con un archivo descriptor separado, requiere mantener 2 fuentes de información separadas sobre una clase y es frecuente que aparezcan problemas de sincronización entre ambas. El programador además de saber 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 una clase Java. Usando anotaciones podemos mantener toda la información en el archivo fuente Java -> necesitamos anotaciones para definir el nombre de la tabla asociada con la clase, las columnas y los tipos SQL que mapean con las propiedades de la clase