

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

## Índice

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Conceptos Basicos</b>	<b>2</b>
<b>3</b>	<b>Paso de Python a Java</b>	<b>2</b>
<b>4</b>	<b>Programación en Java</b>	<b>2</b>
4.1	Hello World . . . . .	3
4.2	For loop . . . . .	3
4.3	If . . . . .	3
4.4	Switch . . . . .	4
4.5	Try-catch . . . . .	4
4.6	Objetos . . . . .	4
4.7	Clases y encapsulamiento . . . . .	5
4.7.1	Herencia . . . . .	5
4.7.2	Clases Abstractas e Instancias . . . . .	5
<b>5</b>	<b>GRASP patterns</b>	<b>6</b>
5.1	Expert . . . . .	6
5.2	Creator . . . . .	6
5.3	Low Coupling . . . . .	6
5.4	High Cohesion . . . . .	7
5.5	Polymorphism . . . . .	7
<b>6</b>	<b>Principios de diseño</b>	<b>7</b>
6.1	Information Hiding . . . . .	7
6.2	"Don't talk to strangers" . . . . .	7
6.3	DRY: Don't repeat yourself . . . . .	7
6.4	LSP: Principio de substitución de Liskov . . . . .	7
<b>7</b>	<b>Random Forests</b>	<b>7</b>
7.1	Machine learning . . . . .	7
7.1.1	Train,test y validation . . . . .	8
7.1.2	Regresión Lineal . . . . .	8
7.1.3	Datasets . . . . .	8
7.2	Decision Trees . . . . .	9
7.2.1	Pureza de un conjunto . . . . .	9
7.3	Random Forest . . . . .	9

# 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 classes 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 características 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 ejecuta un código 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:

```
1         for(int year = 1 ; year < numYears ; year++) {
2             double interest; // interest for this year
3             interest = principal * rate/100.0;
4             valueOfInvestment += interest;
5             System.out.println("Value of the investment after "
6                                 + year + " years is " + valueOfInvestment);
7         }
```

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

```
1         for (tipo_de_elemento i : lista){
2             //Donde i es el elemento de la lista que va a iterar
3         }
```

## 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 (||)

```
1         if (units.equals("inch") || units.equals("inches") || units.
2             equals("in")) {
3             inches = measurement;
4         } else if (units.equals("foot") || units.equals("feet") ||
5             units.equals("ft")) {
6             inches = measurement * 12;
7         } else if (units.equals("yard") || units.equals("yards") ||
8             units.equals("yd")) {
9             inches = measurement * 36;
10        } else if (units.equals("mile") || units.equals("miles") ||
11            units.equals("mi")) {
12            inches = measurement * 12 * 5280;
13        } else {
14            System.out.println("Unknown units " + units);
15            System.exit(-1);
16        }
```

## 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.

```
1      switch (units) {
2          case "inches":
3              inches = measurement;
4              break; // Importante ponerlo para evitar que salte a
                    otro case
5          case "feet":
6              inches = measurement * 12;
7              break;
8          case "yards":
9              inches = measurement * 36;
10             break;
11          case "miles":
12              inches = measurement * 12 * 5280;
13              break;
14          default:
15              System.out.println("Unknown units " + units);
16              System.exit(-1);
17      }
```

## 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

```
1      str = "3,141592"; // note "," instead of "."
2      double x;
3      try {
4          x = Double.parseDouble(str);
5      }
6      catch ( NumberFormatException e ) { // or simply catch (
                    Exception e)
7          System.out.println( "Not a legal number." );
8          x = Double.NaN;
9          // or do not continue because it does not make sense
10         // System.exit(-1);
11     }
```

## 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:

```
1      class Complex {
2          public double real; // DON'T do this at home
3          public double imaginary;
4          public Complex (double r, double im) {
5              real = r;
6              imaginary = im;
7          }
8          public String toString() {
9              // so that we can do println( a complex object )
10          }
```

```

10         return "("+Double.toString(real) + ", "
11         + Double.toString(imaginary)+")";
12     }
13 }

```

## 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 características 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.

```

1      public class Body {
2          protected double mass;
3          protected double[] position;
4          protected double[] velocity;
5          public Body(double mass, double[] initialPosition, double[]
6              initialVelocity) {
7              this.mass = mass;
8              position = initialPosition;
9              velocity = initialVelocity;
10         }
11     }
12
13     public class Star extends Body {
14         private double percentMass;
15         public Star(double mass, double[] initialPosition, double[]
16             initialVelocity, double percentMass) {
17             super(mass, initialPosition, initialVelocity);
18             this.percentMass = percentMass;
19         }
20     }

```

### 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 responsabilidad 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 difíciles 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 difíciles 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 resultados, de esta forma no será necesario comprobar el nombre de la clase con la que estás trabajando, sino que únicamente 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 información 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 código abstrayendo las cosas comunes y poniéndolas en un solo sitio. Esto también se aplica a las responsabilidades, pon cada pieza de información y comportamiento en un único sitio sensible.

## 6.4 LSP: Principio de substitución de Liskov

Si S es un subtipo de T, entonces los objetos de T pueden ser sustituidos por objetos del tipo S sin alterar las propiedades deseadas del programa. **Donde se espera un objeto de una super-clase puede ser sustituido por un objeto de una clase derivada.**

# 7 Random Forests

## 7.1 Machine learning

El **machine learning** es un proceso a través del cual buscas que el ordenador aprenda a ejecutar una determinada tarea el solo en lugar de instruirlo en como hacerlo. De esta forma, el ordenador aprende de los datos y de forma automática. Algunas de las tareas clásicas para las que se emplea el machine learning es la clasificación, la regresión, el clustering y la reducción dimensional.

Al principio, para realizar una tarea tan sencilla como evitar el spam, se necesitaba montar un filtro de spam que se encargara de filtrar aquellos mensajes no deseados. Utilizando el machine learning, solo con una tabla de datos que contengan un conteo de las palabras del mensaje y la clase del email se puede entrenar a un ordenador para que sea capaz de detectar el spam.

La nomenclatura perteneciente a un modelo de machine learning contiene conceptos como:

- **Sample:** La pareja formada por cada elemento y su resultado  $(x_i, y_i)$ . Por ejemplo, en el caso de los emails que hemos mencionado en el parrafo anterior, un sample podría ser  $x_1 = [0, 0, 1, 3...2, 0]$ ,  $y_1 = 0$  siendo el  $y_i = 1$  spam y el  $y_i = 0$  no spam.
- **Supervised learning:** Proceso empleado para encontrar los parametros que mejor mapean  $f(x) = \hat{y}$  tal que  $f(x_i) = y_i$  para el mayor número de samples posibles.

En esta notación  $x_i$  forma el vector de características,  $y_i$  el resultado correcto y  $\hat{y}_i$  la predicción del modelo. La función  $f$  es lo que llamamos un **classifier**.

### 7.1.1 Train, test y validation

En un proceso de machine learning con  $n$  samples, se dividen en 3 partes. La primera de ellas formaría el entrenamiento. En este apartado, el model de machine learning forma relaciones entre los parametros de  $f$  y el resultado correcto ( $y_i$ ) con el fin de hallar los parámetros del polinomio  $f$ . La segunda parte formaría el test. En esta parte el ordenador comprobará que tan bien funciona las relaciones que ha construido con samples desconocidos. Pero ¿por qué un conjunto de elementos para realizar tests?

Pongamos un classifier de k-Nearest Neighbors. En este clasificador el modelo memoriza todos los valores del entrenamiento  $x_i$ . Si se presenta una  $x$  desconocida, busca los  $k$  elementos más cercanos, es decir, los elementos  $x_{j_1}, x_{j_2}, \dots, x_{j_k}$  cuyos parametros más se acercan a los del sample desconocido. Tras esto miramos cual es el resultado más frecuente entre  $\{y_{j_1}, y_{j_2}, \dots, y_{j_k}\}$ . Para valores ya vistos, simplemente con emplear un algoritmo 1-NN será perfecto. Es por esto que necesita realizar tests con elementos desconocidos.

La tercera parte la forma la validación, encargada de controlar los hiperparámetros. Un **hiperparámetro** es un parámetro que se escapa al control humano. Un parámetro normal son los coeficientes de  $f$ , que se encuentran en el entrenamiento. Un hiperparámetro podría ser el número de vecinos que comprobar al emplear el k-NN. En resumen, los hiperparámetros son parámetros del algoritmo, no de la función. Sin embargo, nosotros no realizaremos un apartado de validación para reducir la complejidad del algoritmo.

### 7.1.2 Regresión Lineal

En una regresión lineal, se toma el conjunto de datos y se busca el polinomio que más se ajuste a esos puntos (algo parecido al polinomio interpolador). Al igual que vimos emplear el método de mínimos cuadrados, también se puede utilizar aquí. Si cogieramos una recta (polinomio de grado 1), se ajustaría parcialmente a los puntos con los que se entrena pero se les escaparían los puntos desconocidos del test. Uno de los motivos por el que puede que estos puntos desconocidos se escapen es que sean **outliers**, valores que no se ajustan al comportamiento que presentan el resto de los datos.

Quizá una linea recta es demasiado sencillo, pero si probamos un polinomio de mayor grado nos daremos cuenta de que se ajusta de forma casi perfecta para los valores dados pero funciona muy mal para los valores desconocidos. Este error se conoce como **overfitting**. Sin embargo existen formas de solucionar este problema. Una de la más óptimas es la **regularización**. Este proceso consiste en reducir el número de elementos cogiendo grupos de elementos y tomar la media de estos elementos. Además, también existen hiperparámetros para regular esta regularización.

### 7.1.3 Datasets

Los datasets son la parte más importante del machine learning, pues son los datos que emplearemos para entrenar a nuestro algoritmo. En la presentación sobre [random forests](#) aparece un conjunto de datos sobre



los sonares en la página 18. Este conjunto de datos nos enseña una lección importante sobre trabajar con datasets: es importante mirar los datos antes de entrenar un modelo. En las gráficas podemos ver que entre 50 y 60, los valores son prácticamente idéntico, lo que hará que el algoritmo tenga gran dificultad para diferenciarlos. Es por esto que al no trabajar con esos datos es posible que el algoritmo funcione mejor.

## 7.2 Decision Trees

Un **árbol de decisión** es un modelo de clasificación y regresión formado por dos tipos de nodos: Nodos padre, que realizan una decisión y por lo tanto contienen un par formado por las características y el resultado, y nodos hoja que presentan la clase predicha.

Para poder construir un árbol de decisión partimos de un nodo raíz, le damos todo el conjunto de entrenamiento  $(X, y)$  con  $X[:, k] = \text{columna } k \text{ de } X$ . Después de esto se busca la pareja (la columna  $k$ , un valor de  $k$  llamado  $v$ ) tal que se dividan los datos en dos sectores: las filas con un resultado menor a  $v$  ( $I_{left}$ ) y las filas con un resultado mayor o igual a  $v$  ( $I_{right}$ ). Esto implica que  $y[I_{left}]$  y  $y[I_{right}]$  sean lo más puro posible, es decir, las clases se separen de la forma más clara. Ahora repetiremos el proceso con cada uno de los sectores si se cumple que:

- El sector posee el número mínimo de samples
- No se ha alcanzado el máximo de profundidad
- Ambos sectores creados no son iguales.

### 7.2.1 Pureza de un conjunto

A la hora de separar los sectores debe verse cual es la forma más pura. Para medir esta pureza se emplean diferentes métodos. Un sector será puro si todos sus elementos pertenecen a la misma clase. En el caso de que esto no ocurra, se puede emplear el índice **Gini** para medir su impureza.

$$G = 1 - \sum_{c=1}^C p_c ; p_c = \frac{1}{m} \sum_{i=1}^m 1_{y_i=c} \quad (1)$$

Siendo  $C$  el número de clases que contiene el conjunto,  $m$  el número de samples, y  $p_c$  la frecuencia de la clase  $c$  en  $y$ . Si  $1_{y_i=c} = 1$  si  $y_i = c$ , sino es 0. Lo que se busca es la pareja que minimice el valor Gini de ambos lados.

$$J(k, v) = \frac{m_{left}}{m_{left} + m_{right}} G_{left} + \frac{m_{right}}{m_{left} + m_{right}} G_{right} \quad (\text{Función de coste})$$

Otra forma de comprobar la impureza es la entropía.

$$H = - \sum_{c=1, p_c > 0}^C p_c \log(p_c) \quad (2)$$

## 7.3 Random Forest

Un clasificador/regresor de conjunto combina los resultados de varios clasificadores/regresores diferentes para devolver un output, la mayoría de las veces, la media de los resultados o el valor más votado. La clave detrás de esto es que la combinación de las predicciones de diferentes expertos resulta en una mejor predicción que la de una sola predicción del mejor experto. El random forest emplea un conjunto de diferentes arboles de decisión combinando sus predicciones de acuerdo a lo más votado.

Esto permitirá que al tener diferentes árboles, cada árbol compruebe unos parámetros aleatorios de cada sample. De esta forma se termina observando todos los parámetros. Para implementar esto utilizaremos el **RandomForestGenerator**.

```

1  import numpy as np
2  import sklearn.datasets
3
4
5  iris = sklearn.datasets.load_iris() # its a dictionary
6  print(iris.DESCR)
7  X, y = iris.data, iris.target
8  # numpy arrays : X 150 rows 4 cols, y 150 rows
9
10 ratio_train, ratio_test = 0.7, 0.3
11 # 70% train, 30% test
12
13 num_samples, num_features = X.shape
14 # 150, 4
15
16 idx = np.random.permutation(range(num_samples))
17 # shuffle {0,1, ... 149} because samples come sorted by class!
18
19 num_samples_train = int(num_samples*ratio_train)
20 num_samples_test = int(num_samples*ratio_test)
21 idx_train = idx[:num_samples_train]
22 idx_test = idx[num_samples_train : num_samples_train+num_samples_test]
23 X_train, y_train = X[idx_train], y[idx_train]
24 X_test, y_test = X[idx_test], y[idx_test]
25
26 # Hyperparameters
27 max_depth = 10 # maximum number of levels of a decision tree
28 min_size_split = 5 # if less, do not split a node
29 ratio_samples = 0.7 # sampling with replacement
30 num_trees = 10 # number of decision trees
31 num_random_features = int(np.sqrt(num_features))
32                        # number of features to consider at
33                        # each node when looking for the best split
34 criterion = 'gini' # 'gini' or 'entropy'
35
36 rf = RandomForestClassifier(max_depth, min_size_split, ratio_samples,
37                             num_trees, num_random_features, criterion)

```