APRENDERAPROGRAMAR.COM

CLASES QUE UTILIZAN OBJETOS. RELACIÓN DE USO ENTRE CLASES JAVA. CONCEPTO DE DIAGRAMA DE CLASES. (CU00641B)

Sección: Cursos
Categoría: Curso "Aprender programación Java desde cero"

Fecha revisión: 2029

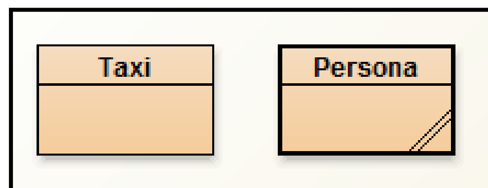
Resumen: Entrega nº41 curso Aprender programación Java desde cero.

Autor: Alex Rodríguez

CLASES QUE UTILIZAN OBJETOS. RELACIÓN DE USO ENTRE CLASES. DIAGRAMAS DE CLASES.

Hemos visto hasta ahora clases que definen tipos donde los campos son variables de tipo primitivo o String. Analicemos ahora la posibilidad de crear clases donde los atributos sean tipos que hayamos definido en otras clases.

Consideraremos que partimos de las clases Taxi y Persona ya escritas y compilando correctamente conforme a los ejemplos vistos en epígrafes anteriores.



Escribe y compila el siguiente código:

```
 //Ejemplo de clase que utiliza tipos definidos en otras clases (usa otras clases)
public class TaxiCond {

    private Taxi vehiculoTaxi;
    private Persona conductorTaxi;

    //Constructor
    public TaxiCond () {
        vehiculoTaxi = new Taxi (); //Creamos un objeto Taxi con el constructor general de Taxi
        conductorTaxi = new Persona (); //Creamos un objeto Persona con el constructor general de Persona
    }

    public void setMatricula (String valorMatricula) { vehiculoTaxi.setMatricula(valorMatricula); }

    //Método que devuelve la información sobre el objeto TaxiCond
    public String getDatosTaxiCond () {
        String matricula = vehiculoTaxi.getMatricula();
        String distrito = vehiculoTaxi.getDistrito();
        int tipoMotor = vehiculoTaxi.getTipoMotor();
        String cadenaTipoMotor = "";
```

© aprenderaprogramar.com, 2006-2029

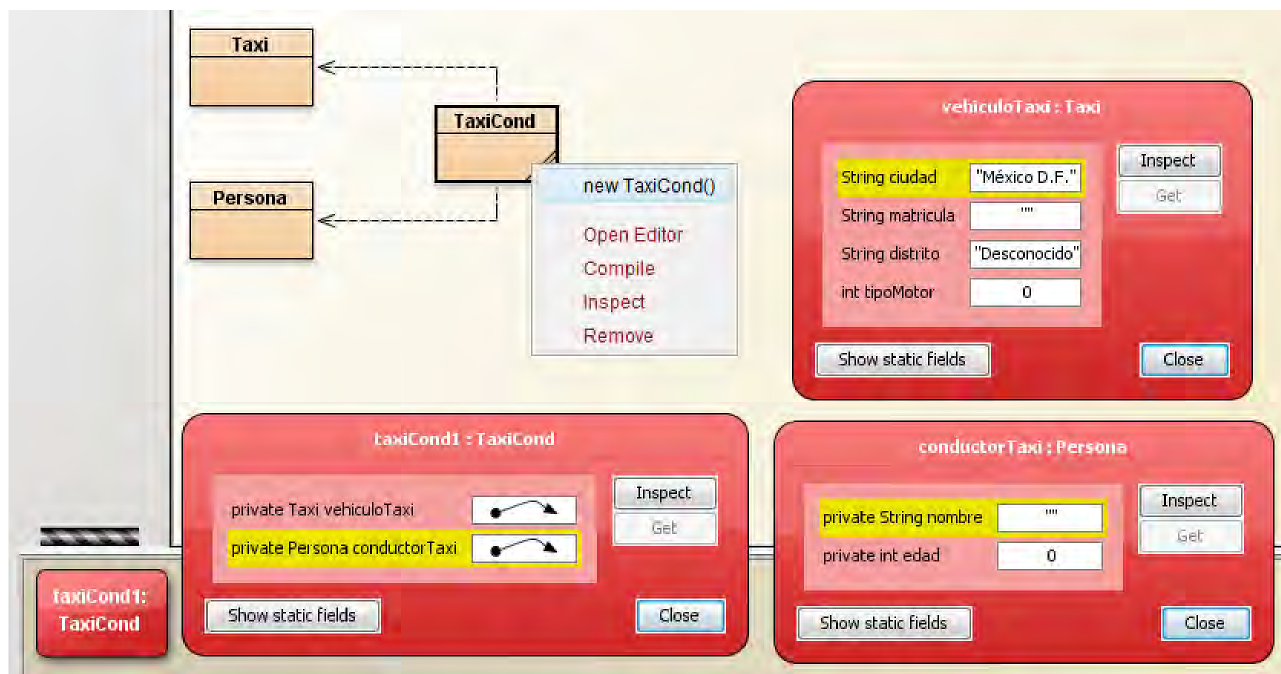
```
        if (tipoMotor ==0) { cadenaTipoMotor = "Desconocido"; }
        else if (tipoMotor == 1) { cadenaTipoMotor = "Gasolina"; }
        else if (tipoMotor == 2) { cadenaTipoMotor = "Diesel"; }

        String datosTaxiCond = "El objeto TaxiCond presenta estos datos. Matrícula: " + matricula +
          " Distrito: " + distrito + " Tipo de motor: " + cadenaTipoMotor;

        System.out.println (datosTaxiCond);
        return datosTaxiCond;
    } //Cierre del método

} //Cierre de la clase
```

Analicemos ahora lo que hace este código. Creamos una clase denominada TaxiCond (que podemos interpretar como "taxi con conductor"). Los objetos del tipo TaxiCond decimos que van a constar de dos campos: un objeto Taxi y un objeto Persona. Fíjate que estamos utilizando el nombre de otra clase como si fuera el tipo de una variable "normal y corriente". Esto es posible porque las clases definen tipos. Desde el momento en que nuestra clase utiliza tipos definidos por otras clases decimos que se establece una relación de uso: TaxiCond usa a Taxi y a Persona. El constructor de TaxiCond inicializa los objetos de este tipo para que consten de un objeto Taxi creado con el constructor por defecto y de una Persona creada con el constructor por defecto.

Para los objetos de tipo TaxiCond hemos definido dos métodos (podríamos haber definido muchos más) que son: el método modificador y con parámetros *setMatricula(String valorMatricula)* y el método observador y sin parámetros *getDatosTaxiCond()*. Un aspecto muy importante del código es que desde el momento en que usamos objetos en una clase, podemos acceder a los métodos públicos propios de esos objetos cuyo código se encontrará en otra clase. Por ejemplo la invocación *vehiculoTaxi.setMatricula(valorMatricula);* llama un método propio de los objetos Taxi que se encuentra definido en otra clase. Es decir, todo objeto puede llamar a sus métodos públicos independientemente de dónde se encuentre.

Una vez compilado el código, en el diagrama de clases se nos muestran unas flechas discontinuas que relacionan la clase Taxicond con las clases Taxi y Persona. Estas flechas discontinuas lo que indican es que hay una relación de uso entre las clases. En algunas circunstancias BlueJ puede mantener erróneamente indicadores de relación que no son ciertos. En estos casos, las flechas pueden eliminarse seleccionándolas y con botón derecho eligiendo la opción Remove. También pueden crearse eligiendo el botón ----> en la parte superior izquierda de la pantalla y a continuación pulsando primero el icono de la "clase que usa" y luego el icono de la "clase que es usada".

Crea un objeto de tipo TaxiCond pulsando sobre el icono de la clase y con botón derecho eligiendo new *TaxiCond()*. A continuación con botón derecho sobre el objeto elige la opción Inspect. La ventana que se nos muestra nos indica que el objeto consta de dos campos, pero en el recuadro correspondiente al valor de dichos campos en vez de un valor nos aparece una flecha curvada. Esta flecha lo que nos indica el que el campo no contiene un valor simple (como un entero) sino un objeto. La flecha **simboliza una referencia al objeto, ya que el objeto no se puede representar directamente** al ser una entidad compleja.



Pulsa sobre la flecha de referencia de cada uno de los campos y luego sobre el botón Inspect de la ventana. Se te abrirán otras dos ventanas donde puedes observar los valores de los campos de cada uno de los objetos que forman el objeto TaxiCond. Pero ten en cuenta que un objeto siempre podría tener como campo otro objeto, es decir, podríamos seguir observando "flechas" una y otra vez al ir inspeccionando objetos y esto sería una situación normal. Estamos trabajando con programación orientada a objetos, por tanto que aparezcan objetos "por todos lados" será normal.

La relación de uso entre clases es una de los tipos de relación más habituales en programación orientada a objetos. Las variables de instancia de un objeto pueden ser tanto de tipo primitivo como tipo objeto. Recordar que la variable que define un objeto no contiene al objeto en sí mismo, sino una referencia al espacio de memoria donde se encuentra. Dado que un objeto es una entidad compleja simbólicamente se representa con una línea que comienza en un punto y termina en una punta de flecha. Un objeto puede crearse e invocar sus métodos públicos desde distintas clases y decimos que esto establece una relación de uso entre clases. Por tanto, el código fuente de una clase puede ser usado desde otras clases.

Un esquema donde se representan las clases y las relaciones que existen entre ellas se denomina **diagrama de clases** y nos sirve para comprender la estructura de los programas. Se dice que el diagrama de clases constituye una vista estática del programa, porque es un esquema fijo de relaciones dentro del programa. Sin embargo, el que exista una relación entre clases no significa que en un momento dado vaya a existir un objeto que materialice esa relación entre clases. Es posible que el programa comience y que pase un tiempo antes de que se cree un objeto que refleje la relación entre

© aprenderaprogramar.com, 2006-2029

clases. Si representáramos los objetos existentes en un momento dado y las relaciones entre ellos tendríamos una vista dinámica del programa. El inconveniente de las vistas dinámicas es que los objetos se crean, destruyen o cambian sus relaciones continuamente, por lo que representarlas resulta costoso. Por este motivo, no utilizaremos las vistas dinámicas. Sin embargo, sí usaremos con frecuencia los diagramas de clases para comprender la estructura de nuestro código.

Nos queda una aclaración por realizar: ¿Por qué si los tipo String son objetos BlueJ nos informa directamente de su contenido en vez de mostrar una flecha? La razón para ello estriba en que el tipo String es un objeto un tanto especial, ya que su contenido es relativamente simple comparado con el de otros objetos. Para facilitar el trabajo BlueJ nos informa directamente de su contenido, pero no podemos olvidar que un String es un objeto y esto tiene una relevancia notable como veremos más adelante.

Para completar la comprensión de la relación de uso entre clases, utiliza los métodos disponibles para el objeto TaxiCond que has creado: establece distintos valores de matrícula con el método setMatricula y visualiza los datos del objeto con el método getDatosTaxiCond. Crea además nuevos métodos que te permitan establecer el distrito y el tipo de motor de los objetos TaxiCond. Te planteamos otra reflexión: al igual que hemos definido un tipo TaxiCond que tiene dos objetos como campos, podemos definir tipos que tengan cinco, diez, quince o veinte objetos como campos. Por ejemplo, un objeto Casa podría definirse con estos campos:

```
private Salon salonCasa;
private Cocina cocinaCasa;
private Baño baño1Casa;
private Baño baño2Casa;
private Jardin jardinCasa;
private Dormitorio dormitorio1Casa;
private Dormitorio dormitorio2Casa;
private Dormitorio dormitorio3Casa;
```

Ten en cuenta que **dentro de un objeto puedes tener n objetos de otro tipo.** En este ejemplo, dentro de un objeto Casa tenemos dos objetos de tipo Baño y tres objetos de tipo Dormitorio. Todos los objetos Dormitorio van a tener los mismos atributos y métodos, pero cada objeto tendrá su propio estado en cada momento.

EJERCICIO

Define tres clases: Casa, SalonCasa y CocinaCasa. La clase SalonCasa debe tener como atributos numeroDeTelevisores (int) y tipoSalon (String) y disponer de un constructor que los inicialice a 0 y "desconocido". La clase CocinaCasa debe tener como atributos esIndependiente (boolean) y numeroDeFuegos (int) y un constructor que los inicialice a false y 0. La clase Casa tendrá los siguientes

© aprenderaprogramar.com, 2006-2029

atributos de clase: superficie (double), direccion (String), salonCasa (tipo SalonCasa) y cocina (tipo CocinaCasa). Define un constructor para la clase Casa que establezca a unos valores de defecto los atributos simples y que cree nuevos objetos si se trata de atributos objeto. Compila el código para comprobar que no presenta errores, crea un objeto de tipo Casa. Comprueba que se inicializan correctamente consultando el valor de sus atributos después de haber creado los objetos. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00642B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

© aprenderaprogramar.com, 2006-2029