UT4: PROGRAMACIÓN ORIENTADA A OBJETOS

Contenido

1. Programación Orientada a Objeto (POO)

La programación Orientada a Objetos es una metodología que basa la estructura de los programas en torno a los objetos.

Los lenguajes de POO ofrecen medios y herramientas para describir los objetos manipulados por un programa. Más que describir cada objeto individualmente, estos lenguajes proveen una construcción (Clase) que describe a un conjunto de objetos que poseen las mismas propiedades.

2. Clase.

Una clase es un tipo definido por el usuario (plantilla) que describe los atributos y los métodos de los objetos que se crearán a partir de ella.

El estado de un objeto viene determinado por sus **atributos,** y su comportamiento está definido por sus **métodos.**

Dentro de las clases también se encuentran los **constructores** que permiten inicializar un objeto.

Los atributos y los métodos se denominan en general **miembros de la clase**.

La definición de una clase consta de dos partes: el nombre de la clase precedido por la palabra reservada **class** y el cuerpo de la clase entre llaves. La sintaxis queda:

```
class nombre-clase {
  cuerpo de la clase
}
```

Dentro del cuerpo de la clase se puede encontrar los atributos y métodos.

Ej:

```
class Circunferencia {
        private double x, y, radio;
        public Circunferencia(){ }

        public Circunferencia(double cx, double cy, double r){
                x=cx;
                y=cy;
                radio=r;
        }
        public void ponRadio(double r){
                radio=r;
        }
        public double longitud(){
                return 2*Math.PI*radio;
        }
}
```

En el ejemplo, se define la **clase Circunferencia**, que puede ser usada dentro de un programa de la misma manera que cualquier otro tipo. Un objeto de esta clase tiene tres atributos (coordenadas del centro y valor del radio), dos constructores y dos métodos.

Los constructores se distinguen fácilmente porque tienen el mismo nombre que la clase.

Los atributos se declaran de la misma manera que cualquier variable. En una clase, cada atributo debe tener un nombre único.

Siguiendo las recomendaciones de la Programación Orientada a Objetos, cada clase se debe implementar en un fichero .java, de esta manera es más sencillo modificar la clase.

El fichero que contiene la clase debe llevar el nombre de la clase pública con la extensión .java.

3. Métodos

Los métodos forman lo que se denomina interfaz de los objetos, definen las operaciones que se pueden realizar con los atributos. Desde el punto de vista de la Programación Orientada a Objetos, el conjunto de métodos se corresponde con el conjunto de mensajes a los que los objetos de una clase pueden responder.

Los métodos implementan la funcionalidad asociada al objeto. Los métodos son el equivalente a las funciones en Programación Estructurada. Se diferencian de ellos en que es posible **acceder** a las **variables de la clase de forma implícita** (atributos).

Cuando se desea realizar una acción sobre un objeto, se dice que se le manda un mensaje invocando a un método que realizará la acción.

Los métodos permiten al programador modularizar sus programas.

Todas las variables declaradas en las definiciones de métodos son variables locales, solo se conocen en el método que las define. Casi todos los métodos tienen una lista de parámetros que permiten comunicar información.

La sintaxis para definir un método es:

```
<modificador-acceso> <modificador > tipoR nombre-método(<parámetros>){
<cuerpodelmétodo>
}
```

Donde <modificador-acceso> indica cómo es el acceso a dicho método: public, private, protected y sin modificador.

<modificador > indica si es estática o no.

<tipoR> es el tipo de dato que retorna el método. Es obligatorio indicar un tipo. Para los métodos que no devuelven nada se utiliza la palabra reservada *void*.

Nombre-método es cómo el programador quiere llamar a su método y <parámetros> son los datos que se van a enviar al método para trabajar con ellos, no son obligatorios.

Para devolver el valor se utiliza el operador **return**. Una vez que se ejecute return, el control se devuelve al código que lo llamó y la ejecución del método termina. Los métodos que no lleven return, tendrán tipo void.

Ej:

```
int suma(int x, int y) {
        return x+y;
}
```

En este ejemplo el método recibe 2 parámetros de tipo int, realiza su suma y devuelve esta suma, que es un valor de tipo int.

```
void imprimir(){
        System.out.println("Este método no devuelve nada y tampoco
        recibe parámetros");
}
```

En este ejemplo el método no devuelve ningún valor, ni recibe parámetros, aun así es necesario poner los paréntesis vacíos.

3.1 Métodos Estáticos o de Clase

Van precedidos del modificador **static**.

Para invocar a un **método** estático no se necesita crear un objeto de la clase en la que se define:

- Si **se invoca desde la clase** en la que se encuentra definido, basta con escribir su nombre.
- Si **se le invoca desde una clase distinta**, debe anteponerse a su nombre, el de la clase en que se encuentra seguido del operador punto (.) La sintaxis es:

  < NombreClase> .metodoEstatico( );

Suelen emplearse para realizar operaciones comunes a todos los objetos de la clase. No afectan a los estados de los objetos.

Por ejemplo, si se necesita un método para contabilizar el número de objetos creados de una clase, se define estático ya que su función, aumentar el valor de una variable entera, se realiza independientemente del objeto empleado para invocarle. Esa variable entera, sería única para todos los objetos (la definimos como estática también).

Otra razón por la que tendríamos que usar métodos estáticos es si se utilizan fuera del contexto de cualquier instancia

Los métodos estáticos solo pueden llamar a otros métodos static directamente, y no se pueden referir a **this** o **super** de ninguna manera.

No es conveniente usar muchos métodos estáticos, pues si bien se aumenta la rapidez de ejecución, se pierde flexibilidad, no se hace un uso efectivo de la memoria y no se trabaja según los principios de la POO.

Las **variables/atributos** también se pueden declarar como static, y es equivalente a declararlas como variables globales, que son accesibles desde cualquier fragmento de código.

Se puede declarar un **bloque static** que se ejecuta una sola vez si se necesita realizar cálculos para inicializar las variables static.

Es decir, el bloque de inicialización estático te permite inicializar los miembros *estáticos* de la clase, algo que no se puede hacer con los constructores normales.

Ej:

```
class Estatica {
        static int a=3, b;
        static {
                System.out.println("Bloque static inicializado");
                b=a*4;
        } // Si no es static no compilará
        static void metodo(int x){
                System.out.println("x= "+x);
                System.out.println("a= "+a);
                System.out.println("b= "+b);
        }

        public static void main(String[] args){
                metodo(42);
        }
}
```

En el ejemplo la clase que tiene dos variables static, un bloque de inicialización static y un método static. La salida del programa es:

```
Bloque static inicializado:
x = 42
a = 3
b = 12
```

3.2 Modificadores de acceso

Una de las formas en que implementaremos la **encapsulación de datos** es definiendo los atributos privado y permitiendo su uso mediante la utilización de accesores (getters) y mutadores (setters). El papel de los accesores y mutadores es devolver y establecer los valores del estado de un objeto. Veamos como programar accesores y mutadores en Java .

Ejemplo:

```
public class Persona {
        private String nombre;
        private String apellido;
}
```

Métodos de acceso (“métodos getter”)

Se utiliza un método de acceso para devolver el valor de un campo privado. Podríamos, por ejemplo, anteponer la palabra "get" al comienzo del nombre del método. Por ejemplo, agreguemos métodos de acceso para nombre, segundo nombre y apellido:

```
public class Persona {
        private String nombre;
        private String apellido;

        public String getNombre() {
                return nombre;
        }

        public String getApellido() {
                return apellido;
        }

}
```

Estos métodos siempre devuelven el mismo tipo de datos que su campo privado correspondiente (por ejemplo, String) y luego simplemente devuelven el valor de ese campo privado.

Métodos mutadores (“métodos setter”)

Se utiliza un método mutador para establecer un valor de un campo privado. En este caso podemos anteponer la palabra "set" al comienzo del nombre del método. Por ejemplo, agreguemos campos mutantes para el nombre y apellido:

```
public class Persona {
	private String nombre;
	private String apellido;

	public String getNombre() {
		return nombre;
	}

	public String getApellido() {
		return apellido;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public void setApellido(String apellido) {
		this.apellido = apellido;
	}


}
```

Estos métodos no tienen un tipo de devolución y aceptan un parámetro que es del mismo tipo de datos que su campo privado correspondiente. Luego, el parámetro se usa para establecer el valor de ese campo privado.

Ahora es posible modificar los valores del nombre de usuario y apellido dentro de un objeto Persona:

```
Person p = new Persona();

p.setNombre("Pepe");

p.setApellido("Sanchidrian");
```

Y también recoger esos valores:

```
String nomPersona = p.getNombre();

String apePersona = p.getApellido();
```

¿Por qué usar accesorios y mutadores?

Podemos pensar que podríamos simplemente cambiar los campos privados de la definición de clase para que sean públicos y lograr los mismos resultados. Hay que recordar que queremos ocultar los datos del objeto tanto como sea posible. Además, el uso de los métodos vistos nos permite:

- Cambiar la forma en que se guardan los datos recibidos en los métodos setters.
- Imponer la validación de los valores antes de colocarlos en nuestros atributos.

Por ejemplo, podemos decidir que el objeto Persona solo puede aceptar nombres de usuario que tengan un máximo de diez caracteres. Agregamos validación en el mutador setNombre para asegurarnos de que el nombre de usuario cumpla con este requisito:

```
public void setApellido(String apellido) {
        if (apellido.length() >10)
                this.apellido = apellido.substring(0,10);
}
```

Ahora, si el nombre de usuario pasado al mutador setNombre tiene más de diez caracteres, se trunca automáticamente.

Se profundizará sobre los modificadores de acceso cuando veamos Herencia y Paquetes en Java. De momento, pondremos que un atributo/método es privado (**private**) si solo lo utilizamos dentro de su clase, si lo queremos usar fuera lo pondremos público (**public**).

Un adelanto:

Modificadores de acceso

	La misma clase	Otra clase del mismo paquete	Subclase de otro paquete	Otra clase de otro paquete
public	X	X	X	X
protected	X	X	X	
default	X	X		
private	X			

4 Objetos

Es una entidad (tangible o intangible) que posee características y acciones que realiza por sí solo o interactuando con otros objetos.

Un **objeto** es una entidad caracterizada por sus atributos propios y cuyo comportamiento está determinado por las acciones o funciones que pueden modificarlo (métodos), así como también las acciones que requiere de otros objetos.

Un objeto constituye una unidad que **oculta** tanto datos como la descripción de su manipulación. Puede ser definido como una **encapsulación** y una **abstracción**: una encapsulación de atributos y servicios, y una abstracción del mundo real.

Para el contexto del Enfoque Orientado a Objetos (EOO) un objeto es una entidad que encapsula datos (atributos) y acciones o funciones que los manejan (métodos). También para el EOO un objeto se define como una instancia o particularización de una clase.

Los objetos de interés durante el desarrollo de software no solo son tomados de la vida real (objetos visibles o tangibles), también pueden ser abstractos. En general, son entidades que juegan un rol bien definido en el dominio del problema. Un libro, una persona, un coche, un polígono son algunos **ejemplos** de objeto.

Cada objeto puede ser considerado como un proveedor de servicios utilizados por otros objetos que son sus clientes. Cada objeto puede ser a la vez proveedor y cliente. De ahí que un programa pueda ser visto como un conjunto de relaciones entre proveedores clientes. Los servicios ofrecidos por los objetos son de dos tipos:

1.- Los datos, que llamamos **atributos**.
2.- Las acciones o funciones, que llamamos **métodos**.

Un objeto consta de una estructura interna (los atributos) y de una interfaz que permite acceder y manipular dicha estructura (los métodos). Para **construir un objeto de una clase** cualquiera hay que llamar a un método de iniciación, **el constructor**. Para ello, se utiliza el operador **new**. La sintaxis es la siguiente:

Nombre-clase nombre-objeto=new Nombre-clase (<valores>);

Donde **Nombre-clase** es el nombre de la clase de la cual se quiere crear el objeto, **nombre-objeto** es el nombre que el programador quiere dar a ese objeto y <valores> son los valores con los que se inicializa el objeto. Dichos valores son opcionales y aparecen en el constructor.

Ej: Circunferencia circ = new Circunferencia();

Se crea el objeto circ, de la clase Circunferencia, con los valores predeterminados.

Cuando se crea un objeto, Java hace lo siguiente:

- **Asignar memoria** al objeto por medio del operador ***new***.
- **Llamar al constructor** de la clase para inicializar los atributos de ese objeto con los valores iniciales o con los valores predeterminados por el sistema: los atributos numéricos a cero, los alfanuméricos a nulos y las referencias a objetos a null.

Si no hay suficiente memoria para ubicar el objeto, el operador new lanza una excepción OutOfMemoryError.

Para acceder desde un método de una clase a un miembro de un objeto de otra clase diferente se utiliza la sintaxis: **objeto.miembro**. Cuando el miembro accedido es un método se entiende que el objeto ha recibido un mensaje, el especificado por el nombre del método y responde ejecutando ese método. En este caso, después del nombre del método siempre se pone paréntesis, aunque no haya parámetros.

Características Generales:

- Un objeto posee **estados**. El estado de un objeto está determinado por los valores que poseen sus atributos en un momento dado.

- Un objeto tiene un **conjunto de métodos**. El comportamiento general de los objetos dentro de un sistema se describe o representa mediante sus operaciones o métodos. Los métodos se utilizarán para obtener o cambiar el estado de los objetos, así como para proporcionar un medio de **comunicación** entre objetos.

- Un objeto tiene un **conjunto de atributos**. Los atributos de un objeto contienen valores que determinan el estado del objeto durante su tiempo de vida. Se implementan con variables, constantes y estructuras de datos (similares a los campos de un registro).

4.1 Asignación

Una vez el objeto está creado, se tiene una **referencia a ese objeto en la variable** donde se asigna. Si se realiza la asignación de un objeto a otro los dos harán referencia al mismo objeto.

Ej:

```
Circunferencia c1 = new Circunferencia(0, 0, 15);
Circunferencia c2 = c1;
c1.ponRadio(25);
System.out.println(c2.radio); // Es el mismo objeto
```

En el ejemplo, la variable c1 apunta a un objeto de la clase Circunferencia. Debido a la asignación, la variable c2 apunta al mismo objeto. Después se modifica el valor del radio del objeto apuntado por c1.
Y por último, se visualiza c2 que será el valor 25.

4.2 Igualdad

Creamos dos objetos iguales (con los mismos valores de sus variables miembro), al preguntar si las variables que apuntan a esos objetos son iguales nos devolverá false, pues aunque tengan el mismo valor no son el mismo objeto.

Ej:
Circunferencia c1 = new Circunferencia(0,0,15);
Circunferencia c2 = new Circunferencia(0,0,15);
if (c1==c2) **/* Esto es falso*/** == Direccion de los objetos

funcionan como los String .equals

4.3 Constructor.

Un Constructor es un método especial en Java empleado para inicializar valores en instancias de objetos. A través de este tipo de métodos es posible generar distintos tipos de instancias para la clase en cuestión.

Los métodos constructores tienen las siguientes **características**:

- Se llaman igual que la clase.
- No devuelven nada, ni siquiera void.
- Puede haber varios constructores, que deberán distinguirse por el tipo de valores que reciban.
- De entre los que existan, solo uno se ejecutará al crear el objeto.
- El código de un constructor, generalmente, suele ser inicializaciones de variables y objetos, para conseguir que el objeto sea creado con dichos valores iniciales.
- Si no se define ningún constructor el compilador crea uno **por defecto** sin parámetros que, al ejecutarse, inicializa el valor de cada atributo de la nueva instancia a 0, false o null, dependiendo de si el atributo es numérico, alfanumérico o una referencia a otro objeto respectivamente, pero dicho constructor desaparece en el mismo momento en que se defina otro constructor, por lo que **si se quiere tener el de por defecto habrá que definirlo**.

La **declaración de constructores** sigue la siguiente sintaxis:

```
<modificador deVisibilidad> NombredelaClase ( <argumentos> ) {
        < declaraciones>
}
```

Donde <modificadorvisibilidad> es el tipo de modificador de acceso del constructor, <Nombredelaclase> es el nombre del constructor y debe coincidir con el de la clase y <argumentos> son las variables que recibe el constructor y contienen los valores con los que se inicializaran los atributos.

Ej:

```
class Circunferencia{
        private double x, y, radio;
        public Circunferencia(){ }
        public Circunferencia(double cx, double cy, double r){
                x=cx; y=cy; radio=r;
        }
}
```

En este ejemplo existen 2 constructores, el primero que es el de por defecto y el segundo que inicializa los atributos x, y y radio con los valores cx, cy y r, respectivamente.

4.4 Referencia this

Java incluye un valor de referencia especial llamado **this**, que se utiliza dentro de cualquier método para referirse al objeto actual. El valor this se refiere al objeto sobre el que ha sido llamado el método actual.

Se puede utilizar this siempre que se requiera una referencia a un objeto del tipo de una clase actual. Si hay dos objetos que utilicen el mismo código, seleccionados a través de otras instancias, cada uno tiene su propio valor único de this.

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a las variables miembros del objeto. Sin embargo, algunas veces no se querrá tener ambigüedad sobre el nombre de la variable miembro y uno de los argumentos del método que tengan el mismo nombre y usaremos la referencia this.

Ej:

```
        class Circunferencia {
                private double x, y, radio;
                public Circunferencia(double x, double y, double radio){
                        this.x=x;
                        this.y=y;
                        this.radio=radio;
                }
        }
```

En el ejemplo, el constructor de la clase inicializa las variables con los argumentos pasados al constructor. Se debe utilizar this en este constructor para evitar la ambigüedad entre los argumentos y los atributos miembro.

También se puede utilizar this para llamar a uno de los métodos del objeto actual. Esto solo es necesario si existe alguna ambigüedad con el nombre del método y se utiliza para intentar hacer el código más claro.