Metodologías de diseño y programación

Ignacio Slater

Departamento de Ciencias de la Computación, Universidad de Chile

3 de marzo de 2021

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
```

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

CentOS

```
sudo yum install git
```

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 **??**

```
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] Por ejemplo *Manjaro*.

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
```

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 **??**).

Para utilizar *Git* primero se debe crear un repositorio, para esto se utiliza el comando `init`. Comencemos entonces por crear un *repo*, para esto abran una terminal y ejecuten:

```
mkdir my-repo && cd my-repo # Crea y se ubica en una carpeta
git init
```

Esto creará un repositorio dentro de la carpeta `my-repo`.

En cualquier momento podemos revisar el estado del repositorio con la instrucción `status`, luego:

```
git status
```

Debiera entregar como resultado:

```
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Veamos ahora como agregar archivos al sistema de versionado. Para esto creen un archivo `README.md` y déjenlo en blanco, con el archivo creado corran nuevamente `git status`.

Hemos creado un archivo, pero este no se agrega inmediatamente al sistema de versionado, esto es bueno ya que, como veremos más adelante, no queremos que todos nuestros archivos se agreguen a *Git*. Para agregar un archivo, esto se hace con el comando `add` y luego el nombre del archivo que se quiera agregar o «`.`» para agregar todos los archivos. Entonces, si queremos agregar el archivo que creamos lo haríamos como:

```
git add README.md
# O equivalentemente
git add .
```

Sin embargo, lo que acabamos de hacer no es suficiente para que los archivos aparezcan en el historial del *VCS*. Lo que hicimos se conoce como *staging*, y es una etapa previa a agregar los archivos a *Git*. Prueben hacer `git status` de nuevo y vean el resultado.

A cada «versión» almacenada en el historial de *Git* se le llama *commit*. Para incluir los archivos de *staging* al sistema de versionado se utiliza el comando `commit`, este comando debe ir acompañado además de un comentario para que el usuario pueda identificar el *commit*. Vamos al ejemplo:

```
git commit -m "My first commit :D"
```

Prueben hacer `git status` de nuevo.

Ahora, hay una razón por la cuál hacer *stage* y *commit* son dos procesos separados: **no es posible deshacer un *commit***. Esto tiene como consecuencia que cualquier archivo o cambio que se haga al repositorio permanecerá en el historial y no podrá borrarse de ahí, incluso si el archivo se borra. Esto es de especial importancia cuando se guarda información de carácter privado en el directorio del *repo*.

Primero, creemos un archivo `secret.txt` y digamos que no queremos que ese archivo se agregue a *Git*. Veamos dos maneras de evitar que este archivo se agregue al historial.

La primera es hacer *unstage* del cambio, esto se puede hacer cuando se hace `add` a algún archivo incorrecto, para nuestro ejemplo sería de la siguiente forma:

```
git add .
```

Esto va a hacer *stage* de todos los archivos. Si hacen `git status` verán que el archivo está esperando a que se le haga *commit*. Ahora, para hacer *unstage* haríamos:

```
git rm --cached secret.txt
```

La otra opción es indicarle a *Git* que ignore ciertos archivos al momento de hacer *stage* (si el archivo ya había sido agregado con `add` entonces tienen que hacer lo indicado antes). Para definir los archivos que serán ignorados *Git* utiliza un archivo especial llamado `.gitignore`, este archivo contiene los nombres de los archivos (o patrones de nombres) que se quiere ignorar. Para ilustrar esto, primero creen el archivo `.gitignore` en el repositorio con el siguiente contenido:

```
# .gitignore
secret.txt
```

Esto le dice a *Git* que no queremos que se mantengan versiones del archivo `secret.txt`. Ahora, esto no es práctico cuando tenemos muchos archivos, para eso podemos usar patrones de nombres,

supongamos que quisiéramos ignorar todos los archivos que contengan la palabra *secret*, esto lo podemos hacer como:

```
# .gitignore
*secret*
```

Para detalles sobre como utilizar *gitignore* refiéranse a la documentación oficial de *Git*.[3] Adicionalmente, existen herramientas para generar archivos `.gitignore` a partir de *templates*, en particular una de las más utilizadas y completas es https://www.gitignore.io.

2.4. Repositorios remotos

Como mencionamos antes, uno de los principales beneficios es poder utilizar *Git* de manera remota. Para esto se necesita utilizar un servidor, y si bien es posible tener el sistema de versionado montado en un servidor propio existen proveedores que ofrecen este servicio y facilitan el trabajo.

Existen varias opciones[4] cuando se quiere utilizar un servidor, pero la más utilizada es por lejos *GitHub*.[5]

GitHub provee muchas funcionalidades pero para este curso utilizaremos solamente las más básicas.

Para detalles sobre cómo utilizar *GitHub* revisen las bibliografías incluidas al final de este capítulo.

Sin importar el proveedor de *Git* que se use los comandos para manejar los repositorios remotos serán siempre los mismos.

Primero veamos cómo «descargar» un repositorio desde el servidor. A la acción de crear una copia del repositorio remoto para utilizarlo como un repositorio local se le conoce como *clonar*. Para esto se utiliza el comando `clone` como se muestra a continuación:

```
git clone https://github.com/octocat/Spoon-Knife.git
```

Ejecutar la instrucción anterior creará una carpeta `Spoon-Knife` que contendrá todos los contenidos del repositorio (incluyendo todas las versiones del repositorio remoto, esto quiere decir que no necesitan hacer `git init`).

Una vez que clonaron el repositorio es posible que se hagan cambios en el repositorio remoto y necesiten actualizar la versión local, para hacer esto se utiliza el comando `pull`. Por ejemplo, siguiendo el ejemplo anterior haríamos:

[3] **gitignore**.
[4] Además de *GitHub* están *GitLab*, *BitBucket*, *Azure* y otros.
[5] **vcs-providers**.

```
cd Spoon-Knife
git pull
```

El último comando que será importante al momento de trabajar con repositorios remotos será `push`, esta instrucción subirá los *commits* realizados en su repositorio local al repositorio remoto. Es importante **siempre hacer `pull`** antes de hacer `push` para evitar conflictos de versiones. La sintaxis es la misma que la de `pull`.

2.5. Ramas

Las ramas (o *branches*) son uno de los aspectos más importantes al trabajar con *Git*, estas cumplen varios objetivos, pero el caso de uso más común es cuando se trabaja en equipo. Cuando utilizamos un repositorio remoto, varias personas pueden clonar y hacer *push* al mismo *repo*, esto puede generar problemas cuando las versiones no son compatibles.

Retrocedamos un poco para explicar el funcionamiento de *Git*. La manera en la que *Git* maneja el historial de versiones es manteniendo un grafo con los *commits* realizados, este grafo tiene una rama principal que se crea al iniciar un nuevo repositorio y por defecto se llama *master*. Lo recomendado es que en la rama principal solamente se encuentren las versiones del programa que están listas para ser utilizadas y se usen otras ramas para mantener las versiones durante el proceso de desarrollo.

Para crear una nueva rama en un repositorio se usa el comando `branch` seguido del nombre que se le desea dar a la rama. Luego, si queremos cambiar de rama utilizaremos el comando `checkout`. Por ejemplo:

```
git branch ramitas-saladas
git checkout ramitas-saladas
```

Al momento de cambiarse de rama, todos los *commits* que se realicen se harán en esa rama. Lo siguiente que querríamos hacer es pasar nuestro trabajo a otra rama (como *master*), para esto utilizaremos la instrucción `merge` seguido del nombre de la rama que queremos mezclar con la nuestra. Al igual que como siempre deben hacer *pull* antes de *push*, es importante que **siempre realicen *merge* en ambas direcciones**, primero pasamos los cambios de la rama de destino a la nuestra, y luego los de la nuestra a la rama de destino, por ejemplo, si quisiéramos actualizar *master* con los cambios que hicimos en nuestra rama haríamos:

```
# Desde ramitas-saladas
git merge master
git checkout master
git merge ramitas-saladas
```

2.6. *Markdown*

Markdown es un estándar liviano que apunta a ser fácil de usar para crear documentos con formato básico. Es similar a *HTML* pero más simple y conciso. En su mayoría, los archivos *Markdown* son documentos de texto plano con algunos caracteres especiales para definir el estilo del texto.

En *GitHub* (y varios otros servidores de *Git*) *Markdown* es el formato estándar de documentación.

Para crear un archivo *Markdown* basta con que el nombre del archivo termine en `.markdown` o `.md`.

Para una referencia de cómo usar *Markdown* refiérase a la documentación de *GitHub*.[6]

2.7. Ejercicios

[6] **gh-markdown**.

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 **??**[1]

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*,

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

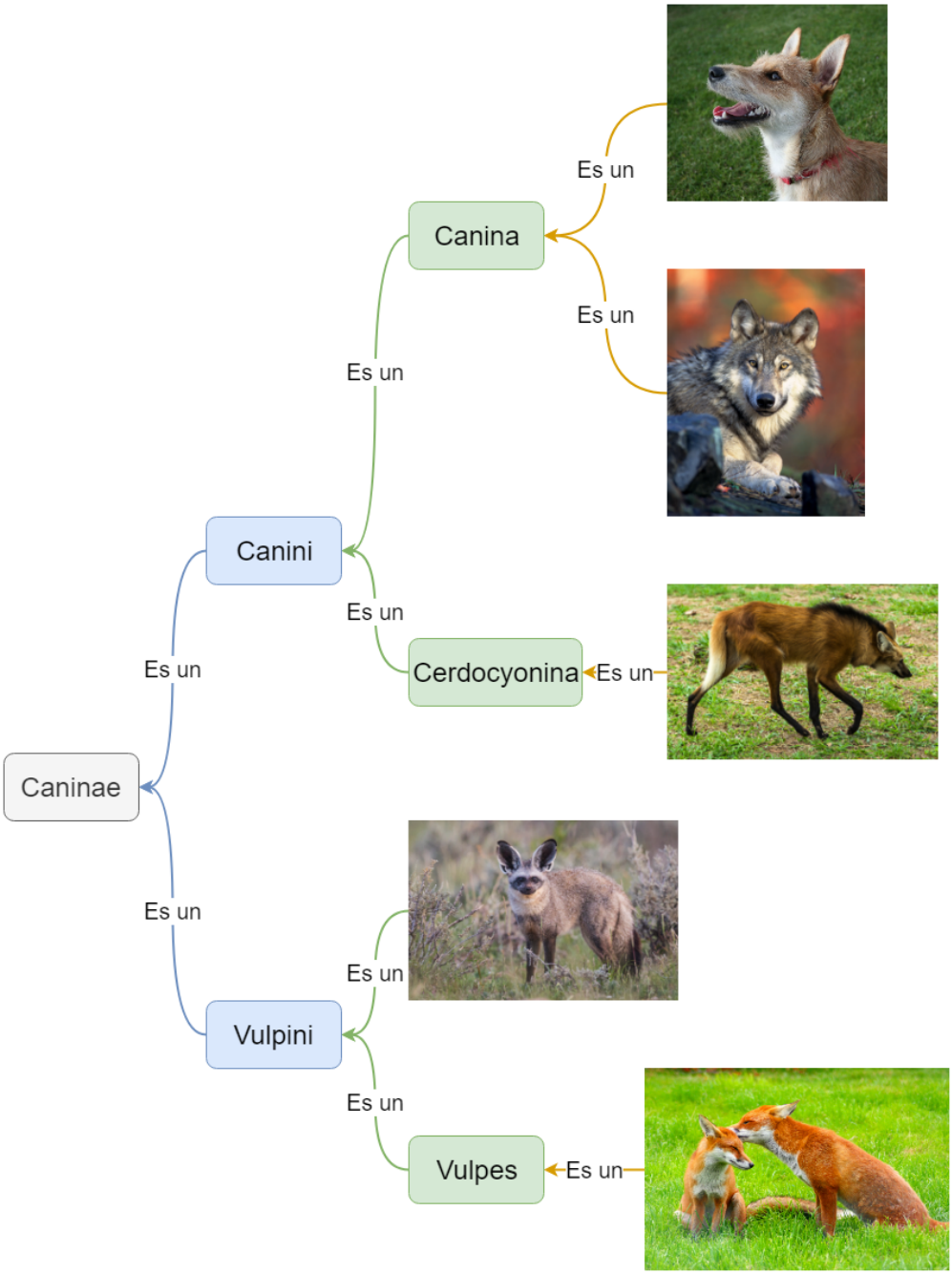

Figura 4.1: Ejemplo de herencia.

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 mé-

todo `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 términos más simples, esto quiere decir que if S es un hijo de T, entonces los objetos de tipo T pueden ser reemplazados por uno de tipo S sin alterar las funcionalidades de un programa.

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).

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.

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

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

Hay una serpiente en mi café

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:

```
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 **??**.

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 **??** 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:

```python
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:

```java
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*:

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

Lo que en *Java* se podría hacer como:

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

El siguiente código es equivalente al anterior:

```java
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:

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

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

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

En *Java* podemos hacer lo mismo como:

```java
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:

```
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]");
}
```

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

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:

```java
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:

```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.

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 **??** 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, par`

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`
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++) {
```

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

```
    components[i] = bigger[i] + smaller[i];
  }
  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 **??**, 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 **??**, 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 **??** 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,` 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

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

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 **??**.

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`.

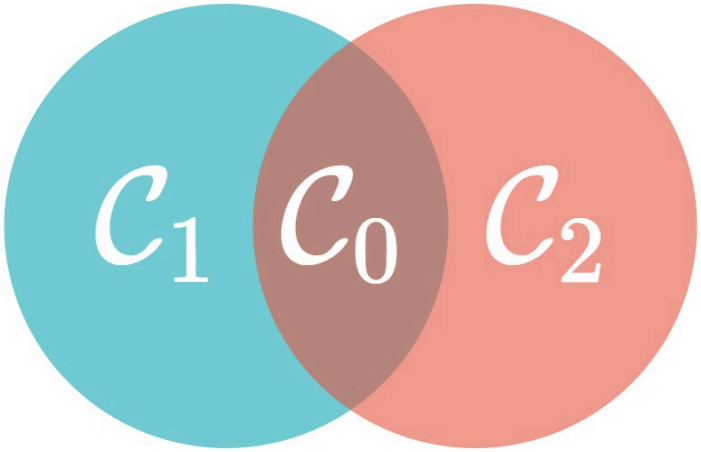

Figura 5.1: Representación de herencia de clases como conjuntos.

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 printSpheri` que imprima en pantalla el vector en coordenadas esféricas de la forma (r, ϕ, θ) sabiendo que:

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

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 que 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 opuestos 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 **??**, 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.

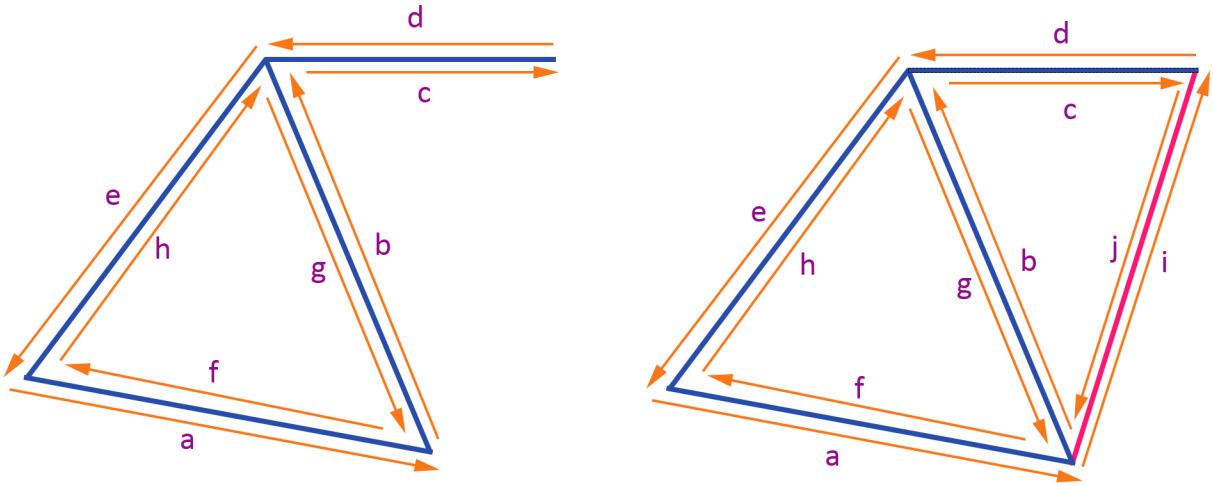

Figura 5.2: Agregar un arco a una estructura *Half-edge*.

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

¿*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 **??** 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.cc3002` 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 **??**)

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

```
// 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 **??**).

```
// 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.
```

```java
/** 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. Por este motivo, una aplicación en *Java* es finalmente una clase con un método `main` que indica que es ejecutable (este método `main` puede estar implícito en algunos contextos, un ejemplo de este caso se verá cuando se revise *testing* en el capítulo **??**). La firma[1] de este método **siempre** debe ser `public static void main(String[] args)` (o lo que es equivalente `public static void main(String... args)`), esto quiere decir que si tenemos una clase con el método:

```java
public void main(String[] args) {
  System.out.println("I am gravely mistaken");
}
```

esta clase no será ejecutable.

6.2.1. Creación de un proyecto

Entenderemos por **proyecto** a cualquier conjunto de archivos y clases que componen una aplicación.

Si es la primera vez que se abre *IntelliJ*, entonces se mostrará el *landing view* del *IDE*, ahí pueden crear un nuevo proyecto directamente haciendo clic en la opción `Create New Project` (vean la figura **??**).

[1] Refiérase a la sección **??**

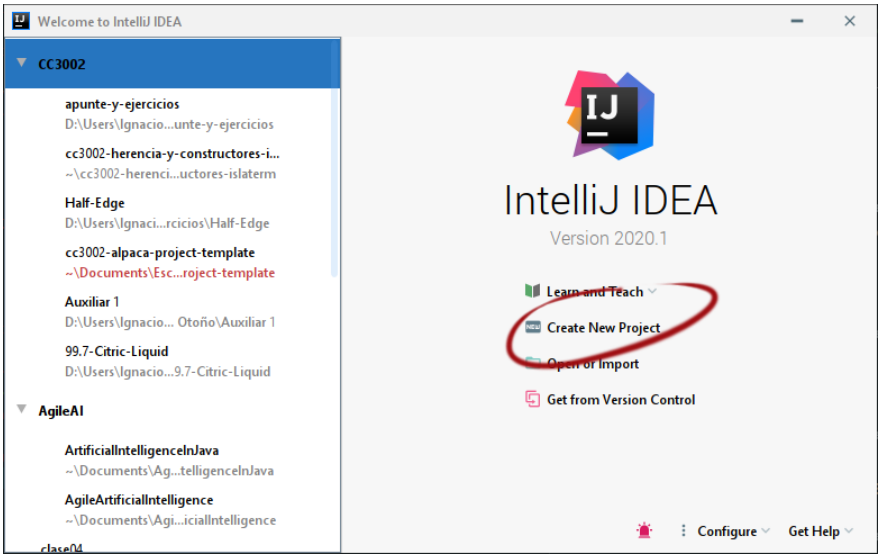

Figura 6.1: *Landing view* de *IntelliJ*

Si ya abrieron algún proyecto con *IntelliJ*, entonces lo más probable es que el *IDE* vuelva a abrir el último proyecto en el que trabajaron. En ese caso, para crear un nuevo proyecto deben ir a `File | New... | Project` (como en la figura **??**).

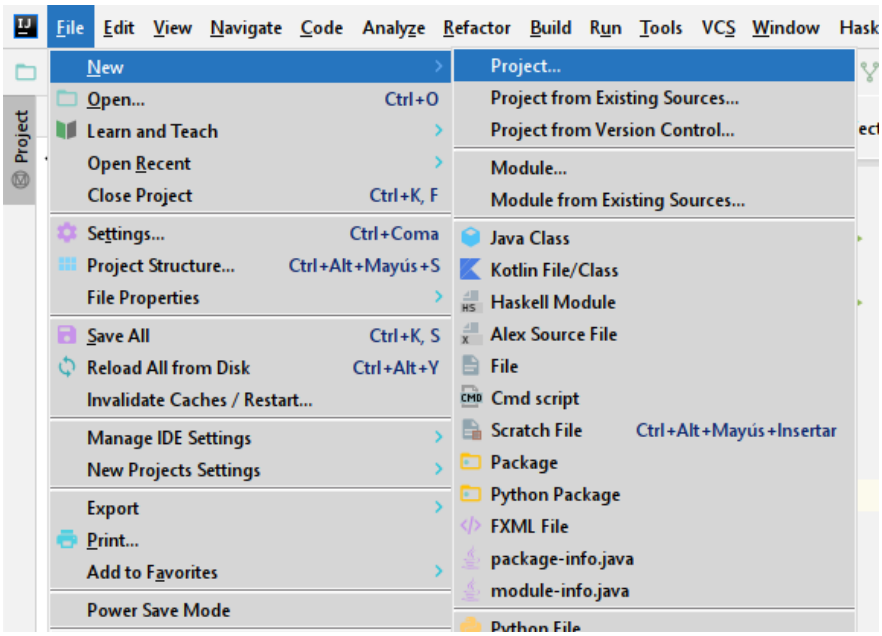

Figura 6.2: Crear un nuevo proyecto desde el menú principal de *IntelliJ*

Una vez que hayan hecho lo anterior, entrarán al menú de creación de proyectos. Aquí se les presentaran muchas opciones (figura **??**), aquí deben seleccionar el *SDK* y las herramientas a utilizar en el proyecto, en el ejemplo el *SDK* es el 14 y no utilizaremos ninguna herramienta adicional.

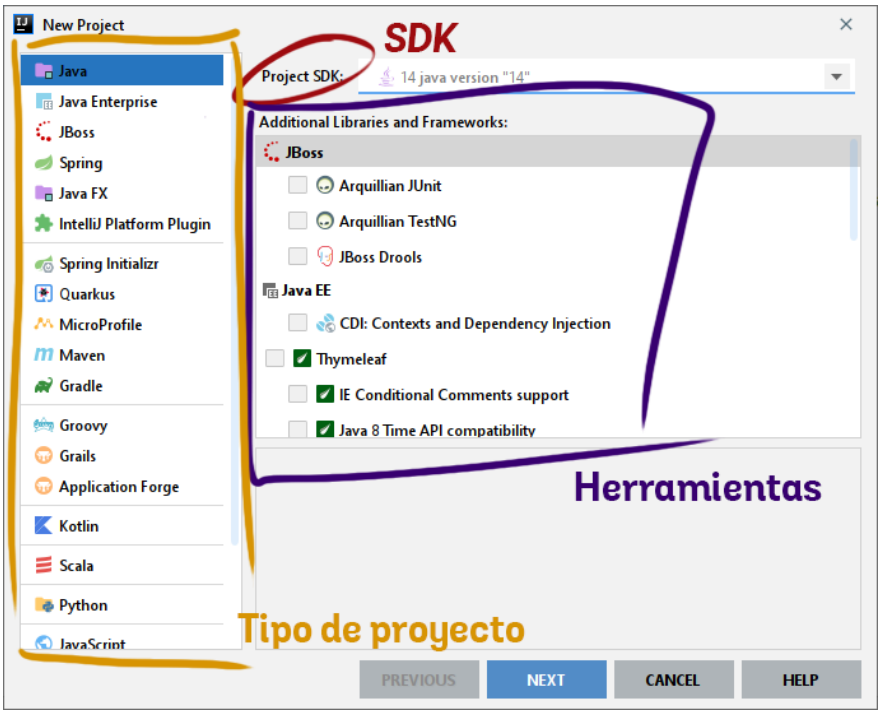

Figura 6.3: Menú de creación de proyecto: tipo de proyecto

Con estas opciones seleccionadas, den clic a `Next` para continuar al paso siguiente de la creación del proyecto.

La pestaña siguiente les dejará seleccionar un *template* para comenzar un proyecto con código predefinido. En este caso no utilizaremos ninguno, así que pueden saltarse esa vista.

La última vista de la creación de proyecto es la que contiene los datos de nuestro proyecto, aquí se puede elegir el nombre del proyecto y la ubicación donde se va a guardar, para este capítulo crearemos un proyecto *Bakemon* como se muestra en la figura **??**.

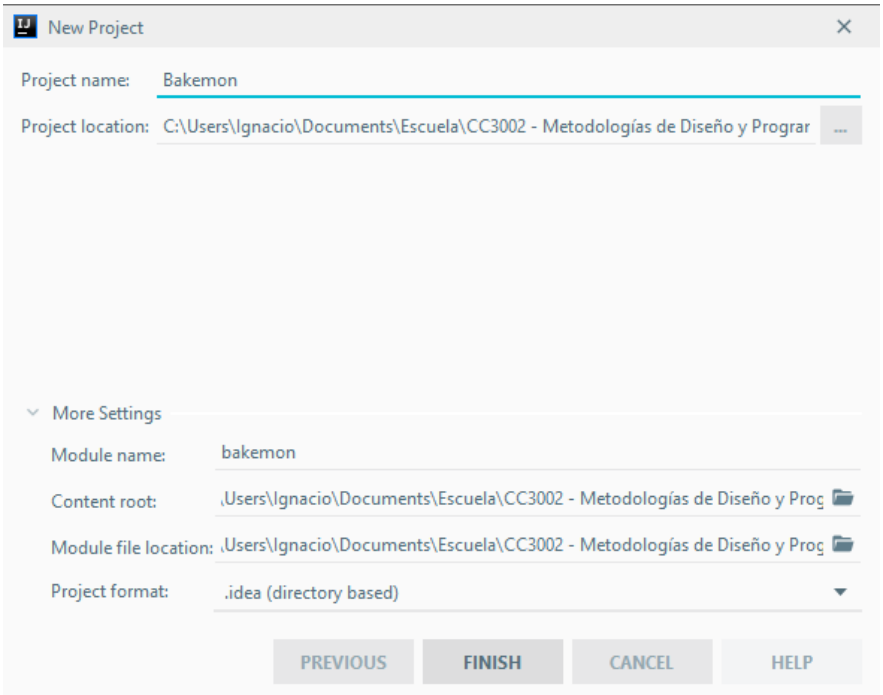

Figura 6.4: Menú de creación de proyecto: datos del proyecto

6.2.2. Configuración de un proyecto

Por defecto, al crear un proyecto se crean varias carpetas y archivos en el directorio que hayamos especificado, pueden ver esto en la pestaña `Project` ubicada a la izquierda del *IDE* (figura **??**), la única carpeta que nos interesará es `src` que es donde guardaremos todo nuestro código.

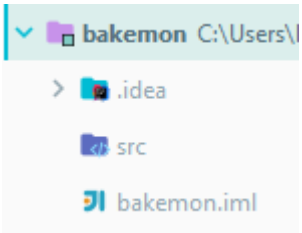

Figura 6.5: Organización inicial de un proyecto

Con el proyecto ya creado puede que sea necesario revisar las configuraciones (en la mayoría de los casos la configuración por defecto de *IntelliJ* debiera ser suficiente, pero es bueno conocer este menú). Para ir a las configuraciones del proyecto accedan mediante el menú principal en `File | Project Structure`.

Este menú tiene varias pestañas, pero nos enfocaremos en 2.

Comencemos por la pestaña `Project Settings | Project`, aquí podrán encontrar 4 opciones a configurar (vean la figura **??**). El primer campo es el nombre del proyecto (por si quisiéramos

cambiarle de nombre en algún momento). El segundo es el *SDK* que indica la versión del *JDK* que vamos a utilizar. La opción siguiente es para decirle al *IDE* las funcionalidades de qué *SDK* utilizar, en general esto va a ser igual que la opción anterior, pero sirve para escribir código compatible con *SDKs* más antiguos (e.g. trabajamos en un proyecto con otra persona que tiene instalado *Java 9* pero nosotros tenemos *Java 11*, en este caso conviene fijar este campo como *Java 9*). Por último tenemos `Project compiler output`, aquí es donde se almacenaran los archivos generados por el compilador, este directorio **debe** existir (la convención es tener una carpeta `out` en la raíz del proyecto, en nuestro caso sería `.../Pokemon/out`).

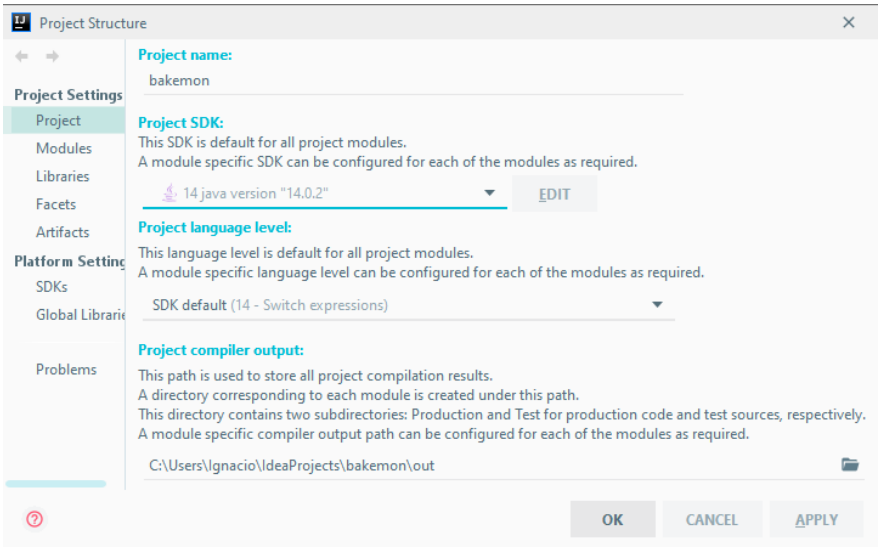

Figura 6.6: Configuraciones del proyecto

La siguiente pestaña es `Project Settings | Modules`, aquí podemos ver la estructura de los directorios del proyecto. Es importante que la carpeta `src` esté marcada como `Sources` para que el *IDE* sepa que ahí es donde se ubica el código de la aplicación (tomen como ejemplo la figura **??**).

El resto de las pestañas también son importantes,[2] y veremos algunas de estas más adelante en el apunte.

6.2.3. Paquetes

Los **paquetes** son una forma de organizar las clases de una aplicación en paquetes[3], de esta forma podemos agrupar clases con características comunes y ayudar a ubicarse dentro de la aplicación. Es una buena práctica siempre utilizar paquetes, incluso cuando nuestra aplicación conste de unos pocos archivos.

Se recomienda utilizar nombres únicos para los paquetes para que (al escribir una librería o publicar código en internet) no haya conflictos de nombres. Generalmente esto se logra utilizando un

[2] ¡Todas las pestañas son válidas!
[3] https://youtu.be/DyDfgMOUjCI?t=74

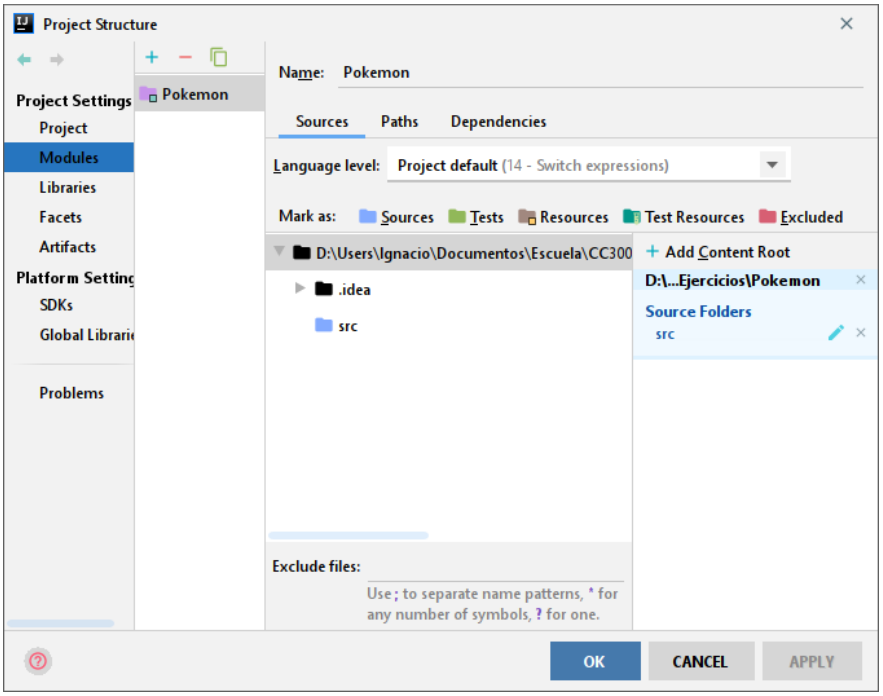

Figura 6.7: Estructura de directorios del proyecto

dominio web.

Para crear un paquete en *IntelliJ* hagan clic derecho en la carpeta `src` y seleccionen `New | Package`, llamaremos al paquete `dcc.<su username>.bakemon`. Al crear un paquete, *IntelliJ* automáticamente creará carpetas con los nombres de los paquetes y agrupará los que tengan prefijos en común.

Con esto cubrimos los conceptos básicos para estructurar una aplicación en *Java*, en las secciones siguientes se profundizará un poco más en el lenguaje y cómo estructurar una aplicación desde un punto de vista de programación orientada a objetos.

6.3. Visibilidad

Como se mencionó en los capítulos anteriores, una de las propiedades importantes de los objetos es la encapsulación, o sea, que los elementos internos de un objeto no puedan ser manipulados desde afuera. Para lograr esto existe el concepto de *visibilidad* (a veces llamada *privacidad* o *acceso*) que define qué elementos de un objeto van a poder verse desde afuera de éste.

Noten que hacer que un elemento sea visible desde fuera de un objeto no rompe la encapsulación, ya que no se están manejando directamente los elementos internos del objeto, sino que es el mismo objeto el que hace visibles ciertas componentes y funcionalidades suyas, y responde a un *mensaje*[4] enviado por el otro objeto con algún resultado.

[4] El concepto de mensaje se verá en más detalle en la sección siguiente

En *Java* existen cuatro modificadores de visibilidad: `public`, `protected`, `private` y *package-private*. Los modificadores de acceso pueden utilizarse para definir clases, métodos y variables, a excepción de las variables locales de un método que siempre serán internas al método e inaccesibles desde fuera de este.

El modificador `public` indica que el elemento definido va a ser visible desde cualquier otra clase. `protected` se usa para definir un elemento que solamente sea accesible desde las clases que heredan de la clase en la que se definió el elemento. La keyword `private` por otro lado define que el elemento será visible solamente desde dentro de la clase en la que se definió.

El modificador *package-private*, que es con el que se definen todos los elementos si no se define explícitamente un modificador de acceso, define al elemento como solo visible dentro del paquete en el que se encuentra la clase. En principio esto debiera otorgar más privacidad que el modificador `protected`, pero en la práctica no es así. Los paquetes definen estructura, no privacidad, esto significa que no aseguran realmente quiénes podrán acceder a los elementos *package-private* y su forma de uso tampoco es clara. Es importante resaltar esto ya que una confusión muy común es creer que los elementos *package-private* de *Java* son equivalentes a los elementos `internal` de *C#* o *Kotlin* cuando en realidad son **totalmente distintos**. En general, la recomendación será siempre evitar utilizar el modificador *package-private* lo más posible.[5] [6]

Un detalle muy importante sobre los modificadores de acceso es el *scope* de estos, es decir, desde dónde puede accederse «realmente» a las componentes internas de un objeto.

TODO:

1. scope de los modificadores
2. getters/setters

6.4. *Method lookup*

Cuando un objeto recibe un mensaje, éste debe ser interpretado de alguna forma, pero para esto se necesita alguna forma de saber cómo responder al mensaje. El proceso de buscar la respuesta adecuada para un mensaje se conoce como *method lookup*.

En *Java* (y el resto de los lenguajes de tipado estático) todos los mensajes **deben** tener una forma de ser interpretados para que el programa compile, pero en el caso de los lenguajes de tipado dinámico como *Python* esto no necesariamente es así.

Al recibir un mensaje, el objeto que lo recibe revisa si tiene algún método para responderlo; si encuentra el método entonces lo ejecuta, sino lo busca en su clase padre. En caso de que no encuentre

[5] **ham-package-private**.

[6] En las versiones más recientes de *Java* se introdujo el concepto de *módulo* que soluciona varios de estos problemas, pero no los utilizaremos en el curso.

el método entonces no puede responder el mensaje y por lo tanto arroja un error, ya sea al compilarse o en tiempo de ejecución dependiendo del lenguaje.

Tomemos como ejemplo las clases `Vector2D` y `VectorND` que vimos en las secciones anteriores. `Vector2D` solamente cuenta con un constructor y no tiene ningún método propio, por lo que cualquier mensaje que reciba no podrá ser interpretado directamente. Ahora, si enviamos el mensaje `add(VectorND)` a `Vector2D` lo va a buscar en dicha clase y no lo va a encontrar, como no lo encuentra se hace delegación[7] para que la clase padre interprete el mensaje. En este caso, el método se encuentra en la clase `VectorND` así que se responde exitosamente el mensaje. Por otro lado, si enviáramos el mensaje `substract(VectorND)`, buscaría en `Vector2D` sin encontrarlo, luego buscaría en `VectorND` y, por último, en la clase `Object`; al no encontrar el método no se puede responder al mensaje y ocurre un error de compilación.

Ejercicio 6.1. ¿Qué ocurre si enviamos el mensaje `equals(VectorND)` a un objeto de la clase `Vector2D`?

6.4.1. `this` y `super`

Todos los objetos de *Java* poseen dos *pseudo-variables* para referenciar al mismo objeto.

6.5. Constructores

En esta sección revisaremos en más detalle los constructores de *Java*.

6.5.1. Constructores vacíos y por defecto

Anteriormente, cuando hablamos de constructores (refiérase a la sección **??**) dijimos que eran métodos especiales que se llamaban al crear una instancia de una clase, esto es parcialmente cierto. En *Java* todos los métodos retornan algo, incluso los marcados como `void` retornan implícitamente un valor que no se puede utilizar y por lo tanto es desechado (el detalle del manejo de memoria en *Java* se verá en la sección **??**), para ejemplificar esto, noten que los siguientes métodos son equivalentes:

```
public void method1() {}
public void method2() {
  return;
}
```

¿Qué es entonces un constructor? Un constructor es un **bloque de código** que se ejecuta al momento de crear una nueva instancia de una clase. ¿Por qué es importante hacer notar esta diferencia? Volveremos a esto un poco más adelante.

[7] **wp-delegation**.

Primero, veamos cómo se define un constructor. La firma de un constructor es de la forma `[modifier] ClassName([parameters]) {[body]}` Vayamos a un ejemplo, volvamos a la clase `Bakemon` que habíamos creado anteriormente y agreguémosle un constructor:

```
public class Bakemon {
  public Bakemon() {
  }
}
```

Nuestro constructor está vacío, eso significa que lo único que hace el constructor es reservar la memoria necesaria para las instancias de ésta clase. Pero ya habíamos visto antes que podíamos crear instancias de la clase sin necesidad de definir un constructor, esto se debe a que *Java* crea un constructor por defecto si no existe ninguno en la clase. Lo anterior se traduce en que el código que hicimos sea equivalente a:

```
public class Bakemon {
}
```

Agreguemos algo de funcionalidad al constructor. Consideren que queremos agregarle un tipo a nuestros *Bakemon*, y digamos por ahora que comenzarán sin tipo (y representemos esto como el *String* `"NONE"`).

```
public class Bakemon {
  private String type;

  public Bakemon() {
    type = "NONE";
  }
}
```

Como estamos definiendo siempre la propiedad `type` con el mismo valor *Java* provee una forma más compacta de escribir esto mismo como:

```
public class Bakemon {
  private String type = "NONE";
}
```

6.5.2. Creando instancias de clases

Ahora, veamos un poco más en detalle qué es lo que está ocurriendo cuando se crea un objeto. Los objetos se crean con la keyword `new`, eso ya lo habíamos visto. ¿Pero qué es exactamente lo que hace la instrucción `new`?

Cuando creamos un objeto, lo que estamos haciendo es enviar el mensaje `new` **a la clase** de la que queremos crear la instancia. Al recibir el mensaje, la clase invoca a su constructor y retorna una instancia de ella (un objeto). Podemos hacer un símil de esto con los objetos (estructuras) de *C*. Consideren el siguiente código:

```
deftype struct {
  char* type;
} Bakemon;

Bakemon createBakemon() {
  Bakemon* bakemon = (Bakemon*) malloc(sizeof(Bakemon));
  bakemon->type = "NONE";
  return *bakemon;
}
```

Los constructores, al igual que los métodos, pueden recibir parámetros, entonces, si quisieramos que nuestro *Bakemon* tenga un nombre y una cantidad de *hit points*. Esto lo podemos lograr con:

```
public class Bakemon {
  private String name;
  private int hitPoints;
  private String type = "NONE";

  public Bakemon(String name, int hitPoints) {
    this.name = name;
    this.hitPoints = hitPoints;
  }
}
```

Ahora, con este nuevo constructor podemos crear instancias de la clase como:

```
Bakemon glamorant = new Bakemon("Glamorant", 10);
Bakemon italicore = new Bamekon("Italicore", 15);
```

¿Qué pasa ahora si intentamos hacer `new Bakemon()`? Nuestro código no compilaría ya que no existe un constructor sin parámetros ya que, como declaramos un constructor explícitamente, *Ja-*

va no crea un constructor por defecto como habíamos visto en la sección anterior. Si quisiéramos lograr el mismo comportamiento anterior tendríamos que declarar un constructor que no reciba parámetros.

En *Java* (a diferencia de *Python* por ejemplo) se pueden declarar múltiples constructores para una clase. Luego, podemos añadir dos constructores adicionales a nuestra clase de la forma

```
public class Bakemon {
  private String name;
  private int hitPoints;
  private String type = "NONE";

  public Bakemon() {
  }

  public Bakemon(String name, int hitPoints) {
    this.name = name;
    this.hitPoints = hitPoints;
  }

  public Bakemon(String name, int hitPoints, String type) {
    this.name = name;
    this.hitPoints = hitPoints;
    this.type = type;
  }
}
```

Noten que tanto en `Bakemon(String, int)` y `Bakemon(String, int, String)` asignamos las variables de instancia `name` y `hitPoints`, esto puede llevar a problemas y, como veremos en más detalles en la siguiente parte del apunte, en general querremos evitar la duplicación de código. Para evitar esto podemos reutilizar constructores de la misma clase con el mensaje `this(...)` que llamará al constructor que tenga los mismos tipos de parámetros que el mensaje que se envía. Es importante notar que `this` y `this(...)` son cosas distintas, la primera es una pseudo-variable (véase sección **??**) mientras que la segunda es un mensaje. Veamos como mejorar los constructores.

```
public class Bakemon {
  private String name;
  private int hitPoints;
  private String type = "NONE";

  public Bakemon() {
  }
```

```
  public Bakemon(String name, int hitPoints) {
    this.name = name;
    this.hitPoints = hitPoints;
  }

  public Bakemon(String name, int hitPoints, String type) {
    this(name, hitPoints);
    this.type = type;
  }
}
```

Ejercicio 6.2. Suponga que agregamos a la clase el método

```
public void setHitPoints(int hitPoints) {
  this.hitPoints = hitPoints;
}
```

¿Puedo llamar al método `setHitPoints(int)` desde el constructor? Por ejemplo:

```
public Bakemon(String name, int hitPoints) {
  this.name = name;
  setHitPoints(hitPoints);
}
```

Ejercicio 6.3. ¿Puedo enviar el mensaje `this(...)` desde un método? ¿Por qué razón?

Ejercicio 6.4. Suponga que modificamos el constructor de la clase por

```
public Bakemon(String name, int hitPoints, String type) {
  this.type = type;
  this(name, hitPoints);
}
```

¿Cambia esto el comportamiento de la clase? ¿Qué sentido tiene ese comportamiento?

Ejercicio 6.5. ¿Pueden haber constructores privados en una clase? ¿Qué utilidad tendría esto?

Ejercicio 6.6. ¿Puede una clase tener sólo constructores privados? ¿En qué contexto podría resultar útil esto?

6.5.3. Constructores y herencia

Ya hemos mencionado anteriormente cómo la herencia es una de las propiedades más poderosas de la programación orientada a objetos, entonces una pregunta natural sería cómo funcionan los constructores cuando hay herencia.

Tomemos como punto de partida el ejemplo de vectores que vimos en el capítulo **??**.

Primero modifiquemos un poco la clase `VectorND` para integrar los modificadores de privacidad que vimos en la sección anterior:

```
public class VectorND {
  protected double[] tail;

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

  public VectorND add(VectorND otherVector) {
    // Igual que antes
  }

  public void print() {
    // Igual que antes
  }
}
```

Ahora, también habíamos definido una clase `Vector2D` que heredaba de `VectorND`, modifiquemos también esta clase para restringir su visibilidad.

```
public class Vector2D extends VectorND {
  public Vector2D(double x, double y) {
    super(new double[]{x, y});
  }
}
```

Noten que el constructor de `Vector2D` hace una llamada a `super(...)`, es importante que no confundan este mensaje con la pseudo-variable `super`. Lo que hace este mensaje es enviar un mensaje a la superclase para que ésta ejecute el constructor que reciba los parámetros que se le pasaron (en este caso `VectorND(double[])`).

Una pregunta que podría surgirnos ahora es la necesidad de llamar explícitamente al constructor de la clase padre. ¿No podríamos simplemente hacer lo siguiente?

```
public Vector2D(double x, double y) {
  this.tail = new double[]{x, y};
}
```

Si intentan correr el código anterior se van a dar cuenta que no compila, la razón de esto es similar a lo que sucedía si no declarábamos explícitamente un constructor en la clase, por defecto todos los constructores hacen una llamada implícita al constructor de su clase padre sin argumentos a no ser que se llame explícitamente a `super(...)`. Esto quiere decir que el código del ejemplo anterior es equivalente a:

```
public Vector2D(double x, double y) {
  super();
  this.tail = new double[]{x, y};
}
```

Luego, como la clase `VectorND` no tiene ningún constructor que no reciba argumentos entonces el código no compila.

Otra pregunta interesante es si los constructores se heredan. Para explicar esto partamos por un ejemplo. La velocidad se puede definir como un vector dentro de un espacio euclidiano de n dimensiones, luego, la velocidad es equivalente a la clase `VectorND` que ya definimos. Podríamos definir una clase para la velocidad que sea una especialización de un vector.

```
public class Velocity extends VectorND {
}
```

Si los constructores se heredan entonces podríamos crear un objeto velocidad utilizando el constructor de la superclase de la forma:

```
Velocity v = new Velocity(new double[]{0, 1, 2, 3});
```

Nuevamente, si intentan correr ese código el programa no va a compilar. Esto se debe a que en *Java*[8] los constructores **no se heredan**. Es muy importante tener esto en cuenta ya que un error común es confundir la llamada implícita a `super()` con herencia de constructores.

Ejercicio 6.7. ¿Se puede enviar el mensaje `super(...)` desde fuera del constructor de la clase?

Ejercicio 6.8. ¿Se le ocurre una razón por la cuál en *Java* no se tiene herencia de constructores? *Hint:* Piense en la definición de constructor que se dio en la sección **??**.

[8] Hay lenguajes que **sí tienen** herencia de constructores (e.g. *Python*).

6.6. Contextos estáticos

Hasta ahora hemos estado utilizando la keyword `static` para escribir nuestros programas, pero no nos hemos detenido a ver lo que significa. ¿Cuál es la diferencia entre declarar `public static void main(S` y `public void main(String[] args)` ? El objetivo de esta sección será explicar a grandes rasgos qué son y cómo se utilizan los *contextos estáticos*.

6.6.1. Métodos estáticos

Comencemos con un ejemplo. Siguiendo con lo que hemos construido hasta ahora, supongan que ahora queremos implementar una poción que cure 10 *hit points* a un *Bakémon*. Un primer acercamiento a este problema podría ser:

```
public class Potion {
  public void useOn(Bakemon target) {
    target.setHitPoints(target.getHitPoints() + 10);
  }
}
```

Ahora, el código anterior puede ser un poco confuso de leer, así que agreguemos un método auxiliar en nuestra clase `Bakemon` para simplificar nuestro código.

```
// com.github.cc3002.bucket.monsters.Bakemon
public void increaseHitPoints(int amount) {
  this.hitPoints += amount;
}
```

Con este cambio podemos simplificar nuestra implementación anterior.

```
// com.github.cc3002.bucket.items.Potion
public void useOn(Bakemon target) {
  target.increaseHitPointsBy(10);
}
```

Utilicemos lo que llevamos, creemos un *Bakémon*, disminuyamos su vida y luego curémoslo con la poción.

```
Bakemon italicore = new Bakemon("Italicore", 15);
Potion potion = new Potion();
italicore.setHitPoints(0);
```

```
potion.useOn(italicore);
System.out.println("hp = " + italicore.getHitPoints());
```

El código anterior funciona como esperamos, pero tiene un problema que a pequeña escala podría parecer irrelevante, pero en proyectos grandes podría generar problemas graves.

Cada vez que creo una poción, ese objeto ocupa espacio en la memoria que permanece ahí durante un tiempo que no podemos definir. ¿Qué pasa si tenemos 100 pociones? ¿y 1000? Más aún, cuando definimos los objetos en el capítulo **??** dijimos que un objeto se componía de dos cosas (un comportamiento y un estado), pero este objeto no tiene estado. ¿Tiene sentido entonces que existan instancias de la clase `Potion`?

Nos gustaría poder definir esto de manera más eficiente en memoria y, de paso, mejorar el diseño de nuestra solución. Es aquí donde entran los métodos estáticos.

Un **método estático** es un método que no participa del *method lookup* ya que va asociado a la clase y no a las implementaciones de esta. Esto se parece a los constructores, pero existen ciertas diferencias, pero antes de ver estas diferencias veamos cómo modificar nuestra clase original para solucionar el error que habíamos cometido.

Para definir un método estático, solamente necesitamos utilizar la *keyword* **`static`** en la definición del método.

```
// com.github.cc3002.bucket.items.Potion
public static void useOn(Bakemon target) {
  target.increaseHitPointsBy(10);
}
```

Este pequeño cambio tiene un gran impacto en la eficiencia del programa y la diferencia tanto en su implementación y su uso son mínimos. Modificando el ejemplo de antes para utilizar el método estático tendríamos:

```
Bakemon italicore = new Bakemon("Italicore", 15);
italicore.setHitPoints(0);
Potion.useOn(italicore);
System.out.println("hp = " + italicore.getHitPoints());
```

Noten que ahora, como el método va asociado a la clase, no instanciamos el objeto para llamar el método.

Tanto los constructores como los métodos estáticos van ligados a la clase y no a instancias de ésta. ¿Entonces qué los diferencia? Si recuerdan la sección anterior, entonces ya lo saben. Los métodos

estáticos son (valga la redundancia) métodos, mientras que los constructores no. ¿Qué implicancias tiene esto?

La primera consecuencia que se puede notar de esto es que las instancias de una clase pueden acceder a los métodos estáticos de ésta (después de todo son instancias **de** la clase), pero esto no se puede hacer en la dirección contraria. Esto último también implica que no se pueden llamar a métodos estáticos desde métodos de instancia de la clase. Un ejemplo de esto último sería intentar hacer

```
// com.github.cc3002.bucket.items.Potion
public void useTwiceOn(Bakemon target) {
  this.useOn(target)
}
```

Pueden probar correr el último método y ver que no funciona.

Ejercicio 6.9. ¿Se pueden heredar métodos estáticos? ¿A qué se debe esto último?

Ejercicio 6.10. Dadas las definiciones anterior. ¿Puede dar una razón sobre por qué el método `main` debe ser estático?

6.6.2. Variables estáticas

Adicionalmente a los métodos estáticos también existen las variables estáticas. Estas variables funcionan de la misma manera que los métodos estáticos ya que tampoco participan del *method lookup*. La diferencia con los métodos es que al ser variables pueden asignarse y accederse a sus valores de la misma forma que cualquier otra variable de la clase.

Un ejemplo de uso de variable estática es `Math.PI`, noten que esto tiene sentido ya que π es una constante y por lo tanto es independiente del contexto en el que se le llame [9].

Retomemos el ejemplo de la poción para mostrar cómo utilizar estas variables.

```
public class Potion {
  public static int healPoints = 10;

  public static void useOn(Bakemon target) {
    target.increaseHitPointsBy(healPoints);
  }
}
```

Luego, podemos usar la clase como:

[9] Volveremos a visitar esta clase en la sección (véase sección **??**)

```
Bakemon italicore = new Bakemon("Italicore", 15);
italicore.setHitPoints(0);
Potion.useOn(italicore);
System.out.println("hp = " + italicore.getHitPoints());
italicore.setHitPoints(0);
Potion.healPoints = 15;
Potion.useOn(italicore);
System.out.println("hp = " + italicore.getHitPoints());
```

Las variables estáticas suelen utilizarse para definir constantes, ya que el valor será el mismo para todas las instancias de la clase, pero este mismo comportamiento puede llevar a errores muy difíciles de encontrar. Veamos un pequeño ejemplo.

```
public class Example {
  private int i = 1;

  public Example() {
    System.out.println(++i);
  }

  public static void main(String[] args) {
    new Example();
    new Example();
  }
}
```

Si ejecutamos el código anterior el programa imprimiría en consola

```
2
2
```

Pero si cambiamos la variable `i` por una estática el resultado es totalmente distinto, si declaramos la variable como `private static int i = 1` el programa va a imprimir

```
2
3
```

¿Por qué pasa esto? Ya mencionamos la razón, las variables estáticas van ligadas a la clase y no a instancias puntuales de ésta, o dicho de otra forma, son comunes para todas las instancias de la clase.

Existen otros usos y tipos de contextos estáticos, pero son menos comunes y no serán de importancia en el curso, así que serán omitidos.

Ejercicio 6.11. ¿Qué sucede si cambiamos el ejemplo de la poción por lo siguiente?

```
public class Potion {
  public int healPoints = 10;

  public static void useOn(Bakemon target) {
    target.increaseHitPointsBy(healPoints);
  }
}
```

¿Por qué ocurre esto?

Ejercicio 6.12. ¿Se pueden heredar variables estáticas? ¿Por qué?

Ejercicio 6.13. ¿Se pueden llamar métodos o variables estáticas desde el constructor?

6.7. Clases abstractas

En las secciones anteriores creamos una clase `Bakemon` de la que extendían tipos específicos de *Bakémon*, esto no está mal. ¿Pero tiene sentido crear instancias de la clase `Bakemon`? O dicho de otra forma: ¿Existe algún contexto en el que necesitaríamos un *Bakémon* sin tipo? La verdad es poco probable.

Para entender este problema alejémonos un poco del ejemplo anterior y vamos a algo más simple, si tenemos un tenedor, una cuchara y un chuchillo, entonces podemos agruparlos diciendo que todos son utensilios. ¿Pero qué es un utensilio? ¿Existe algún utensilio que no sea un cuchillo, una cuchara, un tenedor u otro? Replanteemos esta última pregunta: ¿Existe algún elemento que sea un utensilio solamente? La respuesta es no, no existe una representación concreta (o material) de un utensilio, diremos entonces que es un **concepto abstracto**.

Retomando el ejemplo original, un *Bakémon* sería un concepto abstracto, ya que engloba al conjunto de *Bakémon*, pero no tiene una forma concreta ya que todos los *Bakémon* tienen un tipo.

En *Java* para representar conceptos abstractos existen las **clases abstractas**. Una clase abstracta es una clase que **no puede ser instanciada**, i.e. no puede recibir el mensaje `new`. Aún así, las clases abstractas pueden tener constructores porque, a pesar de no poder recibir el mensaje `new`, sí pueden recibir el mensaje `super(...)`.

Para definir una clase abstracta se utiliza la *keyword* `abstract`, con lo que podemos reescribir nuestra clase `Bakemon`[10] como:

[10] Es una buena práctica que el nombre de las clases abstractas comience con `Abstract`.

```java
public abstract class AbstractBakemon {
  private String name;
  private String type;
  private int hitPoints;

  protected AbstractBakemon(String name, String type, int hitPoints) {
    this.name = name;
    this.type = type;
    this.hitPoints = hitPoints;
  }

  public String getName() {
    return name;
  }

  public String getType() {
    return type;
  }

  public void increaseHitPointsBy(final int amount) {
    hitPoints += amount;
  }

  public int getHitPoints() {
    return hitPoints;
  }

  public void setHitPoints(int hitPoints) {
    hitPoints = hitPoints;
  }
}
```

Noten que el único cambio que hicimos fue agregar la *keyword* `abstract` a la definición de la clase. Con esto podemos reutilizar las funciones de esta clase y evitar la duplicación de código, pero, como veremos ahora y en lo que sigue del curso, es crucial siempre cuestionar nuestro diseño.

En el capítulo **??** dijimos que el fin de utilizar herencia **no es** evitar la duplicación de código, sin embargo esa es la única función que está cumpliendo nuestra clase abstracta ahora. Para que tenga sentido utilizar clases abstractas entonces debe existir al menos una funcionalidad que no esté implementada en la clase abstracta y que se le delegue esa funcionalidad a sus hijos, esto es lo que se conoce como *métodos abstractos*.

Un **método abstracto** es un método que se define en una clase abstracta pero se implementa en las clases que heredan de ésta. En la práctica, un método abstracto se define como la firma del método seguido de un `;`.

Ahora, del ejemplo anterior hay un método que debiera ser abstracto. Si se fijan, el método `getType()` será distinto para todas las clases que la extiendan y, por lo tanto, debiera ser abstracto, así podemos redefinir la clase como:

```
public abstract class AbstractBakemon {
  private String name;
  private int hitPoints;
  private int hitPoints;

  protected AbstractBakemon(String name, int hitPoints) {
    this.name = name;
    this.hitPoints = hitPoints;
  }

  public String getName() {
    return name;
  }

  public abstract String getType();

  public void increaseHitPointsBy(int amount) {
    hitPoints += amount;
  }

  public int getHitPoints() {
    return hitPoints;
  }

  public void setHitPoints(int hitPoints) {
    hitPoints = hitPoints;
  }
}
```

Y ahora modificamos sus clases hijas de la siguiente forma.

```
public class WaterBakemon extends AbstractBakemon {
  public WaterBakemon(String name, int hitPoints) {
    super(name, hitPoints);
```

```
  }

  @Override
  public String getType() {
    return "WATER";
  }
}
```

Noten que eliminamos el parámetro `type`, una mala práctica común es tener métodos y variables innecesarias en nuestro código, esto puede llevar a un mal entendimiento y uso del programa así que es mejor eliminarlas.

Un último detalle importante sobre las clases abstractas es que todos los métodos abstractos **deben** implementarse en las clases que hereden de ésta (ya sea directa o indirectamente). Si no se hace esto, el programa no va a compilar.

Ejercicio 6.14. ¿Tiene sentido tener un constructor público en una clase abstracta? ¿Y uno privado?

6.8. La librería estándar

En el curso de algoritmos y estructuras de datos vieron cómo implementar varias estructuras de datos para resolver problemas comunes, tener una idea general de cómo se implementan y los beneficios y costos de éstas es crucial para el ámbito de programación. Dicho esto, la mayoría de los lenguajes de programación tienen implementadas las estructuras más usuales dentro de sus librerías estándar, y *Java* no es la excepción[11].

La librería estándar de *Java* es demasiado amplia como para cubrirla en este apunte por lo que sólo veremos los contenidos que nos serán de utilidad, en particular listas y diccionarios.

6.8.1. Listas

Hasta ahora hemos utilizado arreglos para almacenar información, los arreglos tienen la ventaja de ser eficientes en cuanto a velocidad y memoria pero la poca flexibilidad que presentan resulta ser poco práctica para la mayoría de los casos.

El problema de la poca flexibilidad de los arreglos puede solucionarse utilizando una estructura de lista. Existen muchas formas de implementar listas, una de las implementaciones más comunes de lista es la basada en arreglos. En *Java* podemos definir esta estructura de la siguiente manera:

```
public class ArrayList {
  private Object[] elements = new Object[8];
```

[11] *C* es un ejemplo de lenguaje que no tiene implementadas estas estructuras.

```
  private int size = 0;

  public Object get(int i) {
    return elements[i];
  }

  public void add(Object o) {
    if (++size == elements.length) {
      grow();
    }
    elements[size - 1] = o;
  }

  private void grow() {
    Object[] newArray = new Object[elements.length * 2];
    for (int i = 0; i < elements.length; i++) {
      newArray[i] = elements[i];
    }
    elements = newArray;
  }
}
```

Esta es una implementación estándar de una estructura de este tipo, ahora, si bien nuestra implementación es correcta en general queremos evitar «reinventar la rueda» ya que es un gasto de tiempo innecesario y es propenso a cometer errores. Para evitar esto *Java* implementa una gran cantidad de las estructuras y algoritmos más comunes.

Volviendo al ejemplo de recién, no es necesario que implementemos la clase `ArrayList` ya que es parte de la librería estándar junto con implementaciones de otros tipos de listas.

Las listas en *Java* implementan la interfaz `List`. Algunos de los métodos que expone esta interfaz son:

```
// Retorna el tamaño de la lista
int size();
// Pregunta si la lista está vacía
boolean isEmpty();
// Pregunta si un elemento está en la lista
boolean contains(Object o);
// Agrega un elemento a la lista
boolean add(E e);
// Remueve un elemento de la lista
```

```
boolean remove(Object o);
// Ordena la lista de acuerdo a su orden natural
// Por defecto todas las implementaciones de List utilizan MergeSort
void sort(Comparator<? super E> c);
// Vacía la lista
void clear();
// Obtiene el elemento en la posición index
E get(int index);
// Cambia el elemento en la posición index por element
E set(int index, E element);
// Retorna el índice del elemento o
int indexOf(Object o);
```

Ahora, si queremos crear una lista de enteros podríamos hacerlo como:

```
List<Integer> ints = new ArrayList<>();
for (int i = 0; i < 50; i++) {
  ints.add(i);
}
```

Luego, si queremos obtener los valores de la lista lo podemos hacer así:

```
for (int i : ints) {
  System.out.println(i)
}
```

Noten que al momento de definir el tipo ocupamos la sintaxis `<Integer>`, esto quiere decir que la lista estará formada por elementos de tipo `Integer`.[12] El parámetro que va entre `<...>` debe ser el nombre de una clase o interfaz (**no puede ser un tipo primitivo**), esto es lo que se conoce como *tipo genérico* y lo revisaremos en más detalle en la sección **??**.

Algunas implementaciones de la interfaz de listas que vale la pena mencionar son:

- `ArrayList`: Lista implementada con arreglos nativos de tamaño dinámico.
- `LinkedList`: Implementación como una lista doblemente enlazada.

En general lo más común es utilizar un `ArrayList`, pero dependiendo del caso puede ser más conveniente una lista enlazada.[13]

[12] Las listas en *Java* son homogéneas.
[13] **so-linked-vs-array**.

6.8.2. Diccionarios

Otra estructura sumamente importante son los diccionarios, estos nos permiten asociar *llaves* con *valores* y así poder acceder a ellos de manera eficiente. Al igual que con las listas existen muchas implementaciones distintas de diccionarios pero una forma simple de verlos es pensarlos como una tabla con dos columnas, donde la primera contiene las llaves y la segunda los valores.

En *Java* los diccionarios se agrupan dentro de la interfaz `Map<K, V>`, donde `K` es el tipo de la llave y `V` el del valor (al igual que como sucede con las listas, las llaves y valores son homogéneos).

Algunos métodos útiles que define la interfaz `Map` son:

```
// Entrega la cantidad de pares llave-valor del diccionario
int size();
// Retorna true si el diccionario está vacío
boolean isEmpty();
// Verifica si una llave existe en el diccionario
boolean containsKey(Object key);
// Análogo al anterior pero para valores
boolean containsValue(Object value);
// Retorna el valor asociado a la llave key
V get(Object key);
// Asocia el valor value con la llave key
V put(K key, V value);
// Elimina la llave key y su valor asociado y retorna el valor.
V remove(Object key);
// Obtiene todos los pares llave-valor.
EntrySet<K, V> entrySet();
```

Noten que dada la naturaleza de los diccionarios una llave puede estar asociada a un solo valor.

A diferencia de las listas, existen bastantes más tipos interesantes de diccionarios, algunos que vale la pena mencionar:

- `HashMap`: Diccionario implementado con una función de *hashing*.
- `ConcurrentHashMap`: Equivalente a `HashMap` pero para operaciones con múltiples *threads*.
- `LinkedHashMap`: Implementación de un diccionario con una función de *hashing* que adicionalmente guarda una lista para conservar el orden en el que se insertan los elementos.
- `EnumMap`: Implementación especial del `HashMap` que usa enumeraciones[14] como llaves y tiene una función de *hashing optimizada*.[15] Este tipo de diccionarios es más limitado pero

[14] Refiérase a la sección **??**

[15] Este tipo de *hashing* toma provecho de lo que se conoce como *universos finitos* y lo verán en el curso de *Diseño y Análisis de Algoritmos*.

asegura una eficiencia mayor a la de un `HashMap` tradicional.

- `TreeMap`: Diccionario implementado con un árbol rojo-negro. Es menos eficiente que un `HashMap` pero ordena los elementos por su llave.

Veamos ahora un pequeño ejemplo de uso de `TreeMap`.

```
Map<String, String[]> courses = new TreeMap<>();
courses.put("CC3501", new String[]{"Daniel C."});
courses.put("CC3102", new String[]{"Alejandro H."});
courses.put("CC3002", new String[]{"Alexandre B.", "Nancy H."});
courses.put("CC3301", new String[]{"Luis M."});
System.out.println(Arrays.toString(courses.get("CC3501")));

courses.remove("CC3501");
courses.put("CC3501", new String[]{"Nancy H."});
System.out.println(Arrays.toString(courses.get("CC3501")));

for (Map.Entry<String, String[]> course : courses.entrySet()) {
  System.out.println(course.getKey() + " "
                     + Arrays.toString(course.getValue()));
}
```

Ejercicio 6.15. Pruebe el mismo ejemplo anterior con otros tipos de diccionarios y vea cómo cambian los resultados.

TODO:

- Sets
- Colas

Capítulo 7

¡Java!

7.1. La *keyword* `final`

7.2. Manejo de memoria

7.3. Excepciones

7.4. Tipos genéricos

7.5. Enumeraciones

7.6. *JARs*

En esta sección veremos lo que son los *Java Archives* más conocidos como *JARs* y las herramientas de las que disponemos para crear y utilizarlos.

Hasta ahora hemos utilizado el *IDE* para ejecutar nuestros programas, esto es bueno cuando estamos desarrollando pero no cuando queremos entregar un producto que sea utilizable por un cliente (no podemos esperar que el cliente instale un *IDE* solamente para correr el programa).

La manera más básica de ejecutar un programa en *Java* es mediante la línea de comandos (*CLI*), finalmente lo que hace *IntelliJ* es brindar una interfaz más amigable para que nosotros no tengamos que manejar el proceso de compilación manualmente. Antes de ver lo que es un archivo *JAR* veamos un poco más en detalle lo que hace el compilador de *Java*.

Si recuerdan, para compilar un programa desde la terminal se debían ejecutar dos instrucciones, primero `javac` y luego `java`. ¿Pero por qué dos comandos en lugar de simplemente uno? Una de las principales razones por las que *Java* fue tan ampliamente adoptado en la industria es porque fue uno de los primeros lenguajes de programación que era independiente del sistema operativo (i.e.

que el mismo código funcionaba en *Windows*, *UNIX*, etc.). Para lograr esto el proceso de ejecución de un programa de *Java* se dividió en dos grandes pasos, compilación y ejecución.

El comando *javac* invoca al compilador crea archivos (`.class`) con *bytecode* ejecutable por la *JVM*. Luego, el comando `java` toma los archivos generados por el comando anterior para convertirlos en binarios nativos del sistema operativo y así poder ejecutarlos. El beneficio de esto es que los archivos generados por el primer paso de la compilación serán independientes del sistema operativo ya que no serán directamente interpretados por éste, sino que por la máquina virtual.

Un *JAR* es una forma de «enfrascar» en un archivo un programa junto con sus metadatos y archivos adicionales que pueda necesitar[1] y así generar un programa más simple de distribuir y ejecutar.

7.6.1. Línea de comandos

La forma más básica de empaquetar un programa en este formato es mediante la línea de comandos.

TODO:

- JARs
- JIT
- CLI
- Gradle

[1] Se suele referir a estos archivos adicionales como recursos del programa.

Parte III

Patrones y metodologías de diseño

Capítulo 8

Test Driven Development

Capítulo 9

Con la vista al frente

En este capítulo introduciremos los conceptos y herramientas necesarias para construir interfaces gráficas en *Java*. Al igual que el resto del apunte, el objetivo de este capítulo no será profundizar en las herramientas que provee *Java* para implementar interfaces gráficas, sino que se verá de forma general para explicar los conceptos relacionados a la arquitectura de un programa que tenga una interfaz gráfica (o *GUI*).

9.1. Modelo-Vista-Controlador

Para explicar lo que es el *Modelo-Vista-Controlador* (o *MVC*) primero debemos familiarizarnos con los conceptos *arquitectura de software* y *patrón arquitectónico*.

Arquitectura de software es un concepto complejo. Definir la arquitectura de una aplicación es uno de los procesos más importantes y difíciles dentro del desarrollo de un proyecto, y no existe un acuerdo respecto a **qué entendemos por arquitectura**. En palabras de *Martin Fowler* (citando a *Ralph Johnson*), la arquitectura son «las decisiones que desearíamos haber tomado bien al principio de un proyecto».[1] Esta definición es algo esotérica pero es debido a la dimensión del problema de definir la arquitectura de nuestro programa, intentando simplificar, podríamos decir que la arquitectura de software es la forma en la que se estructuran, comunican y relacionan todas las componentes de nuestro programa.

[1] **fowler-software-architecture**.