MEDAC
Instituto Oficial de Formación Profesional

Tarea 7

Antonio García Sánchez

Define cuáles son los pilares de la POO.

¿Qué es la Programación Orientada a Objetos?

Se trata de un paradigma de programación, es decir, un modelo o un estilo de programación. Se basa en el concepto de clases y objetos. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de planos de código (clases) para crear instancias individuales de objetos.

A lo largo de la historia, han ido apareciendo diferentes paradigmas de programación. Lenguajes secuenciales como COBOL o procedimentales como Basic o C, se centraban más en la lógica que en los datos. Otros más modernos como Java, C# y Python, utilizan paradigmas para definir los programas, siendo la Programación Orientada a Objetos la más popular.

Con el paradigma de Programación Orientado a Objetos lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma. Esto nos ayuda muchísimo en sistemas grandes, ya que en vez de pensar en funciones, pensamos en las relaciones o interacciones de los diferentes componentes del sistema.

Un programador diseña un programa de software organizando piezas de información y comportamientos relacionados en una plantilla llamada clase. Luego, se crean objetos individuales a partir de la plantilla de clase. Todo el programa de software se ejecuta haciendo que varios objetos interactúen entre sí para crear un programa más grande.

Una vez que hemos visto que es la POO pasemos a ver sus pilares fundamentales que son 4:

- ➔ **Abstracción**

De acuerdo a la RAE, una de las acepciones de abstraer es: “Hacer caso omiso de algo, o dejarlo a un lado.”

Cuando hacemos una abstracción, queremos omitir detalles que no son necesarios para nosotros, y queremos solamente mostrar lo que sí es relevante.

Desde el punto de vista del desarrollo de software, podemos ver que con una clase podemos realizar una abstracción de una entidad del mundo real. Tomemos por ejemplo la clase coche,

esta tiene la posibilidad de guardar datos relacionados a la marca y al año de salida al mercado del coche, pero, ¿Por qué solamente estas dos informaciones? Un coche del mundo real tiene más propiedades, como el color y el modelo. Sin embargo, debemos preguntarnos, ¿Son estas informaciones relevantes para nuestro software? Nuestra clase abstrae todo lo que representa un coche, tomando solamente lo que nos interesa, descartando todo lo demás.

Podemos decir entonces que la abstracción es una técnica que consiste en presentar sólo detalles esenciales al usuario ocultando detalles innecesarios o irrelevantes de un objeto. Ayudando a reducir la complejidad operativa por parte del usuario. La abstracción nos permite proporcionar una interfaz simple al usuario sin solicitar detalles complejos para realizar una acción. En pocas palabras, le brinda al usuario la capacidad de conducir un automóvil sin necesidad de comprender exactamente cómo funciona el motor.

- ➔ **Encapsulación**

El encapsulamiento es el pilar de la programación orientada a objetos que se relaciona con ocultar al exterior determinados estados internos de un objeto, tal que sea el mismo objeto quien acceda o los modifique, pero que dicha acción no se pueda llevar a cabo desde el exterior, llamando a los métodos o atributos directamente.

Podemos decir que encapsulamiento es una forma de ocultación de información entre entidades, mostrándose entre ellas solo la información más necesaria. El encapsulamiento va ligado a los modificadores de acceso, nosotros hemos visto 3

1. Public acceso total
2. Private, acceso únicamente desde la clase que los contiene
3. Protected desde la misma clase o desde las clases que heredan

Aunque existen 3 más

1. Internal, acceso desde el mismo Proyecto
2. Protected internal, combina protected e internal y nos permite acceder desde el mismo Proyecto o de clases que la derivan.
3. Private protected, que permite acceder desde la clase actual o de la que derivan de ella.

➔ **Polimorfismo**

Se trata de la capacidad de un objeto para adquirir múltiples formas o comportamientos. Es como tener una pieza de Lego que puede encajar en varios lugares o tener varios usos según el contexto.

En la programación, esto se traduce en la habilidad de una clase para implementar métodos con el mismo nombre pero con comportamientos distintos, dependiendo de la clase específica con la que se esté interactuando. Esto añade flexibilidad y extensibilidad a tu código, permitiendo adaptarse a diferentes situaciones y requerimientos.

Por lo que he visto el ejemplo típico con el que se explica el polimorfismo son las formas geométricas, ya que el polimorfismo permite calcular áreas independientemente del tipo de forma.

Por ejemplo la clase base llamada Figura encapsula el concepto general de una forma geométrica. Cuando se crean dos dos clases derivadas, Circulo y Rectangulo, que heredan de la clase base Figura. Cada una de estas clases implementa su propia lógica para calcular el área, mostrando así el principio de polimorfismo. Por lo tanto *podemos resumir el polimorfismo diciendo que consiste en diseñar objetos para compartir comportamientos, lo que nos permite procesar objetos de diferentes maneras*

➔ **Herencia**

Permite la creación de nuevas clases basadas en clases existentes, aprovechando y heredando tanto sus atributos como sus métodos. Este concepto se traduce en la reutilización del código y en la construcción progresiva de jerarquías de clases

Ejemplo: tenemos una clase base llamada TarjetaCredito que incluye atributos y métodos comunes a todas las tarjetas de crédito. Estos podrían incluir propiedades como el nombreTitular, numeroTarjeta y fechaExpiracion. Además, podríamos tener métodos como realizarPago(). Si queremos introducir una nueva clase, por ejemplo, TarjetaPremium, que comparte características con la clase base TarjetaCredito pero también tiene funcionalidades específicas adicionales, podemos heredar de la clase base usando la palabra reservada **Extends**

¿Qué son los principios SOLID? ¿Para qué sirven?

Son 5 y se tratan de un conjunto de reglas y buenas prácticas a seguir al diseñar una estructura de clase. Todos tienen el mismo propósito:

"Crear código comprensible, legible y comprobable en el que muchos desarrolladores puedan trabajar en colaboración."

Siguiendo el acrónimo inglés SOLID, son:

El Principio de responsabilidad única (**S**ingle Responsibility Principle)

El Principio Abierto-Cerrado (**O**pen-Closed Principle)

El Principio de sustitución de Liskov (**L**iskov Substitution Principle)

El Principio de segregación de interfaz (**I**nterface Segregation Principle)

El Principio de inversión de dependencia (**D**ependency Inversion Principle)

Principio de responsabilidad única

El Principio de Responsabilidad Única dice que una clase debe hacer una cosa y, por lo tanto, debe tener una sola razón para cambiar.

Esto significa que si una clase es un contenedor de datos, como una clase Libro o una clase Estudiante, y tiene algunos campos relacionados con esa entidad, debería cambiar solo cuando cambiamos el modelo de datos.

Es importante seguir el principio de responsabilidad única. En primer lugar, debido a que muchos equipos diferentes pueden trabajar en el mismo proyecto y editar la misma clase por diferentes motivos, esto podría dar lugar a módulos incompatibles.

Veamos un ejemplo sacado de Gemini para poder entender mejor este principio

```
public class Coche {
    private String marca;
    private String modelo;
    private String color;

    public Coche(String marca, String modelo, String color)
 {
        this.marca = marca;
        this.modelo = modelo;
        this.color
 = color;
    }

    public void arrancar() {
        System.out.println("El coche está arrancando");
    }

    public void acelerar() {
        System.out.println("El coche está acelerando");
    }

    public void frenar() {
        System.out.println("El coche está frenando");
    }
}

public
 class Motor {
    public double calcularConsumo(double velocidad, double tiempo) {
        // Lógica para calcular el consumo de combustible
        // ...
        return 0.0; // Ejemplo de retorno
    }
}

public class Puertas {
    public void abrir() {
        System.out.println("La puerta se ha abierto");
    }

    public void cerrar() {
        System.out.println("La puerta se ha cerrado");
    }
}
```

Como se puede observar cada clase cumple un cometido concreto lo cual es útil por 3 motivos:

Mantenibilidad: Si necesitas modificar la forma de calcular el consumo de combustible, solo cambias la clase Motor.

Reusabilidad: La clase Motor podría ser reutilizada en otros vehículos.

Claridad: Cada clase tiene una responsabilidad clara y concisa, lo que facilita la comprensión del código.

Principio de apertura y cierre

Este principio de apertura y cierre exige que las clases deben estar abiertas a la extensión y cerradas a la modificación.

Modificación significa cambiar el código de una clase existente y extensión significa agregar una nueva funcionalidad.

Entonces, lo que este principio quiere decir es: Deberíamos poder agregar nuevas funciones sin tocar el código existente para la clase. Esto se debe a que cada vez que modificamos el código existente, corremos el riesgo de crear errores potenciales. Por lo tanto, debemos evitar tocar el código de producción probado y confiable (en su mayoría) si es posible.

Pero, ¿cómo vamos a agregar una nueva funcionalidad sin tocar la clase?, puede preguntarse. Por lo general, se hace con la ayuda de interfaces y clases abstractas.

Otro ejemplo

Si estamos desarrollando un sistema de facturación para una empresa. Inicialmente, solo emitimos facturas electrónicas. Nuestra clase Factura podría verse así:

Java

```
public class Factura {
    // ... otros atributos

    public void generarPDF() {
        // Lógica para generar un PDF de la factura
    }
}
```

Ahora, la empresa decide que también quiere emitir facturas en formato XML. Si modificamos directamente la clase Factura para agregar un método generarXML, estaríamos violando el principio de apertura y cierre, ya que estaríamos modificando una clase existente para añadir una nueva funcionalidad.

Para resolver este problema, podemos utilizar interfaces y herencia:

```java
public interface FormatoFactura {
    void generar();
}

public class FacturaPDF implements FormatoFactura {
    @Override
    public void generar() {
        // Lógica para generar un PDF de la factura
    }
}

public class FacturaXML implements FormatoFactura {
    @Override
    public void generar() {
        // Lógica para generar un XML de la factura
    }
}

public class Factura {
    // ... otros atributos
    private FormatoFactura formato;

    public Factura(FormatoFactura formato) {
        this.formato = formato;
    }

    public void generar() {
        formato.generar();
    }
}
```

¿Qué es una interface? Una interfaz en Java define un conjunto de métodos que una clase debe implementar. Es una forma de abstraer (centrarse) en las características comunes de diferentes objetos.

¿Qué es la herencia? La herencia en Java es un mecanismo que permite crear nuevas clases (subclases) a partir de clases existentes (superclases), heredando sus propiedades y métodos.

Principio de sustitución de Liskov

Este Principio establece que las subclases deben ser sustituibles por sus clases base. Este principio nos dice que si tenemos una clase base (padre), cualquier clase hija (hija) debe poder ser utilizada en lugar de la clase base sin que se rompa el programa. Con un ejemplo se entiende mejor.

```java
// Clase base
public class Animal {
    public void hacerSonido() {
        System.out.println("Hace un sonido");
    }
}

// Clase hija
public class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Guau!");
    }
}

// Clase hija
public class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Miau!");
    }
}

// Clase que utiliza animales
public class Zoo {
    public void alimentar(Animal animal) {
        System.out.println("Alimentando al animal");
        animal.hacerSonido();
    }
}
```

Animal es la clase base. Perro y Gato son clases hijas que heredan de Animal. Zoo tiene una función alimentar que recibe un objeto de tipo Animal.

Si creamos un objeto Perro o Gato y lo pasamos a la función alimentar, el código funcionará correctamente porque tanto el perro como el gato son tipos de animales. El LSP se cumple aquí porque podemos sustituir un Animal por un Perro o un Gato sin problemas.

Principio de segregación de interfaces

Este principio nos dice que las interfaces que definimos deben ser lo más específicas posibles y que ninguna clase cliente debería verse obligada a depender de métodos que no utiliza.

Se puede explicar así: Imagina que tienes una interfaz que define todas las acciones que puede realizar un dispositivo electrónico: encender, apagar, ajustar el volumen, conectar a internet, etc. Si tienes una clase que representa un reloj, ¿tiene sentido que esta clase implemente todos estos métodos? Obviamente, un reloj no necesita conectarse a internet, sin embargo un router si. El ISP nos ayuda a evitar este tipo de situaciones

Así se vería visualmente

```
// Interfaz genérica (a evitar según el ISP)
interface DispositivoElectronico {
    void encender();
    void apagar();
    void ajustarVolumen();
    void conectarAInternet();
}

// Interfaz más específica
interface ReproductorAudio {
    void encender();
    void apagar();
    void ajustarVolumen();
}

// Interfaz más específica
interface DispositivoRed {
    void conectarAInternet();
}

// Clase que implementa la interfaz ReproductorAudio
class Radio implements ReproductorAudio {
    // Implementa los métodos de ReproductorAudio
}

// Clase que implementa la interfaz DispositivoRed
class Router implements DispositivoRed {
    // Implementa los métodos de DispositivoRed
}
```

Principio de inversión de dependencia

Este principio establece que las clases de alto nivel (las que contienen la lógica principal de tu aplicación) no deben depender de las clases de bajo nivel (las que implementan detalles específicos, como acceso a bases de datos o redes). En su lugar, ambas deben depender de abstracciones

Ejemplo: Imaginemos que queremos crear una máquina expendedora simple. La máquina necesita una forma de procesar pagos y entregar productos.

```java
// Interfaz (abstracción)
interface Pagador {
    boolean procesarPago(double cantidad);
}

// Implementación concreta (clase de bajo nivel)
class PagadorEfectivo implements Pagador {
    @Override
    public boolean procesarPago(double cantidad) {
        // Simulación de procesamiento de pago en efectivo
        System.out.println("Procesando pago en efectivo de $" + cantidad)
        return true; // Simulación de pago exitoso
    }
}

// Clase de alto nivel
class MaquinaExpendedora {
    private Pagador pagador;

    // Constructor que recibe la interfaz
    public MaquinaExpendedora(Pagador pagador) {
        this.pagador = pagador;
    }

    public void venderProducto(double precio) {
        if (pagador.procesarPago(precio)) {
            System.out.println("Entregando producto");
        } else {
            System.out.println("Pago rechazado");
        }
    }
}
```

La explicación sería la siguiente:

Pagador: Es la interfaz que define el comportamiento de un sistema de pago. Cualquier clase que implemente esta interfaz puede ser utilizada por la máquina expendedora.

PagadorEfectivo: Es una implementación concreta de Pagador que simula el procesamiento de pagos en efectivo.

Máquina Expendedora: Es la clase principal que utiliza un objeto Pagador para procesar pagos y vender productos.

¿Cómo funciona el DIP y la inyección de dependencias aquí?

La clase MaquinaExpendedora no está directamente acoplada a una implementación específica de Pagador. En cambio, depende de la interfaz Pagador.

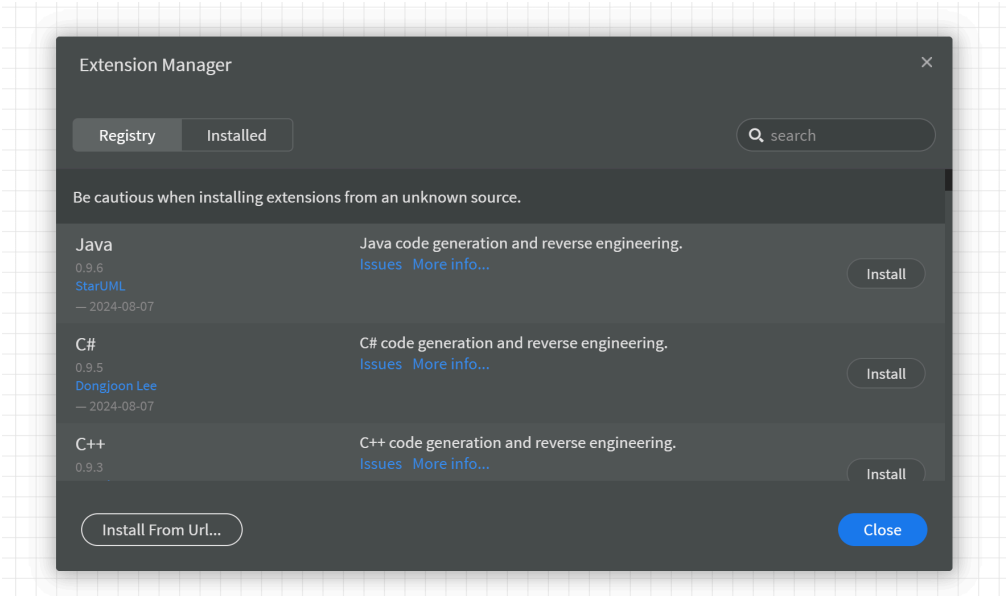
La dependencia se inyecta en el constructor de Máquina Expendedora, lo que permite configurar la máquina con diferentes sistemas de pago en tiempo de ejecución.

Si quisiéramos agregar un nuevo método de pago, como una tarjeta de crédito, solo tendríamos que crear una nueva clase que implemente la interfaz Pagador y pasar una instancia de esa clase al constructor de Máquina Expendedora.

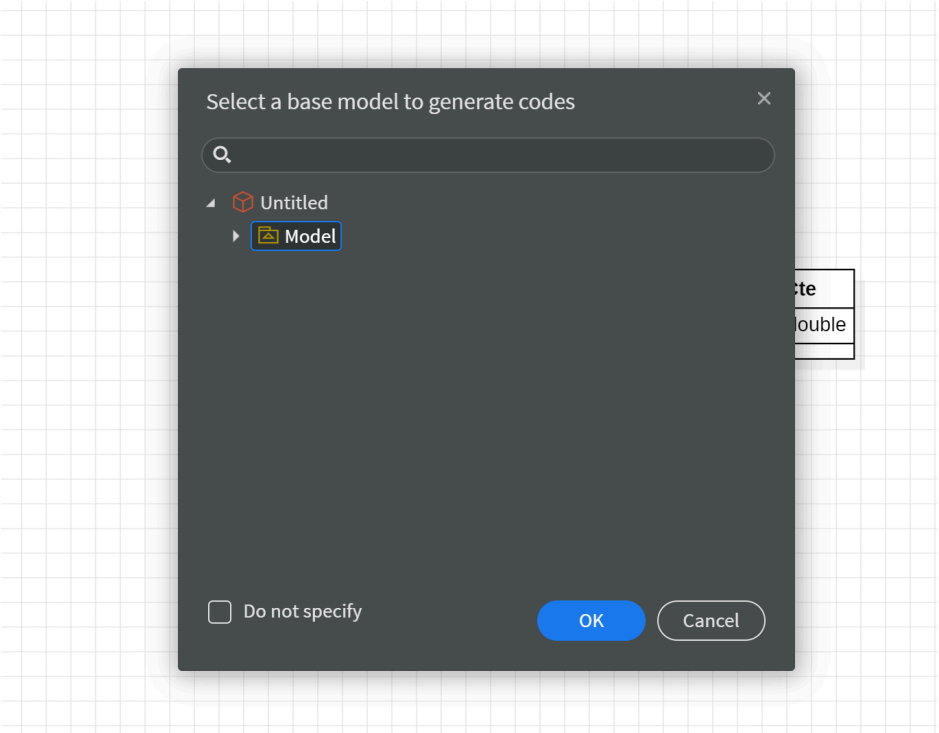
¿Cómo pasar de un diagrama de clases a código en STARUML?

Vamos a ver como se hace con un ejemplo sencillo donde tenemos dos clases la clase Cliente y la clase CuentaCliente. Lo primero sería ir a tools (herramientas) y una vez allí pulsar sobre extension manager

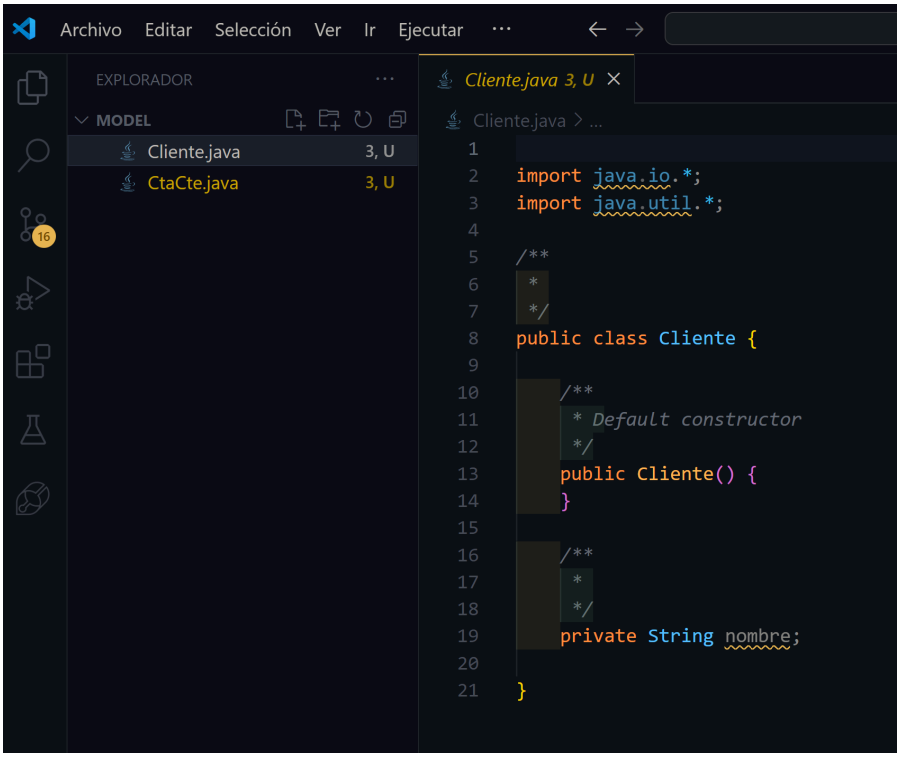
El siguiente paso es instalar la extensión para el lenguaje que se quiera (en este caso Java)

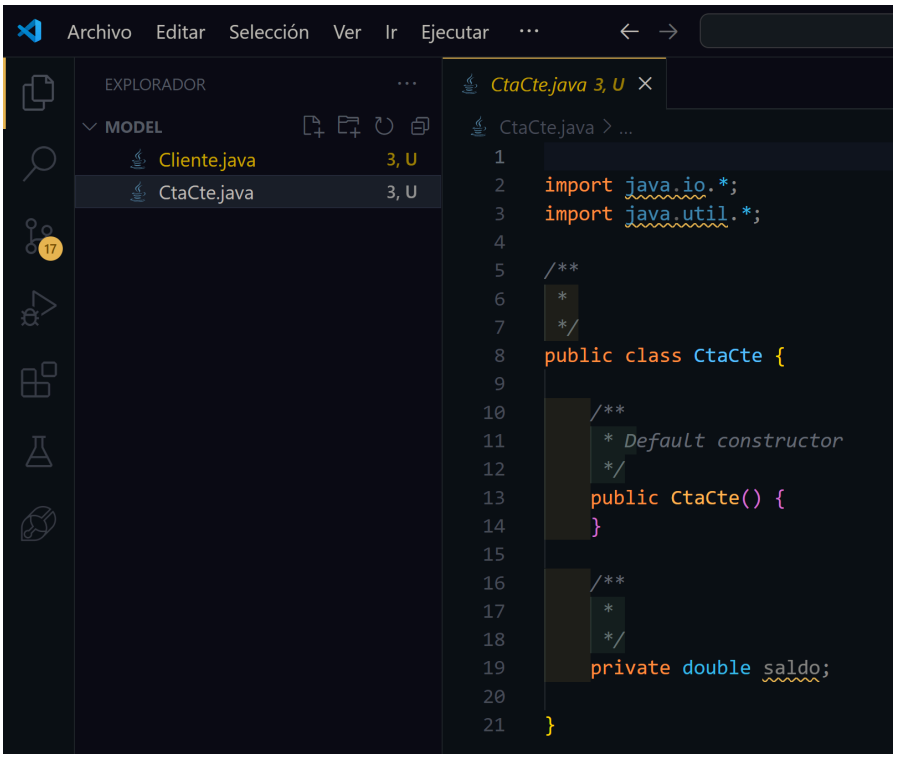


Una vez instalada la extensión y reiniciado el programa saldrá en tools la opción de java y dentro de ella generate code, pulsamos y seleccionamos model nos dirá en qué carpeta queremos guardar el archivo o archivos .java que contendrán el diagrama traspasado a código fuente.



Una vez que se tengan localizados lo único que queda es abrirlos con el ide.


Archivo Editar Selección Ver Ir Ejecutar
EXPLORADOR
MODEL
Cliente.java 3, U
CtaCte.java 3, U
Cliente.java 3, U
Cliente.java > ...
import java.io.*;
import java.util.*;
/**
*
*/
public class Cliente {
/**
* Default constructor
*/
public Cliente() {
}
/**
*
*/
private String nombre;
}


Archivo Editar Selección Ver Ir Ejecutar
EXPLORADOR
MODEL
Cliente.java 3, U
CtaCte.java 3, U
CtaCte.java 3, U
CtaCte.java > ...
import java.io.*;
import java.util.*;
/**
*
*/
public class CtaCte {
/**
* Default constructor
*/
public CtaCte() {
}
/**
*
*/
private double saldo;
}