CC3002 - Metodologías de diseño y programación

Juan-Pablo Silva and Ignacio Slater

Departamento de Ciencias de la Computación, Universidad de Chile

23 de marzo de 2020

Índice general

Introducción

Este apunte busca ser un apoyo para las clases del curso *CC3002 - Metodologías de diseño y programación* del *departamento de ciencias de la computación* de la *Universidad de Chile* dictado por el profesor *Alexandre Bergel*.

El objetivo de este documento es reunir los contenidos del curso, entregar consejos, plantear ejercicios y profundizar en algunos de los contenidos.

Tengan en cuenta que este apunte **no** es un reemplazo para las clases presenciales ni el material generado por el profesor, pues la última palabra sobre qué contenidos serán evaluados y cuál será el enfoque del ramo en cada una de sus iteraciones será la definida por él. Este material solamente es un apoyo para presentar explicaciones y ejemplos alternativos a los vistos en clases.

Este documento no necesariamente cubrirá todos los contenidos del curso ni estará actualizado con respecto a lo visto en clases, pero su intención es abarcar la mayor parte de estos.

La contraparte de este apunte se puede encontrar en https://github.com/CC3002-Metodologias/apunte-y-ejercicios/wiki/CC3002---Metodologías-de-diseño-y-programación

Objetivos del curso

El objetivo principal del curso es servir de introducción a la *ingeniería de software* y al diseño de soluciones a problemas que sean extensibles y mantenibles en el tiempo. Al ser un curso introductorio, no se espera que lxs alumnxs terminen el curso con un conocimiento avanzado sobre diseño de *software*, sino que plantear una manera inicial de como abordar y generar soluciones a problemas de programación complejos en los que la solución correcta la mayoría del tiempo no es la obvia.

Se espera que al terminar el curso las soluciones que planteen a los problemas no sólo cumplan el objetivo de implementar las funcionalidades requeridas sino que tengan un buen diseño (lo que se considere un buen diseño y las formas de generar programas bien diseñados se verán a lo largo del semestre y de este apunte). Dicho esto, al ser un curso de diseño, no existe una única solución correcta para los problemas, sino que cualquier solución bien argumentada

será correcta, pero tengan en cuenta que los contenidos enseñados en el ramo ya tienen argumentos sólidos para justificar su uso y los problemas que se plantearán estarán pensados de tal forma que sea difícil rebatir los argumentos que justifican el uso de estos contenidos en la implementación de la solución.

Específicamente, los objetivos del curso son:

- Aprender a pensar soluciones utilizando programación orientada a objetos.
- Introducir el lenguaje de programación *Java*.
- Entender el significado de extensibilidad y mantenibilidad de un proyecto de *software*.
- Introducir el concepto y presentar herramientas de *testing* y la mentalidad para abordar problemas a partir de estos.
- Mostrar los conceptos básicos del patrón arquitectónico de *modelo-vista-controlador* y como crear interfaces gráficas nativas.
- Presentar algunas herramientas avanzadas y específicas de la librería estándar de *Java*.

Parte I

Herramientas necesarias y recomendadas

Capítulo 1

Java Development Kit (JDK)

Para este curso utilizaremos *Java* como el lenguaje predominante para ilustrar y evaluar los conceptos.

La mayoría de las cosas que se verán en el semestre son agnósticas al lenguaje que se utilice, y no se necesitan conocimientos previos en *Java*. La segunda parte de este apunte se encargará de introducir el lenguaje, su modo de uso y algunas de las herramientas que éste provee.

Se espera que el lector tenga un conocimiento básico de *Python* y nociones generales del lenguaje *C* puesto que algunos de los ejemplos que se darán se contrastarán con el comportamiento de estos respecto a *Java*.

1.1. Instalación

Para instalar *Java*, primero debe instalarse el compilador, la máquina virtual y la librería estándar. Todas estas herramientas vienen incluidas dentro del *JDK*.

Existen muchas maneras de instalar *Java*, y a continuación se mostrarán algunas. Las instrucciones siguientes son para instalar *Java 13*, que es la versión más reciente al momento de escribir este apunte. Para el curso no es necesario que se utilice la última versión, pero es necesario que al menos usen *Java 9* y recomendable que instalen al menos *Java 11*. Si habían instalado anteriormente *Java 8*, les recomentamos que **desinstalen dicha versión** antes de proceder a instalar la más nueva porque puede generar conflictos.

1.1.1. Opción 1: *Open JDK* (Recomendado)

Linux

1.1.2. Windows

Chocolatey (Recomendado)

Si no lo tienen instalado, el primer paso sería instalar el gestor de paquetes *Chocolatey*, para esto se debe abrir *Powershell* como administrador.

```
[Net.ServicePointManager]::SecurityProtocol =
↪  [Net.SecurityProtocolType]::Tls12
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Force
Invoke-WebRequest "https://chocolatey.org/install.ps1" -UseBasicParsing |
↪  Invoke-Expression
```

Una vez que tengan *Chocolatey* instalado, basta ejecutar:

Capítulo 2

Git

Git es un sistema de control de versiones (*VCS*) distribuido para mantener un historial de los cambios que se realizan en los archivos durante el desarrollo de una aplicación.

Se creó con el objetivo de manejar las versiones del *kernel* de *Linux*. Es un proyecto de código abierto y fue adquiriendo popularidad con los años (según un estudio realizado por *StackOverflow*, *Git* es el sistema de versionamiento utilizado por el 70 % de sus usuarios).

Existen muchos otros sistemas de versionamiento que también se utilizan en la industria, en particular *Mercurial*, *Perforce* y *Subversion* son algunos de los más importantes.

2.1. Instalación

Como *Git* es un proyecto *open-source* existen diversas maneras de instalarlo, en particular aquí se ejemplificarán algunas.

2.1.1. Linux

La forma más fácil de instalar *Git* es utilizando el gestor de paquetes del sistema operativo que estén usando. A continuación se muestran los comandos necesarios para cada distribución de *Linux*:

Debian[1]

```
sudo apt install git-all
```

CentOS

```
sudo yum install git
```

[1] *Ubuntu* es un sistema basado en *Debian*

Fedora

```
sudo yum install git-core
```

Arch Linux[2]

```
sudo pacman -Sy git
```

Gentoo

```
sudo emerge --ask --verbose dev-vcs/git
```

2.1.2. Windows

De nuevo, existen muchas maneras de instalar *Git*, a continuación se muestran algunas:

***Chocolatey* (Recomendado)**

```
cinst git.install -y
```

Git bash

Git bash es una interfaz de consola que emula la terminal de *Linux* y que viene con git instalado.

Para instalarla deben descargar el cliente desde el sitio oficial de *Git*.

2.1.3. MacOS

Para Mac existen dos alternativas:

1. Descargar el instalador de Git for Mac.
2. Instalarlo con *brew* ejecutando el comando: `brew install git`

2.2. Configuración

Primero, para comprobar que se haya instalado correctamente, deben ejecutar el comando:

```
git --version
```

El resultado que debiera retornar este comando es algo del estilo (para el caso de *Windows* utilizando *Chocolatey*):

```
git version 2.21.0.windows.1
```

[2]Por ejemplo *Manjaro*.

Dependiendo de la manera en que hayan instalado *Git* es posible que necesiten configurar las credenciales, para esto deben ejecutar los comandos:

```
git config --global user.name "Xen-Tao"
git config --global user.email xentao@depa.na
```

Reemplazando los datos con su nombre y correo. Luego, haciendo `git config -l` verifiquen que su usuario y correo se hayan registrado correctamente.

2.3. Repositorios

Un repositorio (generalmente llamado *repo*) se puede entender como un proyecto con un sistema de versionamiento integrado.

En el caso de *Git*, uno de los principales beneficios es que puede utilizarse de manera local o remota utilizando un servidor o un *host* de repositorios (vea la sección 2.4).

2.4. *GitHub*

2.5. Ejercicios

2.6. Conceptos clave

- ***Git*:** Sistema de control de versiones distribuido Comandos importantes:
  - `git init`: Inicia un repositorio.
- **Conceptos adicionales:**
  - **Sistema distribuido:** Sistema en el que sus componentes están ubicados en varios computadores conectados entre sí. Pueden ver este concepto más en profundidad en el curso *CC5212 - Procesamiento Masivo de Datos*

2.7. Material adicional

Los siguientes enlaces contienen explicaciones alternativas y/o más detalladas de las herramientas disponibles para utilizar *Git*:

- **Documentación oficial de *Git***.
- **Comandos básicos de *Git***.
- **Introducción a *GitHub***: tutorial básico de cómo crear y manejar un repositorio en *GitHub*.

- **Integración de *Git* y *GitHub***: guía de los comandos básicos de *Git* y cómo se relacionan con *GitHub*.
- ***GitHub* flow:** la modalidad (flujo) de trabajo recomendada por *GitHub* para trabajar en equipos. Les recomendamos encarecidamente que utilicen esta modalidad al trabajar individualmente y, en especial, en las tareas de este ramo.
- ***Markdown*:** es la sintaxis utilizada para crear documentación en *GitHub*. Parecida a *HTML* pero más concisa (en realidad es un *superset* de *HTML*).
- ***Issues*:** es una forma de llevar una lista de *tareas* o *problemas* y *metas* en un proyecto. Suelen utilizarse mucho cuando un usuario encuentra un *bug* o solicita la implementación de un *feature* nuevo en algún proyecto.
- ***Fork*:** un *fork* es como clonar un repositorio de otra persona para trabajar paralelamente en el desarrollo de alguna funcionalidad sin cambiar el repo original. Esto es común de hacer cuando se quiere ayudar en el desarrollo de proyectos *open-source*.
- ***GitHub wikis*:** herramienta avanzada para documentar proyectos. La misma que se usa en la wiki del curso.
- **Canal de *YouTube* de *GitHub***.
- **Usar *IntelliJ* para manejar *Git***.
- **Usar *VSCode* para menejar *Git***.
- ***GitKraken*:** interfaz gráfica para manejar repositorios de *Git*. Pueden obtener una licencia gratis postulando a los beneficios de *GitHub Education*.
- ***Mercurial*:** alternativa a *Git* que también es ampliamente usada. *SourceForge* es uno de los ejemplos más importantes de *host* de repositorios de *Mercurial* (el equivalente a *GitHub*).

Parte II

Objetos en Java

Capítulo 3

Programación orientada a objetos

Hasta el momento, gran parte de lo que ustedes conocen es cómo escribir algoritmos en los que realizan acciones siguiendo una lógica. La programación orientada a objetos (OOP) es un **paradigma** de computación que se organiza en base a **objetos** en vez de acciones y **datos** en lugar de lógica.

Esto requiere un gran cambio de enfoque respecto a la programación imperativa tradicional que están acostumbrados a usar puesto que el enfoque estará en cuáles son los objetos que vamos a manipular en vez de la lógica para manipularlos.

3.1. Objetos

En el contexto de programación, un objeto es un elemento que tiene un comportamiento y un estado definido, comúnmente llamados métodos y campos (este último también aparece en la literatura como propiedades o variables de instancia).

El principal objetivo de utilizar objetos es poder crear estructuras para almacenar información.

Existen muchos beneficios de utilizar *OOP*, pero algunas de las propiedades más importantes son:

- **Transparencia:** La información almacenada dentro del objeto no puede ser "vista" desde afuera de éste.
- **Encapsulación:** Cada objeto maneja sus propios datos y funcionalidades.
- **Composición:** Todos los objetos pueden contener a otros (similar al concepto de composición de funciones en matemática: $f \circ g$).
- **Separación de responsabilidades:** Al utilizarse correctamente, *OOP* permite estructurar un programa en secciones, cada una respondiendo a una responsabilidad

específica, lo que añade modularidad al código.

- **Polimorfismo:** Es la capacidad de un objeto de tipo A de verse y poder utilizarse como uno de tipo B. Esto se verá en más detalle cuando se aplique *OOP* en *Java*
- **Delegación:** Cada objeto ejecuta solo las acciones que le corresponden. Si algo no le corresponde, entonces le manda un mensaje a otro (idealmente a quien si le corresponde) que lo haga. Básicamente es un "si no es mi trabajo que lo haga otro".

3.1.1. Interacciones entre objetos

Por las propiedades de transparencia y encapsulación, un objeto no puede acceder directamente a los componentes de otro, pero entonces: ¿Cómo obtengo la información almacenada dentro de un objeto?

Para interactuar con los objetos existe el concepto de *mensaje*, este concepto hace referencia a que en vez de acceder directamente a los componentes internos de un objeto, "se le pide" al objeto que realice una acción (esto se conoce como *message passing*). Luego, cada objeto decide lo que debe hacer en base al mensaje que recibe. La manera en que un objeto decide qué acción tomar con cada mensaje se conoce como *method-lookup* y se explicará más adelante.

3.2. Clases

Para introducir el concepto de clases empecemos con un ejemplo: si preparo dos tortas siguiendo exactamente la misma receta, con los mismos ingredientes y la misma preparación, entonces surge la duda, ¿son estas dos tortas la misma? La respuesta lógica es que no, son tortas individuales distintas entre ellas a pesar de que su preparación haya sido la misma.

Haciendo un símil con los objetos, podríamos decir que los ingredientes de la torta son sus propiedades y la manera de prepararla son sus métodos, entonces tendríamos dos objetos con las mismas propiedades y métodos pero distintos entre sí, aquí es cuando entran las clases. Una clase es una manera de agrupar objetos, lo que informalmente podría llamarse el *tipo del objeto*. Más formalmente, una clase es una entidad en la programación orientada a objetos a través de la cual se definen las características de todos los objetos pertenecientes al grupo definido por la clase, por esta razón un objeto particular suele entenderse como una *instancia de una clase*.

Volviendo al ejemplo de las tortas, la clase podría verse como la receta utilizada para preparar los pasteles.

La introducción del concepto de clases permite agregar otras características fundamentales de *OOP*.

3.2.1. Herencia

En programación la herencia se entiende como la *especialización* de una clase, esto se ilustra en la figura 3.1[1]

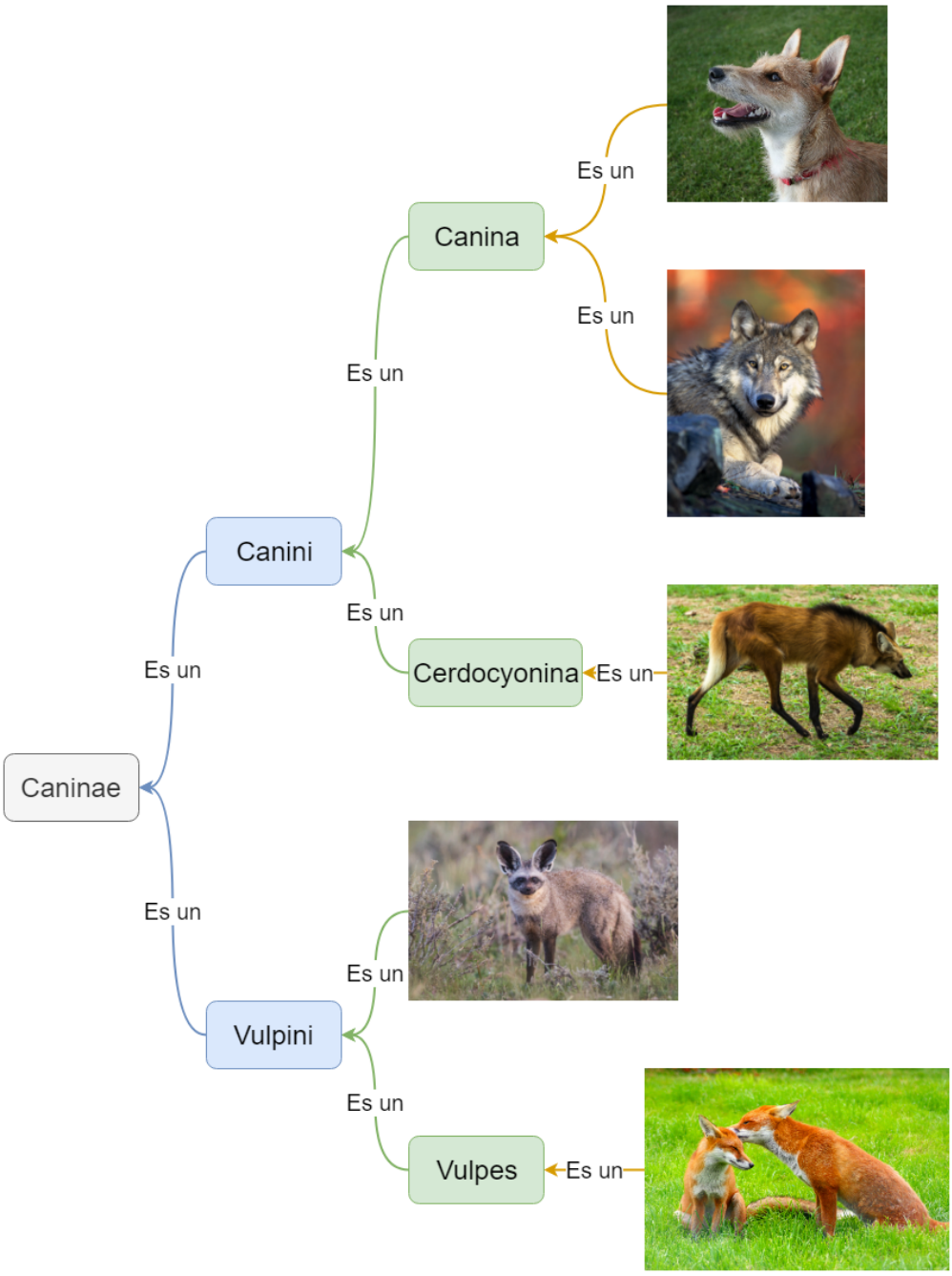

Figura 3.1: Ejemplo de herencia.

[1]Es importante resaltar que los animales no son objetos

En la figura se tiene que cada especie y subespecie es una clase, además se puede apreciar que esto permite darle una organización jerárquica a nuestras clases. Cuando hablamos de herencia en *OOP*, se suele llamar a la clase que está heredando de otra como *clase hija* o *subclase* y a la otra como *clase padre* o *superclase*.

Esta propiedad es importante no sólo por la capacidad de definir subtipos, sino que porque todas las clases hijas **heredan las funcionalidades** de la clase padre. Como esto se cumple para todas las clases de la jerarquía, se tiene que la herencia es una relación transitiva, de modo que $(\forall f,\ f \in \mathcal{C}_0 \mid \mathcal{C}_1 \subset \mathcal{C}_0 \wedge \mathcal{C}_2 \subset \mathcal{C}_1 \implies f \in \mathcal{C}_2)$

Como veremos más adelante, uno de los principales beneficios al momento de utilizar herencia dentro del contexto de programación es evitar la duplicación de código, pero es importante notar que esto es una consecuencia y no el objetivo de utilizar herencia. En un programa bien construido la herencia **debe tener coherencia lógica**, i.e. si bien tanto un avión como un pato pueden volar no tiene sentido crear una superclase para ambos porque son conceptualmente demasiado distintos entre sí.

3.2.2. Principios SOLID

Los **principios SOLID** son convenciones que se tienen en cuenta al momento de utilizar orientación a objetos y que permiten mantener una estructura robusta. Veremos más adelante que existen casos en los que se deben romper algunos de estos principios al momento de diseñar un programa para asegurar la mantenibilidad y extensibilidad del código, pero dentro de lo posible siempre se debe intentar respetar estos principios.

Single-responsibility principle

Una clase debe tener una y solamente una razón para cambiar, por lo que toda clase debe tener una sola responsabilidad.

Para entender esto, considere el siguiente problema: tenemos figuras geométricas y queremos calcular sus áreas.

Una posible solución sería crear una clase que represente una figura geométrica y que de acuerdo al tipo de figura que nos interese, entonces podríamos tener un método:
`calculateArea(String)` que se utilice como `calculateArea("Rectangle")`. Esto funcionaría, pero no respeta el principio, ya que tenemos una sola clase que se encarga de calcular el área de todas las figuras.

Una solución que sí respeta el principio es la de crear clases distintas para cada tipo de figura y que cada una de estas sepa cómo calcular su propia área (en esto se puede aprovechar la herencia).

Open-Closed principle

Los objetos o entidades deben estar abiertos para extenderse, pero cerrados para modificarse.

Este principio hace referencia a que debe ser fácil agregar funcionalidades nuevas o específicas a un programa sin necesidad de cambiar las funcionalidades y propiedades que ya tiene. Uno de los aprendizajes importantes del curso es cómo lograr cumplir con este principio, los capítulos referentes a patrones y metodologías de diseño muestran técnicas y patrones relevantes en este aspecto.

El mismo ejemplo utilizado para el principio anterior aplica aquí. Si tenemos una sola clase que calcula las áreas de todo tipo de figuras, agregar un nuevo tipo de figura implicaría modificar el método `calculateArea(String)`, mientras que si se tiene cada figura como una clase individual, agregar una nueva figura sería simplemente agregar una nueva clase.

Liskov's substitution principle

Sea $q(x)$ una propiedad demostrable para objetos x de tipo T. Entonces $q(y)$ debe ser demostrable para objetos y de tipo S donde S es un subtipo de T.[2]

En terminos más simples, esto quiere decir que las subclases siempre deben ser reemplazables por su clase padre.

Consideren el siguiente problema, queremos crear un modelo para representar aves, crearemos dos en particular: *Palomas* y *Colibríes*. Podemos agrupar estas aves en una clase *Ave* y definir que todas las aves pueden volar. Hasta aquí todo bien.

¿Pero qué pasa si ahora quiero agregar *Pingüinos*? Los pingüinos no pueden volar, pero definimos que todas las aves pueden volar. Nuestra definición inicial rompería entonces el principio de *Liskov*.

Una solución a esto sería crear una nueva clase que represente a las aves voladoras y que sea un subtipo de Ave, de esta forma, un pingüino sería un ave, mientras que una paloma sería un ave voladora.

Interface segregation principle

Un cliente nunca debiera estar forzado a implementar una interfaz que no ocupe o depender de métodos que no ocupe.

Una interfaz es una "promesa" que hace el programador con un cliente (quien use el código) en la que define cuáles son las acciones que puede realizar cualquier clase que implemente dicha interfaz. Este concepto se entenderá mejor cuando veamos aplicaciones de esto.

Tomemos como ejemplo el mismo de las aves del principio anterior. La solución original que se planteó de darle a todas las aves la habilidad de volar no rompia solamente el principio de Liskov sino que este también. Si hubieramos simplemente definido el pingüino como una subclase de ave, habríamos estado forzados a que el pingüino tuviera la capacidad de volar (aún cuando podríamos haber definido el método de tal forma que no hiciera nada).

[2] https://en.wikipedia.org/wiki/Barbara_Liskov

La solución es la misma de antes.

Dependency inversion principle

Las entidades deben depender de abstracciones y no de implementaciones. Los modulos de alto nivel no deben depender de modulos de bajo nivel, sino que deben depender de abstracciones.

Este principio suena más complicado de lo que realmente es. Lo que quiere decir es que al tener dependencias entre clases, la clase que tiene dependencia en otra no debe depender de implementaciones particulares de esta clase, sino que de una abstracción de estas implementaciones.

Usando las aves como ejemplo nuevamente, si tuvieramos una clase con un método `alimentar` para alimentarlas, la implementación de esta clase no debe depender de las implementaciones particulares, e.g. definir un método `alimentar(Paloma)` estaría violando este principio.

La manera correcta de implementar esto sin romper el principio sería crear un único método `alimentar(Ave)`

Capítulo 4

OOP de *Python* a *Java*

En este capítulo veremos como implementar los conceptos vistos en el capítulo anterior en *Java* partiendo desde ejemplos en *Python* para facilitar la transición de uno al otro.

4.1. Introducción

Consideremos un ejemplo básico para comenzar: queremos imprimir un mensaje en consola.

Podríamos hacer el clásico *Hello world!*, pero que fome, en cambio imprimamos otro mensaje.

Creen un archivo `ejemplo_basico.py` y ábranlo con Para imprimir un texto en consola en *Python* haríamos:

```
print("My name is Giorno Giovanna, and I have a dream.")
```

O lo que sería más correcto de acuerdo a las convenciones de *Python*:

```
if __name__ == "__main__":
    print("My name is Giorno Giovanna, and I have a dream.")
```

Luego, si queremos ejecutar el script haríamos:

```
python3 ejemplo_basico.py
```

o en caso de utilizar *Windows*:

```
py -3 ejemplo_basico.py
```

En *Java* reproducir este mismo ejemplo es un tanto más complicado ya que necesitaremos escribir más líneas de código. Para crear un programa equivalente en *Java*, primero crearemos un archivo `EjemploBasico.java`, luego en el editor de texto que prefieran escriban el código:

```
public class EjemploBasico {
```

```
  public static void main(String[] args) {
    System.out.println("My name is Giorno Giovanna, and I have a dream.");
  }
}
```

Veremos en detalle las diferencias entre la sintaxis de ambos ejemplos, pero primero veamos como ejecutar el programa para ver que efectivamente hace lo mismo que el de *Python*, para esto eben ejecutar en consola:

```
javac EjemploBasico.java
java EjemploBasico
```

El primer comando creará un archivo `EjemploBasico.class`, este es un archivo pre-compilado (es importante que en las tareas **NO ENTREGUEN** los archivos `.class`, ya que no los podemos revisar), luego el segundo comando compila y ejecuta el programa. Esto se explicará en el capítulo **??**.

Ahora, veamos las diferencias entre ambos programas. El código en *Python* es bastante fácil de seguir. £Pero por qué en *Java* hay que definir tantas cosas solamente para imprimir un mensaje en consola?

Vamos por partes, lo primero que deben notar es que la línea con el llamado a `println(...)` termina con un `;`, esto debe ser así para todas las instrucciones, esto puede no parecer tan importante a primera vista, pero marca una diferencia enorme respecto a *Python*, ya que a diferencia de *Python* la indentación no es importante. Cuando programamos en *Python* la indentación es lo que define donde compienza y termina una definición, en *Java* en cambio, esto se define entre llaves, donde la apertura de una marca el inicio y el cierre el fin.

Luego, tenemos la definición `public static void main(String[] args)` este es un método especial que será el punto de entrada del programa, por lo que al ejecutar el código se ejecutarán todas las instrucciones definidas dentro del método.

Por último, tenemos que todo esto va dentro de la definición de una clase `EjemploBasico`, esto es necesario ya que *Java* es un lenguaje (casi) totalmente orientado a objetos *fuertemente tipado*. De momento basta que sepan que los programas siguen esa sintaxis, en el capítulo **??** veremos más en detalle el funcionamiento de *Java* y profundizaremos en este tema.

4.2. *Input*

En la sección anterior mostramos un programa que era capaz de imprimir un mensaje en pantalla, pero siempre que lo ejecutemos hará lo mismo. £Qué hacemos para que el programa reciba *input* de un usuario? Para esto tendrémos dos opciones:

4.2.1. Opción 1: Argumentos por consola

La primera opción es la que usan la mayoría de aplicaciones de consola que vienen integradas en los sistemas operativos, i.e. recibir los parámetros como argumentos entregados al momento de ejecutar el programa, por ejemplo:

```
cd path/to/dir
```

Modifiquemos un poco el ejemplo anterior para que reciba parámetros desde consola, en el caso de *Python*, para recibir los argumentos se debe hacer una llamada al sistema, el siguiente ejemplo muestra como hacer esto:

```
import sys

if __name__ == "__main__":
    # Tomamos todos los argumentos menos el primero porque en Python el
    ↪  primer
    # argumento siempre es el nombre del archivo.
    name = " ".join(sys.argv[1:])
    print(f"My name is {name}, and I have a dream.")
```

Y luego podemos ejecutarlo como:

```
python3 ejemplo_basico.py Ignacio Slater
```

Por otro lado, en *Java* tendríamos:

```
public class EjemploBasico {

  public static void main(String[] args) {
    System.out.println("My name is " + String.join(" ", args) + ", and I
    ↪  have a dream.");
  }
}
```

Como pueden ver, en este caso no hay necesidad de hacer una llamada explícita al sistema para obtener los argumentos, ya que el método `main` lo hace por defecto. De manera similar a lo que hicimos para ejecutar el programa en *Python*, ahora ejecutaremos éste como:

```
javac EjemploBasico.java
java EjemploBasico Ignacio Slater
```

En ambos casos el resultado debiera ser el mismo.

4.2.2. Opción 2: Pedir parámetros interactivamente

La otra opción es hacer que el usuario ingrese parámetros durante la ejecución del programa.

En *Python*, si quisieramos hacer eso, tendríamos que modificar el programa anterior como:

```python
if __name__ == "__main__":
    name = input("Write your name: ")
    print(f"My name is {name}, and I have a dream.")
```

Nuevamente, en *Java* el código sería más extenso:

```java
import java.util.Scanner;

public class EjemploBasico {

  public static void main(String[] args) {
    System.out.println("Write your name: ");
    String name = new Scanner(System.in).nextLine();
    System.out.println("My name is " + name + ", and I have a dream.");
  }
}
```

No entraremos en más detalles dentro de estos conceptos ya que no serán utilizados durante el curso.

4.3. Objetos y clases

4.4. Constructores

Supongamos ahora que queremos representar un punto con 3 dimensiones. Esto podemos representarlo con una clase `Point3D`, de la siguiente forma:

```python
class Point3D:
```

Parte III

Patrones y metodologías de diseño