1. Java Entorno Servidor

# 1.2 Anotaciones

Las anotaciones de Java nos permiten agregar información de metadatos en nuestro código fuente.

Son parte de Java desde la versión 5.

| Los metadatos son datos acerca de los datos.  Los metadatos van a agregar *flags o etiquetas* de información adicional sobre los datos reales.  En tiempo de ejecución, una librería o la JVM procesarán estos *flags o etiquetas* de información para tomar decisiones apropiadas sobre el contexto del programa. |
| --- |

## 1.2.1 Tipos de anotaciones

### 1.2.1.1 Anotaciones predefinidas

@Deprecated

La anotación @Deprecated indica que el elemento marcado (clase, método o campo) está en desuso (obsoleto) y ya no se debe usar.

El compilador genera una advertencia cada vez que un programa utiliza un método, una clase o un campo que se ha marcado con la anotación @Deprecated.

Cuando un elemento está en desuso, también debe documentarse con la etiqueta Javadoc @deprecated, como se muestra en el siguiente ejemplo. Tome nota de la diferencia entre mayúsculas y minúsculas con @Deprecated y @deprecated. @deprecated se utiliza con fines de documentación.

| */\*\*  \* @deprecated  \* razón por la que se depreca  \*/* **@Deprecated** **public** **void** unMetodo(){  *//*  } |
| --- |

@Override

Al sobrescribir un método en la clase hija o o implementar una interfaz, debemos usar esta anotación para marcar ese método.

Esto hace que el código sea legible y evita problemas de mantenimiento, como:

Al cambiar la firma del método de la clase principal, debe cambiar la firma en las clases secundarias (donde se usa esta anotación), de lo contrario, el compilador arrojaría un error de compilación.

Este tipo de cambios son difíciles de rastrear si no se ha utilizado esta anotación.

| public **class** **ClasePadre** {   public void unMetodo() {  **System**.out.println("Método de la clase padre");  } }   public **class** **ClaseHija** **extends** **ClasePadre** {   **@Override**  public void unMetodo() {  **System**.out.println("Método de la clase hija");  } } |
| --- |

@SuppressWarnings

Esta anotación indica al compilador que ignore advertencias específicas.

Por ejemplo, en el siguiente código, estoy llamando a un método en desuso (supongamos que el método deprecatedMethod() está marcado con la anotación @Deprecated), por lo que el compilador debería generar una advertencia.

Sin embargo, estoy usando la anotación @SuppressWarnings que eliminaría esa información de obsolescencia y su advertencia.

| **@SuppressWarnings**("deprecation")  **void** metodo() {  myObj.metodoDeprecado(); } |
| --- |

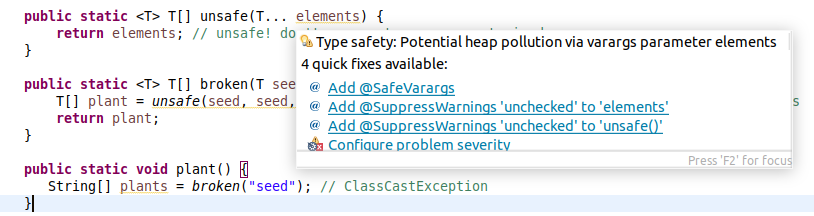
@SafeVarargs

Java 5 introdujo el concepto de *varargs*, o un parámetro de método de longitud variable, así como los tipos parametrizados (genéricos).

Combinar ambos va a causar problemas:

| **public** **static** <T> T[] unsafe(T... elements) {  **return** elements; *// Método inseguro, nunca devolver varargs con tipo parametrizado T* }  **public** **static** <T> T[] broken(T semilla) {  T[] plant = unsafe(semilla, semilla, semilla); *//Se rompe! Será un Object[] independientemente de T*  **return** plant; }  **public** **static** **void** **plant**() {  String[] plants = broken("semilla"); *// Va a la lanzar una ClassCastException* } |
| --- |

El uso de varargs junto con genéricos T… es difícil de verificar para un compilador, por lo que emite advertencias cada vez que se combinan los dos, como en el caso anterior:



Para eliminar esta advertencia, podemos agregar la anotación @SafeVarargs en métodos y constructores finales o estáticos.

@SafeVarargs es como @SupressWarnings que nos permite declarar que una advertencia particular del compilador es un falso positivo.

Una vez que nos aseguremos de que nuestras acciones son seguras, podemos agregar esta anotación:

| **public** **class** Machine<T> {  **private** List<T> versions = **new** ArrayList<>();   **@SafeVarargs**  **public** **final** **void** safe(T... toAdd) {  **for** (T **version** : toAdd) {  versions.add(**version**);  }  } } |
| --- |

@FunctionalInterface

| Una interfaz funcional es aquella interfaz que sólo dispone de un método abstracto |
| --- |

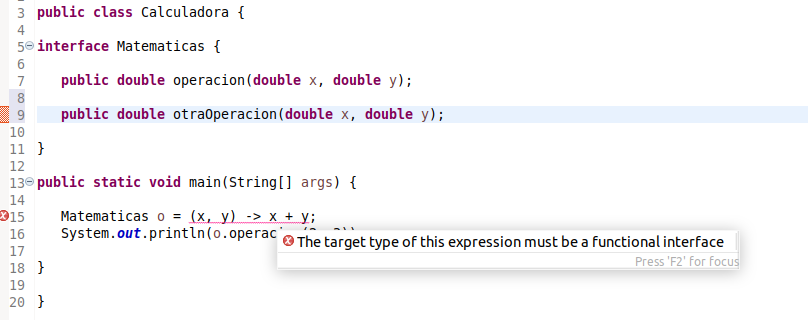
Recordemos que desde Java 8 las interfaces Java han evolucionado mucho y pueden disponer de métodos por defecto (default que se heredan automáticamente implementados en todas las clases que implementen la interfaz).

Por ejemplo:

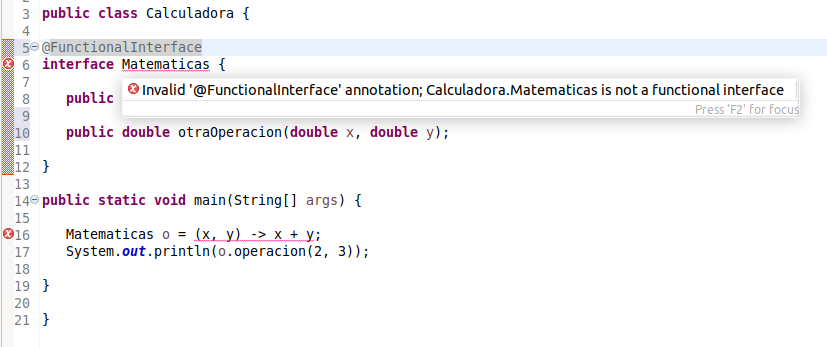
| **public** **class** **Calculadora** {    **interface** **Matematicas** {    **public** **double** **operacion**(**double** x, **double** y);    }    **public** **static** **void** **main**(String[] args) {    Matematicas o = (x, y) -> x + y; // lambda, o función anónima  System.**out**.println(o.operacion(2, 3));  o = (x, y) -> x \* y;  System.**out**.println(o.operacion(4, 6));  }   } |
| --- |

IMPORTANTE:

Si modificamos la interfaz y añadimos un nuevo método (abstracto) a la interfaz Matematicas → el programa no compila: el compilador nos avisa de que esa interfaz ha dejado de ser una interfaz funcional, recordemos que las interfaces funcionales únicamente disponen de un método abstracto, por lo que la asignación de la lambda o función anónima no se puede realizar.



Para evitar este tipo de problemas se usa la anotación @FunctionalInterface que obliga al desarrollador a solo incluir un método abstracto.



| **public** **class** **Calculadora** {    @FunctionalInterface  **interface** **Matematicas** {    **public** **double** **operacion**(**double** x, **double** y);    }    **public** **static** **void** **main**(String[] args) {    Matematicas o = (x, y) -> x + y;  System.**out**.println(o.operacion(2, 3));    }   } |
| --- |

1.2.1.2 Meta-anotaciones

| Meta-anotaciones: anotaciones sobre anotaciones.  Anotaciones que sirven para definir anotaciones y cómo deben comportarse. |
| --- |

Anotaciones para indicar al compilador cómo JVM debe tratar una nueva personalizada anotación.

@Retention

Esta anotación específica cómo se almacena la anotación marcada en el tiempo de ejecución de Java.

Puede estar limitado a:

* sólo al código fuente,
* incrustado en el archivo de clase generado,
* o también disponible en tiempo de ejecución a través de la API de *reflection*.

| **import** java.lang.**annotation**.Retention; **import** java.lang.**annotation**.RetentionPolicy;   *//@Retention(RetentionPolicy.CLASS)* **@Retention(RetentionPolicy.RUNTIME)** *//@Retention(RetentionPolicy.SOURCE)* **public** **@interface** MyCustomAnnotation {  *//some code* } |
| --- |

@Documented

Esta anotación indica que se debe incluir una nueva anotación en los documentos java generados por las herramientas de generación de documentos java (javadoc).

| **import java.lang.annotation.Documented;**   @Documented **public** @**interface** **MyCustomAnnotation** {  *//Some other code* } |
| --- |

@Target

Use la anotación @Target para indicar dónde estará el uso de una nueva anotación.

Después de especificarlo, sólo se podrá usar la nueva anotación en elementos dados.

| **import** java.lang.annotation.ElementType; **import** java.lang.annotation.**Target**;   @**Target**(**value** = {ElementType.**TYPE**, ElementType.METHOD, ElementType.CONSTRUCTOR,  ElementType.ANNOTATION\_TYPE, ElementType.FIELD, ElementType.LOCAL\_VARIABLE,  ElementType.PACKAGE, ElementType.**PARAMETER**}) **public** @**interface** MyCustomAnnotation {  //Some other code } |
| --- |

@Inherited

Cuando se aplica esta anotación a cualquier otra anotación en su definición indicará que la anotación se hereda en el esquema de herencia de clases de Java.

Por ejemplo, @MyCustomAnnotation lleva la meta-anotación @Inherited. Entonces, cuando @MyCustomAnnotation se aplica a cualquier clase MyParentClass, la anotación @MyCustomAnnotation también estará disponible para todas las clases hijas de MyParentClass.

IMPORTANTE:

Ten en cuenta que, de forma predeterminada, las anotaciones aplicadas en la clase principal no están disponibles para las clases secundarias O HIJAS.

| **import java.lang.annotation.Inherited;**   @Inherited **public** @**interface** **MyCustomAnnotation** {  *//Some other code* } |
| --- |

@Repeatable

De forma predeterminada, una anotación se puede aplicar a un elemento Java sólo una vez.

Pero si, por cualquier requisito, se debe aplicar una anotación más de una vez, la definición de la anotación debe usar la meta-anotación @Repeatable.

@Repeatable se agregó en la última versión de Java 8.

| **import** java.lang.annotation.Repeatable;  **@Repeatable**(Schedules.**class**) public **@interface** Schedule {  String dayOfMonth() **default** "first";  String dayOfWeek() **default** "Mon";  int hour() **default** 12; } |
| --- |

| **public** @**interface** **Schedules** {  Schedule[] **value**(); } |
| --- |

A partir de aquí se podría repetir la anotación:

| @Schedule(dayOfMonth="last") @Schedule(dayOfWeek="Fri", hour="23") public void doPeriodicCleanup() { ... } |
| --- |

## 1.2.1.3 Creación de anotaciones (custom annotations)

Las anotaciones se crean utilizando @interface, seguido del nombre de la anotación.

Una anotación también puede tener elementos que parecen métodos, pero no debemos proporcionar implementación para estos elementos.

Por ejemplo, en el siguiente código, tenemos cuatro elementos.



| **import** java.lang.**annotation**.Documented; **import** java.lang.**annotation**.ElementType; **import** java.lang.**annotation**.Inherited; **import** java.lang.**annotation**.Retention; **import** java.lang.**annotation**.RetentionPolicy; **import** java.lang.**annotation**.Target;   **@Documented** **@Target(ElementType.METHOD)** **@Inherited** **@Retention(RetentionPolicy.RUNTIME)** **public** **@interface** Estudiante{  int edad() **default** 18;  String nombre();  String dirección();  String curso() **default** "2º DAW"; } |
| --- |

Todas las anotaciones amplían la interfaz java.lang.annotation.Annotation. Las anotaciones no pueden incluir ninguna cláusula de extensión.

Los elementos que tienen valores predeterminados establecidos al crear anotaciones se pueden omitir al usar la anotación. Por ejemplo, si estoy aplicando la anotación anterior a una clase, lo haría así:

| **@Estudiante(  nombre="Diego.",  direccion="C/ Portugal, Nº 12" )** **public** **class** **MyClase** { ... } |
| --- |

Se pueden tener elementos de tipo array en las anotaciones de la siguiente forma:

| **package org.iesbelen.anotaciones;**  **import java.lang.annotation.Documented;**  **import java.lang.annotation.Inherited;**  **import java.lang.annotation.Retention;**  **import java.lang.annotation.RetentionPolicy;**  **import java.lang.annotation.ElementType;**  **import java.lang.annotation.Target;**  **@Documented**  **//@Target(ElementType.METHOD)**  **@Inherited**  **@Retention(RetentionPolicy.RUNTIME)**  **public @interface Estudiante {**  **int edad() default 18;**  **String nombre();**  **String direccion();**  **String curso() default "2º GS DAW";**    **String[] libros();**    **}** |
| --- |



| **package** org.iesbelen;  **import** org.iesbelen.anotaciones.Estudiante;  **@Estudiante(  nombre = "Diego",  direccion = "C/ Portugal ...",  libros = {"Aprendo Java", "Spring"}  )** **public** **class** **Clase** { } |
| --- |

Si vemos un poco más en detalle las meta-anotaciones:

@Documented

La anotación @Documented indica que los elementos que usan esta anotación deben ser documentados por JavaDoc.

| java.lang.**annotation**.Documented **@Documented** **public** **@interface** Estudiante {  //... } |
| --- |
|  |

| **@Estudiante** **public** **class** **MiClase**{   *//...* } |
| --- |

Si se genera el javadoc de MiClase, la anotación @Estudiante se incluirá en ella.

@Target

Esta meta-anotación especifica dónde se puede usar la anotación.

Por ejemplo, en el código siguiente se define el tipo @Target como METHOD, lo que significa que la anotación sólo se puede usar en métodos.

| **import** **java**.lang.annotation.ElementType; **import** **java**.lang.annotation.Target;  @**Target**({**ElementType**.METHOD}) **public** @**interface** MyCustomAnnotation {  } |
| --- |

| **public** **class** **MyClass** {  **@MyCustomAnnotation**  **public** **void** **myMethod**()  {  *//Doing something*  } } |
| --- |

| Si no define ningún tipo de elemento de destino mediante la meta-anotación @Target, significa que la anotación se puede aplicar a cualquier elemento. |
| --- |

Los tipos de elemento a los que se aplicaría la anotación serían:

ElementType.METHOD

ElementType.PACKAGE

ElementType.PARAMETER

ElementType.TYPE

ElementType.ANNOTATION\_TYPE

ElementType.CONSTRUCTOR

ElementType.LOCAL\_VARIABLE

ElementType.FIELD

@Inherited

La meta-anotación @Inherited indica que la anotación que se utiliza en una clase debe ser heredada por todas sus subclases.

| **java**.lang.annotation.Inherited  @**Inherited** public @interface MiAnotacion{  } |
| --- |

| **@**MiAnotacion **public** **class** **Padre** {   //...  } |
| --- |

| public **class** **Hijo** **extends** **Padre**{   // ...  } |
| --- |

Aquí, la clase Padre usa la anotación @MiAnotación que está marcada, a su vez, con la meta-anotación @inherited. Esto significa que la subclase Hijo hereda @MiAnotacion.

@Retention

Indica cuánto tiempo se conservarán las anotaciones con el tipo anotado.

* RetentionPolicy.RUNTIME:

la anotación debe estar disponible en tiempo de ejecución, para su inspección a través de la reflexión de Java.

* RetentionPolicy.CLASS:

la anotación estaría en el archivo .class pero no estaría disponible en tiempo de ejecución.

* RetentionPolicy.SOURCE:

La anotación estaría disponible en el código fuente del programa, no estaría en el archivo .class ni estaría disponible en el tiempo de ejecución.

Cómo procesar una anotación:

| **package** org.iesbelen.anotaciones;   **import** java.lang.annotation.\*;  **import** java.lang.reflect.\*;   *//Creando la anotación* **@Retention**(RetentionPolicy.RUNTIME)  **@Target**(ElementType.METHOD)  **@interface** MiAnotacion{   **int** **valor**();  }   **public** **class** **HolaMundo** {    *//Usando la anotación*  **@MiAnotacion**(valor=10)   **public** **void** **holaMundo**(**int** valor){  System.out.println("Hola Mundo! " + valor);  }    **public** **static** **void** **main**(String[] args) **throws** NoSuchMethodException, SecurityException {  *//Accessing annotation*  HolaMundo h=**new** HolaMundo();   Method m=h.getClass().getMethod("holaMundo", **int**.class);    MiAnotacion miAnotacion= m.getAnnotation(MiAnotacion.class);   System.out.println("valor es: "+miAnotacion.valor());    h.holaMundo(miAnotacion.valor());    }  } |
| --- |

## 1.2.2 Ejercicios

1. Crea un programa mediante una clase Empresa que use una anotación @Empleado como la siguiente (no quiere decir que sea exactamente igual sino que te inspires en ella, puedes tomar todas las decisiones diseño que consideres oportunas)

@Empleado(

nombre = "Amancio",

apellidos = "Ortega",

dni = "66554433F"

direccion = "AV.DIPUTACION S/N, P.I. SABON 15142, ARTEIXO, LA CORUÑA",

telefono = "981185596",

clase = "Directivo",

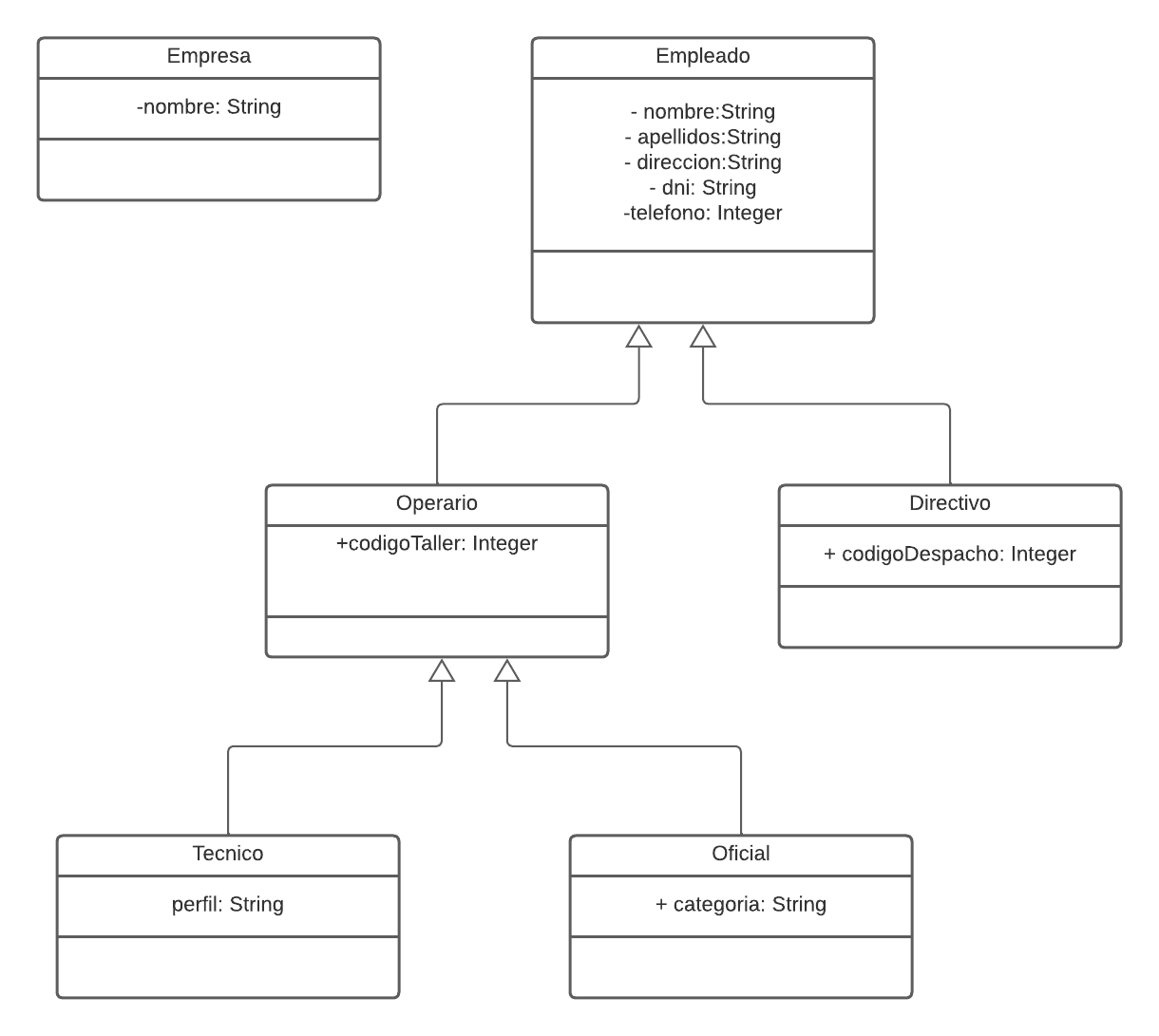
codigoDespacho = "1"

)

Utilizando la anotación proporciona a la clase Empresa los metadatos de empleado de cada clase:

* 1 Directivo,
* 1 Tecnico
* y 1 Oficial.

Debes, por tanto, a Empresa darle de tantas anotaciones @Empleado como necesites para estos 3 perfiles de empleado.



Debe existir un método estático (en algún sitio.. la misma clase Empresa, o en una clase asistente..):

*public static Empresa cargadorDeContexto()*

que procese las anotaciones de Empresa e instancie la empresa conteniendo la lista de empleados.

Empresa debe tener un método toString() que imprime todos los empleados.

Crea todo el código Java que necesites y utiliza Java Reflection sobre la clase para leer la configuración de los empleados dada mediante las anotaciones.

PISTA: Para las anotaciones repetidas tendréis que buscar un método que devuelva un array, no vale el método que te devuelve una sola anotación.

2. Modifica el programa anterior para que en vez de utilizar una anotación @Empleado con un atributo clase = Directivo, Tecnico u Oficial, emplee anotaciones @Directivo, @Tecnico y @Oficial para instanciar a los empleados. Crea todo el código Java que necesites y utiliza Java Reflection sobre la case para leer esta configuración de empleados.

3. Crea una clase AgendaSemanal que cree una lista de tareas semanales en base a una anotación repetitiva @Tarea que indique:

* título de la tarea,
* descripción,
* día de la semana
* y hora.

Crea todo el código Java que necesites y utiliza Java Reflection sobre la clase para leer la configuración de los tareas dada mediante las anotaciones.

4. (Opcional) Realiza un programa que utilice la librería de Java lombok <https://projectlombok.org/> consulta la documentación <https://projectlombok.org/api/lombok/package-summary> y que emplee las anotaciones @NonNull, @Value, @Builder y @Data, explicando en comentarios qué efectos tienen cada una de las anotaciones.

Una posible referencia: <https://leanmind.es/es/blog/simplificando-java-con-lombok/>