Repaso de conceptos de orientación a objetos en Java

Curso 2017-18

Índice

Introducción

Orientación a objetos: metodología de desarrollo de software de gran escala. ¿Cómo se llega a este paradigma de programación? Como consecuencia de la crisis del software y atendiendo al cambio de mentalidad (del **mecanismo** a la persona....).

Introducción

Conceptos clave son:

- **objeto**: representa una entidad en el mundo real, que puede identificarse de forma única. Los objetos se caracterizan por:
  - estado: propiedades (atributos, datos miembro) que lo caracterizan
  - comportamiento: acciones (métodos miembro) que pueden realizarse sobre él
- **clase**: los objetos de un mismo tipo se definen usando una clase común. Una clase es una plantilla que define las propiedades y acciones de un tipo de objetos

Introducción

Se pueden crear varios objetos de una clase (suele ser así). Este proceso se conoce como **instanciación**. A considerar:

- los términos **objeto** e **instancia** se utilizan de forma indistinta
- las clases disponen de métodos especiales, llamados **constructores**, cuya misión es permitir la creación de nuevos objetos
- uso de diagramas UML para representar clases, objetos y relaciones entre clases

Introducción

- un único archivo pueden contener varias clases. Sólo puede haber una clase pública en un archivo; la clase pública y el archivo deben llamarse de la misma forma
- la ejecución de código java implica la existencia de alguna clase que contenga un método **main**

Índice

Creación de objetos

Los constructores son los métodos involucrados en la creación de objetos. Tienen tres características esenciales:

- tienen el mismo nombre que la clase
- no tienen asociado tipo de salida, ni **void**
- se llaman mediante el operador **new**. Los constructores también se encargan de inicializar el valor de los datos miembro

Una clase puede disponer de varios constructores (es lo habitual). Si no se implementa ninguno, Java incorpora un constructor denominado **constructor por defecto**, sin argumentos

Índice

Variables de referencia

Los objetos se manejan mediante **variables de referencia**. Importante:

- los datos y métodos se acceden usando variables de referencia y el operador punto (.)
- las variables de referencia se crean indicando el tipo asociado (la clase), pero sin llamada al constructor
- el valor **null** se usa para indicar que la variable de referencia no está asignada a objeto alguno. Todas las variables de referencia (sea cual sea su clase) tienen **null** como valor por defecto

Índice

Clases predefinidas en Java

Java ofrece ya muchas clases predefinidas, agrupadas en paquetes, que aportan múltiple funcionalidad: **String**, **Random**, **Date**, **ArrayList**, **Vector**, etc.

Las clases en Java se agrupan en **paquetes**. Todas las clases de un paquete están relacionadas de algún modo: paquete **util** (clases de utilidad varias), **io** (clase de entrada/salida), ...

Índice

Datos y métodos estáticos

La palabra reservada **static** alude a datos y métodos asociados a la clase completa y no a instancias particulares. Así:

- **dato miembro estático**: compartido por todas las instancias de dicha clase
- **método estático**: puede ejecutarse aunque no haya ningún objeto de la clase (se llama mediante el operador punto, pero sobre el nombre de la clase)
- las constantes estáticas precisan además de la palabra reservada **final**, que asegura que su valor no será modificado

Índice

Modificadores de visibilidad

Los modificadores de visibilidad permiten matizar el nivel de acceso a los datos y métodos miembro:

- **public**: acceso público desde todas las clases, sea cual sea su paquete
- por defecto (no se indica ninguno): hay acceso público desde las clases del mismo paquete
- **protected**: acceso público desde clases derivadas (desde cualquier paquete)
- **private**: acceso restringido a la propia clase

Índice

Encapsulación

Los datos miembro privados protegen el acceso a las propiedades de las clases y hacen que sean más fáciles de mantener:

- el acceso a los datos miembro privados: métodos de acceso (**set**/**get**)
- el método **set** debe evitar que se almacenen valores sin sentido en el dato miembro que protege

De esta forma, se dice que las clases **encapsulan** sus propiedades y proporcionan métodos seguros de acceso

Índice

Pasos de objetos a métodos

En java sólo hay un forma de paso de argumento: **paso por valor**. Importante:

- si se pasa un objeto a un método, **el argumento formal del método se convierte en realidad en una copia de la referencia al objeto** (no se hace una copia del objeto completo)
- esto indica que si se hace algún cambio sobre el objeto, mediante sus métodos o datos, estos cambios permanecen una vez haya finalizado su ejecución

Pasos de objetos a métodos

- el cambio que no tiene sentido es el de la referencia en sí. Si el argumento referencia se apunta a otro objeto, este cambio se deshará al finalizar la ejecución del método

Índice

Referencia this

La palabra reservada **this** se usa en Java (Scala, C++) para nombrar a la referencia que apunta al objeto en uso. Cuando java tiene que ejecutar un método sobre un objeto:

- crea una referencia, **this**, que apuntará al objeto sobre el que se hizo la llamada
- cuando el método finaliza java retira la referencia **this** y la prepara para apuntar al siguiente objeto con que se trabaje

Esta referencia también puede usarse para invocar métodos, paso de argumento, etc.

Índice

Abstracción y encapsulación

La abstracción de clases es la separación entre la implementación de la clase y su uso. Los detalles de implementación no interesan al usuario de la clase y están **encapsulados**. **Encapsulación** y **abstracción** son dos aspectos del mismo concepto.

Índice

Herencia I

Herencia: posibilidad de crear unas clases a partir de otras ya existentes, reaprovechando el código disponible

- la clase más general se denomina **superclase** y las clases más especializadas se denominan **clases derivadas** o **subclases**
- al construir un objeto de una clase derivada los constructores se llaman en orden de abstracción: de más abstracto a menos abstracto (de **superclase** a **clase derivada**)

Polimorfismo: una variable de un supertipo puede usarse para referirse a un objeto de un subtipo

Principio básico: uso de la herencia con precaución

Índice

Clases abstractas e interfaces

- clase **abstracta**: no completamente definida; no pueden usarse para crear objetos. Pueden incluir datos y métodos completamente definidos
- **interfaces**: caso extremos de clases abstractas. Pueden contener constantes, declaración de métodos y métodos por defecto (mediante la palabra reservada **default**)

Una clase puede implementar múltiples interfaces, pero sólo se puede derivar de una clase. Java no permite **herencia múltiple**.

Índice

Guía para el diseño de clases

En el paradigma de orientación a objetos el diseño se centra en los objetos (propiedades y acciones) y relaciones entre ellos. En el enfoque procedural (o modular) el énfasis se pone en la identificación de las operaciones (funciones).

Guía para el diseño de clases

Los objetos pueden (suelen) contener a otros objetos como datos miembro. Esta relación se denomina de diferentes formas, según el matiz semántico de la relación:

- **asociación**: una clase presenta un dato miembro cuyo tipo es otra clase diferente
- **composición**: igual que en el caso anterior, pero se asume que la clase representa el todo y el dato miembro es una parte (componente); si desaparece el todo desaparece también la parte. Se representa con un rombo en negro en la parte del todo. Ejemplo: relación entre clases Libro - Capítulo

- **agregación**: igual que composición, pero la parte puede pervivir a la destrucción del todo. Se representa con un rombo en blanco en la parte del todo. Ejemplo: relación entre clases Banco - Cliente

En Java hay problemas para indicar de forma explícita la necesidad de destruir un objeto: es una tarea del colector de basura. En C++ sí puede realizarse usando el operador **delete** y los destructores. Todas estas relaciones se plasman en el código de la misma forma: una clase presenta un dato miembro cuyo tipo es de otra clase diferente.

Guía para el diseño de clases

- **herencia**: una clase deriva de otra (o implementa alguna interfaz). Se representa con un triángulo del lado de la clase base.
- **uso**: un método de alguna clase recibe como argumento un objeto de otra o bien define alguna variable local de ella. Se representa mediante una línea discontinua.

Guía para el diseño de clases

Hay algunos principios que deben guiar el desarrollo de un sistema orientado a objetos:

- **DRY: don't repeat yourself**: evitar código duplicado, usando abstracción para detectar funcionalidad común, que se implementará en alguna clase

Este mismo principio se encuentra en **programación modular**: identificar partes de código repetidas y crear funciones para ellas

Guía para el diseño de clases

- **single responsability**: cada objeto debe tener una única responsabilidad (esto implica alta **cohesión**: toda la funcionalidad de la clase está relacionada).

Guía para el diseño de clases

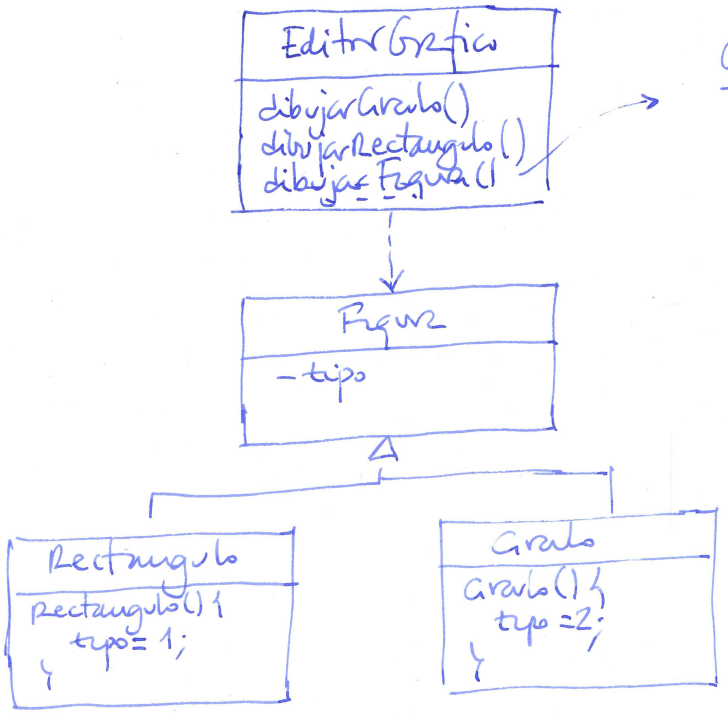
- **open-closed**: permitir modificaciones con cambios mínimos en el código ya hecho

Relacionado con el principio anterior: una clase, una responsabilidad. Si el hecho de incluir un cambio supone la necesidad de tocar varias clases, se dice que el diseño presenta un alto **acoplamiento**.

En otras palabras: la modificación debería limitarse a agregar nuevas clases y que permanezcan sin cambios las ya existentes

EJEMPLO DE PRINCIPIO OPEN – CLOSED

Problema: editor gráfico que gestiona el pintado de diferentes figuras



Cuerpo de dibujarFigura(Figura fig)

```
if (fig.tipo == 1)
    dibujarRectangulo();
else if (fig.tipo == 2)
    dibujarCirculo();
```

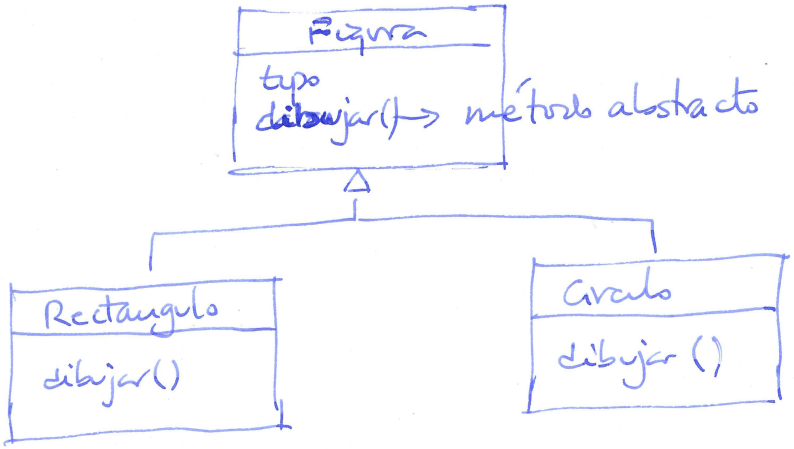
¿Qué ocurre si agregamos nuevos tipos de figuras?

Desventajas:

- para cada nueva figura, hay que rehacer el código de EditorGráfico, su conjunto de pruebas unitarias, etc
- ampliar el sistema implica conocer el funcionamiento interno de EditorGrafico
- aunque el código de la nueva figura funcione de forma correcta, puede haber problemas por EditorGrafico

Guía para el diseño de clases

¡Mejora en el diseño anterior!



¡La responsabilidad de dibujo se delega a cada tipo de Figura!

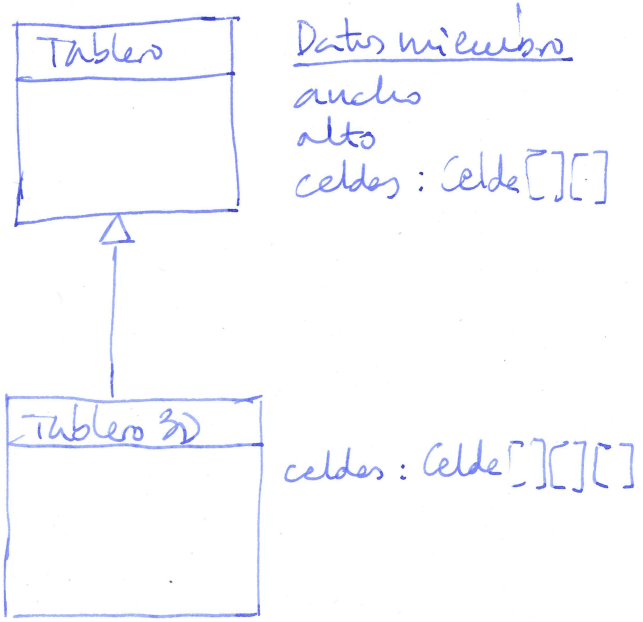
Guía para el diseño de clases

- **Liskov substitution**: los subtipos deberían poder sustituirse por los tipos base sin demadiados problemas

En otras palabras, las clases derivadas no deben romper (ir contra) las definiciones de las clases padre

PRINCIPIO DE SUBSTITUCIÓN DE LISKOV

Problema: Clase Tablero para juegos de mesa
clase Tablero3D para juegos en 3D



Datos miembro
ancho
alto
celdas : Celda[][]

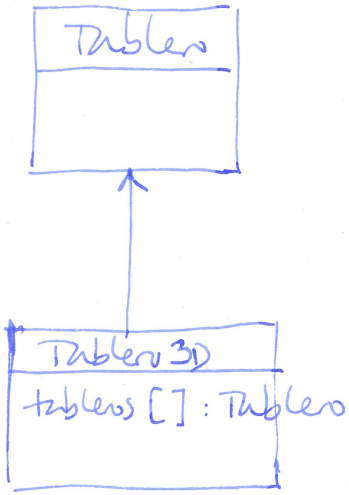
celdas : Celda[][][]

Métodos miembro
obtenerCelda(int, int)
agregarFicha(Ficha, int, int)
quitarFicha(Ficha, int, int)
quitarFichas(int, int)
obtenerFichas(int, int)

Problema: Tablero3D no es realmente un Tablero
- los métodos obtenerCelda, agregarFicha, quitarFicha, quitarFichas y obtenerFichas no tienen sentido en Tablero3D

Guía para el diseño de clases

¡Solución: usar composición frente a herencia!



← de esta forma seguen siendo válidos (útiles) los métodos de Tablero para Tablero3D

Índice

Sobrescritura y sobrecarga

Es importante distinguir entre estos dos términos:

- **sobrescritura** (**override**): definir en las subclases métodos con la misma declaración que otros ya presentes en la **superclase**
- **sobrecarga** (**overload**): definir múltiples métodos de igual nombre, pero con diferente declaración (por ejemplo, diferente número de argumentos, etc)

Son también formas de **polimorfismo**