CC3002 - Metodologías de diseño y programación

Juan-Pablo Silva and Ignacio Slater

Departamento de Ciencias de la Computación, Universidad de Chile

10 de abril de 2020

Índice general

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*.

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 14*, 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 recomendamos que **desinstalen dicha versión** antes de proceder a instalar la más nueva porque puede generar conflictos.

1.1. Linux (x64)

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

Para cualquier distribución de *Linux x64*, desde la terminal:

```
wget https://bit.ly/344NYw9
tar xvf openjdk-14*_bin.tar.gz
sudo mv jdk-14 /usr/lib/jdk-14
```

Luego, para verificar que el binario se haya extraído correctamente:

```
export PATH=$PATH:/usr/lib/jdk-14/bin
java -version
```

Si el binario se instaló correctamente, este comando debiera retornar algo como:

```
openjdk version "14" 2020-03-17
OpenJDK Runtime Environment (build 14+36-1461)
OpenJDK 64-Bit Server VM (build 14+36-1461, mixed mode, sharing)
```

En caso de que no haya resultado, intenten realizar la instalación nuevamente o procedan a la siguiente opción.

Si se instaló correctamente, entonces el último paso es agregar el *JDK* a las variables de entorno del usuario, para esto primero deben saber qué *shell* están ejecutando, pueden ver esto con:

```
echo $SHELL
```

En mi caso, esto retorna:

```
/usr/bin/zsh
```

Luego, deben editar el archivo de configuración de su *shell*, en mi caso ese sería `~/.zshrc` (en *bash* sería `.bashrc`) y agregar al final del archivo la línea:

```
export PATH=$PATH:/usr/lib/jdk-14/bin
```

1.1.2. Opción 2: *Oracle JDK*

Primero deben descargar el *JDK* desde el sitio de *Oracle* (asumiremos que descargaron la versión `.tar.gz`). Luego, desde el directorio donde descargaron el archivo:

```
tar zxvf jdk-14.interim.update.patch_linux-x64_bin.tar.gz
sudo mv jdk-14.interim.update.patch /usr/lib/jdk-14.interim.update.patch
```

Después, de la misma forma que se hizo con la opción 1:

```
export PATH=$PATH:/usr/lib/jdk-14.interim.update.patch/bin
java -version
```

Si este comando funciona, entonces deberán modificar el archivo de configuración de su *shell* para incluir la línea:

```
export PATH=$PATH:/usr/lib/jdk-14.interim.update.patch/bin
```

1.2. *Windows*

1.2.1. Opción 1: *OpenJDK* con *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:

```
cinst openjdk -y
```

Para probar que la instalación se haya hecho de forma correcta, cierren y abran *Powershell* y ejecuten el comando:

```
java -version
```

El input debiera ser algo como:

```
openjdk version "14" 2020-03-17
OpenJDK Runtime Environment (build 14+36-1461)
OpenJDK 64-Bit Server VM (build 14+36-1461, mixed mode, sharing)
```

Si esto no funciona, proceda con la siguiente opción.

1.2.2. Opción 2: *OpenJDK* sin *Chocolatey*

Primero deben descargar los binarios del *JDK* desde aquí.

Con los binarios descargados, extraigan el `.zip` en algún directorio y luego abran *Powershell* como administrador y ubíquense en la carpeta donde extrajeron el archivo. Una vez ahí, ejecuten:

```
Move-Item -Path .\jdk-14 -Destination $Env:Programfiles
[Environment]::SetEnvironmentVariable("JAVA_HOME",
                                      "$Env:Programfiles\jdk-14")
[Environment]::SetEnvironmentVariable(
  "Path",
  [Environment]::GetEnvironmentVariable('Path',
  [EnvironmentVariableTarget]::Machine) + ";$($Env:JAVA_HOME)\bin",
  [EnvironmentVariableTarget]::Machine)
```

Para probar que la instalación se haya hecho de forma correcta, cierren y abran *Powershell* y ejecuten el comando:

```
java -version
```

El input debiera ser algo como:

```
openjdk version "14" 2020-03-17
OpenJDK Runtime Environment (build 14+36-1461)
OpenJDK 64-Bit Server VM (build 14+36-1461, mixed mode, sharing)
```

Si esto no funciona, proceda con la siguiente opción.

1.2.3. Opción 3: *Oracle JDK*

Lo primero es descargar el *JDK* desde el sitio de *Oracle* y una vez descargado ejecuten el instalador y sigan las instrucciones.

Por último, deben agregar el *path* de *Java* a las variables de entorno, para esto abran *Powershell* como administrador y ejecuten:

```
[Environment]::SetEnvironmentVariable("JAVA_HOME",
                                      "$Env:Programfiles\Java\jdk-14")
[Environment]::SetEnvironmentVariable(
  "Path",
  [Environment]::GetEnvironmentVariable('Path',
  [EnvironmentVariableTarget]::Machine) + ";$($Env:JAVA_HOME)\bin",
  [EnvironmentVariableTarget]::Machine)
```

Para probar que la instalación se haya hecho de forma correcta, cierren y abran *Powershell* y ejecuten el comando:

```
java -version
```

El input debiera ser algo como:

```
openjdk version "14" 2020-03-17
OpenJDK Runtime Environment (build 14+36-1461)
OpenJDK 64-Bit Server VM (build 14+36-1461, mixed mode, sharing)
```

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 versionado utilizado por el 70 % de sus usuarios).

Existen muchos otros sistemas de versionado 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:

Opción 1: *Chocolatey* (Recomendado)

Asumiendo que se haya instalado *Chocolatey* como se mostró en la sección 1.2.1

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

Opción 2: *Git for Windows*

Git for Windows es un conjunto de herramientas que incluye *Git BASH* (una interfaz de consola que emula la terminal de un sistema *UNIX* que viene con *Git* instalado), *Git GUI* (una interfaz gráfica para manejar *Git*) e integración con *Windows Explorer* (esto significa que pueden hacer *clic* derecho en una carpeta y abrirla desde *Git BASH* o *Git GUI*).

Para instalarla deben descargar el cliente desde el sitio oficial de *Git for Windows* y seguir las instrucciones del instalador.

2.1.3. *MacOS*

Opción 1: *Homebrew* (Recomendado)

Lo primero es instalar *Homebrew* si no lo han hecho ya, para esto ejecuten:

```
/bin/bash -c "$(curl -fsSL
↪  https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

Ya con *Homebrew* instalado, basta correr el siguiente comando en una terminal:

```
brew install git
```

[2]Por ejemplo *Manjaro*.

Opción 2: *Git for Mac*

Para esto deben descargar el instalador desde aquí y seguir las instrucciones.

Esta opción se sabe que puede tener problemas si se había instalado anteriormente otra versión de *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* instalado con *Chocolatey*):

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

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 versionado 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 manejar *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*).

Capítulo 3

Recomendación: *Cmder* para *Windows*

La consola por defecto de *Powershell*, si bien es funcional, carece de muchas herramientas para trabajar cómodamente que sí ofrecen las terminales de sistemas *UNIX*. Por esta razón recomendamos instalar *Cmder*.

Cmder es un *emulador de consola* que permite correr múltiples *shells* en una misma interfaz, incluyendo *Powershell*, *CMD* y *WSL* entre otras.

Para instalar *Cmder* utilizaremos *Chocolatey*. En una terminal de *Powershell* con permisos de administrador ejecuten:

```
cinst cmder -y
```

Con esto basta para tener *Cmder* instalado, pero una de las principales ventajas de utilizar este emulador de consola es la capacidad de personalizarlo.

Lo primero que haremos será descargar las fuentes de *Powerline* que soportan más caracteres que los que trae *Powershell* por defecto. Para esto ejecuten (puede ser en la misma terminal de *Powershell* o en *Cmder*):

```
git clone "https://github.com/powerline/fonts.git"
Set-Location fonts
.\install.ps1
```

Ahora, desde *Cmder* vayan a la configuración con `Win+Alt+P`. Una vez ahí, en la pestaña `General >Fonts` cambien `Main console font` y `Alternative font` por alguna de las que provee *Powerline* (por ejemplo `DejaVu Sans Mono for Powerline`).

Luego, en la pestaña `Startup` de las configuraciones, seleccionen en la opción `Specified named task` la terminal por defecto que se ejecutará al abrir *Cmder*.

Lo siguiente será instalar herramientas que mejorarán la interacción de *Powershell* con *Git* y entregar de mejor forma la información al momento de usar la consola. Para esto, deberán ejecutar los siguientes comandos:

```
Install-PackageProvider NuGet -MinimumVersion '2.8.5.201' -Force
Set-PSRepository -Name PSGallery -InstallationPolicy Trusted
Install-Module -Name 'posh-git'
Install-Module -Name 'oh-my-posh'
Install-Module -Name 'Get-ChildItemColor'
```

Lo último es configurar el perfil de *Powershell*, esto se hace en un archivo que es el equivalente a `.bashrc` de los sistemas *Linux*. Para abrir este archivo ejecuten:

```
ise $PROFILE
```

Esto abrirá el entorno de *scripting* de *Powershell* (si nunca han configurado la consola, entonces debiera estar vacío). Como último paso, escriban lo siguiente en el archivo de configuración y guarden los cambios.

```
# Helper function to set location to the User Profile directory
function cuserprofile { Set-Location ~ }
Set-Alias ~ cuserprofile -Option AllScope

Import-Module 'posh-git'
Import-Module 'oh-my-posh' -DisableNameChecking

# CHOCOLATEY PROFILE
$ChocolateyProfile = "$env:ChocolateyInstall\helpers\chocolateyProfile.psm1"
if (Test-Path($ChocolateyProfile)) {
  Import-Module "$ChocolateyProfile"
}

remove-item alias:cd -force
function cd($target) {
  if ((Test-Path "$target.lnk") -or $target.EndsWith(".lnk")) {
    if (-not $target.EndsWith(".lnk")) {
      $target = "$target.lnk"
    }
    $sh = New-Object -com wscript.shell
    $fullpath = Resolve-Path $target
    $targetpath = $sh.CreateShortcut($fullpath).TargetPath
    Set-Location $targetpath
  }
  else {
    Set-Location $target
```

```
  }
  <#
    .SYNOPSIS
      Moves to the a target directory.
    .DESCRIPTION
      Sets the current location to the given target folder or the folder
↪   pointed by its
      symlink.
    .PARAMETER target
      The target directory
  #>
}

function U([int]$Code) {
  if ((0 -le $Code) -and ($Code -le 0xFFFF)) {
    return [char] $Code
  }

  if ((0x10000 -le $Code) -and ($Code -le 0x10FFFF)) {
    return [char]::ConvertFromUtf32($Code)
  }

  throw "Invalid character code $Code"
  <#
  .DESCRIPTION
      Helper function to show Unicode characters
  #>
}

Set-PSReadlineOption -BellStyle None
Set-Theme Honukai
```

La última línea solamente define el *tema* de la consola, pueden ver una lista de *temas* disponibles en el repositorio de *Oh-My-Posh*

Parte II

Objetos en *Java*

Capítulo 4

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.

4.1. 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".

4.1.1. 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.

4.2. 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*.

4.2.1. Herencia

En programación la herencia se entiende como la *especialización* de una clase, esto se ilustra en la figura 4.1[1]

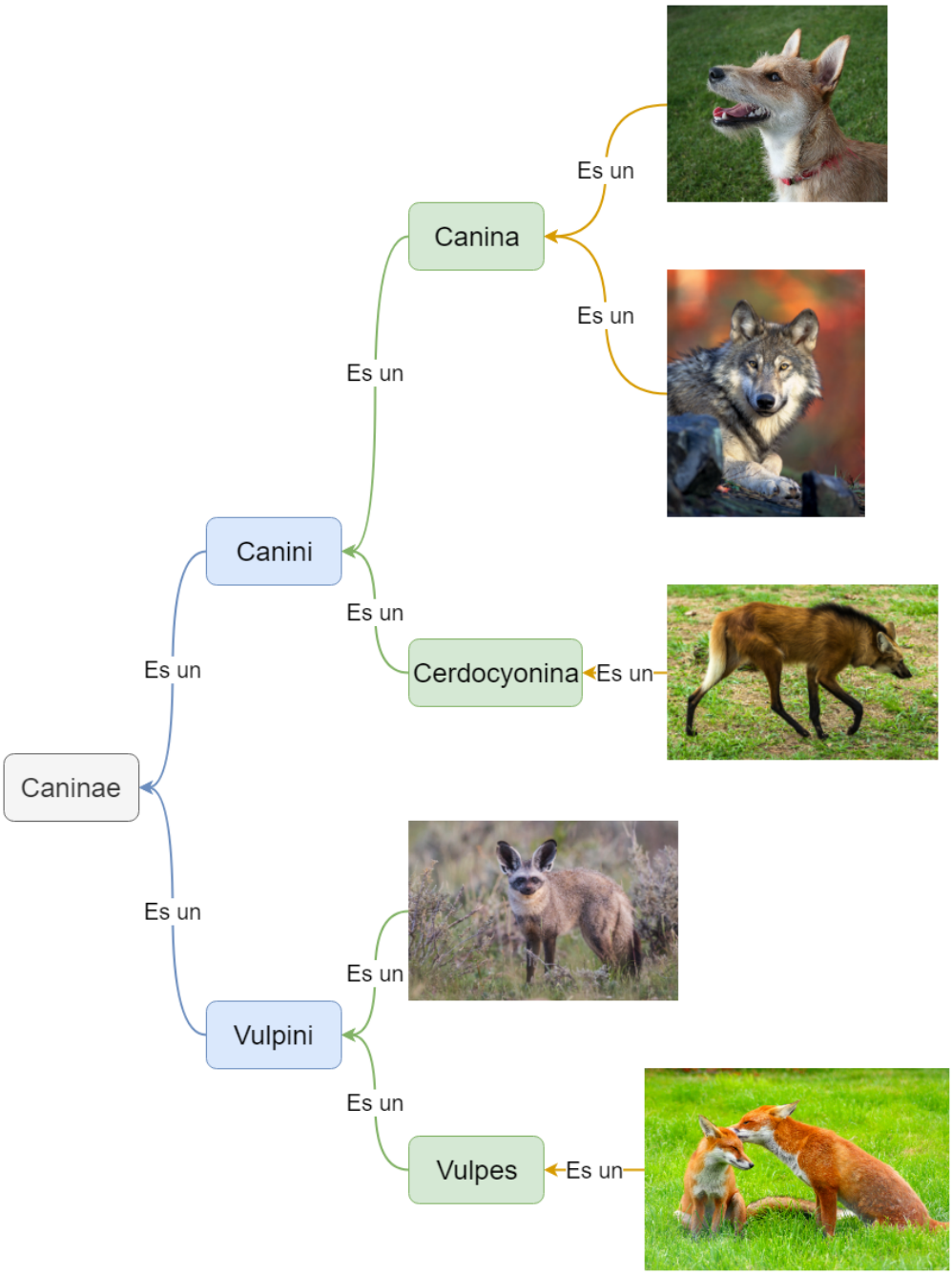

Figura 4.1: Ejemplo de herencia.

[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.

4.2.2. 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 5

OOP: de *Python* a *Java*

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

5.1. Lo básico

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:

```java
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 deben 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 6.

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 dónde comienza 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 6 veremos más en detalle el funcionamiento de *Java* y profundizaremos en este tema.

5.2. Control de flujo

En programación, se le llama *control de flujo* a las instrucciones que dependerán de una condición para "decidir" qué acción debe realizar a continuación el programa, por ejemplo, las instrucciones *if-else* o *while*.

Comencemos por la más simple, implementar una instrucción *if-else* en *Python* se haría de la siguiente forma:

```
if a > b:
    print("a > b")
elif a == b:
    print("a = b")
else:
    print("a > b")
```

En este caso, el inicio y fin de cada rama del bloque *if-else* depende de la indentación, ya mencionamos que esta no es importante en *Java*. El siguiente código es equivalente al anterior:

```
if (a > b) {
  System.out.println("a > b");
} else if (a == b) {
  System.out.println("a == b");
} else {
  System.out.println("a < b");
}
```

Ahora, veamos la instrucción *while*. En *Python*:

```
while i > 0:
  print(i)
  i -= 1  # Equivalente a hacer i = i - 1
```

Lo que en *Java* se podría hacer como:

```
while (i > 0) {
  System.out.println(i);
  i -= 1;
}
```

El siguiente código es equivalente al anterior:

```
while (i > 0) {
  System.out.println(i--);
}
```

Análogamente, también se puede hacer `i++`.

Ejercicio 5.1. Pruebe cambiar la expresión `i--` por `--i`, los resultados son distintos. £Por qué pasa esto?

Java además tiene otra manera de implementar un *loop* conocida como *do-while*, esta funciona exactamente igual que una expresión *while* común con la diferencia de que **siempre** el

bloque dentro del *while* se ejecutará al menos 1 vez. El mismo ejemplo anterior:

```
do {
  System.out.println(i--);
} while (i > 0);
```

Otra instrucción de control de flujo típica es *for*. En *Python* podemos hacer:

```
for i in range(0, 5):
    print(i)
```

En *Java* podemos hacer lo mismo como:

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

Expliquemos un poco esta sintaxis. La primera parte del *for* indica él o los valores iniciales del *loop* (en este caso define una variable `i` con el valor 0), la segunda parte es la condición que se debe cumplir para que se ejecute el código dentro del bloque (aquí, se imprimirá en consola mientras `i` sea menor a 5), y, por último, la tercera parte es la instrucción que se ejecutará cada vez que termine de ejecutarse el bloque (para el ejemplo, en cada iteración se le suma 1 a `i`).

Una de las características más útiles de la instrucción *for* es la capacidad de iterar sobre una estructura de datos, un ejemplo podría ser:

```
pairs = [0, 2, 4, 6, 8]
for i in pairs:
    print(i)
```

En *Java* podemos hacer algo equivalente de la forma:

```
int[] pairs = new int[]{0, 2, 4, 6, 8};
for (int i : pairs) {
  System.out.println(i);
}
```

Adicionalmente, en *Java* existe otra instrucción de control de flujo que no está presente en *Python*, esta se conoce como *switch-case* y es parecido a hacer un *if-else* de una manera más compacta, pero que tiene más restricciones. Un ejemplo de esta instrucción sería:

Ejercicio 5.2. Borre las instrucciones `break` del ejemplo anterior y compruebe si el comportamiento es el mismo.

```java
switch (a) {
  case 0:
    System.out.println("a = 0");
    break;
  case 1:
    System.out.println("a = 1");
    break;
  case 2:
    System.out.println("a = 2");
    break;
  default:
    System.out.println("a is not in [0, 2]");
}
```

5.3. *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 tendremos dos opciones:

5.3.1. 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:

```python
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.

5.3.2. Opción 2: Pedir parámetros interactivamente

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

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

```
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:

```
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.

Ejercicio 5.3. Escriba un programa en *Java* que reciba interactivamente texto desde consola e imprima de vuelta el mensaje entregado, hasta que se ingrese una línea vacía.

Para comprobar si un texto está vacío, pueden hacerlo como `text.isEmpty()`, de manera similar, si quieren comprobar que no está vacío se hace como `!text.isEmpty()`

5.4. Tipos

Tanto *Python* como *Java* son lenguajes orientados a objetos, esto quiere decir que todas las variables que se utilicen en un programa son objetos o tipos primitivos. La diferencia está en que, al ser fuertemente tipado, en *Java* es necesario definir explícitamente el tipo de todas las variables. Esto hace que en *Java* todo deba definirse dentro de una clase.

5.4.1. Tipos primitivos

Los *tipos primitivos* son datos que ocupan una cantidad fija de espacio en la memoria. Esto hace que sean más eficientes de utilizar ya que una vez que se les asigna un lugar en la memoria difícilmente tendrán que moverse a otra dirección (algo que sucederá mucho con los objetos).

La tabla 5.1 muestra los tipos primitivos de *Java* (en *Python* la definición de estos es más complicada, así que se omitirá).

Tipo	Uso de memoria	Rango	Uso	Valor por defecto
`byte`	8-bits	[-128, 127]	Representar arreglos masivos de números pequeños	`0`
`short`	16-bits	[-32.768, 32.767]	Representar arreglos grandes de números pequeños	`0`
`char`	16-bits	[0, 65.535]	Caracteres del estándar *ASCII*	`'\u0000'`
`int`	32-bits	$[-2^{31}, 2^{31}-1]$	Estándar para representar enteros	`0`
`long`	64-bit	$[-2^{63}, 2^{63}-1]$	Representar enteros que no quepan en 32-bits	`0L`
`float`	32-bit	Estándar *IEEE 754x32*	Representar números reales cuando la precisión no es importante	`0.0f`
`double`	64-bit	Estándar *IEEE 754x64*	Representar números reales con mediana precisión	`0.0`
`boolean`	No definido	`true` o `false`	Valores binarios	`false`

Cuadro 5.1: Tipos primitivos en *Java* (los valores por defecto son los que toma la variable si no se le entrega un valor explícitamente **y es un campo** de una clase)

Todos los tipos primitivos en *Java* tienen un objeto «equivalente» para brindar operaciones que no se pueden realizar directamente con los tipos primitivos, e.g. utilizarlos como tipos genéricos (esto se verá en la última parte del apunte).

Noten que la sintaxis en *Java* para los valores *boolean* es `true` y `false`, mientras que en *Python* es `True` y `False`.

Importante. *Los `String` no son tipos primitivos.*

5.4.2. Objetos y clases

Como mencionamos anteriormente, un objeto es una instancia de una clase, por lo que para definir un objeto primero debe definirse la clase a la que pertenece. En ambos lenguajes, las clases se definen con la *keyword* `class`. Para ver la sintaxis en ambos casos comencemos con un ejemplo simple, crear una clase que represente un punto en 2 dimensiones.

En *Python* esto se puede lograr de la siguiente forma:

```python
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Luego, se pueden crear y utilizar instancias de esta clase como en el siguiente ejemplo:

```python
point = Vector2D(1, 3)
print(point.x)
print(point.y)
```

Y en *Java*:

```java
class Vector2D {
  double x, y;

  Vector2D(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

Para crear instancias de una clase en *Java* se utiliza la *keyword* `new`, de esta forma se puede hacer:

```java
Vector2D point = new Vector2D(1, 3);
System.out.println(point.x);
System.out.println(point.y);
```

Tanto `__init__` como `Vector2D(double, double)` son los constructores de la clase y se explicarán en detalle en la siguiente sección.

Métodos

En el capítulo anterior se explicó como, además de almacenar datos, los objetos pueden realizar acciones mediante métodos, en *Python* un método se define como `def method_name(param1, param2, ...): ...`, en *Java* la sintaxis es un poco más tediosa ya que se debe explicitar el tipo de los parámetros que recibe el método y el tipo del dato que retorna de la forma `returnType methodName(param1Type param1, param2Type param2, ...) {...}`.

Agreguemos 2 métodos a la clase `Vector2D`, uno que sume las coordenadas de 2 puntos retornando un nuevo punto, y otro que imprima un punto en pantalla como `(x, y)`.

En *Python*:

```python
# Dentro de la clase Vector2D
def add(self, other_point):
    return Vector2D(self.x + other_point.x,
                    self.y + other_point.y)

def print(self):
    print(f"({self.x}, {self.y})")
```

Y se pueden utilizar estos métodos de la forma:

```python
point = Vector2D(1, 3)
new_point = point.add(Vector2D(-1, 2))
new_point.print()
```

En el caso de *Java* debemos especificar el tipo de retorno o, en el caso de que no retorne nada, `void`. Entonces:

```java
// Dentro de la clase Vector2D
Vector2D add(Vector2D otherPoint) {
  return new Vector2D(x + otherPoint.x, y + otherPoint.y);
}

void print() {
  System.out.println("(" + x + ", " + y + ")");
}
```

Y luego puede utilizarse de la forma:

```
Vector2D point = new Vector2D(1, 3);
Vector2D newPoint = point.add(new Vector2D(-1, 2));
newPoint.print();
```

Nota. *En el caso de Python, los métodos reciben un parámetro* `self` *que se le entrega implícitamente al momento de invocar la función (en el curso CC4101 - Lenguajes de programación se explica la razón detrás de esto).*

Java por otro lado no necesita de ese parámetro, por lo que basta que no se declaren parámetros en la definición del método para indicar que éste no recibe parámetros.

Un caso en particular a tener en cuenta es el lenguaje C, donde si no se especifican los parámetros que recibe una función se interpreta como que puede recibir cualquier número y tipo de parámetros. Si se quiere que efectivamente no se reciban parámetros se debe especificar explícitamente de la forma `returnType function(void) {...}`.

Ejercicio 5.4. Cree un método *printPolar()* que imprima el punto en coordenadas polares de la forma `(r, t)`, donde:

$$r = \sqrt{x^2 + y^2}$$

$$t = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{si } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{si } x < 0 \wedge y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{si } x < 0 \wedge y < 0 \\ \frac{\pi}{2} & \text{si } x = 0 \wedge y > 0 \\ -\frac{\pi}{2} & \text{si } x = 0 \wedge y < 0 \\ \text{indefinido} & \text{si } x = 0 \wedge y = 0 \end{cases}$$

Para esto, ocupe los métodos `Math.sqrt(double)` y `Math.atan(double)`, y el valor `Math.PI`.

5.5. Constructores

En la sección anterior creamos una clase `Vector2D` que se inicializaba con 2 parámetros. El método que se encarga de crear una instancia de una clase se llama constructor.

El constructor es un método especial que no tiene tipo de retorno (esto es distinto a que retorne `void`) y que se llama al momento de crear una instancia de una clase.

En *Python* el constructor de una clase se define como `def __init__(self, param1, param2, ...):`, mientras que en *Java* se hace creando un método con la firma `ClassName(Param1Type param1, Param2Type param2, ...) {...}`. En el caso de *Java*, el método constructor **debe llamarse igual que la clase** a la que pertenece.

Por defecto en *Python* y *Java*, si no se define un constructor explícitamente, las clases tienen un constructor vacío que se crea implícitamente y lo único que hace es reservar un espacio

en memoria para el objeto que se vaya a crear, esto es lo mismo que si se definiera una clase como:

```
class ClassName {
  ClassName() {}
}
```

Importante. *Si se define un constructor de forma explícita entonces no se crea el constructor por defecto.*

El objetivo de los constructores es definir las condiciones iniciales de un objeto, generalmente esto significa iniciar las variables de instancia.

Supongan ahora que queremos crear usuarios para alguna aplicación. Todos los usuarios deben tener un nombre que los identifique y opcionalmente pueden ingresar su género. £Cómo manejamos la existencia de parámetros opcionales?

En *Python* se pueden utilizar parámetros por defecto, por lo que se podría implementar esto como:

```
class User:
    def __init__(self, username, gender="Non specified"):
        self.username = username
        self.gender = gender
```

Esto no es posible de hacer en *Java* ya que no existen parámetros por defecto. En vez de eso lo que se puede hacer (y que no se puede hacer en *Python*) es crear múltiples constructores, entonces:

```
class User {
  String username, gender;

  User(String username, String gender) {
    this.username = username;
    this.gender = gender;
  }

  User(String username) {
    this(username, "Non specified");
  }
}
```

En el próximo capítulo profundizaremos en el funcionamiento de *Java* y explicaremos cómo funciona `this`, pero por ahora lo que necesitan saber es que `this(username, "Non specified")` llama al otro constructor con esos parámetros.

Ejercicio 5.5. Existen lenguajes de programación como *Smalltalk* que no tienen constructores, sino que los objetos se crean vacíos y los parámetros se inician luego con un método «normal».

£Por qué es conveniente que los constructores sean métodos especiales? Discuta

5.5.1. Casos especiales

Existen objetos especiales que pueden iniciarse sin necesidad de llamar explícitamente al constructor. En *Java* este es el caso de las clases `Array` y `String`.

El caso de los *strings* es el más típico ya que cuando creamos un objeto de tipo `String` no hacemos `new String(...)` (aunque se puede), sino que simplemente escribimos texto entre comillas dobles. Es importante notar en este punto que los *strings* se definen entre comillas dobles y los caracteres entre comillas simples, entonces `'a'` es un primitivo pero `"a"` es un objeto.

El otro caso son los arreglos, a diferencia de los *strings*, estos sí se crean utilizando la *keyword* `new`, pero no se llama explícitamente al constructor. En cambio, existen dos maneras de iniciar un arreglo.

Los arreglos tienen un **tamaño fijo** (un arreglo no es lo mismo que una lista), por lo que es necesario definir ese tamaño al momento de crearlo. La sintaxis para hacer esto es `T[] array = new T[size]`, esto va a crear un arreglo de tamaño `size` de elementos de tipo `T` lleno con valores nulos o, en caso de ser un tipo primitivo, con sus valores por defecto. Por ejemplo:

```
int[] ints = new int[3];
ints[0] = 100;
ints[2] = 11;
System.out.println(ints[0]); // Imprime 100
System.out.println(ints[1]); // Imprime 0
```

La otra manera de crear un arreglo es pasándole inmediatamente sus valores, la sintaxis es similar a la anterior con la diferencia de que no se define el tamaño (porque éste se infiere por la cantidad de valores). Así, el mismo arreglo del ejemplo anterior se puede crear de la siguiente forma:

```
int[] ints = new int[] {100, 0, 11};
```

Ejercicio 5.6. Cree una clase `Figure2D` con un método `printType()` que imprima el tipo de la figura. Para esto, considere una figura como un conjunto de puntos (de la clase `Vector2D`), donde `printType()` debe indicar que la figura es un punto si ésta tiene un solo punto, si tiene 2 puntos entonces dice que es una línea y si tiene 3 o más entonces es un polígono.

Resuelva este ejercicio utilizando solo constructores, y variables de instancia de tipo primitivo, `Vector2D`, *strings* y arreglos, y sin utilizar métodos adicionales (tampoco los de las clases `String` o `Array`) ni instrucciones de control de flujo (como `if` o `switch`).

Hint: Utilice varios constructores y ***al menos*** *una variable de instancia.*

5.6. Herencia

En las secciones anteriores dimos una definición para un punto en 2 dimensiones, pero un punto en un plano es algo bastante limitado. En esta sección tomaremos la definición inicial de nuestro punto y la generalizaremos para luego, en base a nuestra nueva definición, crear casos más específicos.

Consideren un vector como una línea que va desde el origen del sistema de coordenadas hasta un punto. Un vector en un espacio euclídeo de dimensión n es una n-tupla de números reales. Dada esta definición podemos definir una clase `VectorND` de la siguiente forma:[1]

```
class VectorND {
  double[] tail;

  VectorND(double[] tail) {
    this.tail = tail;
  }
}
```

Ahora, una característica de los vectores es que pueden sumarse entre ellos. La suma de dos vectores es sólo la suma de sus coordenadas, teniendo en cuenta las dimensiones de estos (un vector de m dimensiones, con $m \leq n$ es un vector de n dimensiones en el que todas las componentes $\mathbf{v}_i$ para $i > m$ son 0).

```
VectorND add(VectorND otherVector) {
  double[] bigger, smaller;
  if (tail.length > otherVector.tail.length) {
    bigger = tail;
    smaller = otherVector.tail;
  } else {
    bigger = otherVector.tail;
    smaller = tail;
  }
  double[] components = new double[bigger.length];
  for (int i = 0; i < smaller.length; i++) {
    components[i] = bigger[i] + smaller[i];
```

[1] La implementación en *Python* la pueden encontrar aquí

```
  }
  for (int i = smaller.length; i < bigger.length; i++) {
    components[i] = bigger[i];
  }
  return new VectorND(components);
}
```

Definamos además un método `print` que imprima el vector de la forma $(x_0, x_1, ..., x_{n-1})$.

```
void print() {
  String result = "(";
  for (int idx = 0; idx < tail.length; idx++) {
    result += tail[idx];
    if (idx < tail.length - 1) {
      result += ", ";
    }
  }
  System.out.println(result + ")");
}
```

Ahora, es poco común trabajar con vectores de n dimensiones, en general trabajaremos con vectores de dos o tres dimensiones, así que sería una buena idea tener clases específicas para dichos tipos. Ahora, ¿Por qué es una buena idea?

Como mencionamos en el capítulo 4, el objetivo de la herencia es la **especialización** de una clase, y eso es precisamente lo que queremos hacer aquí. Tomar un vector de n dimensiones y crear casos específicos para otros con dimensiones fijas (que tendrán propiedades propias de acuerdo a sus dimensiones).

Cambiemos el nombre de nuestra clase `Vector2D` por `Vector2D` y hagamos que sea una subclase de `VectorND`.

En *Python*, esto se haría de la siguiente forma:

```
class Vector2D(VectorND):
    def __init__(self, x, y):
        super().__init__([x, y])
```

Aquí, la clase entre paréntesis es la superclase de `Vector2D`. La línea `super().__init__([x, y])` lo que hace es llamar al constructor de la superclase (`VectorND`) y crear un objeto de tipo `Vector2D` con la lista que se le entregan como parámetros.

En el caso de *Java*, la herencia se define con la *keyword* `extends` de la siguiente forma:

```
class Vector2D extends VectorND {

  Vector2D(double x, double y) {
```

```
    super(new double[] {x, y});
  }
}
```

Como pueden ver, la sintaxis con la que se llama al constructor de la superclase es similar a la manera en la que se hace en *Python*.

Ahora, los métodos `add` y `print` que habíamos definido anteriormente en nuestra clase `Point2D` ya no son necesarios. ¿Por qué pasa esto? La explicación la dimos en el capítulo 4, ahí dijimos que los hijos heredan sus funcionalidades de su padre, así que si borramos el método `add` de nuestra clase `Vector2D`, entonces se llamará al método de la superclase. Entonces, deberíamos poder ejecutar estos métodos de la misma forma que habíamos hecho antes en la sección 5.4.2 y obtener el mismo resultado.

```
Vector2D point = new Vector2D(1, 3);
Vector2D newPoint = point.add(new Vector2D(-1, 2));
newPoint.print();
```

Si intentan correr el código anterior se darán cuenta que no compila. ¿Qué hicimos mal? El problema es que definimos `newPoint` como un objeto de tipo `Vector2D` pero el método `add` retorna un objeto de tipo `VectorND`. Siguiendo esa misma lógica podríamos pensar que hacer `point.add(new Vector2D(-1, 2))` también debiera fallar ya que el método espera recibir un objeto de tipo `VectorND` pero le pasamos uno de tipo `Vector2D`, pero como veremos a continuación ese no es el caso.

```
Vector2D point = new Vector2D(1, 3);
VectorND newPoint = point.add(new Vector2D(-1, 2));
newPoint.print();
```

Este código compila y retorna el resultado esperado. Ahora, veamos por qué al método `add` podemos pasarle un parámetro de tipo `Vector2D` pero no podemos definir `newPoint` como instancia de esa clase. La explicación es simple, `Vector2D` es más específico que `VectorND`.

Podemos pensar las clases como conjuntos. Sean $\mathcal{C}_0$, $\mathcal{C}_1$ y $\mathcal{C}_2$ clases tales que $\mathcal{C}_1$ y $\mathcal{C}_2$ heredan de $\mathcal{C}_0$. Esto quiere decir que

$$\mathcal{C}_0 \subseteq \mathcal{C}_1 \wedge \mathcal{C}_0 \subseteq \mathcal{C}_2 \implies \mathcal{C}_1 \cap \mathcal{C}_2 \equiv \mathcal{C}_0$$

Note que lo anterior implica que para cualquier propiedad $p \in \mathcal{C}_0$, entonces $p \in \mathcal{C}_1 \wedge p \in \mathcal{C}_2$, por lo que sabemos que un objeto de tipo $\mathcal{C}_1$ tiene *al menos* todas las propiedades de $\mathcal{C}_0$, entonces podemos utilizar un objeto de tipo `Vector2D` en el método `add(VectorND)`, ya que sabemos que `Vector2D` puede hacer todo lo que hace `VectorND`, esta propiedad se llama *polimorfismo* y fue presentada en la sección 4.1.

Ahora, esto no se da en la dirección contraria. Si tenemos una propiedad $q \in \mathcal{C}_1$, esta puede o no estar en $\mathcal{C}_2$. Es más,

$$q \in \mathcal{C}_1 \wedge q \in \mathcal{C}_2 \iff q \in \mathcal{C}_0.$$

De esto se desprende que si tenemos un objeto o de tipo $\mathcal{C}_0$, dado que anteriormente mostramos que $\mathcal{C}_1 \cap \mathcal{C}_2 \equiv \mathcal{C}_0$, tenemos $o \in \{\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2\}$. Como no podemos distinguir a cuál de esos conjuntos pertenece o, sólo podemos asumir que pertenece a la intersección de todos los conjuntos, i.e. $\mathcal{C}_0$. De esto podemos concluir directamente que `newPoint` puede ser de tipo `VectorND`, pero no de tipo `Vector2D`.

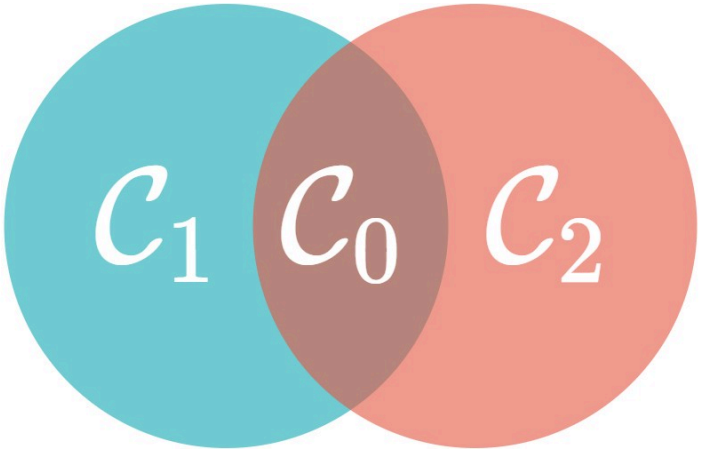

Figura 5.1: Representación de herencia de clases como conjuntos.

Nota. *Un detalle que vale la pena mencionar es que* ***todos los objetos****, tanto en Python como Java, extienden implícitamente a un objeto especial llamado* `object` *en Python y* `Object` *en Java. Así, la definición de la clase* `VectorND` *podría hacerse como* `class VectorND extends Object {...}` *y sería equivalente (en Python es análogo).*

Ejercicio 5.7. Cree una clase `Vector3D` que extienda de `VectorND` e implemente el método `void printSpherical()` que imprima en pantalla el vector en coordenadas esféricas de la forma (r, ϕ, θ) sabiendo que:

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \phi &= \arctan \frac{y}{x} \\ \theta &= \arccos \frac{z}{r} \end{aligned}$$

Para esto puede utilizar los métodos de la librería estándar de *Java* `Math.sqrt(double)`, `Math.atan(double)` y `Math.acos(double)`.

5.7. Ejercicios

Ejercicio 1 Publicaciones

Para este ejercicio se considerarán personas, autores y obras, y se modelará una estructura de clases para representar las relaciones entre ellas.

1. Considere la siguiente implementación de un autor:

```
class Author {
  String name;
  int money;

  Title[] publishedTitles = new Title[8];
  int publishedTitlesCount = 0;
  Title[] boughtTitles = new Title[8];
  int boughtTitlesCount = 0;

  Author(String name, int money) {
    this.name = name;
    this.money = money;
  }

  void write(String name, String content, Title title) {
    title.content += content;
  }

  void publish(Title title, int price) {
    title.price = price;
    if (publishedTitlesCount == publishedTitles.length) {
      Title[] auxArr = new Title[publishedTitles.length * 2];
      for (int idx = 0; idx < publishedTitles.length; idx++) {
        auxArr[idx] = publishedTitles[idx];
      }
      publishedTitles = auxArr;
    }
    publishedTitles[publishedTitlesCount++] = title;
  }

  void distribute(Title title, Store store) {
    store.addTitle(title);
  }
```

```
  void buy(Title title, Store store) {
    if (money >= title.price) {
      store.money += title.price;
      if (boughtTitlesCount == boughtTitles.length) {
        Title[] auxArr = new Title[boughtTitles.length * 2];
        for (int idx = 0; idx < boughtTitles.length; idx++) {
          auxArr[idx] = boughtTitles[idx];
        }
        boughtTitles = auxArr;
      }
      boughtTitles[boughtTitlesCount++] = title;
    }
  }
}
```

Juzgue si esta implementación cumple con los *principios SOLID* y, en caso de qui viole alguno, especifique cuál y proponga una solución (no es necesario que la programe).

2. Considere ahora que un autor es solamente una persona que ha publicado alguna obra. ¿Cambia esto de alguna forma la implementación anterior?

   Hint: ¿Qué tanto diferirían una clase ***`Author`*** *de una clase* ***`Person`****?*

3. Una persona puede escribir múltiples tipos de trabajos, en particular **novelas**, **cuentos**, ***papers*** y **libros científicos**.

   Tanto las novelas como los cuentos son **obras literarias**, los *papers* y libros científicos son **publicaciones científicas** y las novelas y libros científicos son **Libros**, y los 4 son **trabajos escritos**.

   Proponga un esquema de clases para representar esta estructura (basta con un diagrama simple).

4. Diremos que una **revista científica** es una recopilación de *papers* y una **antología** es un conjunto de cuentos. Una antología es una obra literaria y un libro, mientras que una revista científica es una publicación científica. Ambas son obras escritas.

   Modifique el diagrama de su respuesta anterior para incluir estos nuevos tipos.

5. Lo último será definir dos nuevos tipos de entidades, las **editoriales** y las **tiendas**.

   Una **editorial** se encarga de publicar las obras de un autor y distribuirlas a tiendas. Las editoriales **siempre** se especializan en un tipo de obra, siendo estas publicaciones científicas u obras literarias.

   Por otro lado, una **tienda** compra obras a las editoriales, pero solamente si son libros o revistas. Además, una tienda puede o no especializarse en un tipo particular de publicación (literaria o científica).

Bosqueje las clases necesarias para representar este comportamiento y los métodos necesarios para que una editorial publique una obra y una tienda compre de acuerdo a las restricciones que se impusieron en los párrafos anteriores (esto puede ser en *Java*, pseudo-código o como un diagrama pero deben especificarse claramente los tipos de los objetos involucrados en el proceso).

Ejercicio 2 Algebra vectorial

1. El largo (o norma) de un vector $\mathbf{v}$ es la distancia de dicho vector respecto al origen del sistema de coordenadas y se denota como $||\mathbf{v}||$. La norma de un vector se define como:

$$||\mathbf{v}|| = \sqrt{\mathbf{v}_0^2 + \mathbf{v}_1^2 + \cdots + \mathbf{v}_n^2}$$

   Implemente el método `double getLength()` que calcule el largo de un vector de dimensión n.

2. Un *vector cero* es un vector especial de largo arbitrario que cumple la propiedad de que sumarlo con cualquier otro vector da el mismo vector. Formalmente, sea un vector $\mathbf{v}$ de dimensión arbitraria y el vector $\mathbf{0}$ de dimensión indefinida: se dice que $\mathbf{0}$ es un vector cero ssi $\mathbf{v} + \mathbf{0} = \mathbf{v},\ \forall \mathbf{v}$.

   Programe una clase `ZeroVector` que implemente dicha funcionalidad al ser sumado con cualquier otro vector.

3. Agregue un método (en las clases que estime necesarias) `boolean isZeroVector()` que retorne `true` si el objeto es un vector cero y `false` en caso contrario. Note que un objeto de clase `VectorND` también podría ser un vector cero.

4. Dos vectores $\mathbf{a}$ y $\mathbf{b}$ se dicen opuestos si $\forall i \in \mathbb{Z}$ se cumple que $\mathbf{a}_i = -\mathbf{b}_i$. Extienda la clase `VectorND` con un método `boolean isOppositeTo(VectorND)` que retorne `true` si los vectores son paralelos y `false` en caso contrario.

5. El *producto punto* entre 2 vectores $\mathbf{a}$ y $\mathbf{b}$ de dimensiones m y n respectivamente, con $m \leq n$ se define como:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}_1\mathbf{b}_1 + \mathbf{a}_2\mathbf{b}_2 + \cdots + \mathbf{a}_m\mathbf{b}_m$$

   Implemente el método `double dotProduct(VectorND)` que calcule el producto punto entre 2 vectores.

6. El *producto cruz* es una operación entre dos vectores de 3 dimensiones que da como resultado un nuevo vector perpendicular a ambos. Se define el producto cruz entre dos vectores $\mathbf{a}$ y $\mathbf{b}$ como:

$$\mathbf{a} \times \mathbf{b} = (\mathbf{a}_2\mathbf{b}_3 - \mathbf{a}_3\mathbf{b}_2,\ \mathbf{a}_3\mathbf{b}_1 - \mathbf{a}_1\mathbf{b}_3,\ \mathbf{a}_1\mathbf{b}_2 - \mathbf{a}_2\mathbf{b}_1)$$

Cree un método `Vector3D crossProduct(Vector3D)` que haga este cálculo.

Ejercicio 3 Estructura *Half-edge*

Una *estructura Half-edge* establece una relación entre los componentes que describen mallas de polígonos (i.e. vértices, arcos y polígonos) ampliamente usados en la *computación gráfica*.

La ventaja de utilizar esta estructura sobre otras es que permite responder varias preguntas comunes respecto a los polígonos que se representan con esta estructura:

- ¿Qué arcos están conectados a un vértice?
- ¿Qué polígonos están conectados a un vértice?
- ¿Qué vértices están conectados a un polígono?

utilizando una cantidad constante de memoria.

Nota. *Una estructura Half-edge no puede utilizarse para representar superficies no orientables ni superficies con topologías non-manifold.*

Esta estructura obtiene su nombre ya que representa cada arco de una figura como 2 semi-arcos con orientación anti-horaria.

A continuación se plantean los requisitos para implementar esta estructura:

1. Un *Half-edge* es una línea que tiene un vértice de origen y uno de destino, puede estar conectado a 0 ó 1 polígonos y todo *Half-edge* está conectado a otro *Half-edge* asociado a los mismos vértices pero en sentido contrario, a este último se le llama pareja (de ahí el nombre ***Half**-edge*). Un arco se define como un par de *Half-edges* que sean parejas entre sí.

   Defina las clases y constructores necesarios para representar esta estructura considerando vértices de n dimensiones de modo que cada vértice y polígono referencie a lo más a uno de sus *Half-edges*.

   Para simplificar los problemas siguientes es recomendable que agregue un campo `String id` a cada clase para luego poder imprimir en pantalla el *id* de cada objeto.

2. Cree un método que indique si dos vértices son adyacentes (i.e. que están unidos por un arco).

3. Agregue un método que indique si dos polígonos son adyacentes (i.e. tienen al menos un arco en común).

4. Agregue un método que permita definir el *Half-edge* siguiente de otro. Un ejemplo de uso de este método podría ser:

   ```
   halfEdge1.setNext(halfEdge2);
   ```

5. Implemente un método que permita recorrer los arcos conectados a un vértice en sentido antihorario e imprima en pantalla cada uno de los arcos recorridos, de forma tal que un mismo *Half-edge* no sea recorrido más de una vez.

6. Escriba un método que permita recorrer los arcos alrededor de un polígono en sentido antihorario y los imprima en pantalla, nuevamente, asegúrese de que cada *Half-edge* no sea recorrido más de una vez.

7. Cree un método para añadir un arco entre dos vértices. Tenga en cuenta que agregar un nuevo arco implicará cambiar las referencias de algunos de los semi-arcos de los vértices para respetar el invariante de que el recorrido debe hacerse siempre en sentido horario. Para esto último puede tomar como referencia la figura 5.2, note que al agregar el nuevo arco, el semi-arco **a** pasa a apuntar al semi-arco **i** y el semi-arco **c** pasa a apuntar a **j** formándose un nuevo polígono.

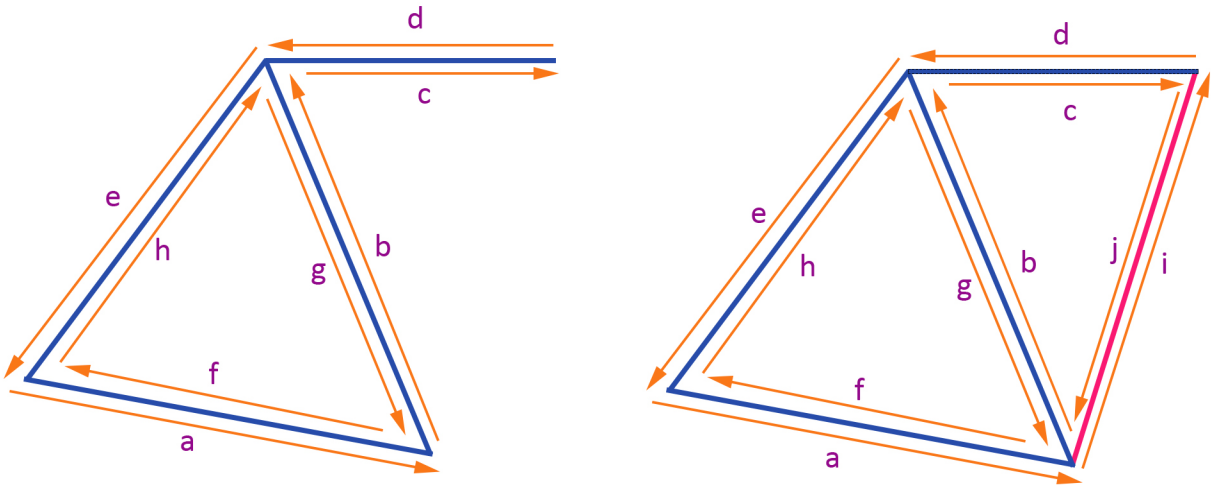

Figura 5.2: Agregar un arco a una estructura *Half-edge*.

8. Implemente un método para remover un arco entre dos vértices. De manera similar a la pregunta anterior, esto implicará cambiar las referencias de algunos de los semi-arcos.

9. Agregue un método para remover un vértice de la estructura. La eliminación de un vértice implica remover todos los semi-arcos conectados a éste.

   Hint: utilice la solución de la pregunta anterior para eliminar los arcos.

Capítulo 6

Profundizando en *Java*

Este capítulo servirá como continuación sobre programación orientada a objetos, pero utilizando las herramientas que provee *Java* y la manera en que este lenguaje se comporta y las herramientas básicas que serán utilizadas durante el curso.

Los contenidos de este capítulo son **específicos a *Java*** y no necesariamente son iguales o análogos a los de otros lenguajes.

Cabe destacar también que son contenidos muy básicos sobre el funcionamiento y las herramientas que proporciona éste. En el capítulo 7 se verán algunos conceptos más avanzados, pero aún así debe considerar que *Java* es mucho más complejo que lo que cubrirá este apunte.

Además, en este capítulo (ya que los ejemplos serán más complejos) se comenzará a utilizar *IntelliJ IDEA* como herramienta para programar, compilar y ejecutar el código que se escriba.

6.1. Convenciones

Antes de explicar cómo funciona *Java*, veamos las convenciones de cómo escribir código. Estas convenciones no es necesario seguirlas, pero es **altamente recomendado** ya que hace más fácil estandarizar la forma en que se escribe un programa cuando se trabaja en equipos.

Las convenciones que ilustraremos aquí son las que utilizan los programadores de *Google*, ya que es uno de los estándares más usados y completos.

Archivos

Nombre de archivos El nombre de los archivos debe ser el nombre de la clase principal del archivo (cada archivo debe tener **a lo más** una clase principal) con la terminación `.java`.

Por ejemplo, la clase `VectorND` que definimos anteriormente debe estar en un archivo llamado `VectorND.java`.

Encoding El *encoding* de los archivos debe ser *UTF-8*.

Espacios en blanco Aparte del salto de línea (que dependerá del S.O.), solamente debe usarse el espacio estándar *ASCII* (`0x20`). Esto implica que no deben utilizarse *tabs* para indentar el código.

Formato

Llaves Siempre se deben utilizar llaves en las instrucciones *if*, *else*, *do* y *while*, incluso si no son necesarios.

Esto quiere decir que el siguiente código:

```
if (a >= 0) return a;
else return -a;
```

debe ser cambiado por:

```
if (a >= 0) {
  return a;
} else {
  return -a;
}
```

Indentación Las instrucciones deben tener una indentación de 2 espacios.

Una instrucción por línea Cada instrucción debe estar seguida de un salto de línea.

Esto quiere decir que:

```
// Esto
int i = 0; double d = 1.0;
// Se debe cambiar por
int i = 0;
double d = 1.0;
```

Límite de columnas Cada línea no debe exceder los 100 caracteres.

Espacios horizontales

- Separar cada *keyword* de los paréntesis que lo sigan en esa línea, e.g. `if (...)` en vez de `if(...)`.
- Separar cada *keyword* de la llave que lo precede, e.g. `} else` en vez de `}else`.

- Colocar un espacio antes de cada apertura de llave (e.g. `if (...) {}` en lugar de `if(...){}`), con 2 excepciones:
  - `@SomeAnnotation({a, b})`.
  - Al definir arreglos como `new int[][] {{1, 2}}`.
- En ambos lados de un operador binario o ternario, e.g. `c = a + b` en vez de `c=a+b`.
- Luego de `,`, `:` y `;`, y luego del cierre de un paréntesis al hacer *casting*, e.g. `for (int i = 0; i < j; i++) {}`, `int a, b`, `a > b ? a : b`, `a = (int) 2.5`.

Nombres

Los nombres en todos los casos siguientes deben ser descriptivos.

```
// Evitar esto
int c = 0; // Product stock counter
// y reemplazarlo por
int productStock = 0;
```

Paquetes Los nombres de paquetes deben contener solo letras minúsculas, e.g. `package com.github.cc3002metodologias` en lugar de `package com.gitHub.CC3002Metodologias`.

Clases Los nombres de las clases utilizan *UpperCamelCase* y deben ser sustantivos.

Métodos Los nombres de métodos se escriben en *lowerCamelCase* y deben ser verbos o acciones.

Variables `static final` Los nombres de estas variables utilizan *CONSTANT_CASE*.

Otras variables Los nombres de variables, parámetros y propiedades utilizan *lowerCamelCase*.

Buenas prácticas

Anotar sobre-escritura Siempre agregar la anotación `@Override` a los métodos que hacen *overriding* a otro (este concepto se verá en la sección 6.4)

No ignorar excepciones Nunca atrapar una excepción para luego ignorarla (el tópico de excepciones se verá en detalle en la sección 7.1), por ejemplo:

```java
// Evitar esto
try {
  methodThatMayFail();
} catch (SomeException e) {}
// Preferir algo como
try {
  methodThatMayFail()
} catch (SomeException e) {
  System.out.println("We failed, but the show must go on.")
}
```

Llamadas calificadas a métodos estáticos Las llamadas calificadas son esas que se hacen a la clase en vez de a instancias de ésta (refiérase a la sección 6.4).

```java
// No hacer esto
Reader reader =  new BufferedReader(new InputStreamReader(System.in));
Reader nullReader = reader.nullReader();
// Reemplazar por esto
Reader nullReader = BufferedReader.nullReader();
```

Javadoc

Formato El formato básico de un comentario *Javadoc* es el siguiente:

```
Para documentaciones complejas.
/**
 * Updates the {@code weight} and {@code bias} parameters
 * according to the learning rate.
 *
 * @param tolerance
 *     the tolerated error to determine if the loss
 *     function converged.
 */
public void updateParameters(double tolerance) {...}

Para documentaciones simples.
/** Calculates the difference between the values of a list. */
private double difference(List<Double> list) {...}
```

Tags Siempre utilizar tags para documentar código que sea más complejo. Los tags deben ir en el orden `@param`, `@return`, `@throws`, `@deprecated`.

Resumen El primer párrafo de la documentación debe ser un resumen. En caso de métodos simples se pueden omitir los *tags* previamente mencionados si se quedan claros en el resumen.

Documentación obligatoria **Todos** los métodos y clases públicas deben estar documentadas, con excepción de los métodos marcados como `@Override`, estos deben ser documentados en caso de que presenten algún comportamiento especial que no sea descrito en el método al que sobre-escribe.

Es recomendable documentar también métodos que no sean públicos para facilitar su comprensión al programador y a quienes revisen el código.

6.2. Estructura de una aplicación

Como se explicó en el capítulo anterior en *Java* (casi) todo son objetos, por esto la aplicaciones se componen de clases que se referencian entre sí.

6.3. Visibilidad

6.4. *Method lookup*

Capítulo 7

Java: Tópicos avanzados

7.1. Excepciones

Parte III

Patrones y metodologías de diseño