Teoría de Programación Orientada a Objetos. 1º de Carrera. Tema 1

Índice

1 Introducción

En esta asignatura, tomaremos un punto de vista para programar utilizando objetos, clases e instancias en vez del punto de vista que emplea procedimientos para gestionar los datos. Este tipo de programación permite gestionar problemas que podrían parecer imposibles de forma cotidiana como puede ser por ejemplo el problema de los n-cuerpos. Para cosas sencillas la programación procedural resulta muy simple, sin embargo para problemas más complejos se vuelve mucho más complicado. Es aquí donde se aprovecha la programación orientada a objetos simplificando las distintas tareas que tienen que llevarse a cabo.

Para ver un ejemplo del código procedural vs. orientado a objetos, ver el ejemplo de la presentación orientació a objecte del día 13 de Febrero

2 Conceptos Basicos

- Encapsulamiento: Construir una clase y los métodos para procesar estos datos.
- Information hiding: Esconder a las otras clases los datos y métodos de las otras clases. Esto se utiliza ya que al no necesitarlas, minimiza el espacio que ocupa el código y evita interferencias.

  Si se accede a los elementos de estas clases independientes puede llevar a problemas, ya que en determinados casos la clase trabajará con un tipo de dato que las otras no utiliza. Esto se soluciona utilizan **setters** y **getters**, que muestran la interfaz de la clase.
- Herencia: Es el mecanismo que emplean las clases para ceder u obtener propiedades de otras clases. Al heredar de otra clase dependiendo del lenguaje se heredan algunas variables y algunos métodos (En python se hereda todo pero en Java viene definido por public y private). Esto permite ahorrar espacio para algunas clasess que empleen los mismos procesos que otras clases. Un **subclase** hereda de una **superclase**
- Composición: Un objeto puede estar compuesto por varios elementos de otro tipo. Por ejemplo un objeto puede presentar distintas caracterísitcas con diferentes tipos de dato (*int, string, list...*). En programación orientada a objetos aparecen dos tipos de asociaciones de contenido. La primera es la **composición**, en la que los elementos que componen un elemento más grande con una conexión fuerte de tal forma que no pueden existir si no pertenecen al grande. La segunda forma es la **agregación**, en la que los objetos están contenidos con una conexión debil, de tal forma que pueden existir sin necesidad de estar contenidos en el elemento grande.
- Polimorfismo: Un mismo nombre puede llevar a diferentes resultados a partir de la función. Por ejemplo, si definieras una clase de la que heredaran otras 2, es posible que ambas clases utilicen el mismo nombre de una función para ejecutar diferentes procesos.

3 Paso de Python a Java

Al contrario que Python, las variables en Java no pueden cambiar su tipo. Al definir una variable, las variables se tienen que definir como privadas o publicas, dependiendo de si se puede heredar de la clase o no, y deben definirse de que tipo son. (Esto viene ejemplificado en orientació a objecte)

NOTA: Si buscas heredar de una interface en Java debes usar *implement*, no *extends*

4 Programación en Java

La programación en Java al contrario que la programación en Python que es interpretado es un lenguaje compilado. La diferencia es que un interprete executa un codigo de alto nivel instrucción a instrucción mientras que un compilador traduce todo el programa a un fichero ejecutable. En el caso de java, el

proceso que se tiene que realizar para poderlo ejecutar es pasarlo por un compilador que devolverá un *Java Bitcode Program*, el cual será ejecutado por el interprete de Java correspondiente dependiendo del sistema operativo en el que se esté operando.

4.1 Hello World

En la presentación de Java aparece un código para poder realizar el clásico programa de "Hello World". En este programa aparece un método publico con la categoría *static*. Este método es un método **de la clase**, no un método **de objetos de la clase**, por este motivo, este método se puede ejecutar independientemente de si existe un elemento de la clase o no. De esta forma podremos ejecutar procesos sin la necesidad de crear objetos.

4.2 For loop

El bucle **for** en Java necesita 3 argumentos para poderse compilar y ejecutar de forma correcta. El primero corresponde a definir la variable que se va iterar y marcar el inicio de esta. El segundo argumento será la condición durante la cual se seguirá ejecutando el bucle. El tercero corresponderá al paso del bucle:

```
for(int year = 1 ; year < numYears ; year++) {
    double interest; // interest for this year
    interest = principal * rate/100.0;
    valueOfInvestment += interest;
    System.out.println("Value of the investment after "
            + year + " years is " + valueOfInvestment);
}
```

El for también se puede emplear para iterar a lo largo de una lista. poniendo el formato:

```
for (tipo_de_elemento i : lista){
    //Donde i es el elemento de la lista que va a iterar
}
```

4.3 If

El comando **if** funciona prácticamente igual que en un lenguaje interpretado. Sin embargo, a la hora de hacer las comparaciones se deberá emplear el método *equals*. En este método, se comparará el elemento al que se le aplique con el argumento que se le pase a la función. Además el if cuenta con ayudas para hacer varias comparaciones en un mismo if utilizando el AND(&&) y el OR (||)

```
if (units.equals("inch") || units.equals("inches") || units.
   equals("in")) {
    inches = measurement;
} else if (units.equals("foot") || units.equals("feet") ||
   units.equals("ft")) {
    inches = measurement * 12;
} else if (units.equals("yard") || units.equals("yards") ||
   units.equals("yd")) {
    inches = measurement * 36;
} else if (units.equals("mile") || units.equals("miles") ||
   units.equals("mi")) {
    inches = measurement * 12 * 5280;
} else {
    System.out.println("Unknown units " + units);
    System.exit(-1);
}
```

4.4 Switch

Funciona igual que el if pero te permite no tener que encadenar varios comandos if como en el ejercicio anterior. En este caso solo se coloca la variable que se está comprobando y se coloca *case* seguido de el dato con el que se quiere comparar. Si no se cumplen ninguno de los casos se puede utilizar *default* para definir que debe hacer en el caso de que no se cumpla ninguna de las anteriores.

```
switch (units) {
    case "inches":
        inches = measurement;
        break; // Importante ponerlo para evitar que salte a
           otro case
    case "feet":
        inches = measurement * 12;
        break;
    case "yards":
        inches = measurement * 36;
        break;
    case "miles":
        inches = measurement * 12 * 5280;
        break;
    default:
        System.out.println("Unknown units " + units);
        System.exit(-1);
}
```

4.5 Try-catch

Funciona para evitar que errores paren el código (*error handling*), de tal forma que se intentará ejecutar lo que venga en el try y, en caso de no conseguirlo, ejecutar el catch

```
str = "3,141592"; // note "," instead of "."
double x;
try {
    x = Double.parseDouble(str);
}
catch ( NumberFormatException e ) { // or simply catch (
   Exception e)
    System.out.println( "Not a legal number." );
    x = Double.NaN;
    // or do not continue because it does not make sense
    // System.exit(-1);
}
```

4.6 Objetos

Para poder trabajar con Java con cualquier tipo de dato es necesario crear un objeto. Sin embargo para algunos tipos sencillos como los *int,string,boolean,...* existen variables que guardan valores por defecto en memoria además de clases *Integer,Double,String....* Un ejemplo de clase empleando esto es el siguiente:

```
class Complex {
    public double real; // DON'T do this at home
    public double imaginary;
    public Complex (double r, double im) {
        real = r;
        imaginary = im;
    }
    public String toString() {
        // so that we can do println( a complex object )
```

```
                return "("+Double.toString(real) + ", "
                + Double.toString(imaginary)+")";
            }
        }
```

4.7 Clases y encapsulamiento

De normal al crear una clase por defecto todos los atributos deben ser privados. Si un atributo debe ser accedido por otra clase emplearemos un *getter*, y si necesitamos cambiarlo emplearemos un *setter*. Estos metodos se llaman *accesors*, y son la forma principal de trabajar con una clase. En las clases además existe el parámetro *this*, que funciona igual que el self en python.

4.7.1 Herencia

La herencia y la composición son los métodos principales empleados para reducir la longitud del código reutilizando otras clases. Por ejemplo, si defines una clase que sea *cuerpo*, pueden derivar de el las clases *estrella* y *planeta*, ya que compartirán las caracterísitcas con *cuerpo* además de las suyas propias. Para heredar de una clase anteriormente definida emplearemos *extends* seguido del nombre de la clase de la que hereda.

```
        public class Body {
            protected double mass;
            protected double[] position;
            protected double[] velocity;
            public Body(double mass, double[] initialPosition, double[]
                initialVelocity) {
                this.mass = mass;
                position = initialPosition;
                velocity = initialVelocity;
            }
        }

        public class Star extends Body {
            private double percentMass;
            public Star(double mass, double[] initialPosition, double[]
                initialVelocity, double percentMass) {
                super(mass, initialPosition, initialVelocity);
                this.percentMass = percentMass;
            }
        }
```

4.7.2 Clases Abstractas e Instancias

Son plantillas para clases concretas derivadas de esta. **No se puede instanciar un objeto de una clase abstracta**. Estas clases pueden:

- Declarar atributos para que se hereden.
- Declarar métodos.
- Cada método puede tener una implementación que también se derive.
- Los métodos que no se implementen deben ser derivados.
- Una clase derivada solo puede tener una superclase, no puede tener múltiples padres.

Un case más estrictos son las **instancias**, las cuales no tienen métodos implementados ni declaraciones de atributos, solo tienen constantes del tipo *static* y *final*.

5 GRASP patterns

Los patrones GRASP son un conjunto de parejas formadas por un problema y una solución acompañada con avisos de cuando y como usarla. Estos patrones se encargan principalmente del reparto de **responsabilidades**. Estas responsabilidades se pueden implementar a través de:

- Atributos: Saber algo
- Asociaciones: Sabe quién sabe o hace algo
- Métodos: Sabe hacer algo.

De esta forma, los GRASP indican a que clases se le deberían dar estas responsabilidades con el objetivo de hacer el código lo más óptimo posible.

5.1 Expert

Problema: ¿Cual es el principio general para asignar responsabilidades a clases?
Solución: Asigna las responsabilidades al experto en la información, la clase que tiene la información necesaria para llevar a cabo la responsabilidad.

A la hora de realizar un código, la realización de cada responabilidad debe realizarla el experto de su propios datos, lo que resultará en que la información se encuentre repartida en diferentes clases. La clave de esto resulta en que una clase no realizará tareas que no sea necesario que realice, lo que resulta en una baja dependencia entre clases, clases más cohesivas y un código más fácil de leer.

5.2 Creator

Problema: ¿Quién debería ser el encargado de generar instancias de alguna clase?
Solución: Se le asignará a B la responsabilidad de crear instancias de A si:

- B agrega o contiene objetos de A
- B utiliza objetos de A de forma muy cercana
- B tiene los datos de inicialización que deben pasarse al constructor de A

5.3 Low Coupling

El **coupling** mide como de relacionadas están dos clases, dependen entre sí o tienen información las unas de otras. Las clases que poseen un coupling muy fuerte:

- Sufren los cambios realizados en las clases relacionadas
- Son más dificiles de ajustar y de mantener
- Son más difíciles de reutilizar

Pero el coupling es necesario si queremos intercambiar mensajes, por lo que debe utilizarse en su justa medida, no demasiado.

Problema: ¿Cómo se permite una baja dependencia, un impacto bajo en los cambios y aumentar el reuso de clases?
Solución: Asigna las responsabilidades de tal forma que el coupling quede reducido. Trata de evitar que las clases sepan tanto como sea posible.

5.4 High Cohesion

La **cohesion** mide como de fuerte están relacionadas las responsabilidades de una misma clase. Una clase con baja cohesión hace muchas cosas no relacionadas o trabajando mucho. Esto hace que sean clases dificiles de entender, mantener y reutilizar

Problema: ¿Cómo se mantiene una clase focalizada, entendible y manejable?
Solución: Asigna las responsabilidades de tal forma que la cohesión se mantenga alta. Intenta evitar que clases hagan muchas cosas o cosas muy diferentes.

5.5 Polymorphism

Problema: ¿Cómo se gestiona el comportamiento basado en tipo (por ejemplo, dentro de una clase), sin emplear if-then-else ni switch que involucre el nombre de la clase o un atributo *tag*?
Solución: Usa los métodos polimórficos. Dale el mismo nombre a métodos de distintas clases que den diferentes resutlados, de esta forma no será necesario comprobar el nombre de la clase con la que estás trabajando, sino que unicamente se necesitará llamar a un método en concreto.

6 Principios de diseño

Con el fin de mejorar el diseño del código a la hora de realizar programación orientada a objetos.

6.1 Information Hiding

Las clases y miembros deben tener minimizada su accesibilidad. Las clases no deben exponer su detalles de implementación interna, un componente solo debe aportar *toda y solo* la infomación necesaria.

6.2 "Don't talk to strangers"

Un objeto A puede llamar a un método de un objeto instancia B, pero nunca deberá utilizar el objeto B para llegar hasta un segundo objeto C.

6.3 DRY: Don't repeat yourself

Evita repetir codigo abstrayendo las cosas comunes y poniendolas en un solo sitio. Esto también se aplica a para las responsabilidades, pon cada pieza de información y comportamiento en un único sitio sensible.

6.4 Liskov