Diseño de clases en Java

Ricardo Pérez López

IES Doñana, curso 2020/2021

Generado el 12 de abril de 2021 a las 13:18:00

Índice general

1. Definición de clases

1.1. Sintaxis básica

⟨*clase*⟩ ::= [⟨*modif_acceso_clase*⟩] **class** ⟨*nombre*⟩ **{**
⟨*miembro*⟩*
}

⟨*nombre*⟩ ::= **identificador**
⟨*miembro*⟩ ::= ⟨*variable*⟩ | ⟨*método*⟩
⟨*variable*⟩ ::= [⟨*modif_acceso_miembro*⟩] [**static**] ⟨*decl_variable*⟩
⟨*método*⟩ ::= [⟨*modif_acceso_miembro*⟩] [**static**] ⟨*def_método*⟩
⟨*modif_acceso_clase*⟩ ::= **public**
⟨*modif_acceso_miembro*⟩ ::= **public** | **private** | **protected**

La definición de una clase es una construcción sintática que define su propio ámbito y que está formada por un bloque de **declaraciones de miembros**, cuada una de las cuales puede declarar una **variable** (también llamada **campo**) o un **método**.

A su vez, cada miembro puede ser **de instancia** o puede ser **estático**.

Ejemplo

```java
public class Hola {
    public int x = 4;
    protected String nombre;

    public void saludo() {
        System.out.println("¡Hola!");
    }
}
```

1.2. Visibilidad de una clase

Una clase siempre pertenece siempre a un paquete, que es el paquete en el que se ha definido.

El uso y definición de paquetes en Java lo estudiaremos con más profundidad posteriormente.

En relación a los paquetes en las que se definen, las clases pueden tener dos tipos de visibilidades:

- **Visibilidad *predeterminada*** (***por defecto*** o ***default***): la clase sólo es accesible desde el interior del paquete en el que se ha definido.
- **Visibilidad *pública***: la clase es accesible desde cualquier paquete.

Para indicar la visibilidad que debe tener una clase, se puede usar un **modificador de acceso**.

1.2.1. Visibilidad predeterminada

Cuando no se utiliza ningún *modificador de acceso* al definir la clase, ésta se define con visibilidad predeterminada.

En un archivo fuente pueden definirse tantas clases con visibilidad predeterminada como se desee.

Además, en ese caso el archivo fuente puede tener cualquier nombre (por supuesto, siempre con extensión `.java`).

1.2.2. Visibilidad pública

Para definir una clase con visibilidad pública, se usa el *modificador de acceso* `public` en la definición de la clase.

En un archivo fuente pueden definirse muchas clases, pero sólo una de ellas puede ser pública.

Además, el archivo fuente debe llamarse igual que la (única) clase pública que contiene.

1.3. Visibilidad de un miembro de una clase

Cada miembro de una clase puede tener uno de estos cuatro tipos de visibilidades:

- **Visibilidad *privada***: el miembro sólo es accesible desde el interior de la clase en la que se ha definido.
- **Visibilidad *predeterminada*** (***por defecto*** o ***default***): el miembro es accesible desde el interior de la clase en la que se ha definido y también desde otras clases que pertenezcan al mismo *paquete*.
- **Visibilidad *protegida***: el miembro es accesible desde el interior de la clase en la que se ha definido, también desde otras clases que pertenezcan al mismo paquete y también desde sus subclases (aunque se hayan definido en paquetes distintos).
- **Visibilidad *pública***: el miembro es accesible desde el interior de la clase en la que se ha definido y también desde cualquier otra clase (siempre que la clase en sí también sea accesible).

La visibilidad es un mecanismo de **encapsulación** que impide que ciertos miembros puedan ser accedidos (o incluso conocidos) fuera de la clase en la que se han definido, o fuera del paquete que contiene la clase en la que se ha definido.

El siguiente cuadro resume las cuatro visibilidades y desde dónde se puede acceder a un miembro definido con una determinada visibilidad en una determinada clase:

Visibilidad	La propia clase	Otras clases del mismo paquete	Subclases de la clase	Otras clases de cualquier paquete
Privada	Sí	No	No	No
Predeterminada	Sí	Sí	No	No
Protegida	Sí	Sí	Sí	No
Pública	Sí	Sí	Sí	Sí

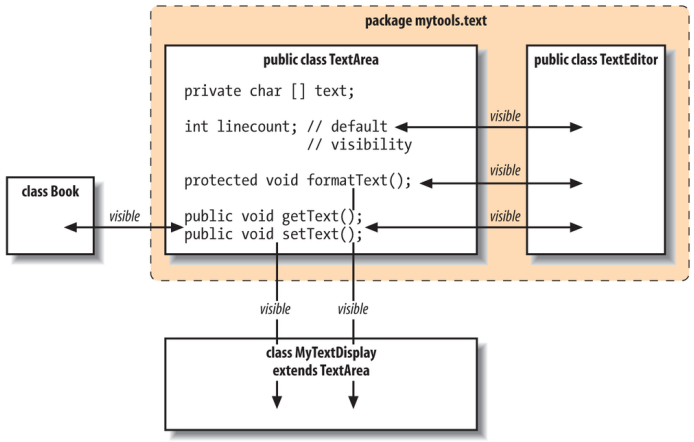
La visibilidad que queremos que tenga un determinado miembro se indica mediante los llamados **modificadores de acceso**.

Los **modificadores de acceso** son palabras clave que acompañan a la declaración de un miembro y que sirven para indicar la visibilidad deseada para ese miembro.

La forma de indicar que se desea que un miembro tenga visibilidad predeterminada es no usar ningún modificador de acceso en su declaración.

Para el resto de visibilidades existe un **modificador de acceso** que puede ir acompañando a la **declaración de cada miembro**:

Visibilidad	Modificador de acceso
Pública	`public`
Privada	`private`
Protegida	`protected`
Predeterminada	*(ninguno)*



Visibilidades en Java

Las variables protegidas de una clase son visibles para sus subclases, pero sólo a través de objetos del tipo de la subclase o sus subtipos.

En otras palabras, **una subclase puede ver una variable protegida de su superclase como una variable heredada, pero no puede acceder a esa misma variable a través de una referencia a la propia superclase**.

Esto puede parecer un poco confuso al principio, ya que puede que no resulte obvio que los modificadores de visibilidad no restringen el acceso entre instancias de la misma clase de la misma manera que restringen el acceso entre instancias de diferentes clases.

Dos instancias de la misma clase pueden acceder a todos los miembros de la otra, incluidos los privados, siempre que se acceda a través de una referencia del tipo correcto.

Dicho de otra manera: dos instancias de `Gato` pueden acceder a todas las variables y métodos de cada uno (incluidos los privados), pero un `Gato` no puede acceder a un miembro protegido de una instancia de `Animal` a menos que el compilador pueda probar que el `Animal` es un `Gato`.

2. Miembros de instancia

2.1. Variables de instancia

En una clase se pueden declarar **variables de instancia**.

Cuando se instancia un objeto a partir de esa clase, dicho objeto contiene las variables de instancia declaradas en su clase, además de otras posibles variables de instancia públicas o protegidas que se hayan podido heredar de sus superclases (y que estarán declaradas en éstas).

La declaración de una variable de instancia en una clase tiene la misma sintaxis que la usada para declarar cualquier otra variable, dentro del cuerpo (bloque) de la definición de la clase y fuera de cualquier método:

⟨*clase*⟩ ::= [⟨*modif_acceso_clase*⟩] **class** ⟨*nombre*⟩ **{**
⟨*miembro*⟩*
}

⟨*miembro*⟩ ::= ⟨*variable*⟩ | ⟨*método*⟩
⟨*variable*⟩ ::= [⟨*modif_acceso_miembro*⟩] [**static**] ⟨*decl_variable*⟩
⟨*modif_acceso_miembro*⟩ ::= **public** | **private** | **protected**

2.1.1. Acceso y modificación

Para acceder a una variable de instancia de un objeto, se usa el operador punto (**.**), como es ya usual y con la sintaxis que ya conocemos:

referencia **.** *variable*

Ejemplo:

```
class Hola {
    public int x = 4;
    protected String nombre;

    public void saludo() {
        System.out.println("¡Hola!");
    }
}
```

2.1.2. Variables de instancia finales

2.2. Métodos

En una clase se pueden definir métodos, que pueden ser:

- Métodos de instancia: los que se invocan *sobre* un objeto.
- Métodos estáticos: los que **no** se invocan sobre un método.

Para que podamos invocar un método sobre un objeto, éste debe disponer de dicho método. Para ello, debe ocurrir una de estas dos cosas:

- El objeto es instancia de una clase que define dicho método.
- La clase hereda el método de una superclase (el método tendrá que ser público o protegido).

La definición de un método de instancia dentro de una clase tiene la siguiente sintaxis:

```
⟨clase⟩ ::= [⟨modif_acceso_clase⟩] class ⟨nombre⟩ {
        ⟨miembro⟩*
}

⟨nombre⟩ ::= identificador
⟨miembro⟩ ::= ⟨variable⟩ | ⟨método⟩
⟨variable⟩ ::= [⟨modif_acceso_miembro⟩] [static] ⟨decl_variable⟩

⟨método⟩ ::= [⟨modif_acceso_miembro⟩] [static] ⟨def_método⟩
⟨modif_acceso_miembro⟩ ::= public | private | protected
⟨def_método⟩ ::= ⟨tipo⟩ identificador ( ⟨lista_parámetros⟩ ) ⟨cuerpo⟩
⟨lista_parámetros⟩ ::= ⟨decl_parámetro⟩[ , ⟨decl_parámetro⟩]*
⟨decl_parámetro⟩ ::= ⟨tipo⟩ identificador
⟨cuerpo⟩ ::= ⟨bloque_no_vacío⟩
⟨bloque_no_vacío⟩ ::= {
        ⟨sentencia⟩+
}
```

2.2.1. La sentencia `return`

Dentro de un método, se usa la sentencia **`return`** para:

- **Finalizar la ejecución del método** y devolver el control al punto del programa desde el que se invocó al método.
- **Devolver** al llamante el **valor de retorno** del método.

Ejemplo:

```java
public class Hola {
    public int x = 4;
    protected String nombre;

    public String saludo() {
        return "¡Hola!";
    }
}
```

Si un método no devuelve **ningún valor**, debe definirse de tipo `void`.

2.2.2. Referencia `this`

Dentro de un método de instancia, la variable especial **`this`** contiene siempre una **referencia al objeto sobre el que se ha invocado al método**.

Cumple el mismo papel que el parámetro especial `self` en Python, pero aquí tiene la peculariedad de que es un **parámetro implícito** que se recibe siempre y que no hay que declararlo en la lista de parámetros.

Por contra, el `self` de Python es un *parámetro explícito*.

A través de la referencia **`this`**, podemos acceder a los campos del objeto y manipularlo directamente.

Ejemplo:

```java
public class Prueba {
    private int x = 4;

    public int getX() {
        return this.x;
    }

    public int setX(int x) {
        this.x = x;
        return this.getX();
    }
}
```

La definición de un método **define un nuevo ámbito**, y ese nuevo ámbito está **anidado dentro del ámbito de la clase** donde se define el método.

Además, **la ejecución del método provoca**, en tiempo de ejecución, **la creación de un nuevo marco** en la pila.

Como en Java no es posible definir métodos anidados, y tampoco existe un ámbito global, eso quiere decir que ese marco será el único que existirá en el entorno.

Cuando no hay ambigüedad, es posible evitar el uso de `this` y acceder directamente al campo sin usar una referencia al objeto.

Para ello, el compilador determina (en tiempo de compilación) a qué variable corresponde cada identificador.

Ejemplo:

```java
1  public class Prueba {
2      private int x = 4;
3
4      public int getX() {
5          return x;        // El compilador deduce que se refiere al campo x
6      }
7
8      public int setX(int x) {
9          this.x = x;      // Aquí hay que romper la ambigüedad
10         return getX();
11     }
12 }
```

En la línea 9, el identificador `x` podría representar dos cosas:

- El parámetro del método.
- La variable de instancia declarada en la clase.

Para romper la ambigüedad, es necesario usar `this.x` para referirse a la variable de instancia, en lugar de sólo `x` (que se referirá al parámetro).

Asimismo, en la línea 10 se puede llamar al método `getX` directamente sin usar `this`, ya que no hay ambigüedad.

2.2.3. Accesores y mutadores

Crear accesores y mutadores en Java es fácil y similar a hacerlo en Python.

Usando los modificadores de acceso, garantizamos la encapsulación de las variables privadas.

Ejemplo:

```java
public class Prueba {
    private int x = 4;           // Variable privada

    public int getX() {          // Accesor público
        return this.x;
    }

    public void setX(int x) {    // Mutador público
        if (x >= 0) {
            this.x = x;
        }
    }
}
```

2.2.4. Constructores y destructores

2.2.5. Sobrecarga

2.2.6. Ámbito de un identificador

2.2.7. Resolución de identificadores

3. Miembros estáticos

3.1. Métodos estáticos

3.2. Variables estáticas

3.3. Variables estáticas finales