Guía práctica de estudio 04:
Clases y objetos

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 04:
Clases y objetos

Objetivo:

Aplicar los conceptos básicos de la programación orientada a objetos en un lenguaje de programación.

Actividades:

- Crear clases.
- Crear objetos o instancias.
- Invocar métodos.
- Utilizar constructores.

Introducción

La programación orientada a objetos se basa en el hecho de que se debe dividir el programa, no en tareas, si no en modelos de objetos físicos o simulados. Si se escribe un programa en un lenguaje orientado a objetos, se está creando un modelo de alguna parte del mundo, es decir, se expresa un programa como un conjunto de objetos que colaboran entre ellos para realizar tareas.

Un **objeto** es, por tanto, la representación en un programa de un concepto, y contiene toda la información necesaria para abstraerlo: datos que describen sus atributos y operaciones que pueden realizarse sobre los mismos.

Los objetos pueden agruparse en categorías y una **clase** describe (de un modo abstracto) todos los objetos de un tipo o categoría determinada.

NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

Objeto

La idea fundamental de la programación orientada a objetos y de los lenguajes que implementan este paradigma de programación es combinar (encapsular) en una única unidad tanto los datos como las funciones que operan (manipulan) sobre los datos. Esta unidad de programación se denomina **objeto**.

Entonces, un **objeto** es una encapsulación genérica de datos y de los procedimientos para manipularlos. En otras palabras, un objeto no es más que un conjunto de **atributos** (variables o datos) y **métodos** (o funciones) relacionados entre sí.

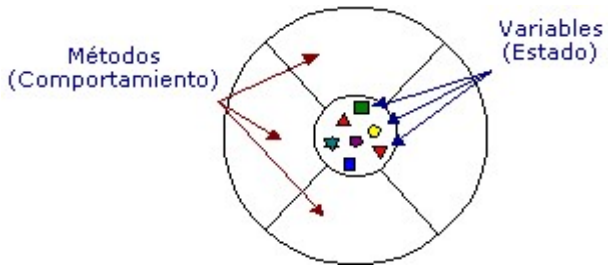

Figura 1. Conceptualización de un objeto en POO.

El **objeto** es el centro de la programación orientada a objetos. Un objeto es algo que se visualiza, se utiliza y que juega un papel o un rol en el dominio del problema del programa. La estructura interna y el comportamiento de un objeto, en consecuencia, no es prioritario durante el modelado del problema.

Clase

En el mundo real existen varios objetos de un mismo tipo, o de una misma **clase**, por lo que una **clase** equivale a la generalización de un tipo específico de objetos. Una **clase** es una plantilla que define las variables y los métodos que son comunes para todos los objetos de un cierto tipo.

Una **clase** es la implementación de un tipo abstracto de datos y describe no solo los **atributos** (datos) de un objeto sino también sus **operaciones** (comportamiento).

En Java para definir una clase se utiliza la palabra reservada **class** seguida del nombre de la clase.

class *NombreDeLaClase*

En UML una clase se representa de manera gráfica con 3 rectángulos dispuestos de manera vertical uno debajo de otro. En el primero se anota el nombre de la clase, en el segundo los atributos y en el tercero las operaciones, es decir:

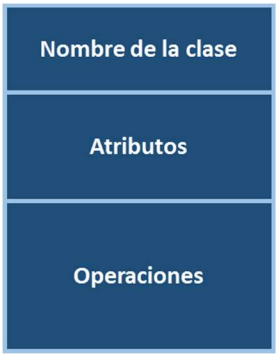

Figura 2. Diagrama de clase en UML.

Instancia

Una vez definida la clase se pueden crear objetos a partir de ésta, a este proceso se le conoce como **crear instancias** de una clase o **instanciar** una clase. En este momento el sistema reserva suficiente memoria para el objeto con todos sus atributos.

Una **instancia** es un elemento de una clase (un objeto). Cada uno de los objetos o instancias tiene su propia copia de las variables definidas en la clase de la cual son instanciados y comparten la misma implementación de los métodos. Sin embargo, cada objeto asigna valores a sus atributos y es totalmente independiente de los demás.

En Java para crear una instancia se utiliza el operador **new** seguido del nombre de la clase y un par de paréntesis.

new *NombreDeLaClase( );*

Mensajes

Normalmente un único objeto por sí solo no es muy útil. Los objetos interactúan enviándose **mensajes** unos a otros. Tras la recepción de un **mensaje** el objeto actuará.

La acción puede ser el envío de otros mensajes, el cambio de su estado, o la ejecución de cualquier otra tarea que se requiera que haga el objeto. Los objetos de un programa interactúan y se **comunican** entre ellos por medio de **mensajes**.

Cuando un objeto A quiere que otro objeto B ejecute uno de sus métodos, el objeto A manda un mensaje al objeto B.

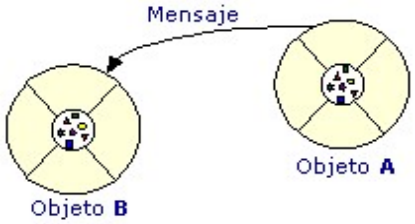

Figura 2. Mensajes entre objetos.

En ocasiones, el objeto que recibe el mensaje necesita más información para saber exactamente lo que tiene que hacer. Esta información se pasa junto con el mensaje en forma de parámetro.

En Java para que un objeto ejecute algún método se utiliza el operador punto:

objeto.nombreDelMetodo(parametros);

Métodos

Los **métodos** especifican el comportamiento de la clase y sus instancias. En el momento de la declaración hay que indicar cuál es el tipo del parámetro que devolverá el método o *void* en caso de que no devuelva nada.

También se debe especificar el tipo y nombre de cada uno de los **parámetros** o **argumentos** del método entre paréntesis. Si un método no tiene parámetros el paréntesis queda vacío (no es necesario escribir *void*). Los parámetros de los métodos son variables locales a los métodos y existen sólo en el interior de estos.

Los argumentos pueden tener el mismo nombre que los atributos de la clase, de ser así, los argumentos *"ocultan"* a los atributos. Para acceder a los atributos en caso de ocultación se referencian a partir del objeto actual ***this***.

En Java se declara un método de manera similar a las funciones en C:

tipoDato nombreDelMetodo(tipoDato parametro1, tipoDato parametro2, …)

Ejemplo:

Se crea una clase Punto para representar un punto en el espacio 2D, cuyos atributos sean las coordenadas (x, y) y contiene un método que imprima los valores de dicha coordenada del punto.

```
public class Punto {
    int x,y;
    public void imprimePunto() {
        System.out.println("Punto [x=" + x + ", y=" + y + "]");
    }
}
```

Posteriormente se crea una clase de prueba para interactuar con la clase Punto. Dentro de esta clase se tiene un método *main* y se crean 2 instancias de la clase Punto. Una vez creadas se da valor a sus atributos y se manda ejecutar su método.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto();
        p.x=5;
        p.y=8;
        p.imprimePunto();

        Punto x  = new Punto();
        x.x=7;
        x.x=2;
        x.imprimePunto();
    }

}
```

Sobrecarga de métodos (overload)

La **sobrecarga de métodos** permite usar el **mismo nombre** de un método en una clase, pero con **diferentes argumentos** (y, opcionalmente, con diferentes valores de retorno). Los métodos sobrecargados hacen más cómoda la implementación de un objeto y, por ende, la utilización de los métodos del mismo.

Los métodos sobrecargados:

- Deben tener el mismo nombre.
- Deben cambiar la lista de argumentos.
- Pueden cambiar el valor de retorno.
- Pueden cambiar el nivel de acceso.

Java permite métodos **sobrecargados** (overloaded), es decir, métodos distintos que tienen el mismo nombre, pero que se diferencian por el número y/o tipo de los argumentos.

Por ejemplo:

public void imprimePunto( )

public void imprimePunto(int x, int y)

public void imprimePunto(float r, float th)

A la hora de llamar a un método sobrecargado, Java sigue algunas reglas para determinar el método concreto que debe llamar:

- Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
- Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (**cast** implícito) y se llama el método correspondiente.
- Si sólo existen métodos con argumentos de un tipo más restringido, el programador debe hacer un **cast** explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
- El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Ejemplo:

```
public class Triangulo {

    float base, altura;

    public float area(){
        return (float)((this.base * this.altura)/2.0);
    }

    public float area(float base, float altura){
        return (float)((base * altura)/2.0);
    }

    public float area(int base, int altura){
        return (float)((base * altura)/2.0);
    }
}
```

```
public class PruebaTriangulo {
    public static void main (String [] args){
        Triangulo triangulo = new Triangulo();
        triangulo.base = 5;
        triangulo.altura = 8;
        System.out.println("base: " + triangulo.base);
        System.out.println("altura: " + triangulo.altura);
        // método área sobrecargado
        System.out.println("area() -> " + triangulo.area());
        System.out.println("area(6, 2) -> " + triangulo.area(6, 2));
        System.out.println("area(5.5f, 3.2f) -> " + triangulo.area(5.5f, 3.2f));
    }
}
```

Métodos de clase (static)

También puede haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama **métodos de clase** o static, estos pueden recibir argumentos, pero no pueden utilizar la referencia *this*.

Un ejemplo típico son los métodos matemáticos de la clase *java.lang.Math* (sin( ), cos( ), exp( ), pow( ), etc.).

Los métodos y variables de clase se crean anteponiendo la palabra **static**. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase.

Ejemplo:

```
public class Circulo {
        static float PI = 3.14159f;
        private float radio;
        …
}
```

En otra clase se puede usar el método o variable:

*System.out.println(**Circulo.PI**);*

Los atributos static tendrán el mismo valor para todos los objetos que se creen de esa clase. Es decir si se modifica el valor de un atributo static, el cambio afectará a todos los objetos de esa clase.

Constructores

Un **constructor** es un método que tiene el mismo nombre que la clase y cuyo propósito es **inicializar** los atributos de un nuevo objeto. **Se ejecuta automáticamente cuando se crea un objeto o instancia de la clase.**

Dependiendo del número y tipos de los argumentos proporcionados se llama al constructor correspondiente.

Si no se ha escrito un constructor en la clase, el compilador proporciona un **constructor por defecto**, el cual no tiene parámetros e inicializa los atributos a su valor por defecto.

Las reglas para los constructores son:

- El constructor tiene el mismo nombre que la clase.
- Puede tener cero o más parámetros.
- No devuelve ningún valor (ni si quiera void).
- Toda clase debe tener al menos un constructor.

Cuando se define un objeto se pasan los valores de los parámetros al constructor utilizando una sintaxis similar a una llamada normal a un método.

Ejemplo:

Para la clase Punto se declara un constructor que reciba los valores de las coordenadas (x, y) y dichos valores se le asignan a los atributos correspondientes (x, y).

```
public class Punto {

    int x,y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void imprimePunto() {
        System.out.println("Punto [x=" + x + ", y=" + y + "]");
    }
}
```

Al hacer esto, dado que ya se cuenta con un constructor que recibe dos parámetros, Java deja de agregar el **constructor por defecto** (el constructor sin parámetros) por lo cual la clase de prueba ahora marcará error.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto();
        p.x=5;
        p.y=8;
        p.imprimePunto();

        Punto x = new Punto();
        x.x=7;
        x.y=2;
        x.imprimePunto();
    }

}
```

En este caso se pueden hacer dos cosas:

- Cambiar la creación de las instancias para que ahora reciban los valores (x, y) desde el momento de su creación.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto(5, 8);
        p.imprimePunto();

        Punto x = new Punto(7, 2);
        x.imprimePunto();
    }

}
```

- Agregar un constructor por defecto (sin parámetros) el cual puede inicializar los valores a un valor por defecto (que se desee) o bien puede estar sin implementación (código).

```
public class Punto {

    int x,y;

    public Punto(){

    }
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void imprimePunto() {
        System.out.println("Punto [x=" + x + ", y=" + y + "]");
    }
}
```

En este último caso, dado que se cuenta con 2 constructores distintos (diferenciados por el número y tipo de parámetros) se tendrían disponibles las dos formas de crear instancias, ya sea con los valores por defecto o pasándolos como parámetros.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto(5, 8);
        p.imprimePunto();

        Punto x = new Punto();
        x.x=7;
        x.y=2;
        x.imprimePunto();
    }

}
```

Destrucción de objetos (liberación de memoria)

Como los objetos se asignan **dinámicamente**, cuando estos objetos se destruyen será necesarios verificar que la memoria ocupada por ellos ha quedado liberada para usos posteriores. El procedimiento es distinto según el tipo de lenguaje utilizado. Por ejemplo, en C++ los objetos asignados dinámicamente se deben liberar utilizando un operador *delete* y en Java y C# se hace de modo automático utilizando una técnica conocida como **recolección de basura** (garbage collection).

Cuando no existe ninguna referencia a un objeto se supone que ese objeto ya no se necesita y la memoria ocupada por ese objeto puede ser recuperada (liberada), entonces el sistema se ocupa automáticamente de liberar la memoria.

Sin embargo, no se sabe exactamente cuándo se va a activar el **garbage collector**, si no falta memoria es posible que no se llegue a activar en ningún momento. Por lo cual no es conveniente confiar en él para la realización de otras tareas más críticas.

Java no soporta destructores pero existe el método *finalize( )* que es lo más aproximado a un destructor. Las tareas dentro de este método serán realizadas cuando el objeto se destruya y se active el **garbage collector**.

Bibliografía

Sierra Katy, Bates Bert
SCJP Sun Certified Programmer for Java 6 Study Guide
Mc Graw Hill

Martín, Antonio
Programador Certificado Java 2.
Segunda Edición.
México
Alfaomega Grupo Editor, 2008

Joyanes, Luis
Fundamentos de programación. Algoritmos, estructuras de datos y objetos.
Cuarta Edición
México
Mc Graw Hill, 2008