Contenido

1. MIEMBROS

Los miembros de una clase son los elementos que la constituyen. La definición de los miembros de una clase permite dotarla de datos y comportamiento. En este documento se abordan los miembros que representan los datos de la clase, mientas que un documento posterior abordaremos los miembros que dotan de comportamiento a la clase (métodos). Los principales tipos de miembros que representan los datos de la clase son los campos, las propiedades y los indexadores.

2. CAMPOS

Un campo es una **variable declarada en el cuerpo de la clase**. El campo representa alguna información acerca del objeto instanciado y es accesible por todos los restantes miembros de la clase. La información almacenada en el campo suele ser utilizada por los métodos de la clase para realizar alguna operación.

Habitualmente, un campo almacena un dato al que debe tener acceso más de un método de clase y, por tanto, se debe almacenar durante más tiempo que el período de duración de un único método.

En la definición de un campo se incluye un modificador de visibilidad[1] que determina el grado de acceso a dicho campo. Aunque es posible asignar cualquier modificador de accesibilidad, se debe utilizar únicamente `private` o `protected` puesto que **un campo no debe ser visible por elementos externos a la instancia**.

Se puede proporcionar un valor inicial a un campo en la misma sentencia de declaración, mediante una ***inicialización inline***. La inicialización inline **tiene lugar antes de llamar al constructor de instancias**. Si el constructor asigna el valor de un campo, sobrescribirá cualquier valor dado durante la inicialización inline del campo.

One more time:

- Aunque es posible asignar cualquier modificador de accesibilidad, a un campo sólo se le debe aplicar visibilidad `private` o `protected`, puesto que un campo **no debe ser invocable desde elementos externos a la instancia.**

```
public class Product
   {
   // Esto es un campo que representa el precio de un producto.
   // Un campo siempre debe ser privado. Cualquier otro miembro de esta clase
   // puede leer o escribir en este campo.
   private double price;

   //...
   }
```

[1] En algunos de los apartados de este documento se hace referencia a modificadores de visibilidad. Trataremos estos modificadores en un documento posterior. Basta por ahora conocer que permiten (`public`) u ocultan (`private`) el acceso a los miembros de una clase desde otra clase. Aunque puede ser fácil intuir su funcionamiento por el contexto, es conveniente volver a repasar este documento una vez que se hayan abordado estos modificadores de visibilidad.

2.1. CAMPOS `final`

El modificador `final` dota al campo al que se le aplica de un comportamiento de **sólo-lectura**, caracterizado por lo siguiente:

- Cualquier campo `final` sólo puede ser asignado **en el constructor o en una inicialización inline**. Cualquier intento de asignación posterior genera un error en tiempo de diseño.

- Si el campo es un **tipo manejado por valor**, el modificador `final` **impide que dicho campo pueda cambiar de valor** tras su inicialización.

- Si el campo afectado por `final` es un tipo referenciado, dicho campo no podrá referenciar a otro objeto tras la instanciación, aunque el objeto referenciado **sí podrá modificar sus propiedades**.

```
public class Test
   {
   // Un campo de tipo primitivo, afectado por el modificador final:
   //
   //    -  Debe ser inicializado inline (en la sentencia de declaración) o bien
   //       en el constructor.
   //    -  Una vez inicializado, no se puede modificar su valor
   //
   // En este caso, el campo se inicializa inline
   private final double field1 = 5.0;

   private final double field2;

   public Test(double value)
      {
      // El campo field2 se inicializa en el constructor.
      this.field2 = value;
      }

   public void testMethod()
      {
      // ERROR: Esta sentencia no compilará porque el campo field2 está afectado
      // por el modificador final y sólo puede ser asignado inline o en el
      // constructor.
      this.field1 = 10;
      }
   }
```

- Siempre se debe valorar la posibilidad de aplicar a un campo el modificador `final`, que indica que su valor sólo puede ser establecido en el constructor o mediante una inicialización *inline*, consiguiendo así dotarlo de **inmutabilidad**. En general, siempre es una buena práctica reducir al máximo la visibilidad y el acceso de los elementos del programa.

3. LA AUTOREFERENCIA `this`

Aunque, en sentido estricto, `this` no es un miembro de una clase, abordaremos aquí esta referencia puesto que su uso es habitual al escribir código en el cuerpo de los miembros de una clase, tal y como veremos en este y otros documentos posteriores. Esta referencia también resulta necesaria al trabajar con constructores y métodos de extensión.

En el cuerpo de cualquier clase está disponible una referencia especial, `this`, que apunta al propio objeto cuyo código está siendo ejecutado.

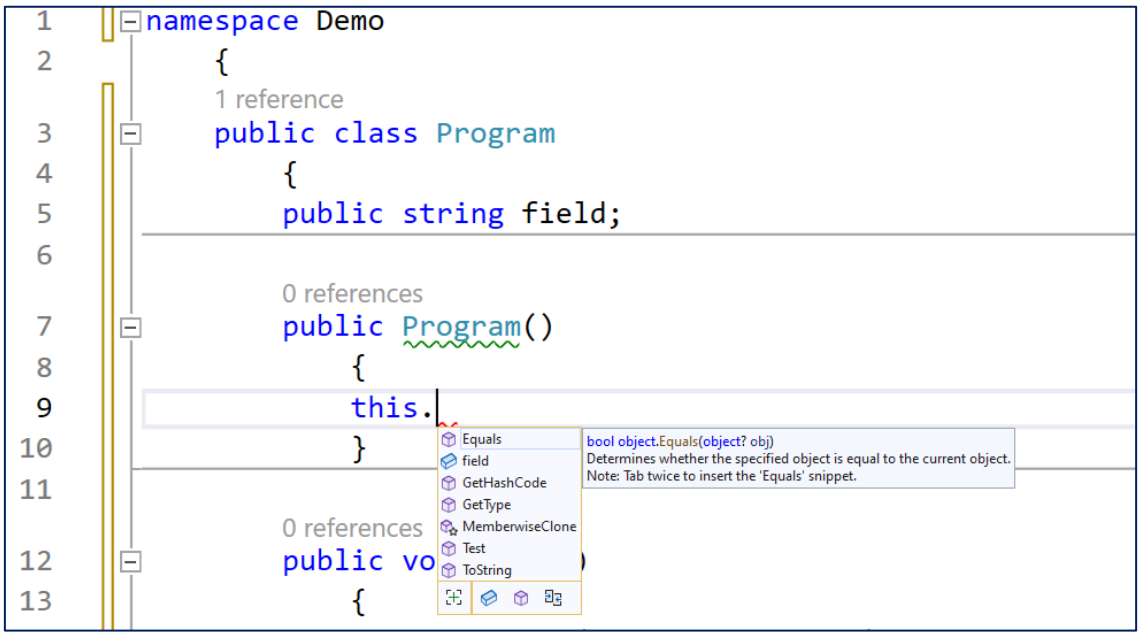
Por tanto, cuando en el código de una clase aparece la expresión:

```
this.<miembro>
```

podemos leerla como "***en este objeto, accede al miembro regular <miembro>***".

Aunque, en muchos casos, el uso de `this` puede parecer superfluo e ***innecesario***, su utilización aporta las siguientes ventajas:

- Es una forma rápida de acceder al listado de miembros del objeto actual y referenciar uno de esos miembros. En la práctica totalidad de IDEs, al invocar a `this`, el completador de código muestra la lista de miembros del objeto actual, junto con su documentación, permitiendo seleccionar de forma rápida al miembro que interese:



- `this` puede facilitar la lectura y comprensión del código: cuando, en el código de una clase, observamos que aparece un identificador (especialmente en un fragmento de código extenso), no tenemos información directa de si ese identificador pertenece a una variable previamente declarada, a una propiedad estática, a una constante o a un tipo externo (tendremos que retroceder en el código en busca del punto en el que se declara ese identificador).

  Sin embargo, si el identificador va precedido de la referencia `this`, sabemos que el elemento referenciado es un miembro regular del objeto actual, excluyendo las demás posibilidades.

  Por ejemplo, al leer el cuerpo del método `testMethod` que se muestra a continuación, sabemos que `field` es un miembro regular definido en la clase actual, sin necesidad de analizar la totalidad del código de la clase en busca del punto en el que se declara `field` :

```
public class Test
   {
   private String field = "hola";

   // [ ... cientos de líneas de código ...]

   public void testMethod(double value)
      {
      // [ ... cientos de líneas de código ...]
      System.out.print(this.field);
      }
   }
```

  En efecto, tal y como indicábamos antes, `this.field` debe leerse como: "***en este objeto, accede al miembro regular*** *`field`*".

  El siguiente fragmento es completamente correcto e idéntico en funcionalidad al anterior, pero, al no utilizar `this`, no sabemos si `field` es una propiedad, una variable previamente declarada, una constante, un miembro estático o un tipo externo: necesitaríamos retroceder y analizar `[…cientos de líneas de código…]` hasta encontrar el punto en el que `field` está declarado.

```
public class Test
   {
   private String field = "hola";

   // [ ... cientos de líneas de código ...]

   public void testMethod(double value)
      {
      // [ ... cientos de líneas de código ...]
      System.out.print(field);
      }
   }
```

Por otro lado, existen algunos escenarios en los que el uso de `this` es **obligatorio**:

- Cuando un método recibe un parámetro cuyo identificador coincide con el identificador de un miembro de la clase, es obligatorio utilizar `this` para **desambiguar**. El identificador al que se accede a través de `this` se refiere al miembro del objeto y el identificador al que se accede directamente, sin `this`, hace referencia al parámetro del método:

```
public class Test
   {
   private String field = "hola";

   // Este método recibe un parámetro que tiene el mismo identificador que un
   // miembro de la clase (field). En esta situación es obligatorio utilizar
   // this para dejar claro si nos queremos referir al campo o al parámetro.
   public void testMethod(string field)
      {
      // muestra el valor del campo field ("hola")
      System.out.print(this.field);

      // muestra el valor del parámetro field
      System.out.print(field);
      }
   }
```

4. MÉTODOS DE ACCESO: SETTERS Y GETTERS

Como hemos visto en el apartado anterior, un campo debe tener visibilidad `private` o `protected`. Esto quiere decir que el campo debe ser leído o escrito únicamente por los miembros de la clase en la que está definido, pero no debe ser accesible directamente desde fuera de la clase.

Si se quiere exponer un campo para que sea accesible desde el exterior de la clase, debe hacerse por medio de un **método de acceso**. Un método de acceso es un método convencional cuya implementación permite realizar lecturas y/o escrituras sobre el valor almacenado en un campo (al que comúnmente se denomina *campo de respaldo*).

Existen dos tipos de métodos de acceso:

- Métodos de acceso set o ***setter***s: son métodos que reciben un valor como parámetro y lo almacenan en el campo de respaldo asociado.

- Métodos de acceso get o ***getter***s: son métodos que devuelven al código llamador el valor almacenado en el campo de respaldo asociado.

Por convenio, el identificador de estos métodos comienza con el prefijo **`get`** o **`set`**, seguido del identificador del campo de respaldo al que se encuentran asociados.

La implementación de un setter o getter consiste en almacenar o devolver el valor de un campo, respectivamente:

```
// Esta clase representa un determinado producto a la venta
public class Product
   {
   // Este es el campo de respaldo sobre el que actúan el setter y getter y que se
   // definen a continuación.
   // Como todos los campos, debe ser privado y, en este ejemplo, almacena el
   // precio del producto.
   private double price;

   // Este método de acceso set se ocupa de almacenar el valor que recibe como
   // parámetro en el campo de respaldo.
   // Observa que el parámetro del setter debe pertenecer al mismo tipo de dato
   // que el campo de respaldo
   public void setPrice(double price)
      {
      this.price = price;
      }

   // Este método de acceso get expone públicamente al código llamador el precio
   // del producto, accediendo al campo de respaldo price.
   // Observa que el getter debe devolver el mismo tipo de dato que el campo de
   // respaldo.
   public double getPrice()
      {
      return this.price;
      }

   //..
   }

public class Program
   {
   public static void main (String[] args)
      {
      Product product = new Product();

      // Invoca al setter para almacenar un precio en el campo de respaldo.
      product.setPrice(25.4);

      // Invoca al getter para recuperar el precio almacenado en el campo de
      // respaldo.
      System.out.print(product.getPrice());
      }
   }
```

En el ejemplo anterior puede comprobarse que, en ningún momento, el código llamador (el código del método `main`) accede directamente al campo `price`, ya que dicho campo es `private` y, por tanto, no es accesible desde fuera de la clase `Product`. **Toda la interacción con el campo se realiza, indirectamente, a través del setter y del getter**.

La razón de operar de esta forma, indirectamente, es que permite realizar (si fuese necesario) una verificación de que el valor asignado/devuelto al/por el campo cumple determinadas restricciones que satisfacen el rango de valores válidos, estableciendo de esa manera una protección ante valores o usos no permitidos.

De esta manera, cualquier operación de validación o tratamiento previo que se haga sobre el valor del campo, queda centralizada en el método de acceso, evitando repetir código o implementar validaciones o procesos dispersos por todo el código fuente.

Incluso si no se realizan validaciones o cálculos sobre el valor del campo, el uso de los métodos de acceso es **prescriptivo**: por un lado, es frecuente que el código evolucione y sea necesario en un futuro realizar validaciones y/o cálculos sobre los valores de un campo, por lo que el empleo de métodos de acceso constituye un modo de *programación defensiva*, que prepara el código para anticiparse a estas situaciones.

Por otro lado, el uso de métodos de acceso aporta una **carga semántica** al código: nos informa que el objeto expone un dato, que conforma su estado interno y que soporta operaciones de lectura y/o escritura.

- Un método de acceso es un miembro de la clase que permite, a tipos observadores externos, leer, escribir o calcular el valor de un campo privado.
- En el método de acceso se puede realizar una verificación y/o tratamiento de los valores leídos o escritos del/al campo.
- En cualquier caso, si se quiere exponer un campo al exterior de la clase, se debe implementar un método de acceso, incluso si no se realiza verificación o tratamiento de los valores.
- También se pueden crear métodos de acceso privados, que se consumen directamente desde otros miembros de la misma clase, para tener centralizadas las operaciones de verificación o tratamiento de los valores del campo.

Modificando el código anterior, podríamos añadir, por ejemplo, una validación para verificar que el precio asignado es correcto (es positivo):

```
// Esta clase representa un determinado producto a la venta
public class Product
   {
   // Este es el campo de respaldo sobre el que actúan el setter y getter que se
   // definen a continuación.
   // Como todos los campos, debe ser privado y, en este ejemplo, almacena el
   // precio del producto.
   private double price;

   // Este método de acceso set se ocupa de almacenar el valor que recibe como
   // parámetro en el campo de respaldo.
   // Observa que el parámetro del setter debe pertenecer al mismo tipo de dato
   // que el campo de respaldo
   public void setPrice(double price)
      {
      if (price > 0)
         this.price = price;
      else
         throw new IllegalArgumentException("price:" + price);
      }

   // Este método de acceso get expone públicamente al código llamador el precio
   // del producto, accediendo al campo de respaldo price.
   // Observa que el getter debe devolver el mismo tipo de dato que el campo de
   // respaldo.
   public double getPrice()
      {
      return this.price;
      }

   //...
   }

public class Program
   {
   public static void main (String[] args)
      {
      Product product = new Product();

      // Invoca al setter para almacenar un precio en el campo de respaldo.
      product.setPrice(25.4);

      // Invoca al getter para recuperar el precio almacenado en el campo de
      // respaldo.
      System.out.print(product.getPrice());
      }
   }
```

*2

[2] Emitir una excepción desde un método de acceso puede no ser adecuado en muchas circunstancias. Se incluye aquí esta implementación a título de ejemplo, pero en un escenario real debe ser objeto de análisis la idoneidad de esta operación.

Cuando una clase sólo expone un getter para un campo, dicho campo se considera de **sólo-lectura**. Inversamente, cuando una clase sólo expone un setter para un campo, dicho campo se considera de **sólo-escritura**. Un campo vinculado a un setter y a un getter es de **lectura** y **escritura**.

- Tanto un campo `final` como un campo de sólo-lectura permiten leer el valor o la referencia almacenados en un campo. La diferencia entre un campo `final` y un campo de sólo-lectura radica en que:
  - Un campo no debe ser accesible desde el exterior de la instancia, mientras que el getter sí puede ser expuesto públicamente.
  - El campo `final`, una vez inicializado, no puede variar su valor o referencia, mientras que una propiedad de sólo lectura puede variar el valor devuelto, dependiendo de su implementación.

Los métodos de acceso se ven afectadas por un ***modificador de visibilidad***, que determina el grado de acceso de la propiedad desde tipos observadores externos y desde tipos derivados.

```
public class Product
   {
   private double price;

   // Este getter expone públicamente el precio para su lectura
   public double getPrice()
      {
      return this.price;
      }

   // Este setter, al ser privado, sólo puede ser utilizado por
   // los miembros de este objeto (por ejemplo, desde un hipotético método
   // 'UpdatePrice'), pero no puede ser invocado desde fuera de la clase.
   private void setPrice(double price)
      {
      this.price = price;
      }

   //...
   }
```

- Como norma general de diseño, los métodos de acceso no deben emitir excepciones.
- El descriptor de acceso debe realizar una operación sencilla de tratamiento o procesamiento del valor. En caso contrario, puede ser más correcto descartar el uso de un método de acceso e implementar en su lugar un método regular.
- Por tanto, si el método de acceso requiere una **operación compleja**, que **puede emitir excepciones** o bien que **requiere múltiples parámetros** para el procesamiento, se debe descartar el uso de métodos de acceso y **optar por el uso de un método de instancia regular**.

- Si un campo pertenece a un tipo que representa una colección de datos, como un array o una lista, evalúa la posibilidad de diseñar el campo con métodos de acceso de sólo-lectura (sólo un getter, sin setter). Exponer un setter con visibilidad pública para estos tipos de datos se considera una mala práctica.

4.1. MÉTODOS DE ACCESO AUTOIMPLEMENTADOS

A diferencia de .NET, en Java no existe el equivalente a propiedades autoimplementadas (no existen getter/setter automáticos con una implementación por defecto). Aunque podemos utilizar las herramientas del IDE para generar esta implementación por defecto, el código resultante es mucho más extenso e incómodo de manejar que su contrapartida en .NET

5. MIEMBROS ESTÁTICOS

Cualquier miembro de una clase, ya sea un campo, una propiedad o un método, puede declararse con el modificador `static` para definirlo **a nivel de clase** en vez de hacerlo a nivel de instancia. Al declarar un miembro estático, existirá **un único miembro,** que se mantiene definido en la clase, pero no existe en los objetos creados a partir de esa clase.

Esto quiere decir que los miembros estáticos no se construyen en los objetos durante la instanciación, es decir, **no están presentes en los objetos** creados a partir de la clase, sino que se encuentran como un dato almacenado en la propia clase.

Dado que un miembro estático se encuentra definido a nivel de clase, para acceder a dicho miembro es necesario hacerlo a través del identificador de la clase donde está definido, como en el siguiente ejemplo:

```
public class Test
   {
   private static double staticField = 5.0;

   public void testMethod()
      {
      // Correcto: un miembro estático debe ser accedido a través del
      // identificador de la clase, ya que se encuentra definido a nivel de
      // clase y no en las instancias de esa clase.
      System.out.print(Test.staticField);

      // ERROR: Esta sentencia no compilará porque el campo staticField es
      // estático y no está presente en los objetos, sino en la propia clase. Por
      // esa razón, el campo no puede ser accedido a través del modificador this.
      // que es una autoreferencia al objeto cuyo código está siendo ejecutado.
      System.out.print(this._staticField);
      }
   }
```

- Siempre que sea posible, es conveniente inicializar el valor de un campo estático mediante una **inicialización *inline***, es decir, se debe declarar e inicializar el campo en la misma sentencia. Hacerlo desde el constructor estático impone una ligera (imperceptible) penalización al rendimiento.

Volveremos a tratar con mayor profundidad los miembros estáticos en un documento posterior.

6. CAMPOS CONSTANTES

A diferencia de .NET, Java no soporta directamente el empleo de campos constantes. Para proporcionar el comportamiento de campo constante, es necesario declarar el campo aplicando sobre él una combinación de los modificadores `static` y `final`.

Al declarar un campo con estos modificadores, el campo debe ser inicializado en la misma sentencia de declaración, mediante una inicialización inline, y **se comporta como un miembro estático de la clase**. Esto implica que un campo constante está presente en la clase, pero no en los objetos instanciados a partir de esa clase.

Ten en cuenta que, además de campos constantes, también se pueden definir constantes en el cuerpo de un método. Las constantes internas a un método se declaran como variables convencionales, pero afectadas por el modificador `final` y **limitan su ámbito de existencia al cuerpo del método que las declara**.