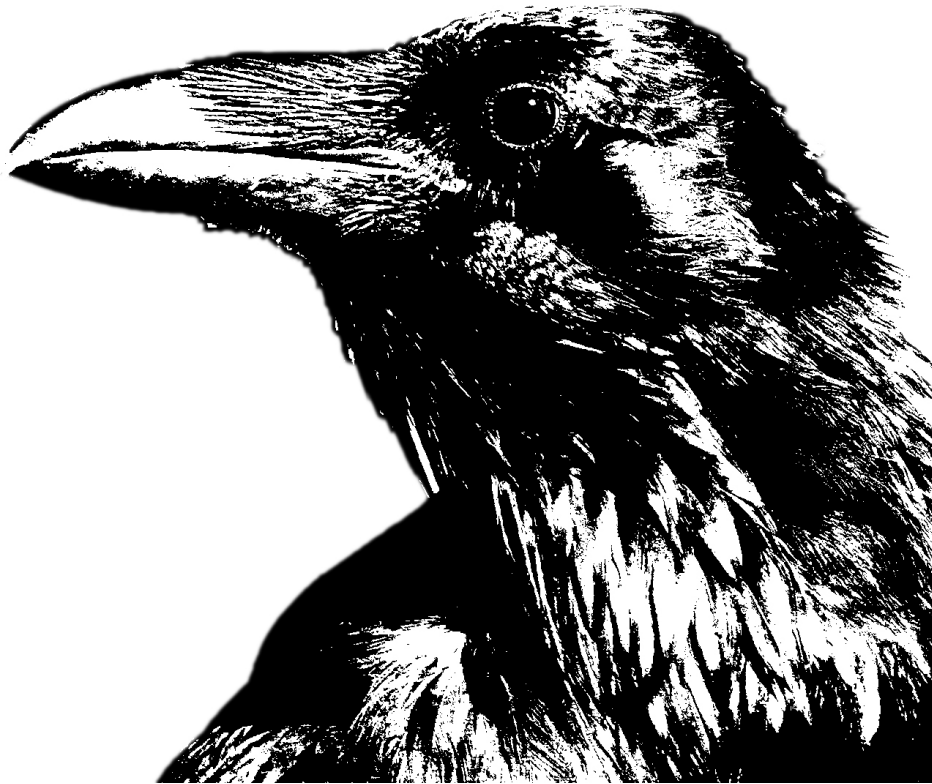


EL TIEMPO PASA Y EL SOFTWARE MUERE

Diseñando programas resistentes al cambio

Inspirado por las clases de Alexandre Bergel

Ignacio Slater Muñoz



Departamento de Ciencias de la Computación
Universidad de Chile

Índice general

I	Por algo se empieza	11
1.	¡Kotlin!	13
1.1.	¿Java?	13
1.2.	¿Kotlin?	13
1.2.1.	Instalando Kotlin	14
1.2.2.	Windows	14
1.2.3.	Linux	16
1.2.4.	MacOS	17
1.3.	Tipos en Kotlin	17
1.3.1.	Tipos explícitos	18
1.3.2.	Tipos inferidos	18
1.3.3.	Funciones	19
1.4.	Complicando las cosas	21
1.4.1.	La función de Fibonacci	21
1.4.2.	Implementación básica (recursiva)	21
1.4.3.	Implementación iterativa	22
1.4.4.	Programación dinámica	23
1.5.	Ejercicios	24
2.	IntelliJ, tu nuevo bff	29
2.1.	No termina de convencerme, pero le voy a dar una oportunidad. ¿Cómo lo instalo?	29
2.1.1.	JetBrains Toolbox	30
2.1.2.	IntelliJ IDEA	30
2.2.	Conociendo IntelliJ	31
2.2.1.	Creando un proyecto	31
2.2.2.	Trabajando con IntelliJ	32
2.3.	Mi primera aplicación	34
2.3.1.	Paquetes	34
2.3.2.	Mi primer Kotlin	35
2.3.3.	Loop principal	37
2.4.	Ejercicios	40
II	OOP no es religión	45
3.	Recuerdo de mi primer objeto	47
3.1.	Sistemas de control de versiones	47
3.1.1.	Git	48
3.1.2.	Instalando Git	48

3.1.3.	Configurando Git	48
3.2.	Creando el proyecto	49
3.3.	Objetos	51
3.4.	Clases	53
3.4.1.	Constructores primarios	54
3.5.	Ejercicios	55
4.	Programación orientada a objetos en principio	59
4.1.	Composición	59
4.2.	Encapsulamiento	61
4.3.	Herencia	63
4.3.1.	Herencia en Kotlin	65
4.3.2.	Method-lookup	67
4.4.	Polimorfismo de subtipos	70
4.5.	Principio de Liskov	77
4.6.	Ejercicios	79
5.	Cuando todo no funciona	87
5.1.	Testing frameworks	88
5.2.	Gradle	88
5.3.	Conociendo a <i>Kotest</i>	95
5.4.	Segunda cita con <i>Kotest</i>	98
5.5.	Mejorando nuestro código	101
5.6.	Ejercicios	103
6.	Nuevo de fábrica: <i>factory method pattern</i>	111
6.1.	Modelando tipos	111
6.2.	Factory method pattern	113
6.3.	Test factories	117
6.4.	Lo bueno y lo malo	118
6.4.1.	Lo bueno	118
6.4.2.	Lo malo	118
6.5.	Ejercicios	119
7.	El capítulo ambiguo	123
7.1.	Ataques: primer intento	123
7.1.1.	Recibiendo daño	123
7.1.2.	Atacando	126
7.2.	Ataques: segundo intento	128
7.3.	<i>Double Dispatch</i>	132
7.4.	Ataques: Tercer intento	135
7.5.	Ejercicios	142
8.	Cuando todo male sal	147
8.1.	Setters	149
8.2.	<i>Java Virtual Machine</i>	153
8.3.	Excepciones	155
8.3.1.	Arrojando excepciones	156
8.3.2.	Atrapando excepciones	157
8.3.3.	Usando excepciones	158
8.4.	Getters	160
8.5.	Ejercicios	162

9. En efecto	167
9.1. Efectos: Primer intento	167
9.2. Strategy Pattern	171
9.3. Efectos: segundo intento	172
9.4. <i>Data classes</i>	174
9.5. <i>Coverage</i>	177
10. Juguemos en el bosque	183
Index	185

¿Cómo contribuir?

Si quieres contribuir a este proyecto, puedes hacerlo de varias formas siempre y cuando respetes el [código de conducta](#) del proyecto.

- Si encuentras algún error ortográfico o de redacción, puedes abrir un *issue* en el repositorio de *Github* o bien, si tienes conocimientos de \LaTeX , puedes hacer un *pull request* con la corrección.
- Si encuentras algún error en el código, puedes abrir un *issue* en el repositorio de *Github* o bien, si tienes conocimientos de *Kotlin*, puedes hacer un *pull request* con la corrección.
- Cualquier sugerencia es bienvenida, basta con abrir un *issue* en el repositorio de *Github*.
- Si quieres contribuir con el contenido del libro, puedes hacerlo de dos formas:
 - Si tienes conocimientos de \LaTeX , puedes hacer un *pull request* con la corrección.
 - Si no tienes conocimientos de \LaTeX , puedes abrir un *issue* en el repositorio de *Github* con la sugerencia de contenido que quieres agregar.

Enlace al repositorio de *Github*: <https://github.com/r8vnhill/software-design-book-es>

La parte del libro que nadie lee

La idea de este «apunte» nació como una *wiki* de *Github* creada por Juan-Pablo Silva como apoyo para el curso de *Metodologías de Diseño y Programación* dictado por el profesor Alexandre Bergel del Departamento de Ciencias de la Computación, Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile.

Lo que comenzó como unas notas para complementar las clases del profesor lentamente fue creciendo, motivado por esxs alumnxs que buscaban dónde encontrar soluciones para esas pequeñas dudas que no les dejaban avanzar.

El objetivo principal del texto sigue siendo el mismo, plantear explicaciones más detalladas, ejemplos alternativos a los vistos en clases y para dejar un documento al que lxs alumnxs puedan recurrir en cualquier momento.

Este libro no busca ser un reemplazo para las clases del curso, es y será siempre un complemento.

Esta obra va dirigida a los estudiantes de la facultad así como para cualquier persona que esté dando sus primeros pasos en programación. El libro presenta una introducción al diseño de software, la programación orientada a objetos y lo básico del lenguaje de programación *Kotlin*.

Antes de comenzar, debo agradecer a las personas que hicieron posible y motivaron la escritura de esto: Beatríz Grabloza, Dimitri Svandich, Nancy Hitschfeld y, por supuesto, Alexandre Bergel y Juan-Pablo Silva.

24 de febrero de 2023, Santiago, Chile

Sobre esta versión

Este libro no partía así, y si alguien leyó una versión anterior se dará cuenta. Solía comenzar con una descripción e instrucciones para instalar las herramientas necesarias para seguir este libro y eso no estaba mal, pero no me agradaba comenzar así, simplemente presentando las herramientas sin ningún contexto de por qué ni para qué las íbamos a utilizar.

Puede parecer ridículo cambiar todo lo que ya había escrito solamente por eso, pero cuando nos enfrentamos a problemas del mundo real esto comienza a cobrar más sentido. Reescribo estos capítulos por una razón simple pero sumamente importante y que será una de las principales motivaciones para las decisiones de diseño que tomaremos a medida que avancemos, en el desarrollo de software **lo único constante es el *cambio***.¹ Una aplicación que no puede adaptarse a los cambios, sin importar que tan bien funcione, está destinada a morir.

¿Qué sucede entonces con las herramientas que vamos a utilizar? Las vamos a introducir, no podemos sacar una parte tan importante, pero no las vamos a presentar todas a la vez, en su lugar las iremos explicando a medida que las vayamos necesitando.

¹Freeman y Robson, «Chapter 1: Intro to Design Patterns»; Metz, «Chapter 1: Object-Oriented Design».

Parte I

Por algo se empieza

Capítulo 1

¡Kotlin!

1.1. ¿Java?

Java es uno de los lenguajes de programación más utilizados en el mundo, se caracteriza por ser un lenguaje basado en clases, orientado a objetos, estática y fuertemente tipado, y (casi totalmente) independiente del sistema operativo.

¿Qué?

Tranquilos, vamos a ir de a poco. Comencemos por uno de los puntos que hizo que *Java* fuera adoptado tan ampliamente en la industria, la independencia del sistema operativo. Cuando *Sun Microsystems*¹ publicó la primera versión de *Java* (en 1996), los lenguajes de programación predominantes eran C y C++ (y en menor medida *Visual Basic* y *Perl*).² Estos lenguajes tenían en común que interactuaban directamente con la API del sistema operativo, lo que implicaba que un programa escrito para un sistema *Windows* no funcionaría de la misma manera en un sistema *UNIX*. *Java* por su parte planteó una alternativa distinta, delegando la tarea de compilar y ejecutar los programas a una máquina virtual (más adelante veremos en más detalle parte del funcionamiento de la *Java Virtual Machine* para entender sus beneficios y desventajas). Esto último hizo que, en vez de cambiar el código del programa para crear una aplicación para uno u otro sistema operativo, lo que cambiaba era la versión de la *JVM* permitiendo así que un mismo código funcionara de la misma forma en cualquier plataforma capaz de correr la máquina virtual.³ Pasarían varios años antes de que surgieran otros lenguajes que compartieran esa característica (destacando entre ellos *Python 2*, publicado el año 2000).

1.2. ¿Kotlin?

Kotlin es un lenguaje de programación *multiplataforma*, estática y fuertemente tipado y con una inferencia de tipos más avanzada que la de *Java*.

¿Pero para qué aprender Kotlin si puedo aprender Java?

Buena pregunta.

¹ Actualmente *Java* es propiedad de *Oracle Corporation*

² *The Most Popular Programming Languages - 1965/2022 - New Update -*.

³ Actualmente casi todos los sistemas operativos son capaces de usar la *JVM*, en particular el sistema *Android* está implementado casi en su totalidad para usar esta máquina virtual.

Kotlin es un lenguaje desarrollado por *JetBrains* pensado para ser totalmente interoperable con *Java*, esto quiere decir que puedo importar código escrito en *Java* desde *Kotlin* y vice versa. En fin, existen muchísimas razones para aprender *Kotlin*, pero en vez de enumerarlas creo que es mejor que las vayan descubriendo en el transcurso de este libro.

1.2.1. Instalando Kotlin

Lo primero que necesitaremos para trabajar en *Kotlin* es el compilador.⁴

Ya veo

La forma más fácil de instalar el compilador de *Kotlin* es instalando *IntelliJ* (como veremos en el capítulo 2), pero la solución más fácil no siempre es la correcta. Instalar *IntelliJ* para aprender lo más básico de *Kotlin* es como intentar andar en motocicleta cuando todavía estamos aprendiendo a andar en bicicleta con rueditas (pero tranquilxs, hacia el final de este capítulo ya habremos encontrado nuestro equilibrio espiritual).

1.2.2. Windows

Windows Package Manager (winget)

Comencemos por una herramienta esencial para trabajar en *Windows*. *Windows Package Manager* es el gestor de paquetes de *Windows*.

Increíble

Como el nombre anterior es demasiado largo, lo llamaremos *winget* y esperaremos que no le moleste.

El gestor *winget* viene instalado por defecto en *Windows 11* y en las versiones más recientes de *Windows 10*, para ver que esté instalado necesitaremos abrir *Powershell* (PS) y ejecutar el siguiente comando:

```
winget -?
```

Si el comando anterior **no** da error significa que *winget* está instalado, si es así pueden pasar a la siguiente sección, lxs espero.

¿Ya se fueron?

Ok, ahora veremos cómo instalar *winget* en caso de que no lo tengan instalado en su sistema.

Pongan atención para no perderse.

Prepárense.

Para instalar *winget* deben acceder a la *Tienda de Windows* y descargar el *Instalador de aplicación*.

¡Perfecto! Ya completamos el primer paso para aparentar ser el hacker que siempre quisiste.

Windows Terminal

Para instalar lo que necesitaremos basta con tener *Powershell*, la terminal integrada de *Windows* (por favor, NUNCA usen *cmd*), pero actualmente *Windows* provee una nueva terminal llamada *Windows Terminal*⁵ (WT). WT viene instalada en las versiones más nuevas de *Windows 10* y en todas las versiones de *Windows 11*. Pueden usar el *Buscador*

⁴Un compilador es un programa que «traduce» código escrito en un lenguaje de programación a otro (e. g. traducir un programa a lenguaje de máquina para ejecutarlo).

⁵Es increíble pensar que a alguien le pagaron por idear ese nombre.

de Windows para abrir la aplicación «Terminal» o, si quieren hacerle creer a su abuela que son capaces de hackear la NASA,⁶ pueden abrir *Powershell* y ejecutar la instrucción:

```
wt.exe
```

Si no encuentran la aplicación o si el comando anterior arrojó un error, pueden instalar la terminal usando *winget*, para esto deben ejecutar el siguiente comando en *Powershell*:

```
winget install Microsoft.WindowsTerminal
```

A partir de ahora, cada vez que hable de *Powershell* me estaré refiriendo a una sesión de *Powershell* corriendo en *Windows Terminal*.

Oh-My-Posh

Utilizar *Windows Terminal* en lugar de la *terminal de Windows* (jaja) por defecto es una mejora, pero podemos mejorar aún más. Una de las ventajas de WT es la capacidad de personalización que otorga a sus usuarios, esto podemos hacerlo a mano o usar alguna herramienta que lo haga por nosotros, aquí les voy a mostrar una de esas herramientas.

Oh-My-Posh es una herramienta para personalizar *Windows Terminal* de forma simple, como les voy a mostrar en un instante. Para instalar *Oh-My-Posh* abran WT y ejecuten el siguiente comando:

```
winget install JanDeDobbeleer.OhMyPosh -s winget
```

Ahora que tenemos *Oh-My-Posh* instalado, hay que decirle a *Powershell* que lo use, aquí voy a usar *Notepad* porque viene instalado con todas las versiones de Windows, cosa que espero no volver a hacer en la vida. En PS:

```
# Para poder ejecutar scripts (sdwheelerSetExecutionPolicyMicrosoftPowerShell)
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Force
# Creamos el archivo sólo si no existe
if (-not (Test-Path $PROFILE)) {
    New-Item -ItemType File -Path $PROFILE -Force
}
notepad.exe $PROFILE
```

Luego escribamos al comienzo del archivo el siguiente código:

```
oh-my-posh.exe init pwsh `
  --config -join($env:POSH_THEMES_PATH,
    "\powerlevel10k_rainbow.omp.json") | `
  Invoke-Expression
```

Lo último que necesitas es instalar alguna de las fuentes de <https://www.nerdfonts.com>, ahí descomprimen el archivo, seleccionan todos los archivos descomprimidos y con clic derecho debiera darles la opción de instalar.

⁶En caso de una investigación por parte de cualquier entidad federal o similar, el autor de este libro no tiene ninguna participación con este grupo ni con las personas que lo integran, no sé cómo estoy aquí, probablemente agregado por un tercero, no apoyo ninguna acción por parte de los lectores de este libro.

Ahora la próxima vez que abran la terminal podrán ver los cambios que hicimos. Si quieren darle una identidad propia a su terminal pueden revisar la [documentación](#) de *Oh-My-Posh*.

Chocolatey

Lo primero que necesitaremos para instalar las herramientas que usaremos será un gestor de paquetes, utilizaremos *Chocolatey*.

Para partir abran una ventana de *Powershell* como administrador. Una vez abierta, deben ejecutar las instrucciones:⁷

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

Esto otorgará los permisos necesarios y descargará e instalará el gestor de paquetes.

Para comprobar que el programa se haya instalado correctamente corran el comando:

```
choco -?
```

Ahora toca instalar el *JDK*, con chocolatey esto es simple, solamente deben ejecutar:

```
choco install openjdk -y
```

Luego, para ver que se haya instalado correctamente pueden hacer `java -version`, aquí es muy importante que la versión que aparezca **no sea** la versión 1.8 o anteriores, en caso de que esa sea la versión instalada entonces lo recomendado es desinstalar todas las versiones de *Java* instaladas y repetir la instalación.

¡Perfecto! Ahora sólo nos falta instalar el compilador de *Kotlin*. Nuevamente, para instalar el compilador confiaremos en nuestro amigo *Chocolatey*. Podemos instalar el compilador desde *Powershell* con el comando:

```
choco install kotlinc
```

Luego, podemos comprobar la instalación con:

```
kotlinc -version
```

1.2.3. Linux

Para instalar *Kotlin* en sistemas *Linux* se recomienda utilizar *SDKMAN!*⁸. Instalar *SDKMAN!* es relativamente simple, solo deben correr los comandos que presentamos a continuación y seguir los pasos que aparezcan en pantalla.

⁷<https://github.com/islaterm/software-design-book-es/blob/master/install/windows/chocolatey.ps1>

⁸El «!» es parte del nombre


```
curl -s "https://get.sdkman.io" | bash
# Cambie roac por su nombre de usuario
source "/home/roac/.sdkman/bin/sdkman-init.sh"
```

Como es costumbre, podemos probar que se instaló correctamente con:

```
sdk version
```

Con *SDKMAN!* instalado, obtener *Kotlin* es tan simple como:

```
sdk install kotlin
```

Y comprobamos la instalación haciendo:

```
kotlinc -version
```

1.2.4. MacOS

Para instalar *Kotlin* en OSX se pueden seguir las mismas instrucciones que se dieron para *Linux*, o se puede utilizar *Homebrew*:

```
brew update
brew install kotlin
```

Por último comprobamos la instalación:

```
kotlinc -version
```

1.3. Tipos en *Kotlin*

Al igual que *Java*, *Kotlin* es un lenguaje de tipado estático y fuerte, pero con una inferencia de tipos mucho más poderosa que la de *Java*. El sistema de tipos de *Kotlin* es un tema sumamente complejo que se escapa del alcance de este libro, así que nos limitaremos a dar una pequeña introducción a éste y lo iremos desarrollando un poco más a lo largo del libro a medida que vaya siendo necesario.

Lo primero que debemos saber es que *Kotlin* tiene dos tipos de variables: valores y variables.

Un **valor**⁹ (o *read-only variable*) es una variable que no puede ser **reassignada** una vez que se le ha asignado un valor. Esto es más fácil entenderlo con un ejemplo, lo primero es abrir la terminal interactiva de *Kotlin* con el comando `kotlinc` y escribir lo siguiente:

```
val i: Int = 8000
i = 9000
```

⁹Término propio

Como podemos ver, al intentar reasignar el valor de `i` obtenemos un error, esto es porque `i` es un valor y no una variable, por lo que no puede ser reasignada. El error que obtenemos es el siguiente:

```
error: val cannot be reassigned
i = 9000
^
```

Importante. No debemos confundir un valor de solo lectura con una constante, ya que incluso si una variable es declarada como `val` puede ser modificada si su tipo es mutable. Volveremos a esto más adelante.

Por otro lado, una **variable** puede ser reasignada, para ver esto basta cambiar `val` por `var` en la declaración de `i`:

```
var i: Int = 8000
i = 9000
```

En este caso, el código se ejecuta sin problemas c:

1.3.1. Tipos explícitos

Todas las variables en *Kotlin* deben tener un tipo definido, y una vez definido no puede ser cambiado.¹⁰

El tipo de una variable se define después del nombre de la variable y antes del valor que se le asigna, y se separa de éste con dos puntos (:). Podemos ver esto en el ejemplo anterior, donde definimos el tipo de `i` como `Int`.

Existen muchos tipos en *Kotlin*, pero por ahora nos limitaremos a los más básicos:

- `Int`: El conjunto de todos los números enteros de 32 bits. Formalmente: $i : \text{Int} \iff i \in \mathbb{Z} \wedge -2^{31} \leq i \leq 2^{31} - 1$.
- `Double`: El conjunto de todos los números reales de *doble precisión*.¹¹ En este caso, la definición formal es un poco más complicada, pero es cercana a: $d : \text{Double} \iff d \in \mathbb{R} \wedge 4,9 \times 10^{-324} \leq |d| \leq 1,8 \times 10^{308}$.
- `String`: El conjunto de todas las cadenas de caracteres. Formalmente: $s : \text{String} \iff s \in \mathbb{L}^*$, donde \mathbb{L} es el conjunto de todos los caracteres *Unicode* que pueden ser representados en *UCS-2*.
- `Boolean`: El conjunto de dos valores: `true` y `false`.
- `Unit`: El conjunto de un único valor: `Unit`. Esta noción es un poco más complicada, pero se puede entender como el tipo de variables que no almacenan información. Otro nombre común para este tipo es *void*, pero no utilicemos ese nombre porque *Kotlin* no tiene un tipo *void*.

Importante. Noten que todos los tipos en *Kotlin* empiezan con mayúscula, lo que debemos respetar a la hora de crear nuestros propios tipos.

1.3.2. Tipos inferidos

Como mencionamos en la sección anterior: «**Todas** las variables en *Kotlin* deben tener un tipo definido». Sin embargo, *Kotlin* tiene una inferencia de tipos bastante poderosa, por lo que en la mayoría de los casos no es necesario definir explícitamente el tipo de una variable. Esto es muy útil, ya que va a reducir muchísimo la cantidad de texto que tenemos que escribir para hacer nuestros programas; esto también va a hacer que nuestros programas sean más fáciles de leer y entender al reducir la cantidad de código *boilerplate* que tenemos que procesar. Créanme cuando les digo que la mayor parte del tiempo que trabajan lo van a pasar leyendo código.

Pero ES MUY IMPORTANTE no abusar de la inferencia de tipos, ya que esto puede terminar teniendo el efecto contrario al que queremos: hacer que nuestros programas sean más difíciles de leer y entender, i.e. gastar más tiempo leyendo D:

¹⁰ Este tipo de atrocidades sólo pueden suceder en lenguajes de tipado dinámico como *Python* o *Ruby*.

¹¹ https://en.wikipedia.org/wiki/Double-precision_floating-point_format

Para ver cómo funciona la inferencia de tipos, podemos reescribir el ejemplo anterior de la siguiente manera:

```
var i = 8000
i = 9000
```

No vamos a ahondar más sobre este tema, ya que sería extender innecesariamente este capítulo (y porque no soy experto en el tema), pero a lo largo del libro iremos viendo ejemplos de cómo usar la inferencia de tipos correctamente, en que casos podemos usarla, y en cuales no.

1.3.3. Funciones

Un último tipo de dato que vamos a ver en este capítulo son las funciones. Las funciones son un tipo de dato que toma una cantidad de parámetros y devuelve un valor. Formalmente, sea FT el tipo de las funciones, tenemos que:

$$FT(A_1, \dots, A_n) \rightarrow R$$

con A_i los tipos de los parámetros de la función, y R el tipo de retorno de la función.

En *Kotlin* tenemos dos formas de definir funciones: utilizando la palabra reservada `fun`¹² o utilizando una expresión lambda. En esta sección veremos sólo la primera forma, y dejaremos la segunda para más adelante.

Lo más básico

Una función se define de la siguiente forma:

```
fun <nombre>(<parámetros>): <tipo de retorno> {
    <cuerpo de la función>
}
```

Ahora, podemos definir una función simple que imprima un mensaje en pantalla:

```
fun jojoReference(name: String): Unit {
    println("My name is " + name + ", and I have a dream.")
    return Unit
}
```

Veamos qué está pasando en este ejemplo:

- La palabra reservada `fun` nos indica que estamos definiendo una función.
- El nombre de la función es `jojoReference`. Noten que el nombre de la función está escrito así, esto se llama *camelCase* y es una convención que se usa en *Kotlin* para nombrar variables y funciones.
- La función toma un parámetro, que se llama `name` y es de tipo `String`.
- No vamos a utilizar el resultado de la función, así que podemos definir el tipo de retorno como `Unit`.
- El cuerpo de la función es el bloque de código que se ejecuta cuando llamamos a la función. En este caso, la función imprime un mensaje en pantalla (utilizando la función `println(String): Unit`¹³).
- La función termina con la palabra reservada `return`, que indica que la función termina su ejecución y devuelve el valor que se encuentra después de la palabra reservada. En este caso, la función devuelve el valor `Unit`.

Ahora, podemos llamar a la función de la siguiente manera:

¹²Trad. Diversión

¹³El nombre viene de *print line*, ya que lo que hace es imprimir una línea de texto en la pantalla, i.e. texto seguido de un salto de línea.

```
jojoReference("Giorno Giovanna")
```

Lo otro más básico

Pasemos a la parte que a nadie le gusta, documentar el código. Para documentar algo en *Kotlin* escribiremos un comentario que comience con `/**` y termine con `*/`. Dentro de este comentario escribiremos qué hace la función, qué parámetros toma, y qué valor devuelve. En el ejemplo anterior podríamos escribir lo siguiente:

```
/**
 * Prints a Jojo's Bizarre Adventure reference to the console with the given name.
 */
fun jojoReference(name: String): Unit {
    println("My name is " + name + ", and I have a dream.")
    return Unit
}
```

Es de vital importancia documentar el código, ya que el código que escribimos hoy puede ser utilizado por nosotros o por otras personas dentro de un año, y si no documentamos el código no vamos a saber qué hace cada función, y esto puede hacer que tengamos que botar código a la basura.

En el resto del libro omitiremos la documentación en la mayoría de los ejemplos para no extender innecesariamente el texto, pero esto lo haremos sólo porque el iremos explicando el código en cada ejemplo, pero en la vida real **siempre** debemos documentar el código.

Lo no tan básico

Lo último que nos queda es simplificar el código que escribimos en el ejemplo anterior porque siempre podemos hacerlo mejor.

Primero, como no utilizaremos el valor de retorno de la función, podemos omitir la última línea.

Luego, podemos utilizar la característica llamada *string interpolation* para simplificar la concatenación de cadenas de texto. La *interpolación de cadenas* es una característica que nos permite insertar variables dentro de una cadena de texto, y *Kotlin* nos permite hacer esto con el símbolo `$`. Por ejemplo, si tenemos una variable `name` de tipo `String` y queremos concatenarla con una cadena de texto, podemos hacerlo de las siguientes maneras:

```
val name = "Giorno Giovanna"
println("My name is " + name + ", and I have a dream.")
println("My name is $name, and I have a dream.")
```

Como podemos ver, la segunda forma es mucho más simple y legible que la primera.

Por último, si el tipo de retorno de la función es `Unit`, podemos omitirlo.

Con esto, podemos escribir la función de la siguiente manera:

```
fun jojoReference(name: String) {
    println("My name is $name, and I have a dream.")
}
```

¿Podemos simplificarlo más? Sí, pero dejaremos eso para más adelante ;)

1.4. Complicando las cosas

1.4.1. La función de Fibonacci

La función de Fibonacci es uno de los ejemplos más conocidos de una función recursiva (función que se llama a sí misma). Se define de la siguiente manera:

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n > 1 \end{cases} \quad (1.1)$$

En las secciones siguientes vamos a ver cómo implementar esta función en Kotlin usando distintas técnicas.

1.4.2. Implementación básica (recursiva)

La implementación más sencilla es la recursiva, esta es la que se muestra en la ecuación 1.1.

Empecemos por entender un poco lo que es una instrucción de control de flujo.

Instrucción de control de flujo

Una instrucción de control de flujo es una instrucción que controla el orden en el que se ejecutan las instrucciones de un programa.

En *Kotlin* existen cuatro tipos de instrucciones de control de flujo:

- *if-else*: es una **expresión** que evalúa una condición y ejecuta una instrucción si la condición es verdadera y otra si es falsa.
- *when*: es una **expresión** que evalúa una condición y ejecuta una instrucción dependiendo del valor de la condición. Pueden pensar en esto como una versión más potente del *if-else*.
- *for*: es una **instrucción** que ejecuta una directiva un número determinado de veces.
- *while*: es una **instrucción** que ejecuta una directiva mientras una condición sea verdadera.

Para la implementación recursiva de la función de Fibonacci vamos a usar la expresión *if-else*. La implementación de la función de Fibonacci en *Kotlin* es la siguiente:

```
fun recursiveFibonacci(n: Int): Int {  
    if (n <= 1) {  
        return n  
    } else {  
        return recursiveFibonacci(n - 1) + recursiveFibonacci(n - 2)  
    }  
}
```

Veamos lo que está pasando:

- La función *recursiveFibonacci* recibe un número entero y devuelve un número entero.
- La función evalúa si el número recibido es menor o igual a 1.

- Si el número es menor o igual a 1, la función devuelve el número recibido.
- Si el número es mayor a 1, la función devuelve la suma de la función llamada con el número recibido menos 1 y la función llamada con el número recibido menos 2.

Algo importante sobre las expresiones *if-else* y *when* es que son expresiones

omg!

esto significa que pueden ser evaluadas y devolver un valor, a esto se le llama reducción. Con esto en mente, veremos que podemos utilizar la expresión *if-else* para asignar valores a variables. Veamos un ejemplo:

```
fun printIsPositive(n: Int) {
    val isPositive = if (n > 0) {
        true
    } else {
        false
    }
    println(isPositive)
}
```

Recordemos que si el tipo de retorno es Unit no es necesario especificarlo.

1.4.3. Implementación iterativa

Para la implementación iterativa utilizaremos la expresión *for*. La idea es mantener dos variables, `minusOne` y `minusTwo`, que representan los dos últimos números de la sucesión de Fibonacci, y una variable `result` que representa el número actual.

La implementación es la siguiente:

```
fun iterativeFibonacci(n: Int): Int {
    var minusTwo = 0 // f(0)
    var minusOne = 1 // f(1)
    var result = 0 // f(n)
    for (i in 1..n) {
        result = minusTwo + minusOne
        minusTwo = minusOne
        minusOne = result
    }
    return result
}
```

Veamos lo que está pasando:

- La función *iterativeFibonacci* recibe un número entero y devuelve un número entero.
- La función inicializa las variables `minusOne`, `minusTwo` y `result` con los valores correspondientes a los números de Fibonacci 0, 1 y 0.
- La función itera desde 1 hasta el número recibido. El operador `..`, llamado `rangeTo`, crea un rango de números que va desde el número de la izquierda hasta el número de la derecha, incluyendo ambos (i.e. `1..5` es el rango $[1, 5] = \{1, 2, 3, 4, 5\}$).

- En cada iteración la función actualiza las variables `minusOne`, `minusTwo` y `result` con los valores correspondientes a los números de Fibonacci de la iteración actual.
- Al finalizar la función devuelve el valor de la variable `result`.

1.4.4. Programación dinámica

La programación dinámica es una técnica de programación que consiste en guardar los resultados intermedios de una función para evitar calcularlos nuevamente. En el caso de la función de Fibonacci, sabemos que para calcular el valor de $f(6)$ necesitamos calcular el valor de $f(5)$ y $f(4)$. Si guardamos el valor de $f(5)$ podemos calcular el valor de $f(6)$ sin tener que calcular nuevamente el valor de $f(5)$. De esta manera, podemos calcular el valor de $f(6)$ sin tener que calcular el valor de $f(5)$ ni el valor de $f(4)$.

Para implementar la programación dinámica vamos a utilizar un arreglo de enteros, este es un nuevo tipo de dato de *Kotlin*. Un arreglo es una colección de elementos del mismo tipo, podemos crear un arreglo de enteros de la siguiente manera:

```
val array = IntArray(10)
```

En este caso creamos un arreglo de 10 elementos de tipo entero.

Ahora volvemos al tema de **mutabilidad**.

Mutabilidad

Un objeto es mutable si su estado puede cambiar después de ser creado, en contraste, un objeto es inmutable si su estado no puede cambiar después de ser creado.

Un arreglo es un tipo de dato mutable, es decir, podemos cambiar la información que contiene.

Para acceder a un elemento del arreglo utilizamos el operador `[]`, por ejemplo:

```
val array = IntArray(3)
array[0] = 1 // El primer elemento del arreglo es el elemento 0
array[1] = 2
array[2] = 3 // El último elemento del arreglo es el elemento 2
```

Si intentamos acceder a un elemento que no existe, por ejemplo `array[3]`, obtendremos un error.

Con esto podemos implementar la función de Fibonacci usando programación dinámica:

```
fun dynamicFibonacci(n: Int): Int {
    val memo = IntArray(n + 1)
    memo[0] = 0
    memo[1] = 1
    for (i in 2..n) {
        memo[i] = memo[i - 1] + memo[i - 2]
    }
    return memo[n]
}
```

Con esto ya estamos listxs para enfrentar al mundo.

1.5. Ejercicios

Ejercicio 1 Identificando tipos

1. Para cada una de las siguientes expresiones, indique el tipo de la variable x:

- a) `var x = true`
- b) `var x = 1`
- c) `var x = "1.0"`
- d) `var x = 1.0`
- e) `var x = Unit`
- f) `var x = IntArray(10)[0]`
- g) `var x = "${IntArray(10)}"`

2. Investigue qué son los siguientes tipos y de un ejemplo de uso de cada uno:

- a) *Float*
- b) *Long*
- c) *Char*
- d) *Byte*
- e) *Short*
- f) *Long*

Utilice la terminal interactiva de *Kotlin* (`kotlinc`) para verificar sus respuestas.

3. Utilizando la notación $FT(A_1, \dots, A_n) \rightarrow R$ indique el tipo de las siguientes funciones, reemplazando `_` por el tipo correspondiente:

- a) `fun f1(x: Int, y: Int): _ {
 return x + y
}`
- b) `fun f2(x: Double): _ {
 println(x)
}`
- c) `fun f3(x: Double, y: Double): _ {
 return println(x + y)
}`
- d) `fun f4(x: Int, y: Int): _ {
 return x + y
 println(x + y)
}`

4. Indique dos tipos inmutables y uno mutable.

Ejercicio 2 ¿Qué imprime?

Indique si las siguientes instrucciones arrojan un error o no, y si no lo hacen, indique qué imprime cada una al ser ejecutadas en la terminal interactiva.

1.

```
val x = 1
val y = 2
val z = 3
println(x + y + z)
```
2.

```
val x = 0
while (x < 10) {
    println(x)
    x = x + 1
}
```
3.

```
val x = 0
for (i in x..10) {
    println(i)
}
```
4.

```
fun jojoReference(name: String): Unit {
    return Unit
    println("My name is " + name + ", and I have a dream.")
}
```
5.

```
fun f(x: Int): Double {
    return x
}
```
6.

```
val a = for (i in 0..10) {
    i
}
```

Ejercicio 3 Fibonacci

1. Simplifique la implementación de la función recursiva de Fibonacci usando if-else como expresión. Hint: utilice una expresión de la forma `return if (condicion) {expresion1} else {expresion2}`.
2. ¿Qué sucede si se llama a la función de Fibonacci con un número negativo? ¿Cómo se puede solucionar este problema? Discuta, no debe implementar la solución.

Bibliografía

754-2019 - IEEE Standard for Floating-Point Arithmetic | IEEE Standard | IEEE Xplore 7542019IEEEStandard

754-2019 - IEEE Standard for Floating-Point Arithmetic | IEEE Standard | IEEE Xplore. URL: <https://ieeexplore.ieee.org/document/8766229> (visitado 08-02-2023).

baeldung: Statically Typed Vs Dynamically Typed Languages | Baeldung on Computer Science baeldungStaticallyTypedVs2021

baeldung. *Statically Typed Vs Dynamically Typed Languages* | Baeldung on Computer Science. 29 de oct. de 2021. URL: <https://www.baeldung.com/cs/statically-vs-dynamically-typed-languages> (visitado 08-02-2023).

Built-in Types and Their Semantics - Kotlin Language Specification BuiltInTypesTheira

Built-in Types and Their Semantics - Kotlin Language Specification. URL: <https://kotlinlang.org/spec/built-in-types-and-their-semantics.html#built-in-integer-types-builtins> (visitado 08-02-2023).

Chocolatey Software, Inc.: Chocolatey - The Package Manager for Windows chocolateysoftwareinc.ChocolateyPackageManager

Chocolatey Software, Inc. *Chocolatey - The Package Manager for Windows*. Chocolatey Software. URL: <https://chocolatey.org/> (visitado 26-04-2022).

Dev.Java: The Destination for Java Developers DevJavaDestination

Dev.Java: The Destination for Java Developers. URL: <https://dev.java/> (visitado 07-02-2023).

Home - SDKMAN! The Software Development Kit Manager HomeSDKMANSoftware

Home - SDKMAN! The Software Development Kit Manager. SDKMAN! URL: <https://sdkman.io/> (visitado 08-02-2023).

Homebrew Homebrew

Homebrew. Homebrew. URL: <https://brew.sh/> (visitado 08-02-2023).

IntArray - Kotlin Programming Language IntArrayKotlinProgramming

IntArray - Kotlin Programming Language. Kotlin. URL: <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-int-array/index.html> (visitado 08-02-2023).

Java (Programming Language) JavaProgrammingLanguage2023

Java (Programming Language). En: Wikipedia. 10 de ene. de 2023. URL: [https://en.wikipedia.org/w/index.php?title=Java_\(programming_language\)&oldid=1132820859](https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=1132820859) (visitado 07-02-2023).

Anot.: Page Version ID: 1132820859.

KevinLaMS: Use the Winget Tool to Install and Manage Applications**kevinlamsUseWingetTool2023**

KevinLaMS. *Use the Winget Tool to Install and Manage Applications*. 3 de feb. de 2023. URL: <https://learn.microsoft.com/en-us/windows/package-manager/winget/> (visitado 07-02-2023).

Reading 9: Mutability & Immutability**ReadingMutabilityImmutabilitya**

Reading 9: Mutability & Immutability. URL: <https://web.mit.edu/6.005/www/fa15/classes/09-immutability/> (visitado 08-02-2023).

sdwheeler: Set-ExecutionPolicy (Microsoft.PowerShell.Security) - PowerShell**sdwheelerSetExecutionPolicyMicrosoftPowerShell**

sdwheeler. *Set-ExecutionPolicy (Microsoft.PowerShell.Security) - PowerShell*. URL: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy> (visitado 07-02-2023).

Strong and Weak Typing**StrongWeakTyping2023**

Strong and Weak Typing. En: Wikipedia. 4 de feb. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Strong_and_weak_typing&oldid=1137354135 (visitado 08-02-2023).

Anot.: Page Version ID: 1137354135.

The Most Popular Programming Languages - 1965/2022 - New Update -**MostPopularProgramming**

The Most Popular Programming Languages - 1965/2022 - New Update -. URL: <https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2022-new-update/> (visitado 07-02-2023).

Type Inference - Kotlin Language Specification**TypeInferenceKotlin**

Type Inference - Kotlin Language Specification. URL: <https://kotlinlang.org/spec/type-inference.html#type-inference> (visitado 08-02-2023).

Type System - Kotlin Language Specification**TypeSystemKotlin**

Type System - Kotlin Language Specification. URL: <https://kotlinlang.org/spec/type-system.html#flexible-types> (visitado 08-02-2023).

What Is the lowerCamelCase Naming Convention?**WhatLowerCamelCaseNaming**

What Is the lowerCamelCase Naming Convention? WhatIs.com. URL: <https://www.techtarget.com/whatis/definition/lowerCamelCase> (visitado 08-02-2023).

Capítulo 2

IntelliJ, tu nuevo *bff*

Hasta ahora es probable que tengan experiencia utilizando algún editor de texto como *Notepad++*, *Sublime* o *Visual Studio Code*, eso es bueno, pero no suficiente para enfrentar problemas complejos (esto no es por desmerecer a esos editores de texto, también son herramientas poderosas apropiadas para otros contextos). Es aquí cuando surge la necesidad de tener un *entorno de desarrollo integrado* (IDE), un IDE es como un editor de texto cualquiera, pero poderoso. *IntelliJ* es un IDE desarrollado por *JetBrains* especializado para desarrollar programas en *Java*, *Kotlin* y *Scala*, y es la herramienta que usaremos de aquí en adelante.

Pero VSCode nunca me ha fallado ¿Por qué no puedo usarlo también para esto si le puedo instalar extensiones para programar en Kotlin?

La razón es simple, pero muy importante, como dije un poco más arriba, *IntelliJ* es una herramienta diseñada específicamente para desarrollar en *Kotlin*, esto hace que sea muchísimo más completo que otros editores de texto y que tenga facilidades para correr, configurar y testear¹ nuestros proyectos. De nuevo, no me malinterpreten, *VSCode* es una herramienta sumamente completa y perfectamente capaz de utilizarse en el mundo profesional, pero es bueno que conozcan otras alternativas también. Es posible utilizar cualquier otra herramienta que permita hacer lo mismo que *IntelliJ* para seguir este libro, pero será responsabilidad del lector/a/e adaptar lo visto aquí para funcionar con dichas herramientas.²

2.1. No termina de convencerme, pero le voy a dar una oportunidad. ¿Cómo lo instalo?

¡Esa es la actitud!

Si estas leyendo este libro y eres estudiante, tengo excelentes noticias para ti. Si no eres estudiante, tengo no tan excelentes noticias para ti.

IntelliJ viene en dos sabores, sabor *Community* y sabor *Ultimate*. Si eres un ser humano común y corriente (o un gato con acceso a internet) puedes descargar e instalar *IntelliJ IDEA Community Edition* desde el sitio oficial de *JetBrains*. ¿Muy complicado? Totalmente de acuerdo. ¿Por qué descargar el IDE desde la página web cuando *JetBrains* nos entrega una herramienta para facilitarnos el proceso?

¹ Esto será sumamente importante en capítulos futuros

² Dicho esto, este libro es abierto a la colaboración y cualquier aporte que contribuya al aprendizaje de los lectores es bienvenido. En caso de querer colaborar pueden hacerlo mediante un *Pull Request* al repositorio de *GitHub* de este libro (<https://github.com/islatern/software-design-book-es>), respetando las normas establecidas en el *Código de conducta*.

2.1.1. JetBrains Toolbox

JetBrains Toolbox es una herramienta que facilita la instalación de los productos de *JetBrains*, manejar distintos proyectos y también hace más fácil actualizar las herramientas.

Instalar *Toolbox* se puede hacer descargando la herramienta desde el [sitio oficial](#).

Alternativamente, también podemos instalarla desde la terminal.

Windows

```
winget install JetBrains.Toolbox
```

Linux/Mac

```
BASE_URL="https://data.services.jetbrains.com/products/download"
PLATFORM="?platform=linux&code=TBA"

wget --show-progress -qO ./toolbox.tar.gz $BASE_URL$PLATFORM

TOOLBOX_TEMP_DIR=$(mktemp -d)
tar -C "$TOOLBOX_TEMP_DIR" -xf toolbox.tar.gz
rm ./toolbox.tar.gz
"$TOOLBOX_TEMP_DIR"/*/jetbrains-toolbox

rm -r "$TOOLBOX_TEMP_DIR"
```

Nota. *JetBrains* provee de licencias para estudiantes de manera gratuita, para esto, lo único que deben hacer es registrarse en el [sitio oficial](#) con su correo institucional (por ejemplo, para estudiantes de la Universidad de Chile, puede ser un correo [@ug.uchile.cl](mailto:ug.uchile.cl)) y luego solicitar la licencia. Con esto podrán acceder a todas las herramientas de *JetBrains* y no tendrán que preocuparse por la licencia.

Para quienes no son estudiantes, *JetBrains IntelliJ IDEA Community Edition* es completamente gratis, pero no podrán acceder a todas las herramientas que provee *Ultimate*. Lo que suena malo, pero ninguna de las herramientas que no están disponibles en la versión *Community* son necesarias para el desarrollo de este libro. A partir de este momento, cuando se mencione *IntelliJ*, se estará haciendo referencia a cualquiera de las dos versiones.

2.1.2. IntelliJ IDEA

En la figura 2.1 pueden ver la interfaz de *JetBrains Toolbox*, ahí basta con buscar la versión de *IntelliJ* que quieran instalar y hacer click en el botón *Install*.

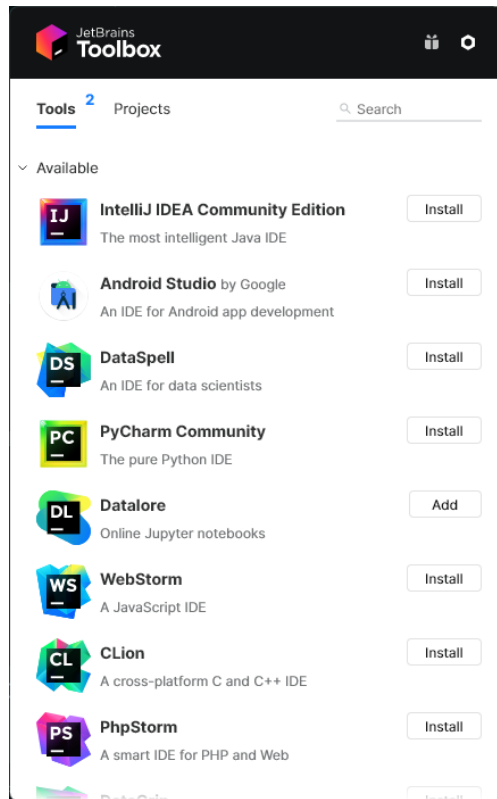


Figura 2.1: Interfaz de *JetBrains Toolbox*.

2.2. Conociendo *IntelliJ*

2.2.1. Creando un proyecto

El primer paso para utilizar *IntelliJ* es abrirlo

amazing

esto puede tomar algunos minutos dado que el *IDE* ocupa muchos recursos.

Una vez abierto debieran ver una ventana similar a la de la figura 2.2. Aquí deberán seleccionar la opción «*New Project*».

Luego deberíamos estar en la ventana principal de creación de proyectos (figura 2.3), aquí tendremos muchas opciones disponibles pero en este libro las ignoraremos y sólo le pondremos atención a proyectos de *Kotlin*. En esta ventana, primero seleccionaremos la opción *New Project* (1), ahí tendremos la opción de nombrar nuestro proyecto (2), pongámosle «*intellij-basics*». Luego, colocamos la ubicación donde queremos que se cree el proyecto (3), en este caso se guardará en *E:\Code\Teaching* (donde se creará una sub-carpeta con el nombre del proyecto). Por ahora, desmarquemos la opción *Create Git repository* (4) ya que no vamos a utilizarlo todavía. Lo siguiente es seleccionar el lenguaje y *build system* que vamos a utilizar (5), en este caso seleccionaremos *Kotlin e IntelliJ* respectivamente, por ahora no nos vamos a detener en lo que es un *build system*. Finalmente, debemos seleccionar la distribución de *JDK* que vamos a utilizar (6), en este caso seleccionaremos la última versión de *JDK* disponible (19 en el ejemplo).

Con esto podemos darle *Create* al proyecto, esperamos unos segundos para que se cocine y listo, ya tenemos nuestro primer proyecto en *IntelliJ*.

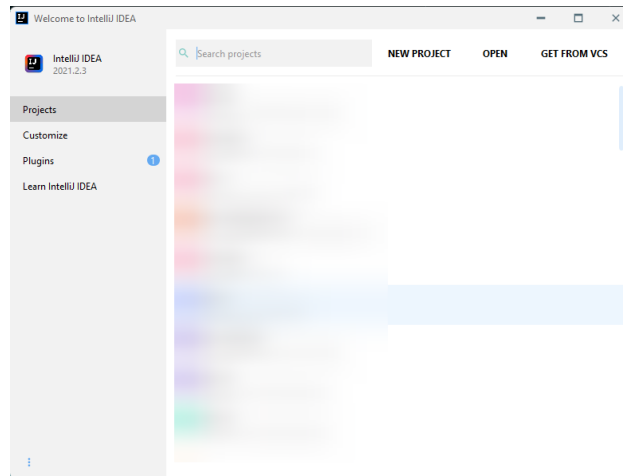


Figura 2.2: Ventana inicial de *IntelliJ*.

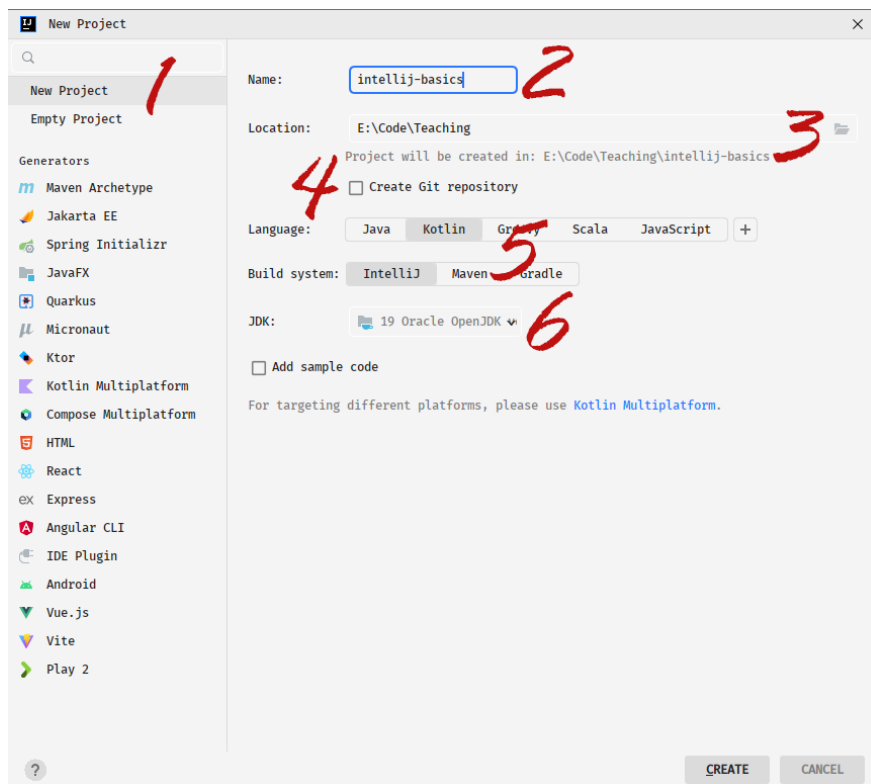


Figura 2.3: Ventana de creación de proyectos

2.2.2. Trabajando con *IntelliJ*

Hasta ahora hemos utilizado la terminal para programar en kotlin, esto es bastante útil pero *IntelliJ* nos ofrece muchas más herramientas para programar. La primera que veremos es la *consola REPL* (*Read-Eval-Print-Loop*), esto es similar a utilizar *Kotlin* en la terminal pero con muchas más herramientas.

Para acceder a la consola REPL, debemos ir a la sección «Tools» de la barra de herramientas, ir a la opción «Kotlin» y

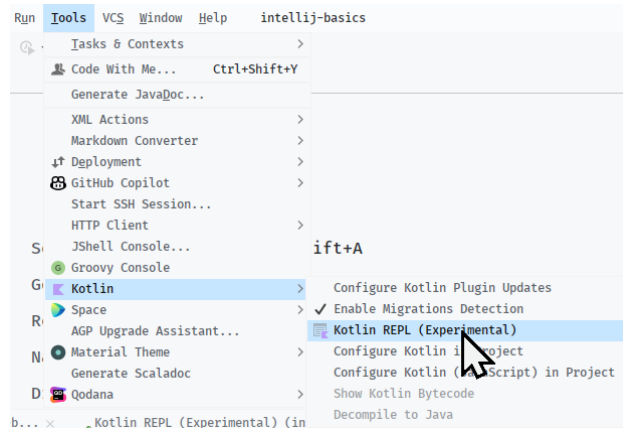


Figura 2.4: ubicación de la *consola REPL* en *IntelliJ*.

seleccionar la opción «*Kotlin REPL (Experimental)*» (figura 2.4).

Search Everywhere

Otra forma de acceder a la *consola REPL* es utilizando la herramienta *Search Everywhere* (figura 2.5), esta herramienta nos permite buscar casi todo en *IntelliJ* y nos permite acceder a herramientas y archivos de manera rápida. Para utilizarla, debemos presionar Shift + Shift.

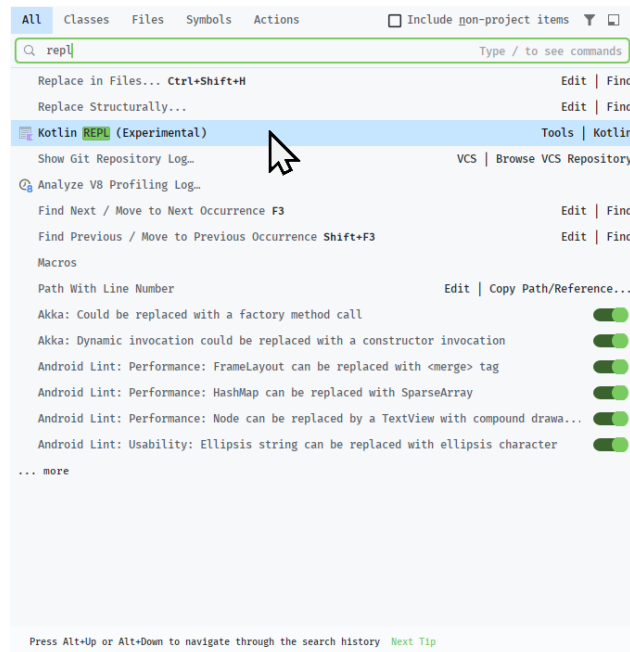


Figura 2.5: Resultado de *Search Everywhere* para la búsqueda de *repl*.

Ahora intenten reescribir el código del apartado 1.4 para ver cómo se siente trabajar con un *IDE* (reescribanlo, no lo copien y peguen). Luego pueden ejecutar el código y ver el resultado en la consola presionando el botón «*play*» en la esquina superior izquierda de la pestaña.

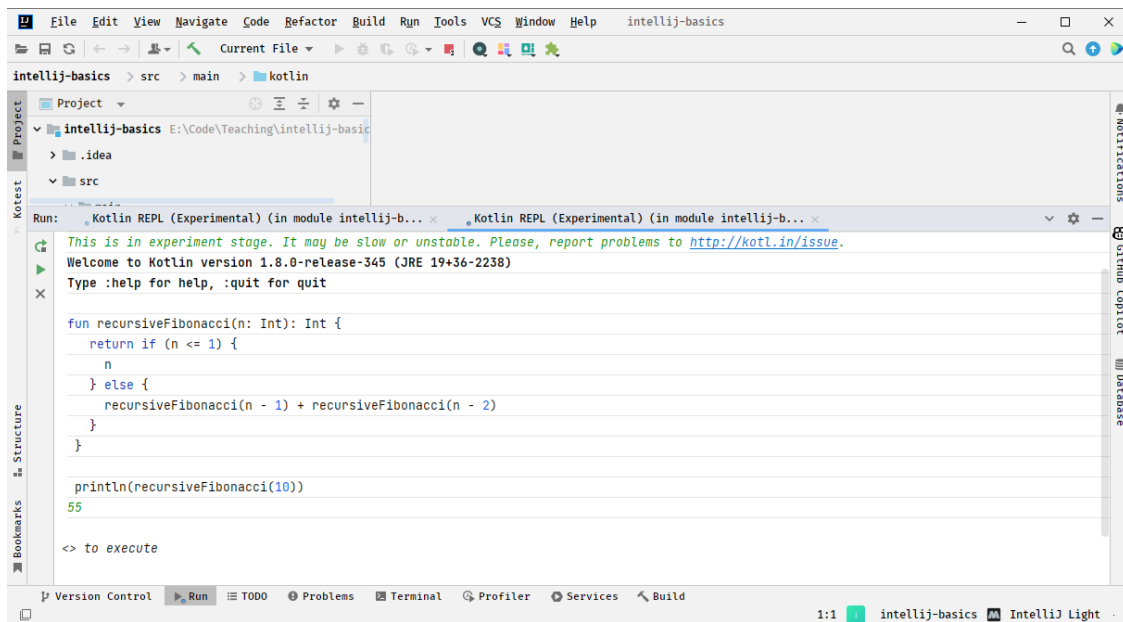


Figura 2.6: Ejemplo de uso de la *consola REPL* en *IntelliJ*.

2.3. Mi primera aplicación

Ahora veremos cómo crear una aplicación con *IntelliJ*, será algo muy simple, pero nos servirá como base para comenzar el segundo arco de este libro.

Lo primero que necesitamos es entender la estructura del proyecto. En la carpeta del proyecto debieran haber 3 elementos: el archivo `intellij-basics.iml`, y las carpetas `.idea` y `src`. Su utilidad es:

- `.idea`: contiene la configuración de *IntelliJ*.
- `src`: contiene el código fuente de nuestra aplicación.
- `intellij-basics.iml`: es el archivo de configuración del proyecto.

En la mayoría de los casos, no nos van a interesar mucho los archivos de configuración, así que interactuaremos solamente con el directorio `src`.

2.3.1. Paquetes

De la misma forma en que un computador se organiza en carpetas, un proyecto se organiza en paquetes.

El principal objetivo de los paquetes es darle organización a nuestro código, porque, al igual que tener una carpeta repleta de archivos (te estoy mirando a ti que tienes el escritorio tapizado en íconos), tener todo el código de nuestra aplicación en una carpeta (o en el mismo archivo `D:`) es barbarie.

La gracia de usar paquetes en vez de simplemente carpetas es que podemos incorporar la lógica de los paquetes en nuestro código. Además, si se utilizan correctamente podemos evitar gran parte de los problemas que podrían surgir al momento de interactuar con librerías externas.

Kotlin (al igual que *Java* y *Scala*) permiten organizar nuestro código en paquetes, mientras que otros lenguajes como *Python* y *C++* no.³ El estándar para nombrar paquetes es el mismo para *Java*, *Kotlin* y *Scala*:

³En lugar de paquetes *Python* provee módulos y *C++* tiene *namespaces*, ambas cumplen los mismos objetivos que los paquetes, pero se utilizan de forma distinta. Es importante informarse de esas diferencias cuando están aprendiendo un nuevo lenguaje.

- El nombre del paquete debe ser único (se recomienda usar el dominio de la empresa o persona que lo creó).
- El nombre del paquete debe ser escrito en minúsculas.
- El nombre del paquete no debe incluir guiones bajos (_).
- El nombre del paquete puede incluir guiones (-), pero no se recomienda.

Veamos algunos ejemplos:

```
package cl.ravenhill.intellij.basics; // Bien
package cl.ravenhill.intelliJBasics; // Mal, el nombre no debe incluir mayúsculas
package cl.ravenhill.intellij_basics; // Mal, el nombre no debe incluir guiones bajos
package cl.ravenhill.intellij-basics; // Permitido, pero no recomendado
```

Busquemos la carpeta `src/main/kotlin` y creemos un paquete llamado `cl.ravenhill.intellij.basics` haciendo click derecho sobre la carpeta `kotlin` y seleccionando `New -> Package`.

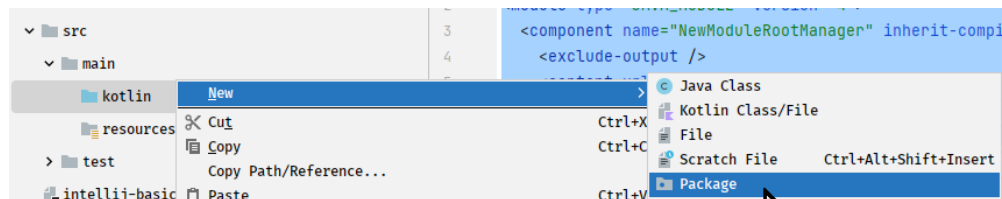


Figura 2.7: Creando un nuevo paquete

Es importante acostumbrarse a crear paquetes para organizar el código por dos razones:

- Siempre que se trabaje con librerías externas, es muy probable que se encuentren con paquetes que no son de su propiedad, y que no pueden modificar.
- Cuando una aplicación crece, los paquetes nos permiten organizar el código de forma lógica.

2.3.2. Mi primer *Kotlin*

Ahora que ya sabemos cómo crear paquetes, vamos a crear nuestro primer archivo *Kotlin*. Para ello, haremos click derecho sobre el paquete que acabamos de crear y seleccionaremos `New -> Kotlin File/Class` (figura 2.8), y en el dialogo que aparece crearemos un archivo `InteractiveFibonacciCalculator` (figura 2.9).

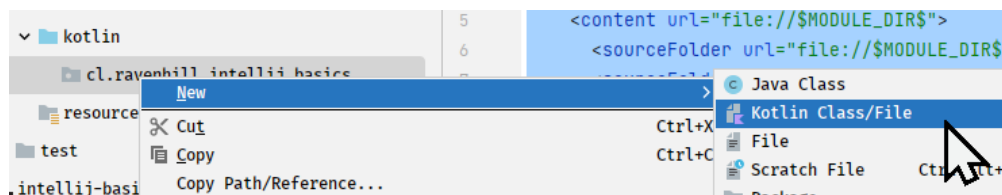


Figura 2.8: Creando un nuevo archivo *Kotlin*

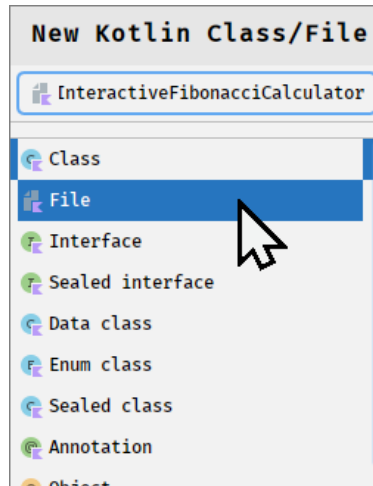


Figura 2.9: Creando un nuevo archivo *Kotlin*

Esto creará un archivo `InteractiveFibonacciCalculator.kt` en la carpeta `src/main/kotlin/cl/ravenhill/intellij/basics`. Noten que el nombre del archivo utiliza *PascalCase*.⁴

Ahora vamos a escribir el código que calcula la sucesión de Fibonacci de forma interactiva. Lo primero que debemos conocer aquí es la función `main`. Esta función es el punto de entrada, es decir, es la función que se ejecuta cuando se corre el programa.

Existe más de una forma de escribir la función `main` en *Kotlin*, pero utilizaremos la más simple, que es definir una función `main(): Unit`. Podemos partir con algo como:

```
package cl.ravenhill.intellij.basics

fun main(): Unit {
    println("Welcome to the Fibonacci Calculator!")
}
```

¡Y listo! Tenemos nuestra primera aplicación en *Kotlin*. Para ejecutarla, buscaremos el botón *Run* ubicado al lado izquierdo de la declaración de la función `main` (figura 2.10), le daremos click y, en la pestaña que se abre, seleccionaremos *Run 'InteractiveFibonacci...'* (las otras opciones las ignoraremos por ahora).

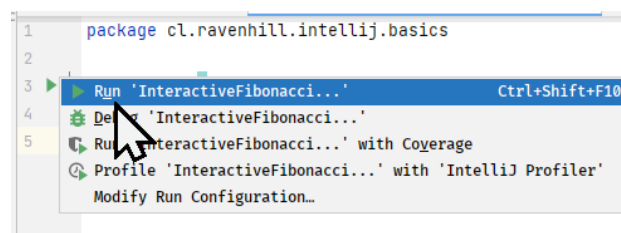


Figura 2.10: Ejecutando la aplicación

⁴What Is Pascal Case?

2.3.3. Loop principal

Para esta parte, reutilizaremos el código del apartado 1.4.

```
fun main() {
    println("Welcome to the Fibonacci Calculator!")
    println("Please enter a number to calculate the Fibonacci sequence for or 'q' to
    ↪ quit")
    while (true) {
        val input = readln()
        if (input == "q") {
            println("Goodbye!")
            break
        }
        val n = input.toInt()
        println("The Fibonacci number for $n is ${recursiveFibonacci(n)}")
    }
}
```

Veamos qué hace este código:

- Imprime un mensaje de bienvenida.
- Imprime un mensaje pidiendo un número.
- Inicia un loop infinito.
- Lee la entrada del usuario con la función `readln(): String`.
- Si la entrada es `q`, imprime un mensaje de despedida y sale del loop con la palabra reservada `break`.
- Si la entrada no es `q`, convierte la entrada a un número y calcula el número de Fibonacci para ese número.
- Imprime el resultado.

¿Y si ahora queremos que el usuario pueda elegir entre calcular el número de Fibonacci de forma recursiva o iterativa? Para esto podemos utilizar una expresión *if-else*, pero una forma más elegante de hacerlo es utilizando la expresión *when*.

```
fun main() {
    println("Welcome to the Fibonacci Calculator!")
    println("""
    |Usage:
    | - Enter a number to calculate the Fibonacci number
    | - Enter 'r' to use the recursive algorithm
    | - Enter 'i' to use the iterative algorithm
    | - Enter 'q' to quit
    """).trimMargin()
    while (true) {
        when(readln()) {
            "q" -> {
                println("Goodbye!")
                break
            }
        }
    }
}
```

```

    "r" -> {
        println("Using recursive algorithm")
        println("Fibonacci number: ${recursiveFibonacci(readln().toInt())}")
    }
    "i" -> {
        println("Using iterative algorithm")
        println("Fibonacci number: ${iterativeFibonacci(readln().toInt())}")
    }
    else -> {
        println("Invalid input")
    }
}
}
}
}

```

Veamos las partes esenciales de este código:

- Primero notemos que el segundo print está escrito con triple comilla doble, esto se llama *raw string literal* y nos permite escribir strings con saltos de línea y tabulaciones sin tener que escaparlos. La función `trimMargin()` elimina los espacios en blanco al inicio de cada línea. Para entender mejor lo que hace la función `trimMargin()`, podemos revisar la documentación desde el IDE, para esto simplemente colocamos el cursor sobre la función como en la figura 2.11.
- La expresión *when* es una expresión que funciona como un *if-else* pero con múltiples condiciones. En este caso, si la entrada es `q`, se imprime un mensaje de despedida y se sale del loop. Si la entrada es `r`, se imprime un mensaje indicando que se está usando el algoritmo recursivo, se lee un número y se calcula el número de Fibonacci para ese número. Si la entrada es `i`, se imprime un mensaje indicando que se está usando el algoritmo iterativo, se lee un número y se calcula el número de Fibonacci para ese número. Si la entrada no es ninguna de las anteriores, se imprime un mensaje indicando que la entrada es inválida.

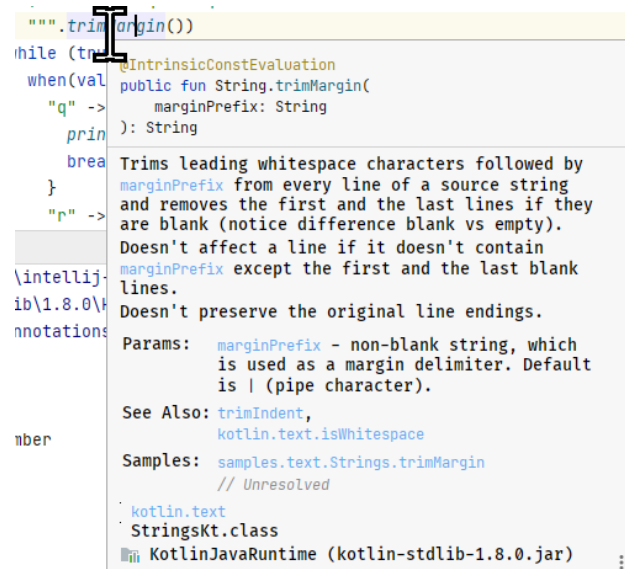


Figura 2.11: Documentación de la función `trimMargin()`.

Con esto termina la parte introductoria de este libro, en la siguiente parte veremos cómo utilizar las herramientas de

desarrollo que nos ofrece IntelliJ IDEA para crear aplicaciones más complejas y robustas, tratando de sacarle el máximo provecho a Kotlin e IntelliJ IDEA.

Pero antes, algunos ejercicios para practicar lo que hemos visto hasta ahora :)

2.4. Ejercicios

Ejercicio 4 Potencias

1. Implemente una función `iterativePow(x: Int, n: Int): Int` que calcule la potencia de un número de forma iterativa. *Hint: Utilice un ciclo for que vaya desde 1 hasta la potencia.*
2. Implemente una función `recursivePow(x: Int, n: Int): Int` que calcule la potencia de un número de forma recursiva.
3. La n -ésima potencia de un número x se puede calcular utilizando la estrategia divide y vencerás de la siguiente manera:

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x^{n/2} \cdot x^{n/2} & \text{si } n \text{ es par} \\ x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x & \text{si } n \text{ es impar} \end{cases}$$

Implemente una función recursiva `divideAndConquerPow(x: Int, n: Int): Int` que calcule la n -ésima potencia de un número x utilizando esta estrategia. *Hint: Para saber si un número es par o impar podemos utilizar el operador módulo (%) que nos da el resto de la división de dos números, entonces si $n \% 2$ es igual a 0 n es par.*

4. Tenemos que:

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x^{n/2} \cdot x^{n/2} & \text{si } n \text{ es par} \\ x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x & \text{si } n \text{ es impar} \end{cases}$$

Con esto, podemos implementar una función recursiva `fastPow(x: Int, n: Int): Int` que calcule la n -ésima potencia de un número x utilizando esta estrategia con la siguiente especificación:

$$\text{fastPow}(x, n) = \begin{cases} \text{fastPow}(\frac{1}{x}, -n) & \text{si } n < 0 \\ 1 & \text{si } n = 0 \\ \text{fastPow}(x \cdot x, \frac{n}{2}) & \text{si } n \text{ es par} \\ x \cdot \text{fastPow}(x \cdot x, \frac{n-1}{2}) & \text{si } n \text{ es impar} \end{cases}$$

Implemente esta función.

5. Programe una interfaz interactiva que permita calcular la potencia de un número de forma iterativa, recursiva, utilizando la estrategia divide y vencerás y utilizando la estrategia de potencias rápidas.

Ejercicio 5 Números binarios

Un número binario es un número que está escrito en base 2. Por ejemplo, el número binario 101 es igual a $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$.

1. Implemente una función `binaryToDecimalFor(binary: IntArray): Int` que reciba un arreglo de enteros que representan un número binario y calcule su equivalente en base 10 utilizando un ciclo for.
2. Implemente una función `binaryToDecimalWhile(binary: IntArray): Int` que reciba un arreglo de enteros que representan un número binario y calcule su equivalente en base 10 utilizando un ciclo while.
3. Implemente una función `binaryToDecimalRecursive(binary: IntArray): Int` que reciba un arreglo de enteros que representan un número binario y calcule su equivalente en base 10 utilizando recursión. Para esto, considere lo siguiente:

$$\text{binToDec}(\text{binary}) = \begin{cases} 0 & \text{si } \text{binary} \text{ es vacío} \\ \text{binary}[n-1] \cdot 2^{n-1} + \text{binToDec}(\text{binary}[0..n-2]) & \text{sino} \end{cases}$$

Hint: Para obtener el subarreglo `binary[0..n - 2]` puede utilizar la función `binary.sliceArray(0 until n - 1)`⁵.

4. Implemente una función `decimalToBinary(decimal: Int): IntArray` que reciba un número entero y calcule su equivalente en base 2. Esto se puede hacer de la siguiente forma:

1. Divida el número entre 2 y guarde el cociente y el resto.
2. Repita el paso anterior hasta que el cociente sea 0.
3. Guarde los restos en un arreglo de enteros.
4. Invierta el arreglo de enteros.

Hints:

- Utilice un ciclo `while` para resolver el problema.
- Para invertir un arreglo puede utilizar la función `array.reversedArray()`.

5. Implemente una función `decimalToBinaryRecursive(decimal: Int): IntArray` que reciba un número entero y calcule su equivalente en base 2 utilizando recursión. *Hint: Utilice una función auxiliar que reciba el número y el arreglo de enteros.*
6. El equivalente en base 2 de un número entero se puede calcular utilizando la estrategia divide y vencerás de la siguiente manera:

$$decToBin(x) = \begin{cases} 0 & \text{si } x = 0 \\ decToBin(x/2) \cdot 10 + x \text{ mód } 2 & \text{si } x > 0 \end{cases}$$

Implemente una función `decimalToBinaryDivideAndConquer(decimal: Int): IntArray` que reciba un número entero y calcule su equivalente en base 2 utilizando esta estrategia.

7. El equivalente en base 2 de un número entero *i* se puede calcular sin utilizar un arreglo de la siguiente forma:

1. Cree 2 variables *power* y *binary* que inicialmente valgan 0.
2. Mientras *i* sea mayor que 0:
 - a) Guarde el resultado de *i* módulo 2 en una variable *rem*.
 - b) Calcule $10^{power} \cdot rem$ y sume el resultado a *binary*.
 - c) Incremente *power* en 1.
 - d) Divida *i* entre 2 y guarde el resultado en *i*.
3. Retorne *binary*.

Implemente una función `decimalToBinaryWithoutArray(decimal: Int): Int` que reciba un número entero y calcule su equivalente en base 2 utilizando esta estrategia.

Puede utilizar el siguiente código para probar sus funciones:

```
fun main() {  
    // Output: 1024  
    println(iterativePow(2, 10))  
    println(recursivePow(2, 10))  
    println(divideAndConquerPow(2, 10))  
}
```

⁵until es una función (infija) que recibe un número n y retorna un rango que va desde 0 hasta n - 1

```
println(fastPow(2, 10))  
}
```

```
fun main() {  
    val binary = intArrayOf(1, 0, 1, 1, 0, 1, 1, 1)  
    // Output: 123  
    println(binaryToDecimalFor(binary))  
    println(binaryToDecimalWhile(binary))  
    println(binaryToDecimalRecursive(binary))  
    // Output: [1, 1, 1, 1, 0, 1, 0, 1]  
    println(decimalToBinary(123).contentToString())  
    println(decimalToBinaryRecursive(123).contentToString())  
    println(decimalToBinaryDivideAndConquer(123).contentToString())  
    // Output: 1110101  
    println(decimalToBinaryWithoutArray(123))  
}
```

Bibliografía

GitHub Flow

GitHubFlow

GitHub Flow. GitHub Docs. URL: <https://ghdocs-prod.azurewebsites.net/en/get-started/quickstart/github-flow> (visitado 12-05-2022).

Integrated Development Environment

IntegratedDevelopmentEnvironment2023

Integrated Development Environment. En: *Wikipedia*. 27 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Integrated_development_environment&oldid=1135934172 (visitado 08-02-2023).

Anot.: Page Version ID: 1135934172.

IntelliJ IDEA

IntelliJIDEACapable

IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains. JetBrains. URL: <https://www.jetbrains.com/idea/> (visitado 19-05-2022).

JetBrains Toolbox App

JetBrainsToolboxApp

JetBrains Toolbox App: Manage Your Tools with Ease. JetBrains. URL: <https://www.jetbrains.com/toolbox-app/> (visitado 19-05-2022).

Searching Everywhere | IntelliJ IDEA

SearchingEverywhereIntelliJ

Searching Everywhere | IntelliJ IDEA. IntelliJ IDEA Help. URL: <https://www.jetbrains.com/help/idea/searching-everywhere.html> (visitado 08-02-2023).

What Is Pascal Case?

WhatPascalCase

What Is Pascal Case? TheServerSide.com. URL: <https://www.theserverside.com/definition/Pascal-case> (visitado 08-02-2023).

Parte II

OOP no es religión

Capítulo 3

Recuerdo de mi primer objeto

Hasta este punto hemos visto cómo crear un proyecto en *IntelliJ* y como diseñar programas simples. Pero en el mundo real no van a trabajar en proyectos simples.

En este capítulo veremos cómo comenzar a trabajar en proyectos más complejos y cómo utilizar herramientas que nos harán mantener¹ la cordura.

3.1. Sistemas de control de versiones

Para las siguientes secciones utilizaremos como ejemplos un juego de cartas inspirado en *Yu-Gi-Oh!* desarrollando y publicado por *Konami*. Lo llamaremos *Kygo*.

Comencemos por crear un nuevo proyecto siguiendo los mismos pasos del capítulo anterior (con la opción de *Git* desmarcada) en *IntelliJ* y llamémoslo *kygo*.

Algo que me ha pasado muchas veces y que probablemente les pasará a ustedes es que tienen un proyecto que está funcionando perfectamente, pero que necesitan agregarle una nueva funcionalidad. Trabajamos muy duro en la nueva funcionalidad y la terminamos, pero al momento de probarla nos damos cuenta de que la nueva funcionalidad rompió el código que ya teníamos. Esto es un problema muy común y que puede ser muy frustrante, ya que nos obliga a volver a escribir el código que ya teníamos funcionando. Sería ideal poder volver a un punto anterior en el tiempo y recuperar el código que ya teníamos funcionando...

Sistemas de control de versiones al rescate.

Sistema de control de versiones

Un *sistema de control de versiones* (VCS) es un programa que nos permite guardar el estado de nuestro proyecto en diferentes puntos del tiempo.

Esto tiene varios beneficios, pero el principal² es que podemos guardar el estado de nuestro proyecto en un punto del tiempo y luego volver a ese punto en el tiempo si es necesario.

Existen varios sistemas de control de versiones como *Git*, *Mercurial*, *Subversion*, *Perforce* y *CVS*. En este libro utilizaremos *Git* ya que es el sistema de control de versiones más popular y el que más se utiliza en la industria.

¹Y a veces perder.

²imho

3.1.1. Git

Git es un sistema de control de versiones distribuido, esto significa que la completitud del proyecto y su línea de tiempo se pueden compartir entre todos los miembros de un equipo. Esto significa que es una herramienta extremadamente útil para sincronizar el trabajo de un equipo.

3.1.2. Instalando Git

Windows

Para instalar *Git* en Windows, podemos hacerlo utilizando *winget*, para esto abran PS y ejecuten el siguiente comando:

```
winget install Git.Git
```

Por último comprobemos que la instalación fue exitosa ejecutando el siguiente comando:

```
git version
```

Linux (Debian)

Para instalar *Git* en Linux, podemos hacerlo utilizando el gestor de paquetes de su distribución, para esto abran una terminal y ejecuten lo siguiente:

```
sudo apt-get update  
sudo apt-get install git-all
```

Por último comprobemos que la instalación fue exitosa ejecutando el siguiente comando:

```
git version
```

MacOS

Para instalar *Git* en MacOS, podemos hacerlo utilizando *Homebrew*, para esto abran una terminal y ejecuten el siguiente comando:

```
brew install git
```

Por último comprobemos que la instalación fue exitosa ejecutando el siguiente comando:

```
git version
```

3.1.3. Configurando Git

Git necesita saber quién es el culpable de los cambios que se realizan en el proyecto. Para esto debemos configurar el nombre y el correo electrónico del autor. Para esto abran una terminal y ejecuten los siguientes comandos:


```
git config --global user.name "r8vnhill"  
git config --global user.email "reachme@ravenhill.cl"
```

Con esto ya podemos empezar a usar *Git*, a lo largo del capítulo y del libro iremos conociendo más sobre esta herramienta y cómo utilizarla.

3.2. Creando el proyecto

Primero debemos crear un nuevo proyecto en *IntelliJ*, esto lo podemos hacer de la misma forma que hicimos en la sección anterior.

Con el proyecto creado, podemos partir creando un paquete `cl.ravenhill.kygo` en el directorio `src/main/kotlin`.

Ahora debemos iniciar el repositorio de *Git* en el directorio del proyecto. Para esto primero hagamos click derecho en el directorio del proyecto y seleccionemos la opción *Copy Path/Reference...* (figura 3.1). Luego, en la ventana que se abrió, hagamos click en el botón *Absolute Path* (figura 3.2).

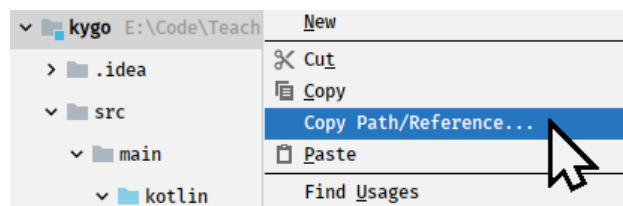


Figura 3.1: Copiando la ruta del proyecto

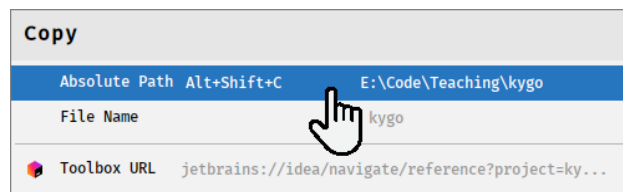


Figura 3.2: Copiando la ruta absoluta del proyecto

Ahora abramos una terminal y ejecutemos el siguiente comando:

Windows

```
Get-Clipboard | Set-Location  
git init
```

Linux

```
cd $(xclip -o -selection clipboard)
git init
```

MacOS

```
cd $(pbpaste)
git init
```

Veamos lo que está pasando en el comando anterior:

- Primero tomamos el contenido del portapapeles y lo usamos como argumento al comando `cd`³/`Set-Location`⁴ para cambiar el directorio actual a la ruta del proyecto.
- Luego ejecutamos el comando `git init`. El comando `git init` inicializa un repositorio de *Git* en el directorio actual, esto significa que *Git* comenzará a monitorear los cambios que se realicen en el directorio actual.

Nos queda verificar que el repositorio de *Git* se haya creado correctamente. Para esto ejecutemos el comando `git status` (nuestrx otrx mejor amigx), esto nos entregará un mensaje como el siguiente:

On branch master

No commits yet

Untracked files:

```
(use "git add <file>..." to include in what will be committed)
.idea/
kygo.iml
```

nothing added to commit but untracked files present (use "git add" to track)

Aquí lo que nos está diciendo *Git* es que existen archivos que no están siendo monitoreados y que, por lo tanto, no forman parte del repositorio.

Podríamos agregar estos archivos directamente al repositorio, pero en general no es una buena idea. Como dijimos, esos elementos son de configuración y si estamos trabajando con otras personas podría causar problemas si todos tienen configuraciones diferentes. Por esto nos gustaría poder no sólo no agregarlos, sino que asegurarnos de que nunca⁵ se agreguen.

Para esto debemos crear un archivo llamado `.gitignore` en el directorio raíz del proyecto. Este archivo contiene una lista de patrones que *Git* usará para ignorar archivos que coincidan con esos patrones. En nuestro caso, vamos a ignorar todos los archivos que estén en el directorio `.idea/` y el archivo `kygo.iml`. Para esto, abramos el archivo `.gitignore` y agreguemos las siguientes líneas:

```
.idea/
kygo.iml
```

³Change directory (*Cd Man Page - Linux - SS64.Com*)

⁴sdwheeler, *Set-Location (Microsoft.PowerShell.Management) - PowerShell*.

⁵Existen formas de agregarlos de todos modos, pero por qué quieres hacer eso D:

Ahora, si volvemos a ejecutar el comando `git status` veremos que el único archivo que *Git* nos está indicando que no está siendo monitoreado es el archivo `.gitignore`.

Digámosle ahora a *Git* que empiece a monitorear el archivo `.gitignore`. Para esto ejecutemos el comando `git add .gitignore`. Este comando le dice a *Git* que agregue el archivo `.gitignore` al repositorio. Ahora, si volvemos a ejecutar el comando `git status` veremos que no hay archivos que no estén siendo monitoreados, pero sí faltan archivos a los que hacer *commit*.

Un *commit* es una forma de decirle a *Git* que guarde los cambios que se han realizado en el repositorio. Para hacer un *commit* ejecutemos el comando `git commit -m "Adds .gitignore"`, donde `-m` indica el mensaje del *commit*. Es sumamente importante que los mensajes de los *commits* sean descriptivos, ya que estos nos ayudarán a entender qué cambios se realizaron en el repositorio.⁶

Ahora, si volvemos a ejecutar el comando `git status` veremos que no hay archivos que no estén siendo monitoreados y que tampoco hay archivos a los que hacer *commit*.

3.3. Objetos

Hasta ahora, gran parte de lo que hemos visto ha sido programación imperativa, es decir, programación basada en el concepto de *procedimientos* o *funciones*. Hemos implementado *algoritmos* que nos permiten resolver problemas como series de instrucciones que se ejecutan de acuerdo a una lógica. Sin embargo, en la vida real, los problemas que debemos resolver no son tan simples como una serie de instrucciones, sino que son más complejos y requieren de una mayor abstracción para poder ser resueltos.

En este capítulo veremos la programación orientada a objetos, un paradigma de programación que nos permite abstraer⁷ problemas complejos en objetos que interactúan entre sí.

Programación orientada a objetos

La programación orientada a objetos es un paradigma de computación que se organiza en base a objetos en vez de acciones y datos en vez de lógica. Aquí lo que realmente nos importa son los objetos que queremos manipular más que la lógica para manipularlos.

Un objeto es una *abstracción* que contiene información y maneras de manejar esta información. La información contenida dentro de un objeto no es visible desde afuera. Un objeto se compone de su estado (campos) y su comportamiento (métodos).

En *Kotlin* podemos crear objetos con la palabra reservada *object* seguida del nombre del objeto. Para crear un objeto desde *IntelliJ* primero crearemos un nuevo proyecto de la misma forma que lo hicimos en el capítulo anterior, nombrémoslo «oop-introduction». Con el proyecto creado crearemos un nuevo paquete llamado `cl.ravenhill.oop.essential`, hacemos click derecho sobre el paquete y seleccionamos la opción *New* → *Kotlin Class/File* → *Object* y lo llamaremos *Dice*. Esto nos creará un archivo llamado *Dice.kt* dentro del paquete que acabamos de crear con el siguiente contenido:

```
package cl.ravenhill.oop.essential

object Dice {
}
```

⁶Lo cuál será extremadamente útil para mantener una buena comunicación con los demás miembros del equipo, o incluso para nosotros mismos en el futuro. También nos ayudará a documentar los cambios que se realizan (por ejemplo) de la versión 1.0 a la 1.1.

⁷Abstraer es el proceso de eliminar detalles innecesarios de un objeto o sistema para enfocarnos en sus características esenciales (*Object-Oriented Programming*). Este será un concepto clave en todo lo que resta del libro.

Como podemos ver, el objeto *Dice* no tiene ningún campo ni método, por lo que no es muy útil. Para agregar campos y métodos a un objeto debemos agregarlos dentro de las llaves del objeto.

Agreguemos un campo llamado *sides* que será un número entero que representará la cantidad de caras del dado. Para agregar un campo a un objeto debemos agregar la palabra reservada *var* seguida del nombre del campo, podemos omitir el tipo de dato ya que *Kotlin* lo infiere automáticamente y es obvio por contexto. Digamos que nuestro dado tiene 6 caras, por lo que nuestro campo *sides* tendrá el valor por defecto de 6.

```
object Dice {  
    var sides = 6  
}
```

Ahora agreguemos un método llamado *roll* que nos permita tirar el dado.

```
/**  
 * Object that represents an n-sided dice.  
 *  
 * @property sides Number of sides of the dice.  
 */  
object Dice {  
    var sides = 6  
  
    /**  
     * Rolls the dice.  
     *  
     * @return A random number between 1 and the number of sides of the dice.  
     */  
    fun roll(): Int {  
        return (1..sides).random()  
    }  
}
```

Como podemos ver, el método *roll* no recibe ningún parámetro y retorna un número entero. Para generar un número aleatorio entre 1 y 6 podemos usar la función *random* de la clase *IntRange* que nos permite generar un número aleatorio entre un rango de números.

Noten las anotaciones *@property* y *@return* que agregamos al objeto y al método. Estas anotaciones nos permiten agregar documentación más detallada a nuestro código. Estas anotaciones son opcionales, pero es una buena práctica agregarlas para hablar un lenguaje común con otros desarrolladores.

Luego, podemos utilizar el objeto que creamos de la siguiente forma:

```
fun main() {  
    println("Dice roll: ${Dice.roll()}")  
    Dice.sides = 20  
    println("Dice roll: ${Dice.roll()}")  
}
```

Importante. Los objetos son estáticos, es decir, no podemos crear instancias de ellos, por lo que no podemos tener múltiples objetos del mismo tipo. Los objetos son útiles cuando queremos crear una única instancia de un objeto que se comparte por

toda la aplicación.

Por último, hacemos:

```
git add .
git commit -m "FEAT Adds Dice object"
```

3.4. Clases

Una clase es un modelo que define la estructura de un objeto. Podemos pensarla como una plantilla que podemos rellenar con datos para crear objetos. Una clase es una abstracción de un objeto, es decir, es una representación de un **tipo** de objeto. Por ejemplo, podemos definir una clase llamada *Player* que represente a un jugador de nuestro juego.

Para definir una clase debemos usar la palabra reservada *class* seguida del nombre de la clase. En *IntelliJ* debemos hacer click derecho sobre el paquete `cl.ravenhill.kygo` y seleccionar la opción *New -> Kotlin Class/File* y seleccionar la opción *Class*. Esto creará el siguiente archivo:

```
package cl.ravenhill.kygo

class Player {
}
```

Ahora podemos agregar campos a la clase:

```
class Player {
    var name: String = ""
    var health: Int = 0
}
```

Como podemos ver, los campos de una clase se definen de la misma forma que los campos de un objeto.

Ahora, podemos crear objetos a partir de la clase *Player*:

```
fun main() {
    val player1 = Player()
    player1.name = "Mugi"
    player1.health = 8000
    println(player1.name)
    println(player1.health)

    val player2 = Player()
    player2.name = "Jaiva"
    player2.health = 7000
    println(player2.name)
    println(player2.health)
}
```

3.4.1. Constructores primarios

Un constructor es un bloque de código (**no** una función) que se ejecuta al momento de crear un objeto a partir de una clase. En *Kotlin* podemos definir dos tipos de constructores: **primarios** y **secundarios**.

Por ahora veamos cómo definir un constructor primario, ya que podemos introducir los constructores secundarios más adelante cuando los necesitemos. Un constructor primario se define dentro de un bloque *init* (de nuevo, **no** es una función). Un constructor primario puede recibir parámetros, los cuales se definen en la declaración de la clase, con una sintaxis similar a la de los métodos. Esto se ve mejor con un ejemplo:

```
class Player(name: String, health: Int) {  
    val name: String  
    val health: Int  
  
    init {  
        this.name = name  
        this.health = health  
    }  
}
```

Vamos viendo lo que está pasando:

- Primero definimos los campos de la clase, los cuales son *name* y *health*.
- Luego definimos los campos de la clase.
- Dentro del bloque *init* asignamos los valores de los parámetros a los campos de la clase.

Pero hay dos cosas que probablemente les hagan ruido:

- ¿Qué son *this.name* y *this.health*?
- ¿Estamos asignando nuevos valores a variables de sólo lectura?

La segunda pregunta es fácil de responder: no, no estamos asignando nuevos valores a variables de sólo lectura. Como el constructor primario se ejecuta al momento de crear un objeto, los campos de la clase aún no se han inicializado, por lo que podemos asignarles un valor.

La primera pregunta es un poco más compleja de responder, porque necesitamos entender el concepto de *method-lookup* que veremos más adelante. Por ahora, basta con decir que *this.name* y *this.age* hacen referencia a los campos de la clase, y no a los parámetros del constructor.

Nota. Existen otras (mejores) formas de definir un constructor primario, pero por ahora nos limitaremos a ésta para no introducir tantos conceptos nuevos.

Ahora podemos repetir el ejemplo anterior, pero usando el constructor primario que acabamos de definir:

```
fun main() {  
    val player1 = Player("Mugi", 8000)  
    println(player1.name)  
    println(player1.health)  
  
    val player2 = Player("Jaiva", 7000)  
    println(player2.name)  
    println(player2.health)  
}
```

```
}
```

Por último:

```
git add .  
git commit -m "FEAT Creates Player class"
```

3.5. Ejercicios

Ejercicio 6 Calculadora

1. Cree un proyecto `calculator` en *IntelliJ* de la forma que se vió en este capítulo.
2. Inicie un repositorio *Git* en el proyecto.
3. Cree un archivo `.gitignore` en el directorio raíz del proyecto y agregue `.idea/` y `*.iml` a la lista de archivos ignorados.
4. Agregue el archivo `.gitignore` al repositorio.
5. Haga un *commit* con el mensaje `REPO Adds .gitignore`.
6. Cree un paquete `cl.ravenhill.calculator` en el proyecto.
7. Cree un objeto `Calculator` en el paquete `cl.ravenhill.calculator`.
8. Agregue un método `add` que reciba dos números enteros y retorne la suma de estos.
9. Agregue un método `subtract` que reciba dos números enteros y retorne la diferencia de estos.
10. Agregue un método `multiply` que reciba dos números enteros y retorne el producto de estos.
11. Agregue un método `divide` que reciba dos números enteros y retorne el cociente de estos.
12. Agregue un método `modulo` que reciba dos números enteros y retorne el resto de la división de estos.
13. ¿Por qué utilizamos un objeto en lugar de una clase para modelar la calculadora?
14. Agregue el archivo `Calculator.kt` al repositorio y haga un *commit* con el mensaje `FEAT Adds Calculator`.

Ejercicio 7

1. Cree un proyecto `cars` en *IntelliJ* de la forma que se vió en este capítulo.
2. Inicie un repositorio *Git* en el proyecto.
3. Cree un archivo `.gitignore` en el directorio raíz del proyecto y agregue `.idea/` y `*.iml` a la lista de archivos ignorados.
4. Agregue el archivo `.gitignore` al repositorio.
5. Haga un *commit* con el mensaje `REPO Adds .gitignore`.

6. Cree un paquete `cl.ravenhill.cars` en el proyecto.
7. Cree una clase `Car` en el paquete `cl.ravenhill.cars`.
8. Agregue los siguientes campos a la clase `Car`:
 - `brand` de tipo `String`.
 - `model` de tipo `String`.
 - `year` de tipo `Int`.
 - `color` de tipo `String`.
 - `price` de tipo `Double`.
9. Cree un constructor primario para la clase `Car` que reciba los campos `brand`, `model`, `year`, `color` y `price`.
10. Agregue un método `print()` a la clase `Car` que imprima en la consola los valores de los campos de la siguiente forma:

```
Car(brand=Toyota, model=Corolla, year=2018, color=White, price=1000000.0)
```
11. Agregue el archivo `Car.kt` al repositorio y haga un *commit* con el mensaje `FEAT Adds Car`.
12. ¿Por qué utilizamos una clase en lugar de un objeto para modelar un auto?

Bibliografía

astrand: Astrand/Xclip **astrandAstrandXclip2023**

astrand. *Astrand/Xclip*. 7 de feb. de 2023. URL: <https://github.com/astrand/xclip> (visitado 08-02-2023).

Atlassian: Git Add | Atlassian Git Tutorial **atlassianGitAddAtlassian**

Atlassian. *Git Add | Atlassian Git Tutorial*. Atlassian. URL: <https://www.atlassian.com/git/tutorials/saving-changes> (visitado 08-02-2023).

Atlassian: Git Commit | Atlassian Git Tutorial **atlassianGitCommitAtlassian**

Atlassian. *Git Commit | Atlassian Git Tutorial*. Atlassian. URL: <https://www.atlassian.com/git/tutorials/saving-changes/git-commit> (visitado 08-02-2023).

Atlassian: Git Init | Atlassian Git Tutorial **atlassianGitInitAtlassian**

Atlassian. *Git Init | Atlassian Git Tutorial*. Atlassian. URL: <https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init> (visitado 08-02-2023).

Cd Man Page - Linux - SS64.Com **CdManPage**

Cd Man Page - Linux - SS64.Com. URL: <https://ss64.com/bash/cd.html> (visitado 08-02-2023).

Classes | Kotlin **ClassesKotlin**

Classes | Kotlin. Kotlin Help. URL: <https://kotlinlang.org/docs/classes.html> (visitado 08-02-2023).

Create a New Project | IntelliJ IDEA **CreateNewProject**

Create a New Project | IntelliJ IDEA. IntelliJ IDEA Help. URL: <https://www.jetbrains.com/help/idea/new-project-wizard.html> (visitado 10-02-2023).

Distributed Version Control **DistributedVersionControl2023**

Distributed Version Control. En: *Wikipedia*. 17 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Distributed_version_control&oldid=1134292917 (visitado 08-02-2023).

Anot.: Page Version ID: 1134292917.

Git **Git2023**

Git. En: *Wikipedia*. 27 de ene. de 2023. URL: <https://en.wikipedia.org/w/index.php?title=Git&oldid=1135882639> (visitado 08-02-2023).

Anot.: Page Version ID: 1135882639.

Git. URL: <https://git-scm.com/> (visitado 08-02-2023).

Metz: Chapter 1: Object-Oriented Design

metzChapterObjectOrientedDesign2013

Sandi Metz. «Chapter 1: Object-Oriented Design». En: *Practical Object-oriented Design in Ruby: An Agile Primer*. Addison-Wesley, 2013, págs. 1-14. ISBN: 978-0-321-72133-4. Google Books: [rk9sAQAAQBAJ](#).

Object Expressions and Declarations | Kotlin

ObjectExpressionsDeclarations

Object Expressions and Declarations | Kotlin. Kotlin Help. URL: <https://kotlinlang.org/docs/object-declarations.html> (visitado 04-02-2023).

Object-Oriented Programming

ObjectorientedProgramming2023

Object-Oriented Programming. En: *Wikipedia*. 17 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=1134190065 (visitado 04-02-2023).

Anot.: Page Version ID: 1134190065.

Pbpaste Man Page - macOS - SS64.Com

PbpasteManPage

Pbpaste Man Page - macOS - SS64.Com. URL: <https://ss64.com/osx/pbpaste.html> (visitado 08-02-2023).

sdwheeler: Set-Location (Microsoft.PowerShell.Management) - PowerShell

sdwheelerSetLocationMicrosoftPowerShell

sdwheeler. *Set-Location (Microsoft.PowerShell.Management) - PowerShell*. URL: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/set-location> (visitado 08-02-2023).

Version Control

VersionControl2023

Version Control. En: *Wikipedia*. 31 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Version_control&oldid=1136616430 (visitado 08-02-2023).

Anot.: Page Version ID: 1136616430.

Capítulo 4

Programación orientada a objetos en principio

En este capítulo seguiremos desarrollando nuestro juego de cartas y comenzaremos a ver cómo aprovechar las ventajas de la programación orientada a objetos.

4.1. Composición

De momento tenemos jugadores en nuestro juego, pero nuestros jugadores no tienen cartas. Un primer intento de solución podría ser el siguiente:

```
class Player(  
    name: String,  
    health: Int,  
    deck: MutableList<String>,  
    cardsText: MutableMap<String, String>,  
    cardsAttacks: MutableMap<String, Int>  
) {  
    val name: String  
    var health: Int  
    val deck: MutableList<String>  
    val cardsText: MutableMap<String, String>  
    val cardsAttacks: MutableMap<String, Int>  
  
    init {  
        this.name = name  
        this.health = health  
        this.deck = deck  
        this.cardsText = cardsText  
        this.cardsAttacks = cardsAttacks  
    }  
}
```

Aquí introducimos dos nuevos tipos de datos: *MutableList*¹ y *MutableMap*.² Un *MutableList* es una lista que puede ser modificada, es decir, podemos agregar y eliminar elementos de la lista. Una lista, al igual que un arreglo, es homogénea, es decir, todos los elementos de la lista deben ser del mismo tipo. El tipo de elementos de la lista se indica entre <>.

Un *MutableMap* es un diccionario que puede ser modificado. Un diccionario es una estructura de datos que asocia una clave con un valor. En este caso, el tipo de la clave y el tipo del valor se indican entre <>.

Ahora, podemos crear un jugador:

```
val deck = mutableListOf("White-eyes Blue Dragon", "Light Magician")
val cardsText = mutableMapOf(
    "White-eyes Blue Dragon" to "Legendary dragon of destruction",
    "Light Magician" to "The ultimate sorcerer with a powerful attack"
)
val cardsAttacks = mutableMapOf(
    "White-eyes Blue Dragon" to 3000,
    "Light Magician" to 2500
)
val player = Player("Mugi", 8000, deck, cardsText, cardsAttacks)
println(player.name)
println(player.health)
println(player.deck)
println(player.cardsText)
```

Sin embargo, este modelo tiene algunos problemas. Por ejemplo, cada vez que agregamos una carta al mazo, debemos agregar también su texto y su ataque a los diccionarios. ¿Qué pasa entonces si se nos olvida agregar una carta al diccionario de texto?

Composición

La **composición** es la propiedad de todos los objetos de contener a cualquier otro objeto. Podemos pensar la composición de objetos como la composición de funciones matemáticas. Por ejemplo, la función $f(x) = (x + 1)^2$ es una composición de las funciones $g(x) = x + 1$ y $h(x) = x^2$.

Para solucionar este problema podemos crear una clase que represente a una carta y tenga campos para el texto y el ataque. De la misma forma que antes, podemos crear una nueva clase utilizando *IntelliJ*, nombremos a la clase *Card* y agreguemos los campos *name*, *text* y *attack*.

```
class Card(name: String, text: String, attack: Int) {
    val name: String
    val text: String
    val attack: Int

    init {
        this.name = name
        this.text = text
        this.attack = attack
    }
}
```

¹ *MutableList* - Kotlin Programming Language.

² *MutableMap* - Kotlin Programming Language.

```
}
```

Una vez que tenemos la clase *Card*, podemos modificar la clase *Player* para que utilice la nueva clase.

```
class Player(  
    name: String,  
    health: Int,  
    deck: MutableList<Card>  
) {  
    val name: String  
    var health: Int  
    val deck: MutableList<Card>  
  
    init {  
        this.name = name  
        this.health = health  
        this.deck = deck  
    }  
}
```

Note que ahora nuestro código es más conciso y fácil de leer, esto es una de las ventajas que nos ofrece la composición. La composición es una de las propiedades más importantes de la programación orientada a objetos, ya que nos permite separar problemas complejos en problemas más simples.

En realidad desde un comienzo estábamos usando composición sin saberlo, ya que la clase *Player* es una composición de las clases *String*, *Int*, *MutableList* y *MutableMap*. El problema está en que no la estábamos aprovechando al máximo.

¿Ahora la estamos aprovechando al máximo :0?

No realmente, eso es algo que iremos desarrollando a lo largo del libro.

Por último, hacemos:

```
git add .  
git commit -m "FEAT Adds deck of cards to Player class"
```

4.2. Encapsulamiento

Supongamos que queremos agregar un método para que una carta pueda atacar a un jugador. Esto es bastante simple de implementar, solo debemos restar los puntos de ataque de la carta al jugador. Definamos el método *attack* en la clase *Card* de la siguiente forma:

```
fun attack(player: Player) {  
    player.health -= attack  
}
```

Nota. El operador *--* (resta y asignación) es una abreviación de *a = a - b*.

Ahora podemos atacar a un jugador con una carta:

```
val deck = mutableListOf(
    Card("White-eyes Blue Dragon", "Legendary dragon of destruction", 3000)
)
val player = Player("Mugi", 8000, deck)
val attacker =
    Card("Light Magician", "The ultimate sorcerer with a powerful attack", 2500)
attacker.attack(player)
println(player.health)
```

Sin embargo, este código tiene un problema. ¿Qué pasa si el ataque de la carta es mayor que los puntos de vida del jugador? En este caso, el jugador tendrá puntos de vida negativos y eso podría causar problemas en el juego.

Para solucionar este problema podemos agregar una condición en el método `attack`:

```
fun attack(player: Player) {
    if (attack < player.health) {
        player.health -= attack
    } else {
        player.health = 0
    }
}
```

Sin embargo, ahora tenemos un problema diferente. ¿Qué pasa si hay otras formas en las que el jugador puede perder puntos de vida? Entonces tendríamos que agregar una condición en cada método que modifique los puntos de vida del jugador. Esto haría nuestro código muchísimo más difícil de mantener.

Encapsulamiento

El **encapsulamiento** es la propiedad de los objetos de «empaquetar» sus datos y métodos, manejando y controlando el acceso a ellos.

¿Cómo podemos aplicar la encapsulación en este caso? Lo primero será definir un método para modificar los puntos de vida del jugador, esto podemos hacerlo creando un método *takeDamage* en la clase *Player*:

```
fun takeDamage(damage: Int) {
    if (damage < health) {
        health -= damage
    } else {
        health = 0
    }
}
```

Ahora podemos modificar el método `attack` para que utilice el nuevo método:

```
fun attack(player: Player) {
    player.takeDamage(this.attack)
}
```

Pero esto no es suficiente, ya que los puntos de vida del jugador pueden ser modificados desde cualquier parte del código sin utilizar el método `takeDamage`. Para solucionar esto podemos utilizar un modificador de acceso llamado *private*. Este modificador de acceso nos permite definir que un campo o método solo puede ser accedido desde la **misma clase**.³

Para hacer que un campo o método sea privado debemos agregar el modificador de acceso `private` antes de su definición. Por ejemplo, podemos hacer que el campo `health` de la clase `Player` sea privado de la siguiente forma:

```
class Player(  
    name: String,  
    health: Int,  
    deck: MutableList<Card>  
) {  
    val name: String  
    private var health: Int  
    val deck: MutableList<Card>  
    ...  
}
```

Ahora el campo `health` solo puede ser accedido desde la clase `Player`.

Yay

Pero introducimos otro problema.

Bro

Hicimos que el campo `health` sólo pueda ser asignado desde la clase `Player`, pero además hicimos que sólo pueda ser «visto» desde la misma clase. Esto significa que no podemos hacer cosas como `println(player.health)` desde la función `main`. Podemos solucionar esto creando un método `getHealth` en la clase `Player`, pero *Kotlin* nos ofrece una forma más simple de hacer esto. La idea es decir que una variable es pública, pero que su valor sólo puede ser asignado (*set*) desde la misma clase.

```
var health: Int private set
```

El encapsulamiento es un concepto clave en el desarrollo de software, porque (como veremos en este libro) siempre vamos a querer *encapsular el cambio*. En este caso, si ahora quisieramos hacer que los puntos de vida del jugador sí puedan ser negativos, solo tendríamos que modificar el método `takeDamage` y no tendríamos que modificar los múltiples métodos que podrían querer modificar los puntos de vida del jugador.

Ahora podemos hacer *commit* de los cambios:

```
git add .  
git commit -m "FEAT Adds attack method to Card class"
```

4.3. Herencia

Consideremos ahora que el mazo puede tener cartas de diferentes tipos: cartas de monstruos, magias y trampas. Por ahora, diremos que la única diferencia entre estas cartas es que las cartas de monstruos tienen un ataque, mientras que las cartas de magia y trampa no.

³Que un campo o método sea privado no significa que no pueda ser accedido desde otros objetos, solo significa que no puede ser accedido desde objetos de otras clases.

Primero, simplifiquemos un poco nuestro constructor para ir reduciendo la cantidad de código que tenemos que escribir:

```
class Card(name: String, text: String, attack: Int) {
    val name: String = name
    val text: String = text
    val attack: Int = attack

    fun attack(player: Player) {
        player.takeDamage(attack)
    }
}
```

Noten que ahora nos deshicimos del bloque `init` y definimos los campos de la clase directamente en el cuerpo de la clase, esto es equivalente a lo que teníamos antes.

Ahora sí, veamos cómo podemos definir los tipos de cartas que queremos. Una solución es agregar un campo *type* a la clase *Card*, esto podría ser un *String*, además podríamos definir que si intentamos atacar con una carta mágica o trampa el daño que se le hace al jugador es 0.

```
class Card(name: String, text: String, attack: Int, type: String) {
    val name: String = name
    val text: String = text
    val attack: Int = attack
    val type: String = type

    fun attack(player: Player) {
        if (type == "Monster") {
            player.takeDamage(attack)
        } else {
            println("You can't attack with a $type card")
        }
    }
}
```

Esta implementación tiene varios problemas, pero enfoquémonos en dos:

- Utilizamos un *String* para representar el tipo de carta, esto es un problema porque no podemos asegurar que el valor que se le pase al constructor sea uno de los tipos válidos y tendríamos que agregar una condición para verificar que el tipo sea válido.
- Si queremos agregar un nuevo tipo de carta, tendríamos que modificar el código de la clase *Card* y agregar una nueva condición en el método *attack*.

Pero ambos problemas se reducen a lo mismo: no encapsulamos el cambio. Si queremos agregar un nuevo tipo de carta, tenemos que modificar el código de todas las clases que dependen de la clase *Card*. Esto claramente no escala bien.

Una forma de enfrentar este problema es utilizar **herencia**.

Herencia

La herencia es la propiedad de los objetos de heredar características de otros objetos. Diremos que un objeto *T* hereda de otro objeto *S* si *T* es una especialización de *S*. A *T* se le conoce como **subclase** de *S* y a *S* como **superclase** de *T*.

Veamos un ejemplo de herencia en la vida real. Consideremos la familia de animales *Caninae*, todos los miembros de esta familia pueden mover la cola (creo), pero puede que lo hagan por distintos motivos.

Consideremos ahora a tres animales de esta familia: el perro, el lobo y el zorro. Todos ellos pueden mover la cola, pero el perro lo hace para saludar, el lobo para intimidar y el zorro para jugar. Podemos decir entonces que la propiedad de mover la cola es una propiedad de la familia *Caninae* que es heredada por todos los miembros de esta familia.

Github Copilot me sugiere escribir «En programación, la herencia es una forma de reutilizar código», pero esto es incorrecto. La herencia tiene como efecto secundario la reutilización de código, pero no es su objetivo. El objetivo de la herencia es especializar una clase, y para esto es necesario que la herencia tenga coherencia lógica.

Para ver un ejemplo de herencia sin coherencia lógica, pensemos que tanto el perro como el lobo pueden aullar. Si nuestro objetivo es sólo reutilizar código, podríamos decir que perro es una subclase de lobo, pero esto no tiene sentido, ya que los perros y los lobos son especies distintas.

4.3.1. Herencia en Kotlin

En *Kotlin* existen dos tipos de clases:

- **Clases finales:** estas clases no pueden ser heredadas.
- **Clases abiertas:** estas clases pueden ser heredadas.

Por defecto, todas las clases son finales, para hacer una clase abierta se utiliza la palabra reservada *open*. Por ejemplo, la clase *Card* que definimos antes podría ser abierta:

```
open class Card(name: String, text: String, attack: Int) {...}
```

Ahora, creemos una nueva clase *MonsterCard* que herede de *Card*, esto lo haremos de la siguiente forma:

```
class MonsterCard(name: String, text: String, attack: Int) : Card(name, text, attack) {  
}
```

Vamos viendo:

1. Definimos la clase *MonsterCard* como haríamos normalmente.
2. Luego de los parámetros del constructor, escribimos dos puntos (:) y el nombre de la clase de la que queremos heredar.
3. Finalmente, entre paréntesis escribimos los parámetros del constructor de la clase de la que queremos heredar.

Importante. Una clase puede tener una única superclase, pero una superclase puede tener múltiples subclases.

Antes de seguir volvamos a la clase *Card*. En la clase *Card* no se declara explícitamente una superclase, la keyword siendo *explícitamente*. Esto es porque en *Kotlin* todas las clases heredan de la clase *Any*,⁴ que es la superclase de todas las clases, y por lo tanto, la superclase de *Card*. Esto significa que lo que hicimos antes es equivalente a:

⁴Built-in Types and Their Semantics - Kotlin Language Specification.

```
open class Card(name: String, text: String, attack: Int) : Any() {...}
```

Noten que agregamos paréntesis después del nombre de la superclase, esto es porque *Any* tiene un constructor vacío, y por lo tanto, debemos llamarlo explícitamente. Lo primero que se ejecuta en el constructor de una clase es el constructor de la superclase, antes de ejecutar cualquier otra línea de código.

Importante. Como al crear una clase lo primero que hacemos es llamar explícitamente al constructor de la superclase, esto quiere decir que no estamos heredando el constructor de la superclase. Esto se debe a que, al no ser un método, los constructores no se heredan.

Como vimos, cuando una clase hereda de otra, hereda todas las propiedades y métodos de la clase padre. Esto significa que no tenemos que volver a definir las propiedades *name*, *text* y *attack* en la clase *MonsterCard*, ya que estas propiedades ya están definidas en la clase *Card*. Lo mismo ocurre con el método *attack*, ya que este método está definido en la clase *Card* y por lo tanto, está disponible para la clase *MonsterCard*. Así tendremos algo como:

```
open class Card(name: String, text: String, attack: Int) {  
    val name: String = name  
    val text: String = text  
    val attack: Int = attack  
  
    fun attack(player: Player) {...}  
}
```

```
class MonsterCard(name: String, text: String, attack: Int) : Card(name, text, attack)
```

¿Notaron que la clase *MonsterCard* no tiene llaves? Esto es porque la clase *MonsterCard* no tiene código adicional, por lo tanto, no necesitamos definir un cuerpo para la clase.

Por último modifiquemos el método *attack* de la clase *Card* para que sólo las cartas de monstruos puedan atacar.

```
fun attack(player: Player) {  
    when(this) {  
        is MonsterCard -> player.takeDamage(this.attack)  
        else -> println("You can't attack with this card")  
    }  
}
```

Aquí tenemos una cosa nueva, la keyword *is*. Esta keyword nos permite verificar si una variable es de un tipo determinado.

Quizás ya vienen venir esto, pero tenemos un problema. ¿Qué sucede si queremos agregar nuevos tipos de cartas? Pues tendríamos que agregar nuevas condiciones al método *attack* de la clase *Card*. Es decir, seguimos sin encapsular el cambio.

Single-Responsibility Principle

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

El problema aquí lo podemos resumir en que la clase *Card* tiene la responsabilidad de conocer a todos los tipos de cartas que existen, esto es innecesario ya que cada clase conoce su propio tipo.

4.3.2. Method-lookup

Antes de continuar debemos entender algunos conceptos nuevos.

El primer concepto nuevo es el de **firma**. Todas las funciones tienen una firma, que es la combinación de su nombre y sus parámetros. Por ejemplo, la función `attack` tiene la firma `attack(Int)`. Toda función debe tener una firma única dentro de una clase, es decir, no puede haber dos funciones con el mismo nombre y los mismos parámetros en una misma clase, aunque estas funciones tengan distinto tipo de retorno. Por ejemplo, no podemos tener dos funciones `attack(Int)` en la misma clase, aunque una tenga tipo de retorno `Int` y la otra `String`. Esto es porque la firma de una función es lo único que importa a la hora de llamar a una función, y no el tipo de retorno.

Por ejemplo, si tenemos dos funciones `attack(Int)` y `attack(String)`, podemos llamar a ambas funciones de la siguiente manera:

```
attack(1)
attack("1")
```

Esto lo que se conoce como **sobrecarga de funciones** (o **overloading**).

Sobrecarga de funciones

La sobrecarga de funciones es una característica de los lenguajes de programación que permite definir varias funciones con el mismo nombre, pero con distinto número de parámetros o distinto tipo de parámetros. En Kotlin, la sobrecarga de funciones se realiza de forma automática, por lo que no es necesario declarar explícitamente que una función está sobrecargada. En otros lenguajes, como *Pascal*, es necesario declarar explícitamente que una función está sobrecargada.

La sobrecarga de funciones debe evitarse dentro de lo posible, ya que el proceso de resolver la sobrecarga de funciones es complejo⁵ y puede llevar a errores difíciles de detectar. También puede llevar a confusión, ya que no todos los lenguajes de programación resuelven la sobrecarga de funciones de la misma manera, o incluso pueden no resolverla en absoluto (como es el caso de *Python*). Esto no quiere decir que no debemos usar la sobrecarga de funciones, sino que debemos usarla con cuidado y solo cuando sea necesario.

Nota. A partir de ahora utilizaremos la siguiente sintaxis para referenciar miembros de una clase:

`<Class>::<method>(<Parameter Types>): <Return Type>.`

Por ejemplo, la función `attack(Int)` de la clase `Card` se representaría como `Card::attack(Int): Unit`.

Lo siguiente que debemos entender es el concepto de **mensaje**. Por la propiedad de encapsulamiento, sabemos que no podemos acceder a los atributos de una clase sin pasar por un «filtro». Esto es, no podemos acceder a los atributos de una clase directamente, sino que debemos hacerlo a través de mensajes (message passing).

Method-lookup

El proceso de buscar una función para responder a un mensaje se conoce como **method-lookup**. El method-lookup se realiza de la siguiente manera:

1. Se busca la función en la clase actual.
2. Si no se encuentra, se busca en la superclase.

⁵Overload Resolution - Kotlin Language Specification.

3. Si no se encuentra, se busca en la superclase de la superclase.
 4. Y así sucesivamente hasta llegar a la clase Any.
 5. Si no se encuentra en ninguna de las clases, se produce un error.
- En los lenguajes de tipado estático, el method-lookup siempre se resuelve en tiempo de compilación.

Nota. Un compilador es un programa que traduce un programa de un lenguaje a otro. En general cuando hablamos de compilación nos referimos a la traducción de un lenguaje de alto nivel a un lenguaje de bajo nivel con el objetivo de que pueda ser ejecutado por una máquina.

Por otro lado, un interprete es un programa que ejecuta directamente un programa de un lenguaje de alto nivel.

Ejemplos de lenguajes compilados son C, C++, Java, C# y Kotlin. Ejemplos de lenguajes interpretados son Python, Ruby, MATLAB y Lisp.

Una manera (no tan lejana de la realidad) de entender el method-lookup es pensar que cada objeto tiene una tabla en la que una columna son los mensajes que puede recibir y otra columna son las funciones que se ejecutan cuando se recibe ese mensaje. Cuando se envía un mensaje a un objeto, el objeto busca en su tabla la función que se debe ejecutar para ese mensaje y la ejecuta. Si no encuentra la función en su tabla, busca en la tabla de su superclase y así sucesivamente hasta llegar a la tabla de Any.

Ahora, antes dijimos que la firma de una función debe ser única dentro de una clase. ¿Pero qué pasa cuando una clase tiene un método con la misma firma que un método de su superclase?

Sobrescritura de funciones

La sobrescritura (overriding) de métodos es una característica de los lenguajes de programación que permite definir una función con la misma firma que una función de la superclase. En Kotlin, la sobrescritura **no** se realiza de forma automática, por lo que es necesario declarar explícitamente que una función está sobrescrita con la palabra reservada *override*.

Nota. Al igual que con las clases, existen las funciones abiertas y las finales.

Ahora sí, volvamos a nuestro ejemplo. Teníamos que los ataques funcionaban de la siguiente manera:

- Si la carta es un *Monstruo*, el ataque se realiza con el ataque del monstruo.
- Si no es un monstruo, entonces no se puede atacar.

Podemos pensar esto de la siguiente forma: ninguna carta puede atacar, excepto los monstruos. ¿Cómo podemos expresar esto en código?

```
open class Card(name: String, text: String, attack: Int) {
    val name: String = name
    val text: String = text
    val attack: Int = attack

    open fun attack(player: Player) {
        println("You can't attack with this card")
    }
}
```

```
class MonsterCard(name: String, text: String, attack: Int) : Card(name, text, attack) {
    override fun attack(player: Player) {
        player.takeDamage(this.attack)
    }
}
```

Probemos nuestro código:

```
val dragon = MonsterCard(
    "White-eyes Blue Dragon", "Legendary dragon of destruction", 3000)
val reflect = Card(
    "Reflect Strength", "When a Monster Card attacks you, it is destroyed", 0)
val deck = mutableListOf(dragon, reflect)
val player = Player("Mugi", 8000, deck)
println("$player has ${player.health} health points")
dragon.attack(player)
println("$player has ${player.health} health points")
reflect.attack(player)
println("$player has ${player.health} health points")
```

Si ejecutamos el código, obtenemos algo como lo siguiente:

```
cl.ravenhill.kygo.Player@cc34f4d has 8000 health points
cl.ravenhill.kygo.Player@cc34f4d has 5000 health points
You can't attack with this card
cl.ravenhill.kygo.Player@cc34f4d has 5000 health points
```

¿Qué?

Como ya hemos dicho, todos los objetos heredan de *Any*, esto implica que heredan las propiedades y métodos de *Any*. En particular, heredan el método `Any::toString(): String`.

Cuando hacemos `println("$player")`, en realidad lo que estamos haciendo es hacer `println(player.toString())`. El método `toString()` es un método que poseen todos los objetos que devuelve un string que representa al objeto. Por defecto, el método `Any::toString()` devuelve el nombre de la clase seguido de @ y un número que indica donde está ubicado el objeto en memoria. En nuestro caso, el objeto `player` está ubicado en la dirección de memoria `cc34f4d`.

Sobrescritura de toString

Si queremos que nuestro objeto se imprima de una forma más amigable, podemos sobrescribir el método `toString()`.

¿Cómo hacemos esto? Simplemente sobrescribiendo el método `toString()` de la siguiente forma:

```
override fun toString(): String {
    return name
}
```

4.4. Polimorfismo de subtipos

La **serialización** es el proceso de convertir un objeto en un formato que pueda ser almacenado o transmitido y reconstruido posteriormente en el mismo o en un objeto similar. La serialización es un proceso muy útil para guardar el estado de un objeto en un archivo o transmitirlo a través de una red.

Supongamos que queremos guardar nuestras cartas en archivos de distintos formatos: XML, JSON y YAML. Los formatos que queremos son los siguientes:

```
<!-- XML -->
<MonsterCard>
  <name>White-eyes Blue Dragon</name>
  <text>Legendary dragon of destruction</text>
  <attack>3000</attack>
</MonsterCard>
```

```
// JSON
{
  "name": "White-eyes Blue Dragon",
  "text": "Legendary dragon of destruction",
  "attack": 3000,
  "type": "MonsterCard"
}
```

```
# YAML
!!MonsterCard
name: White-eyes Blue Dragon
text: Legendary dragon of destruction
attack: 3000
```

Con esto podemos hacer el proceso de serialización de la siguiente manera:

```
open class Card(name: String, text: String, attack: Int) {
  ...
  open fun toXml(): String {
    return """
      |<Card>
      |  <name>$name</name>
      |  <text>$text</text>
      |  <attack>$attack</attack>
      |</Card>
      """.trimMargin()
  }
  open fun toJson(): String {
    return """
      |{
      |  "name": "$name",
      |  "text": "$text",
```

```

        | "attack": $attack
        | "type": "Card"
        |}
    """.trimMargin()
}
open fun toYaml(): String {
    return """
        !!Card
        |name: $name
        |text: $text
        |attack: $attack
        """.trimMargin()
}
}

```

Y para las cartas de monstruo:

```

class MonsterCard(name: String, text: String, attack: Int) : Card(name, text, attack) {
    ...
    override fun toXml(): String {
        return """
            |<MonsterCard>
            |  <name>$name</name>
            |  <text>$text</text>
            |  <attack>$attack</attack>
            |</MonsterCard>
            """.trimMargin()
        }
    override fun toJson(): String {
        return """
            |{
            |  "name": "$name",
            |  "text": "$text",
            |  "attack": $attack
            |  "type": "MonsterCard"
            |}
            """.trimMargin()
        }
    override fun toYaml(): String {
        return """
            !!MonsterCard
            |name: $name
            |text: $text
            |attack: $attack
            """.trimMargin()
        }
    }
}

```

¿Pero qué finalidad tiene definir las funciones toXml(), toJson() y toYaml() en la clase Card si las vamos a sobrees-

cribir en la clase que hereda de ella? Como verán, tenemos un problema.

La idea sería poder definir una función en la clase carta sin tener que implementarla. Esto podemos lograrlo con **clases abstractas**.

Clase abstracta

Una clase abstracta es una clase incompleta que no puede ser instanciada.

Una función abstracta es una función que no tiene implementación. Para que una clase abstracta tenga sentido, debe tener al menos una función abstracta.

En *Kotlin* las clases abstractas se definen con la palabra clave `abstract` y deben comenzar con la palabra clave `Abstract`.⁶ Comencemos por cambiar el nombre de la clase `Card` a `AbstractCard`, para esto podemos darle click derecho al nombre de la clase y seleccionar la opción `Refactor -> Rename` como en la figura 4.1, esto cambiará el nombre de la clase en todos los archivos donde se use además de cambiar el nombre del archivo.

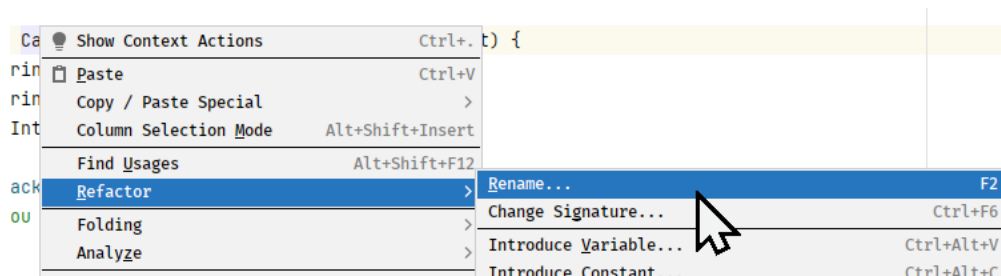


Figura 4.1: Renombrando la clase `Card` a `AbstractCard`.

Ahora, agregamos la palabra clave `abstract` antes de la palabra clave `class` y cambiamos los métodos `toXml()`, `toJson()` y `toYaml()` a funciones abstractas:

```
abstract class AbstractCard(name: String, text: String, attack: Int) {  
    ...  
    abstract fun toXml(): String  
    abstract fun toJson(): String  
    abstract fun toYaml(): String  
}
```

Importante. Las funciones abstractas deben ser implementadas en las clases que hereden de la clase abstracta.

¿Qué sucede si ahora además quiero poder guardar las cartas en un archivo? Podríamos definir métodos para guardar las cartas en archivos de distintos formatos:

```
abstract class AbstractCard(name: String, text: String, attack: Int) {  
    ...  
    abstract fun toXmlFile(fileName: String)  
    abstract fun toJsonFile(fileName: String)  
    abstract fun toYamlFile(fileName: String)  
}
```

⁶Esto último es para hacer más fácil distinguir entre clases abstractas y clases concretas.

Pero (para variar) tenemos un problema. Cada vez que queramos agregar un nuevo formato, tendremos que agregar una nueva función a la clase abstracta y luego implementarla en cada clase que herede de ella. Esto es un problema porque si tenemos muchas clases que hereden de la clase abstracta, tendremos que modificarlas todas cada vez que se agregue un nuevo formato.

Open-closed principle

Las clases deben estar abiertas para extensión pero cerradas para modificación.

Una solución es delegar la responsabilidad de guardar las cartas en archivos a una clase que se encargue de eso, aprovechando la propiedad de composición y polimorfismo.

Polimorfismo de subtipos

El polimorfismo de subtipos es la propiedad de un objeto de tipo A de verse y ser usado como un objeto de tipo B siempre y cuando A sea un subtipo de B .

En términos prácticos, esto significa que podemos crear una función $f(B)$ y pasarle un objeto de tipo A siempre y cuando A sea un subtipo de B .

Pero antes de continuar ordenemos nuestro programa. Como notarán, a medida que avanzamos en el desarrollo del programa, la cantidad de archivos que tenemos va creciendo de forma acelerada. Esto es un problema porque si queremos modificar una clase, tenemos que ir a buscarla en todos los archivos. Una solución es agrupar las clases con características similares en paquetes.

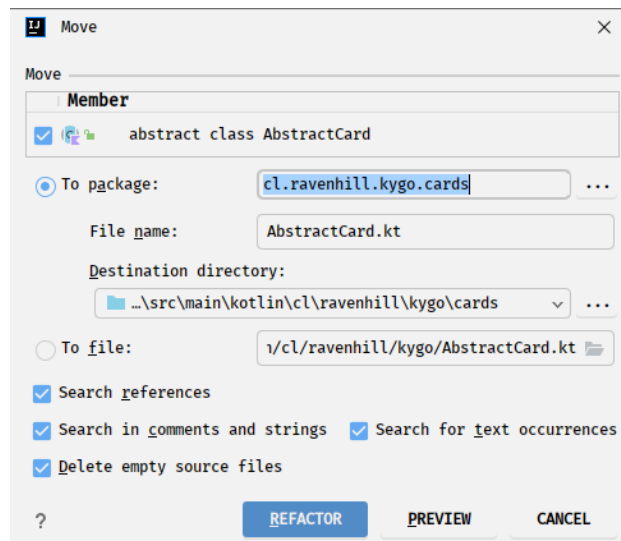


Figura 4.2: Moviendo las clases AbstractCard y MonsterCard al paquete cards.

Hagamos un paquete cards haciendo click derecho en el paquete `cl.ravenhill.kygo` y arrastremos los archivos `AbstractCard.kt` y `MonsterCard.kt` dentro de él, esto desplegará una ventana como la de la figura 4.2 donde deberemos comprobar los cambios que se realizarán y luego darle click al botón *Refactor*. Luego hagamos un paquete `serializer` en el paquete `cl.ravenhill.kygo`.

Luego, creemos la clase `AbstractCardSerializer` en el paquete `serializer`, recordando agregar la palabra clave `abstract` antes de la palabra clave `class`:

```

package cl.ravenhill.kygo.serializer

import cl.ravenhill.kygo.cards.AbstractCard
import java.io.File

abstract class AbstractCardSerializer {
    fun toFile(card: AbstractCard, filename: String) {
        val file = File(filename)
        file.writeText(serialize(card))
    }

    abstract fun serialize(card: AbstractCard): String
}

```

Aquí tenemos algo nuevo: la clase File y el método File::writeText(String). La clase File es una clase que representa un archivo en el sistema de archivos, y el método File::writeText(String) es un método que recibe un string y lo escribe en el archivo.

Ahora podemos definir clases que hereden de AbstractCardSerializer para cada formato:

```

class XmlCardSerializer : AbstractCardSerializer() {
    override fun serialize(card: AbstractCard): String {
        return """
            |<Card>
            |  <name>${card.name}</name>
            |  <text>${card.text}</text>
            |</Card>
            """.trimMargin()
    }
}

```

Y por último, podemos modificar la clase AbstractCard para que use la clase AbstractCardSerializer:

```

abstract class AbstractCard(
    name: String, text: String, attack: Int,
    serializer: AbstractCardSerializer) {
    ...
    var serializer = serializer

    fun toFile(filename: String) {
        serializer.toFile(this, filename)
    }

    fun serialize(): String {
        return serializer.serialize(this)
    }
}

```

Aquí nos topamos nuevamente con nuestro amigo `this`. En *Kotlin* existen dos pseudo-variables que nos permiten referenciar a un objeto: `this` y `super`. ¿Cuál es la diferencia entre ellas? Fácil, `this` hace referencia al objeto que recibe el mensaje, mientras que `super` hace referencia al objeto que recibe el mensaje.

¿Qué?

`this` y `super`

Tanto `this` como `super` son pseudo-variables que hacen referencia al objeto que recibe un mensaje. Lo que hace distintos a `this` y `super` es el *method-lookup*. En el caso de `this`, el *method-lookup* comienza en la clase del objeto que recibe el mensaje, mientras que en el caso de `super`, el *method-lookup* comienza en la superclase del objeto que recibe el mensaje.

Ahora, nuestra clase `MonsterCard` debiera quedar así:

```
class MonsterCard(
    name: String,
    text: String,
    attack: Int,
    serializer: AbstractCardSerializer
) : AbstractCard(name, text, attack, serializer) {
    override fun attack(player: Player) {
        player.takeDamage(this.attack)
    }
}
```

¿Lo ven venir? Tenemos un pequeño problema: estamos usando una clase abstracta como tipo. Este problema es un poco más complejo de ver, pero es importante entenderlo. Al declarar el tipo de la variable `serializer` como `AbstractCardSerializer`, estamos diciendo que la variable es de un tipo que definimos como algo incompleto. ¿Existe una manera de representar tipos que agrupen a todos los tipos que hereden de una clase abstracta y que no sean abstractos?

Interfaces

Una interfaz es un contrato entre un cliente y un proveedor. Un proveedor declara las propiedades que tienen todos los objetos que representa y un cliente puede crear cualquier objeto que cumpla con el contrato de la interfaz.

Importante. *Las interfaces son una forma de tener polimorfismo de subtipos sin tener que usar herencia de clases (porque una interfaz no es una clase).*

Las clases abstractas e interfaces son similares, pero tienen una diferencia importante. Como ya sabemos, una clase puede extender de solo una superclase, pero como una interfaz no es una clase, sino que un contrato, una clase puede implementar muchas interfaces siempre y cuando no rompa con el contrato de ninguna de ellas.

En *Kotlin* las interfaces se definen con la palabra clave `interface` en vez de `class`.

Podemos crear una interfaz `CardSerializer` haciendo click derecho en el paquete `serializer` y seleccionando *New -> Kotlin File/Class*. Luego, en la ventana que se despliega, seleccionamos *Interface* y le damos un nombre a la interfaz, en este caso `CardSerializer`. Eso nos creará el siguiente archivo:

```
package cl.ravenhill.kygo.serializer

interface CardSerializer {
}
```

Y ahora agregamos los métodos `serialize(AbstractCard): String` y `toFile(AbstractCard, String)` a la interfaz:

```
interface CardSerializer {
    fun serialize(card: AbstractCard): String
    fun toFile(card: AbstractCard, filename: String)
}
```

Ahora, podemos modificar la clase `AbstractCardSerializer` para que implemente la interfaz `CardSerializer`, esto lo hacemos de la misma forma que heredamos de una clase, con la diferencia de que (al no ser clases) las interfaces no tienen constructor:

```
abstract class AbstractCardSerializer: CardSerializer {
    override fun toFile(card: AbstractCard, filename: String) {
        val file = File(filename)
        file.writeText(serialize(card))
    }

    abstract override fun serialize(card: AbstractCard): String
}
```

Y ahora podemos modificar la clase `AbstractCard` para que use la interfaz `CardSerializer`:

```
abstract class AbstractCard(
    name: String, text: String, attack: Int,
    serializer: CardSerializer
) {...}
```

Como siempre, terminamos con:

```
git add .
git commit -m "FEAT Adds card serializers"
```

¡Te toca!

(4.1)

Cree una interfaz `Card` para no usar la clase `AbstractCard` como tipo.

Solución: <https://github.com/r8vnhill/kygo/commit/523a8f441e67e9583a0c2edf53dab0b89e181aec?diff=split>

Ejercicio 4.1. Implemente las clases `JsonSerializer` y `YamlSerializer` que extiendan de `AbstractCardSerializer` y que implementen el método `serialize`.

4.5. Principio de Liskov

Veamos otro problema en nuestra implementación. Hasta ahora definimos 3 tipos de cartas: *Monstruo*, *Mágica* y *Trampa*. Además definimos un método `Card::attack(Player)` que es invocado cuando una carta ataca a un jugador. ¿No es raro que las cartas *Mágica* y *Trampa* también tengan un método `attack`?

En efecto, no tiene sentido que una carta *Mágica* ataque a un jugador, aunque definamos un método `attack` que no haga nada. Una de las razones por las que no tiene sentido es porque estamos rompiendo el *principio de Liskov*

Principio de sustitución de Liskov

Sea $\phi(x)$ una propiedad verificable de los objetos x de tipo \mathcal{T} . Entonces, $\phi(y)$ debe ser verificable para objetos y de tipo \mathcal{S} donde \mathcal{S} es un subtipo de \mathcal{T} .

¿Qué?

En nuestro caso, la propiedad $\phi(x)$ es la propiedad de que las cartas pueden atacar a un jugador, y \mathcal{T} es el tipo `Card`. Pero la verdad es que no todas nuestras cartas pueden atacar a un jugador, por lo que no podemos decir que $\phi(y)$ es verificable para todos los objetos de tipo `Card`. Por lo tanto, no podemos decir que el principio de Liskov se cumple. Noten que hay casos en los que el *principio de Liskov* se cumplirá por restricciones impuestas por el lenguaje, pero hay otros casos (como en nuestro ejemplo) en los que el principio de Liskov no se cumple por restricciones lógicas que nosotros mismos impusimos.

¿Cómo podemos solucionar este problema? La solución en este caso es simple: quitar el método `attack` de la clase `Card` y definirlo solamente en la clase `MonsterCard`. De la misma forma, el parámetro `attack` de la clase `MonsterCard` no tiene sentido para las cartas *Mágica* y *Trampa*, por lo que también lo podemos quitar.

Así, la clase `Card` quedaría de la siguiente forma:

```
// Cambiar a clase abstractaAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
open class Card(name: String, text: String, serializer: CardSerializer) {
    val name = name
    val text = text
    var serializer = serializer

    fun toFile(filename: String) {
        serializer.toFile(this, filename)
    }

    fun serialize(): String {
        return serializer.serialize(this)
    }
}
```

Y la clase `MonsterCard` quedaría de la siguiente forma:

```
class MonsterCard(
    name: String,
    text: String,
    attack: Int,
    serializer: CardSerializer
```

```
) : Card(name, text, serializer) {  
    val attack = attack  
  
    fun attack(player: Player) {  
        player.takeDamage(this.attack)  
    }  
}
```

COMMIT

El *principio de Liskov* es uno de los principios más importantes de la programación orientada a objetos, y es muy importante que lo tengamos en cuenta a la hora de diseñar nuestras aplicaciones. En el siguiente capítulo veremos cómo podemos comenzar a diseñar aplicaciones pensando un poco en el futuro y en cómo todo se puede caer a pedazos de un momento a otro.

4.6. Ejercicios

Importante. Recuerde hacer commit después de cada pregunta.

Ejercicio 8 Orden de ejecución

Para los siguientes fragmentos de código, indique si compila y, en caso afirmativo, indique el orden de ejecución de las líneas de código.

1.

```
class Animal(name: String) {  
    private val name = name  
    fun pairTo(animal: Animal) {  
        println("A $name with a ${animal.name}")  
    }  
}  
fun main() {  
    Animal("Dog").pairTo(Animal("Cat"))  
}
```
2.

```
open class A {  
    init {  
        println("A")  
    }  
}  
class B : A() {  
    init {  
        println("B")  
    }  
}  
fun main() {  
    B()  
}
```
3.

```
open class A {  
    init {  
        println("A")  
    }  
}  
class B : A() {  
    init {  
        println("B")  
    }  
}  
class C : B() {  
    init {  
        println("C")  
    }  
}  
fun main() {  
    C()  
}
```
4.

```
open class A {
```

```

    fun m1
}
class B : A() {
    override fun foo() {
        println("B.foo")
    }
}
fun main() {
    B().foo()
}

5. open class A {
    open fun m1(): String {
        return "A.m1"
    }
    open fun m2(): String {
        return "A.m2 > ${this.m1()}"
    }
    open fun m5(): String {
        return "A.m5 > ${this.m2()}"
    }
}
open class B : A() {
    override fun m1(): String {
        return "B.m1"
    }
    fun m3(): String {
        return "B.m3 > ${super.m1()}"
    }
    fun m4(): String {
        return "B.m4 > ${super.m2()}"
    }
    override fun m5(): String {
        return "B.m5 > ${super.m5()}"
    }
}
class C : B() {
    override fun m2(): String {
        return "C.m2 > ${this.m1()}"
    }
}
fun main() {
    println("1. ${C().m1()}")
    println("2. ${B().m1()}")
    println("3. ${A().m1()}")
    println("4. ${C().m2()}")
    println("5. ${B().m2()}")
    println("6. ${A().m2()}")
    println("7. ${B().m3()}")
    println("8. ${C().m4()}")
    println("9. ${C().m5()}")
}

```



```

6. open class A {
    fun m(o1: A, o2: B) : String {
        return "A.m(A, B)"
    }
}
class B : A() {
    fun m(o1: A, o2: A) : String {
        return "B.m(A, A)"
    }
}
fun main() {
    println("1. ${B().m(A(), A())}")
    println("2. ${B().m(A(), B())}")
}

```

Ejercicio 9 Aves

Considere la siguiente implementación de distintos tipos de aves:

```

interface Bird {
    fun fly()
}

class Duck : Bird {
    override fun fly() {
        println("I'm flying")
    }
}

class Pidgeon : Bird {
    override fun fly() {
        println("I'm flying")
    }
}

class Penguin : Bird {
    override fun fly() {
        println("I can't fly")
    }
}

class Ostrich : Bird {
    override fun fly() {
        println("I can't fly")
    }
}

```

¿Rompe esto con algún principio visto en este capítulo? ¿Por qué? ¿Cómo podría solucionar este problema?

Ejercicio 10 Puntos en el espacio

1. Defina una clase Point2D que represente un punto en el plano.

2. Defina el método `Point2D::toString(): String` que devuelva una representación del punto en el formato `Point2D(x, y)`.

3. La distancia entre dos puntos p y q se define como:

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Defina un método `Point2D::distanceTo(Point2D): Double` que calcule la distancia entre dos puntos. *Hint: puede importar `kotlin.math.sqrt` para usar la función raíz cuadrada de la librería estándar de Kotlin.*

4. Defina la función `Point2D::distanceToOrigin(): Double` que calcule la distancia entre un punto y el origen del sistema de coordenadas.
5. Considere la siguiente implementación de la interfaz `Point`:

```
interface Point {  
    val coordinates: DoubleArray  
    fun distanceTo(other: Point): Double  
    fun distanceToOrigin(): Double  
}
```

Modifique la clase `Point2D` para que implemente la interfaz `Point`.

6. Defina una clase `Point3D` que implemente la interfaz `Point`. Para esto considere que la distancia entre dos puntos en el espacio se define como:

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}$$

7. Defina una clase `PointND` que represente un punto en un espacio de n dimensiones. Para esto considere que la distancia entre dos puntos en el espacio se define como:

$$d(p, q) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}$$

y que puede calcularse de la siguiente forma:

1. Dados dos puntos p y q , almacene las coordenadas del más corto (menos dimensiones) en una variable `shorter` y las del más largo (más dimensiones) en una variable `longer`.
2. Cree una variable `sum` que almacene el valor 0.
3. Para cada coordenada i en `shorter`:
 - a) Calcule la diferencia entre la coordenada i de `shorter` y la coordenada i de `longer`.
 - b) Eleve el resultado al cuadrado.
 - c) Sumelo a la variable `sum`.
4. Para cada coordenada i en `longer` que no haya sido considerada en el paso anterior:
 - a) Eleve la coordenada al cuadrado.
 - b) Sumelo a la variable `sum`.
5. Calcule la raíz cuadrada de `sum`.

8. Modifique su estructura de clases para representar `Point2D` y `Point3D` como subclases de `PointND`.
9. Defina una clase `Line2D` que represente una línea a partir de dos puntos.
10. Cree un método `Line2D::getLength(): Double` que calcule la longitud de la línea. *Hint: puede usar el método `PointND::distanceTo(PointND)` para calcular la distancia entre dos puntos.*
11. Implemente el método `Line2D::distanceTo(Point2D): Double` que calcule la distancia entre una línea y un punto. Para esto considere que para toda línea que pase por los puntos p y q , la distancia entre la línea y el punto r se define como:

$$d(p, q, r) = \frac{|(q_y - p_y)r_x - (q_x - p_x)r_y + q_x p_y - q_y p_x|}{\sqrt{(q_y - p_y)^2 + (q_x - p_x)^2}}$$

Ejercicio 11 Algebra Vectorial

1. Un vector euclideano es un objeto matemático que tiene una magnitud y una dirección. Otra forma de pensar en un vector es como una flecha que va desde el origen del sistema de coordenadas hasta un punto en el espacio. Defina una clase `Vector3D` que represente un vector en el plano.
2. Defina un método `Vector3D::toString(): String` que devuelva una representación del vector en el formato `Vector3D(x, y, z)`.
3. Dos vectores son iguales si sus coordenadas son iguales. Defina un método `Vector3D::equalTo(Vector3D): Boolean` que determine si dos vectores son iguales.
4. La suma de dos vectores se define como:

$$\mathbf{v} + \mathbf{w} = (\mathbf{v}_x + \mathbf{w}_x, \mathbf{v}_y + \mathbf{w}_y, \mathbf{v}_z + \mathbf{w}_z)$$

Defina un método `Vector3D::plus(Vector3D): Vector3D` que calcule la suma de dos vectores.

5. La resta de dos vectores se define como:

$$\mathbf{v} - \mathbf{w} = (\mathbf{v}_x - \mathbf{w}_x, \mathbf{v}_y - \mathbf{w}_y, \mathbf{v}_z - \mathbf{w}_z)$$

Defina un método `Vector3D::minus(Vector3D): Vector3D` que calcule la resta de dos vectores.

6. El producto de un vector por un escalar se define como:

$$\mathbf{v} \cdot \alpha = (\alpha \mathbf{v}_x, \alpha \mathbf{v}_y, \alpha \mathbf{v}_z)$$

Defina un método `Vector3D::times(Double): Vector3D` que calcule el producto de un vector por un escalar.

7. El largo (magnitud o norma) de un vector se define como:

$$||\mathbf{v}|| = \sqrt{\mathbf{v}_x^2 + \mathbf{v}_y^2 + \mathbf{v}_z^2}$$

Defina un método `Vector3D::length(): Double` que calcule el largo de un vector.

8. Normalizar un vector es el proceso de convertirlo en un vector unitario. Un vector unitario es aquel cuyo largo es 1. Para normalizar un vector, se divide cada coordenada por el largo del vector. Defina un método `Vector3D::normalize(): Vector3D` que devuelva un vector unitario con la misma dirección que el vector original.

9. El producto punto entre dos vectores se define como:

$$\mathbf{v} \cdot \mathbf{w} = v_x w_x + v_y w_y + v_z w_z$$

Defina un método `Vector3D::dot(Vector3D): Double` que calcule el producto punto entre dos vectores.

10. El producto cruz entre dos vectores se define como:

$$\mathbf{v} \times \mathbf{w} = (v_y w_z - v_z w_y, v_z w_x - v_x w_z, v_x w_y - v_y w_x)$$

Defina un método `Vector3D::cross(Vector3D): Vector3D` que calcule el producto cruz entre dos vectores.

11. El ángulo entre dos vectores se define como:

$$\theta = \arccos \left(\frac{\mathbf{v} \cdot \mathbf{w}}{||\mathbf{v}|| ||\mathbf{w}||} \right)$$

Defina un método `Vector3D::angleTo(Vector3D): Double` que calcule el ángulo entre dos vectores. *Hint: puede importar `kotlin.math.acos` para calcular el arcocoseno.*

12. Defina un método `Vector3D::perpendicularTo(Vector3D): Boolean` que determine si dos vectores son perpendiculares. Para esto considere que dos vectores son perpendiculares si y solo si su producto punto es cero.
13. Defina un método `Vector3D::oppositeTo(Vector3D): Boolean` que determine si dos vectores son opuestos. Para esto considere que dos vectores son opuestos si y solo si

$$\mathbf{v}_x = -\mathbf{w}_x \wedge \mathbf{v}_y = -\mathbf{w}_y \wedge \mathbf{v}_z = -\mathbf{w}_z$$

14. El vector cero es aquel que tiene todas sus coordenadas en cero. Defina un método `Vector3D::isZero(): Boolean` que determine si un vector es el vector cero.
15. Defina un método `Vector3D::parallelTo(Vector3D): Boolean` que determine si dos vectores son paralelos. Para esto considere que dos vectores son paralelos si y solo si su producto cruz es el vector cero.

Bibliografía

- | | |
|--|---|
| Built-in Types and Their Semantics - Kotlin Language Specification | BuiltinTypesTheira |
| <i>Built-in Types and Their Semantics - Kotlin Language Specification</i> . URL: https://kotlinlang.org/spec/built-in-types-and-their-semantics.html#built-in-integer-types-builtins (visitado 08-02-2023). | |
| Encapsulation (Computer Programming) | EncapsulationComputerProgramming2023 |
| <i>Encapsulation (Computer Programming)</i> . En: Wikipedia. 18 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Encapsulation_(computer_programming)&oldid=1134481916 (visitado 04-02-2023). | |
| Anot.: Page Version ID: 1134481916. | |
| Janssen: OOP Concepts for Beginners | janssenOOPConceptsBeginners2018 |
| Thorben Janssen. <i>OOP Concepts for Beginners: What Is Composition?</i> Stackify. 16 de ene. de 2018. URL: https://stackify.com/oop-concepts-composition/ (visitado 08-02-2023). | |
| Metz: Chapter 2: Designing Classes with a Single Responsibility | metzChapterDesigningClasses2013 |
| Sandi Metz. «Chapter 2: Designing Classes with a Single Responsibility». En: <i>Practical Object-oriented Design in Ruby: An Agile Primer</i> . Addison-Wesley, 2013, págs. 15-34. ISBN: 978-0-321-72133-4. Google Books: rk9sAQAAQBAJ . | |
| MutableList - Kotlin Programming Language | MutableListKotlinProgramming |
| <i>MutableList - Kotlin Programming Language</i> . Kotlin. URL: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/index.html (visitado 08-02-2023). | |
| MutableMap - Kotlin Programming Language | MutableMapKotlinProgramming |
| <i>MutableMap - Kotlin Programming Language</i> . Kotlin. URL: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/index.html (visitado 08-02-2023). | |
| Overload Resolution - Kotlin Language Specification | OverloadResolutionKotlin |
| <i>Overload Resolution - Kotlin Language Specification</i> . URL: https://kotlinlang.org/spec/overload-resolution.html#overload-resolution (visitado 08-02-2023). | |
| Shvets: SOLID Principles | shvetsSOLIDPrinciples2021 |
| Alexander Shvets. «SOLID Principles». En: <i>Dive Into Design Patterns</i> . 2021, págs. 51-70. | |

Capítulo 5

Cuando todo no funciona

Como ya vimos, cuando desarrollamos aplicaciones, la complejidad de las mismas va creciendo a medida que se van agregando nuevas funcionalidades. Esto hace que el código se vuelva más difícil de entender y de mantener. Además, cuando se agregan nuevas funcionalidades, es muy común que se rompan las funcionalidades ya existentes D: Esto es un problema, ya que no podemos estar seguros de que las funcionalidades que ya existen siguen funcionando correctamente.

En este capítulo vamos a ver cómo podemos pensar nuestras aplicaciones de una manera que nos permita encontrar estos problemas lo antes posible, y cómo podemos arreglarlos.¹

El *Test-Driven Development* (TDD) es una técnica de desarrollo de software en la que vamos a tratar de probar nuestros programas antes de escribirlos.

¿Pero a qué me refiero con *probar* un programa?

Cuando hablamos de probar un programa, nos referimos a que vamos a ejecutar el programa con distintos valores de entrada, y vamos a verificar que el programa nos devuelva los valores esperados. Por ejemplo, si tenemos una función que suma dos números, podemos probarla con distintos pares de números, y verificar que la suma de esos números sea la que esperamos. Aquí diremos que cada par de números es un *caso de prueba*. Más formalmente, un caso de prueba es una especificación de los valores de entrada, condiciones de ejecución y valores de salida esperados de un programa.

En el caso de la función de suma, podríamos escribir los siguientes casos de prueba:

- Sumar 1 y 2, y verificar que el resultado sea 3.
- Sumar 2 y 2, y verificar que el resultado sea 4.
- Sumar 0 y 0, y verificar que el resultado sea 0.
- Sumar 1 y 0, y verificar que el resultado sea 1.

En el caso de la función de suma, los casos de prueba son muy simples, pero en la práctica, los casos de prueba pueden ser mucho más complejos.

Entonces, cuando hablamos de probar un programa, nos referimos a escribir los casos de prueba y registrar el resultado de ejecutar estos casos.

¹O al menos empezar a arreglarlos.

Test-Driven Development

El *Test-Driven Development* (TDD) es una metodología de desarrollo de software en la que los requisitos del software son traducidos a casos de prueba antes de escribir el código del programa.

El TDD se separa en tres pasos:

1. Escribir unos pocos casos de prueba y verificar que todos fallen.²
2. Escribir el código mínimo necesario para que el caso de prueba pase.
3. Refactorizar³ el código para que sea más legible y fácil de mantener.

El TDD es difícil de explicar, así que en lugar de aburrirles con una explicación teórica extensa, vamos a ir desarrollando todas las soluciones de aquí en adelante usando TDD. Así, vamos a ir viendo cómo funciona el TDD en la práctica.

¿Pero por qué deberíamos usar TDD? Existen muchas razones para usar TDD, pero mencionemos tres que considero importantes:

- Fallar rápido: Si escribimos los casos de prueba antes de escribir el código, vamos a encontrar los errores lo antes posible, y vamos a poder arreglarlos antes de que se vuelvan más difíciles de arreglar.
- Evitar imparcialidad en los tests: Si escribimos los tests después de escribir el código, es muy probable que los tests estén sesgados a favor del código que acabamos de escribir. Esto es un problema, ya que no podemos estar seguros de que los tests realmente estén verificando que el código funciona correctamente. Aquí nos importa el comportamiento esperado del código, no cómo está implementado.
- Documentación: Los tests son una forma de documentar el código. Si escribimos los tests antes de escribir el código, podemos usarlos como una forma de especificar el comportamiento del código.

5.1. Testing frameworks

Un *framework* es una abstracción en la que un software provee una funcionalidad genérica que puede ser extendida/-modificada por el usuario, obteniendo así un software específico para un problema. Podemos pensar en un framework como un esqueleto de un programa, que podemos completar con nuestras propias ideas.

Los *testing frameworks* son frameworks que nos permiten escribir casos de prueba para nuestros programas. Actualmente existen *testing frameworks* para casi todos los lenguajes de programación utilizados en la industria.

En este libro vamos a utilizar el framework *Kotest* para escribir los tests.

5.2. Gradle

Un *build system* es una colección de herramientas que nos permiten automatizar el proceso de compilación de un programa. Existen muchos *build systems* disponibles, pero en este libro vamos a utilizar *Gradle*.

Comencemos por crear un nuevo proyecto para trabajar en este capítulo. El proyecto que vamos a crear será un juego inspirado en *Pokémon*⁴ al que llamaremos *Bucket Monsters* o *Bakémon*.

Para crear el proyecto, vamos a utilizar *IntelliJ*. En la barra de herramientas, seleccionamos *File* → *New* → *Project...*. En la ventana que aparece, en la pestaña *New Project*, crearemos un proyecto llamado *bakemon*, marcaremos la opción *Create Git repository* y seleccionaremos *Gradle* como *build system*. Luego, en la sección *Advanced settings*, pondremos *cl.ravenhill* como *GroupId* y *bakemon* como *ArtifactId* (figura 5.1).

²Si escribimos tests que pasan antes de implementar la funcionalidad, hay algo *sus* en nuestros tests.

³Refactorizar es una técnica de desarrollo de software que consiste en reescribir el código sin cambiar su comportamiento.

⁴Actualmente propiedad de The Pokémon Company

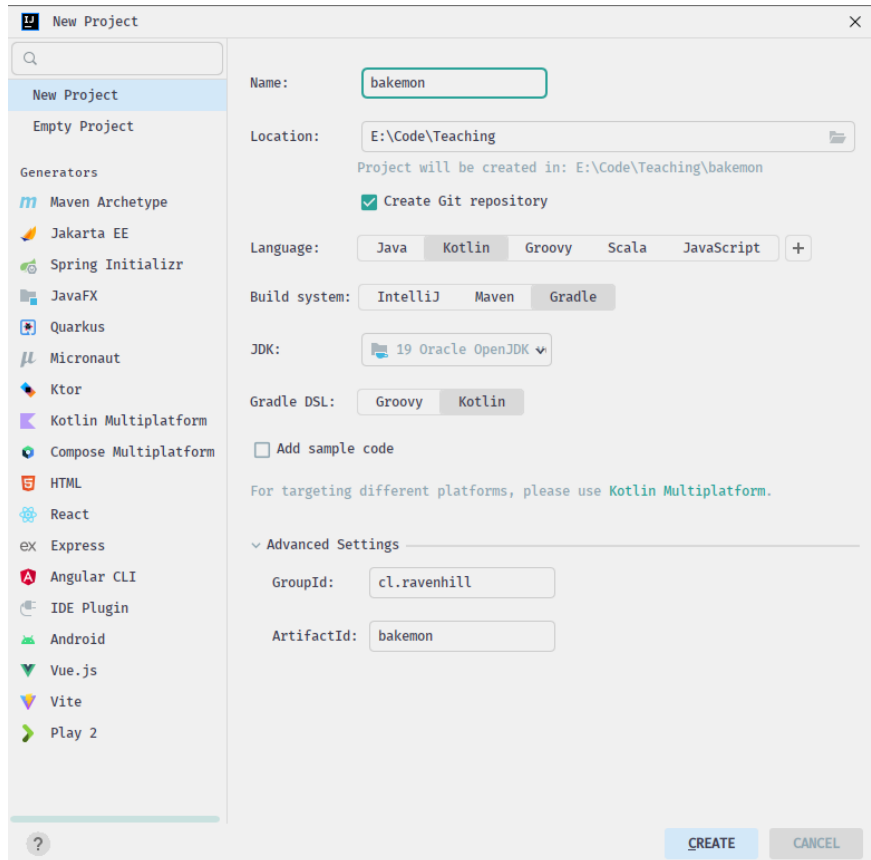


Figura 5.1: Creando un proyecto Gradle en IntelliJ

Es probable que al crear el proyecto *IntelliJ* nos reclame por la versión del JDK (figura 5.2), eso lo solucionaremos un poco más adelante, por ahora ignorémoslo y continuemos.



Figura 5.2: Warning de Gradle por la versión del JDK

Esto nos creará un proyecto Gradle con lo siguiente:

- Un directorio `.git` que contiene la información del repositorio Git.
- Un directorio `.gradle` donde se almacenan las dependencias del proyecto.
- Una carpeta `gradle` que contiene archivos que *IntelliJ* utiliza para ejecutar los comandos de Gradle.
- Un archivo `.gitignore` con algunas reglas por defecto.

- Los archivos `build.gradle.kts` y `settings.gradle.kts` que contienen la configuración del proyecto.
- Un archivo `gradle.properties` que contiene las propiedades del proyecto.
- Los archivos `gradlew` y `gradlew.bat` que son los ejecutables de Gradle.
- Los directorios `.idea` y `src` que ya conocíamos.⁵

Por ahora ignoraremos la mayoría de estos archivos, y nos centraremos en corregir el warning que nos apareció al crear el proyecto. Este warning se debe a que *IntelliJ* creó el proyecto con una versión de *Gradle* que no es compatible con las versiones más nuevas del JDK. Solucionarlo es simple, primero iremos a <https://services.gradle.org/distributions/>, ahí podremos ver las versiones de *Gradle* disponibles. Una vez ahí, buscaremos el archivo correspondiente a la versión de *Gradle* más reciente, en mi caso `gradle-8.0-rc-3-bin.zip` (es importante que sea la versión que termine en `-bin.zip`). Copien el nombre del archivo y busquen el archivo `gradle/wrapper/gradle-wrapper.properties` dentro de su proyecto. Ahí, en la línea que dice `distributionUrl=`, reemplacen el nombre del archivo por el que copiaron antes. Los contenidos del archivo deberían quedar así:

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-8.0-rc-3-bin.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

Lo siguiente será hacerle *build* al proyecto para que se descargue la versión de *Gradle* que acabamos de especificar.

Para esto, utilizaremos la herramienta *Search Everywhere* de *IntelliJ* y buscaremos *Activate Gradle Window*, ahí debería aparecernos una única opción llamada *Gradle*. Si le hacemos click se abrirá una pestaña como la de la figura 5.3. Ahí, en la barra superior, seleccionaremos la opción *Execute Gradle Task* y luego escribiremos `build` en el campo de texto que aparece (figura 5.4). Y ahora, a esperar...

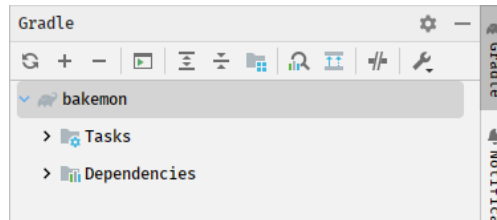


Figura 5.3: La ventana de Gradle

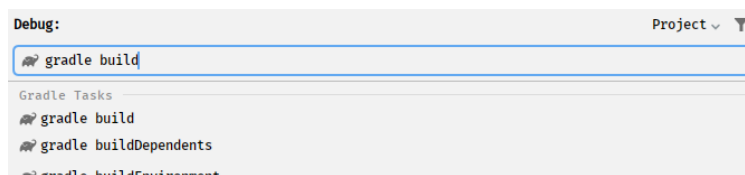


Figura 5.4: Ejecutando el comando build

Una vez que el comando termine, nos debería aparecer un mensaje como el de la figura 5.5.

Con eso solucionamos el warning. Ahora, revisemos el archivo `build.gradle.kts` que se creó junto con el proyecto. Este archivo contiene la configuración del proyecto, y es el que utilizaremos para agregar las dependencias que necesitamos.

⁵Noten que el archivo `bakemon.iml` no se crea ya que ese archivo pertenece al *build system* nativo de *IntelliJ*.

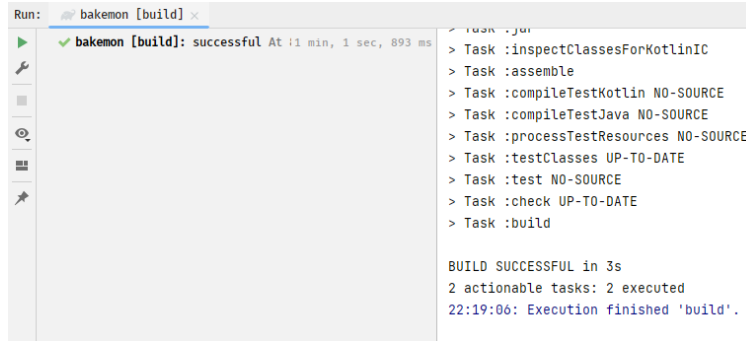


Figura 5.5: El comando build terminó exitosamente

Revisemos el archivo por partes. Primero, en la parte superior, encontramos la línea `plugins` que contiene la configuración de los plugins que utilizaremos. En este caso, sólo tenemos el plugin `kotlin` que es el que nos permite utilizar *Kotlin* en el proyecto.

```
plugins {
    kotlin("jvm") version "1.8.0"
}
```

Luego tenemos dos líneas que identifican nuestro proyecto, la primera es el nombre del grupo (generalmente el nombre de la organización) y la segunda es la versión del proyecto. Debieran verse así:

```
group = "cl.ravenhill"
version = "1.0-SNAPSHOT"
```

Luego tenemos la configuración de las dependencias del proyecto. Una dependencia es un archivo que contiene código que nos permite utilizar funcionalidades que no están incluidas en el lenguaje ni en nuestro proyecto, i.e. una librería. Para esto, primero debemos especificar el repositorio de donde se descargarán las dependencias, en este caso, utilizaremos el repositorio de *Maven Central*⁶ que es el repositorio más grande de dependencias de *Java* y *Kotlin*. Luego debemos especificar las dependencias que queremos utilizar, en este caso, el proyecto viene configurado con la dependencia de testing de *Kotlin*.

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation(kotlin("test"))
}
```

Aquí, la función `testImplementation` es una función que nos permite especificar dependencias de testing, esto hará que las dependencias que especifiquemos sólo serán accesibles dentro de los tests.

A continuación debemos indicar el motor de testing que utilizaremos, en este caso, utilizaremos *JUnit*.

⁶*Maven Central Repository Search.*

```
tasks.test {
    useJUnitPlatform()
}
```

Por último tenemos la configuración del plugin de *Kotlin*, en este caso, le decimos que la versión del JDK que utilizaremos será la 8.

```
kotlin {
    jvmToolchain(8)
}
```

Ahora, hagamos unos pequeños cambios al archivo para adaptarlo a nuestro proyecto. Primero, cambiemos la versión del JDK a la que estemos utilizando, en mi caso, la 19.

```
kotlin {
    jvmToolchain(19)
}
```

Luego, borremos la dependencia de testing de *Kotlin* ya que no la utilizaremos. Y por último, agreguemos las dependencias a *Kotest*:

```
dependencies {
    testImplementation("io.kotest:kotest-runner-junit5:5.5.5")
    testImplementation("io.kotest:kotest-assertions-core:5.5.5")
}
```

Veamos qué es lo que acabamos de hacer. Primero, agregamos la dependencia principal de *Kotest* que es la que nos permite utilizar el framework de testing. Luego, agregamos la dependencia de *Kotest* que nos permite utilizar las aserciones de *Kotest*, una aserción es una función que nos permite verificar que una condición se cumpla.

Ahora, hagamos *build* nuevamente para que se descarguen las dependencias que acabamos de agregar. No vamos a entrar en más detalles de cómo funciona *Gradle* ya que no es el objetivo de este libro y porque no vamos a utilizar nada más avanzado que lo que ya vimos, pero lo que debemos recordar es que:

- *Gradle* es un *build system* que nos permite automatizar tareas de compilación, testing, etc.
- *Gradle* utiliza un archivo de configuración llamado `build.gradle.kts` que contiene la configuración del proyecto.
- Cada vez que modificamos los archivos de configuración de *Gradle* debemos ejecutar el comando `build` para que se apliquen los cambios.

Ahora nos queda modificar el `.gitignore` para que se adapte mejor a nuestro proyecto. ¿Pero qué agregamos al `.gitignore`? Cuando tenemos proyectos más grandes, es complicado saber qué archivos debemos agregar al `.gitignore`, por suerte, este es un problema con el que ya se han topado otros y existen herramientas que nos ayudan a generar un `.gitignore` adecuado para nuestro proyecto.

La herramienta que utilizaremos es un plugin de *IntelliJ*.⁷ Para instalar el plugin, abriremos el menú *File* y seleccionaremos la opción *Settings...* (figura 5.6). Luego, en la ventana que se abrirá, seleccionaremos la opción *Plugins* y en la pestaña *Marketplace* buscaremos `.ignore`, una vez lo encontremos podemos instalarlo (figura 5.7).

⁷Distinto a un plugin de *Gradle*.

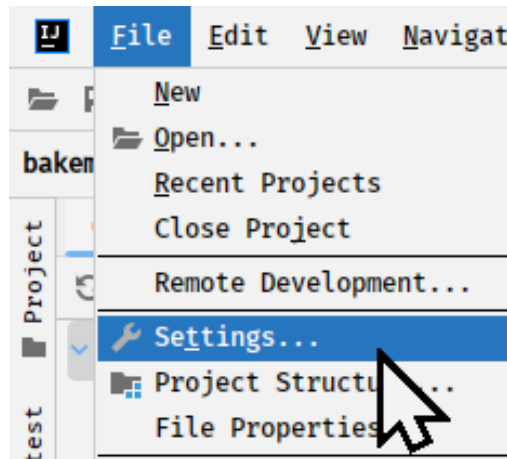


Figura 5.6: Menú de Settings

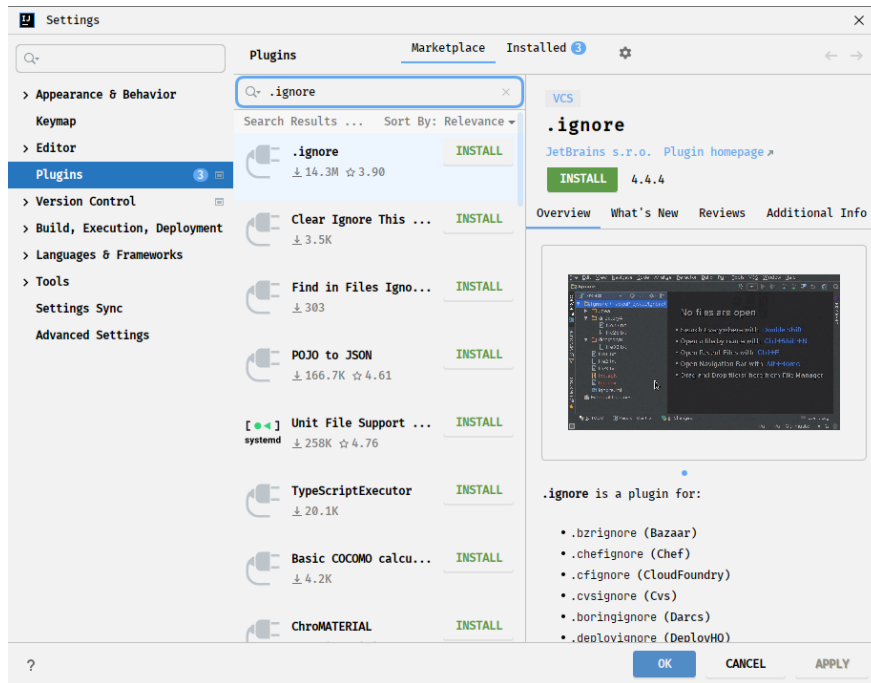


Figura 5.7: Instalación del plugin `.ignore`

(maybe reiniciar intellij)

Una vez instalado el plugin, utilizaremos *Search Everywhere* para buscar la opción `.ignore File` (figura 5.8).⁸ Luego, en la ventana que se abrirá, seleccionaremos la opción `.gitignore File` (figura 5.9). Esto abrirá otra ventana en la que podremos elegir los templates para los lenguajes, ambientes y herramientas que utilizaremos en nuestro proyecto. Aquí marcaremos las opciones *Kotlin*, *Gradle*, *JetBrains* y el sistema operativo que estemos utilizando (figura 5.10). Esto modificará el archivo `.gitignore` para agregar las reglas correspondientes a las opciones que seleccionamos.

Con esto, podemos hacer *commit* de los cambios que hicimos al proyecto.

⁸Asegurémonos de tener seleccionada la carpeta principal del proyecto, o el archivo se creará en otra carpeta.

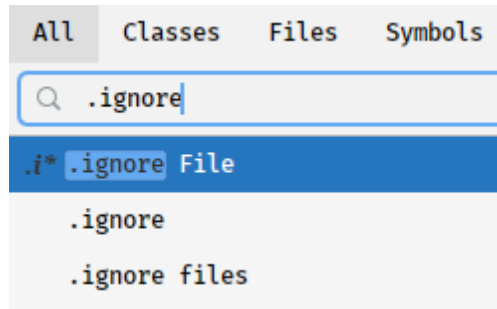


Figura 5.8: Opción *.ignore File*

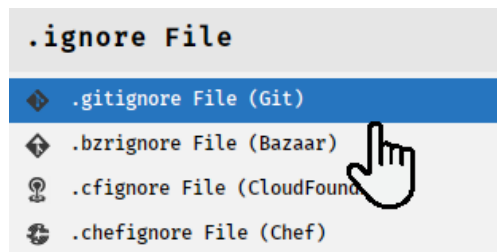


Figura 5.9: Opción *.gitignore File*

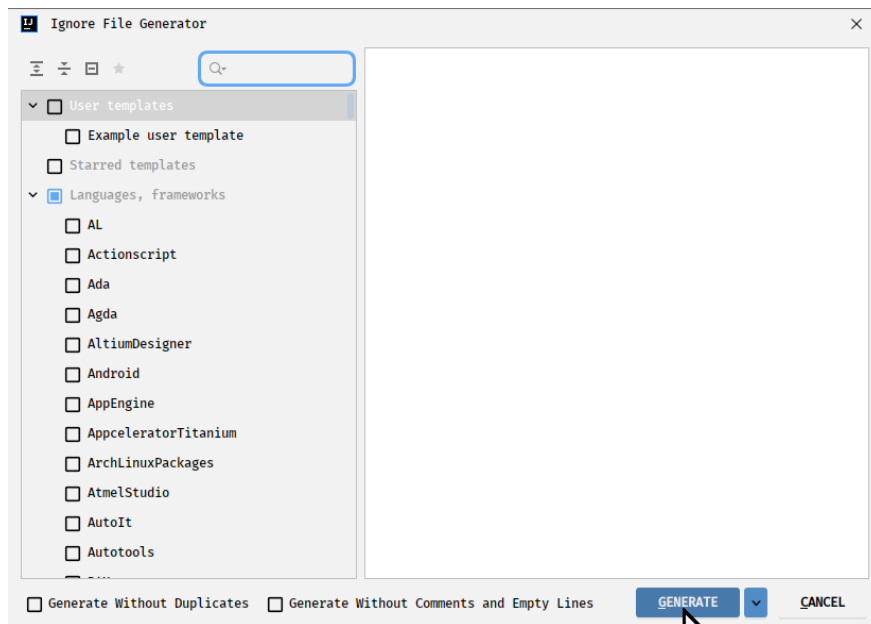


Figura 5.10: Generación del *.gitignore*

```
git add .
git commit -m "PROJECT Adds Gradle configuration"
```

5.3. Conociendo a *Kotest*

Ahora veremos cómo utilizar *Kotest* para testear nuestras aplicaciones siguiendo una metodología de desarrollo basada en pruebas. Para esto, comencemos por definir lo más básico de nuestro juego.

Pero antes, instalaremos un nuevo plugin de *IntelliJ* que nos permitirá sacarle más provecho a *Kotest*. Para esto, vayamos a *File* → *Settings* → *Plugins*, y busquemos *Kotest* en el *Marketplace*. Una vez instalado, *IntelliJ* nos preguntará si queremos reiniciar el IDE, hagámosle caso.

Cuando desarrollamos aplicaciones, las cosas se pueden complicar muy rápido. Es por esto que es muy importante partir siempre por lo más simple. En este caso, vamos a comenzar con una clase que represente a un *Bakémon*. ¿Pero qué es lo más simple? Simple, una clase que represente a un *Bakémon* sin ningún método ni atributo. Así que vamos a comenzar por ahí.

Primero, busquemos el directorio `src/test/kotlin` y creemos un nuevo paquete llamado `cl.ravenhill.bakemon`. Ahora, dentro de este paquete, creemos una nueva clase llamada *BakemonTest*. Aquí es probable que *IntelliJ* nos pregunte si queremos agregar el archivo a *Git*, esto es equivalente a hacer *git add* en la línea de comandos, así que hagámosle caso.

Ahora, deberíamos tener una clase que se ve así:

```
package cl.ravenhill.bakemon

class BakemonTest {
}
```

A continuación necesitamos decirle al framework que esta clase es un test. Para esto, heredaremos de la clase *FunSpec* de la siguiente forma:

```
package cl.ravenhill.bakemon

import io.kotest.core.spec.style.FunSpec

class BakemonTest : FunSpec({})
```

Algo que notar aquí es que le estamos pasando `{}` como argumento al constructor de la clase *FunSpec*. Esto se conoce como **función lambda** y es una forma de pasar un bloque de código como argumento a una función o constructor, por ahora no nos preocupemos por esto, más adelante veremos en más detalle funciones lambda.

Ahora vamos a crear nuestro primer test. Para esto, primero pensemos en qué es lo que podemos testear de una clase vacía. En este caso, lo primero que queremos ver es que el constructor funcione correctamente. ¿Pero cómo testeamos que un constructor de una clase vacía funciona correctamente? La solución dependerá de la clase que estemos testeando, pero una solución común sería ver que dos objetos creados a partir de la misma clase son iguales. Lo siguiente es poner en palabras lo que queremos testear, en este caso, podemos decir: *Dos objetos creados a partir de la misma clase deben ser iguales*. Una vez hemos planteado nuestro test, podemos agregar ese test a nuestra clase de la siguiente forma:

```
class BakemonTest : FunSpec({
    test("Two objects created from the same class should be equal") {}
})
```

Y con eso tenemos nuestro primer test. Probemos qué sucede si lo ejecutamos. Para esto, podemos hacer click al botón «play» al lado izquierdo de la definición de la clase y seleccionando la opción «Run 'BakemonTest'». Esperamos un poco

a que se corran los tests y podemos ver que el test que acabamos de crear pasa (figura 5.11). ¡Felicidades! Tenemos el primer error de este capítulo.

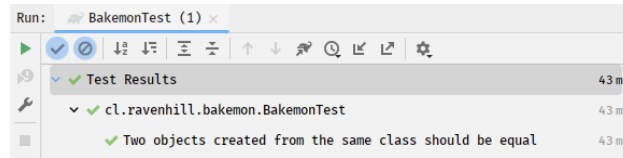


Figura 5.11: Primer test

Como dijimos antes, el primer paso del TDD es crear un test que falle. En este caso es fácil ver por qué nos pasó esto. El test que acabamos de crear no tiene ninguna aserción, por lo que siempre va a pasar. Solucionar esto es simple, simplemente agreguemos una aserción que verifique que dos objetos creados a partir de la misma clase son iguales. Para esto, utilizaremos la función *shouldBe* que verifica que dos objetos sean iguales.

```
test("Two objects created from the same class should be equal") {  
    val a = Bakemon()  
    val b = Bakemon()  
    a shouldBe b  
}
```

Ahora, si tratamos de correr el test recibiremos un error diciendo «*Unresolved reference: Bakemon*». Esto es porque la clase *Bakemon* no existe. Pero lo importante es que esto significa que nuestro test falla.

¿Yay?

Con esto podemos pasar al siguiente paso del TDD, que es crear el código mínimo necesario para que el test pase. Creemos un paquete `cl.ravenhill.bakemon` en el directorio `src/main/kotlin` y dentro de ese paquete creemos una clase llamada *Bakemon*. De nuevo, esto nos creará una clase vacía:

```
package cl.ravenhill.bakemon  
  
class Bakemon {  
}
```

Ahora, si corremos el test, veremos que sigue fallando pero con un error distinto. En este caso obtendremos lo siguiente:

```
expected:<cl.ravenhill.bakemon.Bakemon@4578b1f>  
but was:<cl.ravenhill.bakemon.Bakemon@27559659>  
Expected :cl.ravenhill.bakemon.Bakemon@4578b1f  
Actual   :cl.ravenhill.bakemon.Bakemon@27559659  
<Click to see difference>
```

¿Por qué pasa esto? ¿No son ambos objetos iguales? La respuesta es que sí, pero *Kotlin* no tiene forma de saberlo, ya que no le hemos dicho bajo qué criterio dos objetos son iguales.

Otra cosa que podemos notar es que el error utiliza un nombre poco intuitivo para nuestras clases. Esto es lo primero que solucionaremos. Puede que ya se hayan dado cuenta de que el nombre que utiliza *Kotest* para nuestros objetos es muy parecido al que vimos en el apartado 4.3.2, y la causa es la misma: no hemos sobrescrito el método `Any::toString()`: `String` para nuestras clases. Hagamos eso:


```
class Bakemon {
    override fun toString(): String {
        return "Bakemon"
    }
}
```

Ahora, si corremos el test, tendremos un mensaje más amigable:

```
expected:<Bakemon> but was:<Bakemon>
Expected :Bakemon
Actual   :Bakemon
<Click to see difference>
```

Lo siguiente, es decirle a *Kotlin* cómo comparar dos objetos de la clase *Bakemon*. Pero para esto entendamos primero qué significa que dos objetos sean iguales.

En *Kotlin* existen dos maneras de verificar si dos objetos son iguales: igualdad referencial e igualdad estructural.

La **igualdad referencial** se refiere a que dos objetos son iguales si y sólo si ocupan el mismo espacio en memoria. Esto significa que si tenemos dos variables que referencian al mismo objeto, entonces esos dos objetos son iguales. En *Kotlin*, la igualdad referencial se verifica con el operador `===`.

Por otro lado, la **igualdad estructural** se refiere a que dos objetos son iguales si y sólo si los atributos que lo identifican son iguales. A diferencia de la igualdad referencial, esta condición debemos definirla nosotros. En *Kotlin*, la igualdad estructural se verifica con el operador `==`. Existen dos métodos que definen la igualdad estructural en *Kotlin*:

- `Any::equals(Any?): Boolean` que verifica que dos objetos sean iguales de acuerdo a la igualdad estructural.
- `Any::hashCode(): Int` que retorna un número entero que identifica a un objeto. Este número generalmente no lo utilizaremos explícitamente, pero la librería estándar de *Kotlin* espera que si dos objetos son iguales de acuerdo al método `equals`, entonces sus códigos de hash deben ser iguales.

Importante. Si no definimos la igualdad estructural explícitamente, entonces *Kotlin* utilizará la igualdad referencial.

Importante. El método `shouldBe` espera que dos objetos sean iguales de acuerdo a la igualdad estructural.

Nota. El método `equals` recibe un parámetro de tipo `Any?` y no de tipo `Any`. `Any?` es el supertipo de todos los tipos de *Kotlin*, mientras que `Any` es la superclase de todos los objetos en *Kotlin*. Esta diferencia es importante, ya que no todo en *Kotlin* es un objeto, pero no nos detendremos en eso por ahora.

La igualdad estructural siempre se define de la misma forma:

- Verificar si el objeto con el que estamos comparando es igual referencialmente. Esto lo hacemos para mejorar el rendimiento de la comparación.
- Verificar si el objeto con el que estamos comparando es de la misma clase. Esto lo hacemos porque dos objetos de distinta clase podrían tener los mismos atributos.
- Verificar si los atributos que identifican a los objetos son iguales.

En nuestro caso, la clase *Bakemon* no tiene atributos, por lo que no haremos la tercera verificación. Traduzcamos esto a código:

```
override fun equals(other: Any?): Boolean {
    return if (other === this) {
        true
    }
```

```

    } else {
        other is Bakemon
    }
}

```

Ahora, si corremos el test, veremos que pasa.

Nos falta implementar el método `hashCode`, pero recuerden que estamos aplicando el TDD, por lo que primero debemos crear un test que falle. Hagamos eso:

```

test("Two objects created from the same class should have the same hashCode") {
    val a = Bakemon()
    val b = Bakemon()
    a shouldHaveSameHashCodeAs b
}

```

Ahora, si corremos el test, veremos que falla. Con esto, podemos implementar el método `hashCode`. Como dijimos, el método `hashCode` debe ser consistente con la igualdad estructural, esto significa que debemos considerar qué identifica a un objeto. Para esto, revisemos el método `equals` que acabamos de implementar. ¿Qué identifica a un objeto de la clase *Bakemon*? Podríamos pensar que nada, ya que no tiene atributos, pero no es así. Hay una cosa que identifica a un objeto de la clase *Bakemon*, y es su clase. Por lo tanto, podemos implementar el método `hashCode` de la siguiente forma:

```

override fun hashCode(): Int {
    return Objects.hash(Bakemon::class)
}

```

Noten que en lugar de definir nuestra propia manera de calcular el código de hash, estamos utilizando la función `Objects.hash` que está definida en la librería estándar de *Kotlin*.

Ahora, si corremos el test, veremos que pasa.

Para terminar, haremos *commit* de los cambios que hemos hecho hasta ahora:

```

git add .
git commit -m "FEAT Adds toString, equals and hashCode to Bakemon"

```

5.4. Segunda cita con *Kotest*

Ahora que ya escribimos nuestros primeros tests, empecemos a complicar un poco las cosas. Digamos ahora que todos los *Bakémon* tienen un nombre, puntos de salud, nivel y puntos de ataque.

Comencemos modificando los tests que ya escribimos para que tengan en cuenta estos nuevos campos:

```

test("Two objects created from the same class should be equal") {
    val a = Bakemon("Kokodile", 25, 5, 4)
    val b = Bakemon("Kokodile", 25, 5, 4)
    a shouldBe b
}

```

```

}
test("Two objects created from the same class should have the same hashCode") {
    val a = Bakemon("Kokodile", 25, 5, 4)
    val b = Bakemon("Kokodile", 25, 5, 4)
    a shouldHaveSameHashCodeAs b
}

```

Como esperaríamos, esto falla porque no hemos definido los campos en el constructor. El siguiente paso, es escribir el código mínimo necesario para que los tests pasen. En este caso, debemos modificar el constructor de Bakemon para que reciba los nuevos parámetros y los guarde en las propiedades correspondientes, y también debemos modificar los métodos equals(), hashCode() y toString() para que tengan en cuenta los nuevos campos.

Primero, notemos que los parámetros que agregamos al constructor están resaltados en rojo. Esto es porque el constructor de Bakemon no acepta parámetros. Para solucionar esto, hagamos click derecho sobre alguno de los parámetros en rojo y seleccionemos la opción *Show Context Actions* (figura 5.12), y luego, en el menú que aparece, seleccionemos la opción *Add parameters to constructor 'Bakemon'* (figura 5.13). Esto abrirá la ventana de la figura 5.14, ahí podemos darle nombre a los parámetros haciendo click sobre el parámetro, démosle los nombres name, health, level y attack.

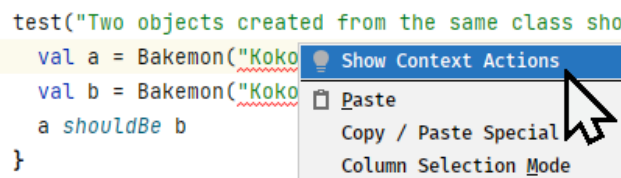


Figura 5.12: Mostrar acciones de contexto

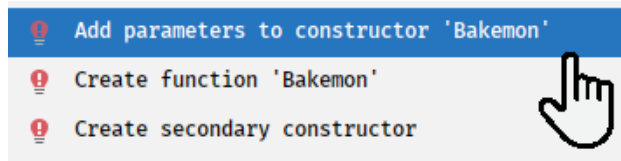


Figura 5.13: Agregar parámetros al constructor

Perfecto, nuestro constructor ahora recibe parámetros. Lo siguiente que debemos hacer es guardar los parámetros en las propiedades correspondientes. Hagamos eso:

```

class Bakemon(name: String, health: Int, level: Int, attack: Int) {
    val name = name
    val health = health
    val level = level
    val attack = attack
    ...
}

```

Ahora, agreguemos los parámetros a los métodos equals(), hashCode() y toString().



Figura 5.14: Cambiar firma del constructor

```
override fun toString(): String {
    return "Bakemon(name='$name', health=$health, level=$level, attack=$attack)"
}

override fun equals(other: Any?): Boolean {
    return when {
        this === other -> true
        other !is Bakemon -> false
        else -> name == other.name &&
            health == other.health &&
            level == other.level &&
            attack == other.attack
    }
}

override fun hashCode(): Int {
    return Objects.hash(Bakemon::class, name, health, level, attack)
}
```

Ahora sí, los tests deberían pasar.

Como siempre, terminemos haciendo *commit* de los cambios.

```
git add .
git commit -m "FEAT Adds name, health, level and attack to Bakemon"
```

5.5. Mejorando nuestro código

Hasta ahora hicimos los pasos 1 y 2 del TDD, nos falta el paso 3: **refactorizar**. Aquí nos centraremos en mejorar la calidad de nuestro código sin cambiar su comportamiento, esto significa que nuestros tests deben seguir pasando.

Primero comenzaremos por cambiar nuestros tests porque, aunque no lo crean, tienen algunos problemas. El primero es que estamos repitiendo las llamadas al constructor en cada test, sería ideal hacer este llamado en un sólo lugar y luego usar el objeto que se crea en cada test (recordemos que queremos encapsular el cambio). Una primera idea sería definir una variable común para todos los tests e inicializarla una sola vez en el cuerpo de la función lambda.

```
class BakemonTest : FunSpec({
    val bakemon = Bakemon("Kokodile", 25, 5, 4)

    test("Two objects created from the same class should be equal") {
        val b = Bakemon("Kokodile", 25, 5, 4)
        bakemon shouldBe b
    }
    test("Two objects created from the same class should have the same hashCode") {
        val b = Bakemon("Kokodile", 25, 5, 4)
        bakemon shouldHaveSameHashCodeAs b
    }
})
```

Pero esto tiene un gran problema, como la variable bakemon es común a todos los tests, si un test modifica el estado de bakemon, los otros tests pueden fallar. Una forma de solucionar eso sería resetear el estado de bakemon en cada test, pero con eso volvemos a nuestro problema inicial ya que no estamos encapsulando el cambio.

Para solucionar esto, *Kotest* provee una función llamada `beforeTest`, esta función se ejecuta antes de cada test y en general se usa para inicializar el estado de los objetos que vamos a usar en todos los tests. Es importante notar que `beforeTest` se ejecuta antes de cada test, por lo que si tenemos 10 tests, `beforeTest` se ejecutará 10 veces. Con esto, podemos reescribir nuestros tests de la siguiente forma:

```
class BakemonTest : FunSpec({
    lateinit var bakemon: Bakemon

    beforeTest {
        bakemon = Bakemon("Kokodile", 25, 5, 4)
    }

    test("Two objects created from the same class should be equal") {
        val b = Bakemon("Kokodile", 25, 5, 4)
        bakemon shouldBe b
    }
    test("Two objects created from the same class should have the same hashCode") {
        val b = Bakemon("Kokodile", 25, 5, 4)
        bakemon shouldHaveSameHashCodeAs b
    }
})
```

Pero notarán algo nuevo aquí, la palabra reservada `lateinit`. Esta palabra reservada es una promesa que hacemos a Kotlin de que la variable bakemon va a ser inicializada más adelante, por lo que podemos definirla sin pasarle un valor

inicial.

Ahora, nos queda un último problema, estamos repitiendo mucha información entre los tests. En particular estamos repitiendo el nombre del Bakemon, su nivel, su ataque y su defensa. Esto podría no parecer un problema al principio, pero si tenemos 20 tests es fácil que nos equivoquemos escribiendo alguno de los parámetros, basta que uno de sus dedos se mueva un poquito y estaremos comprobando si *Kokodile* es igual *Kolodile*. La forma más fácil de arreglar esto es definir los parámetros como valores en el cuerpo de la función lambda y luego usarlos en los tests. De esta forma, si queremos cambiar algún parámetro, lo hacemos en un sólo lugar y no tenemos que preocuparnos por cambiarlo en todos los tests, es decir, estamos encapsulando el cambio.

```
class BakemonTest : FunSpec({
    lateinit var bakemon: Bakemon

    val bakemonName = "Kokodile"
    val bakemonHealth = 25
    val bakemonLevel = 5
    val bakemonAttack = 4

    beforeTest {
        bakemon = Bakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
    }

    test("Two objects created from the same class should be equal") {
        val b = Bakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
        bakemon shouldBe b
    }
    test("Two objects created from the same class should have the same hashCode") {
        val b = Bakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
        bakemon shouldHaveSameHashCodeAs b
    }
})
```

Si corremos los tests, debieran seguir pasando.

Ahora que tenemos nuestros tests más limpios, podemos pasar a refactorizar nuestro código.

Lo primero que vamos a hacer es simplificar el constructor de Bakemon. Notemos que el constructor de Bakemon en realidad hace muy poco, solo inicializa las variables de instancia. Cuando lo único que hace el constructor primario de una clase es inicializar las variables de instancia, *Kotlin* provee una sintaxis especial para simplificar el código. En lugar de definir un constructor primario, podemos definir en la lista de parámetros del constructor cuáles de los parámetros inicializarán variables de instancia anteponiendo `var` o `val` a cada parámetro según corresponda, esto creará automáticamente un constructor primario que inicializará las variables de instancia con los valores de los parámetros. De esta forma, podemos reescribir el constructor de Bakemon de la siguiente forma:

```
class Bakemon(val name: String, val health: Int, val level: Int, val attack: Int) {...}
```

Podemos ejecutar los tests y ver que siguen pasando.

Para lo siguiente, vamos a aprovechar que las funciones en *Kotlin* son valores⁹. Cuando una función en *Kotlin* solo tiene una expresión, podemos utilizar una sintaxis más compacta para definirla ¿¿¿y es???? que podemos omitir el cuerpo

⁹Esto significa que *Kotlin* es un lenguaje de primera clase, esto es algo que veremos más adelante.

de la función y escribir la expresión directamente después de los paréntesis de los parámetros como si estuviéramos asignando una variable. Tomando como ejemplo la función `toString` de `Bakemon`, podemos reescribirla de la siguiente forma:

```
override fun toString(): String =  
    "Bakemon(name='$name', health=$health, level=$level, attack=$attack)"
```

Pero leyendo el código se puede inferir que la función `toString` devuelve un `String`, por lo que podemos omitir el tipo de retorno de la función. Es importante que esto lo hagamos solo cuando el tipo de retorno de la función es obvio, ya que si no lo es, puede ser muy confuso para quien lea el código.

Sabiendo esto, podemos reescribir la clase `Bakemon` de la siguiente forma:

```
class Bakemon(val name: String, val health: Int, val level: Int, val attack: Int) {  
  
    override fun toString() =  
        "Bakemon(name='$name', health=$health, level=$level, attack=$attack)"  
  
    override fun equals(other: Any?) = when {  
        this === other -> true  
        other !is Bakemon -> false  
        else -> name == other.name &&  
            health == other.health &&  
            level == other.level &&  
            attack == other.attack  
    }  
  
    override fun hashCode() = Objects.hash(Bakemon::class, name, health, level, attack)  
}
```

Nuevamente, podemos ejecutar los tests y ver que siguen pasando, y hacer *commit* de nuestros cambios.

```
git add .  
git commit -m "REFACTOR Simplifies Bakemon class"
```

Y con esto ya completamos nuestra primera iteración de *Test-Driven Development*. En este capítulo rompimos y reparamos nuestro código múltiples veces, en los capítulos siguientes seguiremos utilizando esta metodología e iremos profundizando más en el tema a medida que nuestros programas se vayan volviendo más complejos, siempre manteniendo la idea de que el código debe ser fácil de entender y de modificar, además de asegurarnos de ir encapsulando el cambio siempre que sea posible.

5.6. Ejercicios

Ejercicio 12 Listas

Considere una clase `IntArrayList` que representa una lista basada en arreglos que implementa la interfaz `IntList`.

```
/**  
 * This interface represents a list of integers.
```

```

*/
interface IntList {
    /**
     * Adds a new value to the list.
     */
    fun add(value: Int)
    /**
     * Gets the value at the given index.
     */
    fun get(index: Int): Int
    /**
     * Removes the value at the given index.
     */
    fun remove(index: Int): Int
    /**
     * Returns the size of the list.
     */
    fun size(): Int
    /**
     * Returns the capacity of the list.
     */
    fun capacity(): Int
}

```

Para su implementación considere que una lista basada en arreglos es una estructura de datos que tiene un arreglo de enteros y dos variables: una que representa el tamaño de la lista y otra que representa la capacidad del arreglo. Puede encontrar más información sobre listas basadas en arreglos en el siguiente video: <https://www.youtube.com/watch?v=pN2BHqWWGHU>

Considere que el constructor de la clase `IntArrayList` se puede utilizar de la siguiente forma:

```
val list = IntArrayList(intArrayOf(1, 2, 3))
```

Recuerde hacer *commit* de su trabajo después de cada pregunta.

1. Cree un proyecto *lists* en *IntelliJ* de la forma que se vió en este capítulo utilizando *Gradle* como sistema de construcción y agregue las dependencias de *Kotest*.
2. Cree una clase `IntArrayTest` que extienda de la clase `FunSpec`.
3. Escriba un test que verifique que el tamaño de una lista vacía es cero.
4. Escriba un test que verifique que el tamaño de una lista es el número de elementos con los que se inicializó.
5. Escriba un test que verifique que la capacidad de una lista vacía es 1.
6. Escriba un test que verifique que la capacidad de una lista es el doble del tamaño del arreglo con el que se inicializó.
7. Escriba un test que verifique que se pueden obtener elementos de una lista.
8. Escriba un test que verifique que el tamaño de listas creadas con 1 y 2 elementos es 1 y 2 respectivamente.
9. Escriba un test que verifique que se pueden agregar elementos a una lista. Para esto, utilice un ciclo **for** para agregar elementos a la lista. Vea el código al final del ejercicio para obtener una idea de cómo hacerlo.
10. Escriba un test que verifique que se pueden remover elementos de una lista.

11. Implemente la clase `IntArrayList` de forma que pase todos los tests escritos en el ejercicio anterior. Puede utilizar el siguiente código como punto de partida:

```
class IntArrayList(elements: IntArray) : IntList {
    private var capacity = if (elements.isEmpty()) 1 else elements.size * 2
    var elements = IntArray(capacity)

    init {
        for (i in elements.indices) {
            this.elements[i] = elements[i]
        }
    }
}
```

A continuación se presenta la estructura que debieran tener sus tests:

```
class IntArrayListTest : FunSpec({
    lateinit var list: IntList
    beforeTest {
        list = IntArrayList(intArrayOf())
    }
    test("The size of an empty list should be 0") {...}
    test("The capacity of an empty list should be 1") {...}
    test("The capacity of a list starts at double the size of the array") {
        list = IntArrayList(intArrayOf(1, 2, 3))
        // Verificar que la capacidad es 6
    }
    test("The size of the list is the number of elements it contains") {
        list = IntArrayList(intArrayOf(1, 2, 3))
        // Verificar que el tamaño es 3
    }
    test("An element can be added to the list") {
        for (i in 1..10) {
            list.add(i)
            // Verificar que el tamaño es i
            // Verificar que el último elemento es i
        }
    }
    test("The capacity of the list is doubled when it is full") {
        list = IntArrayList(intArrayOf(1, 2))
        // Verificar que la capacidad es 4
        list.add(3)
        list.add(4)
        // Verificar que la capacidad es 8
    }

    test("An element can be removed from the list") {
        list = IntArrayList(intArrayOf(1, 2, 3))
        list.remove(1) shouldBe 2
        // Verificar que el tamaño es 2
        // Verificar que los elementos son 1 y 3
    }
    test("The capacity of the list is reduced by half when it is 1/4 full") {
```

```

    list = IntArrayList(intArrayOf(1, 2, 3, 4))
    for (i in 1..3) {
        list.remove(0)
    }
    // Verificar que la capacidad es 2
}

test("Elements of the list can be obtained by index") {
    list = IntArrayList(intArrayOf(1, 2, 3))
    // Verificar que los elementos son 1, 2 y 3
}
})

```

Ejercicio 13 Bases de datos

Para este ejercicio implementaremos una base de datos de videojuegos.

Parte I

Partiremos modelando la clase `Game` que representa un videojuego. Considere que un videojuego tiene un nombre, un género, un año de lanzamiento y una calificación.

1. Cree un proyecto *gamedb* en *IntelliJ* de la forma que se vió en este capítulo utilizando *Gradle* como sistema de construcción y agregue las dependencias de *Kotest*.
2. Cree una clase `GameTest` que extienda de la clase `FunSpec`.
3. Escriba un test que verifique que se puede crear un videojuego con el nombre "Super Mario Bros.", el género "Plataformas", el año de lanzamiento 1985 y la calificación 10.0, para esto utilice el método `beforeTest` y compruebe que dos objetos creados con los mismos parámetros son iguales. Recuerde definir los parámetros pasados a los tests como `val` dentro del cuerpo de la expresión lambda.
4. Escriba un test que verifique que dos videojuegos con los mismos atributos tienen el mismo hashcode.
5. Implemente los métodos `hashCode` y `equals` de la clase `Game` de forma que pase los tests escritos en el ejercicio anterior. Para esto considere que dos videojuegos son iguales si tienen el mismo nombre y año de lanzamiento.
6. Escriba un test que verifique que dos videojuegos con distinto nombre no son iguales y que tienen distinto hashcode.
7. Escriba un test que verifique que dos videojuegos con distinto género no son iguales y que tienen distinto hashcode.
8. Escriba un test que verifique que dos videojuegos con distinto año de lanzamiento no son iguales y que tienen distinto hashcode.
9. Escriba un test que verifique que dos videojuegos con distinta calificación no son iguales y que tienen distinto hashcode.

Parte II

Ahora implementaremos la base de datos de videojuegos, para esto crearemos un objeto `GameDatabase` que tendrá un diccionario de videojuegos cuya clave será el código hash del videojuego y el valor será el videojuego en sí.

1. Cree una clase `GameDatabaseTest` que extienda de la clase `FunSpec`. A continuación se presenta un código que puede utilizar como punto de partida:

```

class GameDatabaseTest : FunSpec({
    val games = listOf(
        Game("The Legend of Zelda: Breath of the Wild", "Aventura", 2017, 10.0),
        Game("Dark Souls III", "RPG", 2016, 9.2),
        Game("Doki Doki Literature Club!", "Visual Novel", 2017, 9.1),
        Game("Final Fantasy X", "RPG", 2001, 8.0)
    )

    beforeTest {
        GameDatabase.init(games)
    }
})

```

2. Considere los siguientes tests:

```

test("The database is empty when it is created with an empty list") {
    GameDatabase.init(listOf())
    GameDatabase.entries.size shouldBe 0
}

test("The database is initialized with the games in the list") {
    GameDatabase.entries.size shouldBe 4
    for (game in games) {
        GameDatabase.entries shouldContainKey game.hashCode()
    }
}

```

Implemente la funcionalidad mínima para que los tests pasen. Para esto puede utilizar el siguiente código como punto de partida:

```

object GameDatabase {
    val entries = mutableMapOf<Int, Game>()
}

```

Hint: Utilice los métodos `MutableMap::clear()` y la expresión `entries[game.hashCode()] = game` para implementar la inicialización de la base de datos.

3. Escriba un test que verifique que se puede agregar un videojuego a la base de datos.
4. Implemente el método `add` del objeto `GameDatabase` de forma que pase el test.
5. Escriba un test que verifique que se puede eliminar un videojuego de la base de datos. Para esto puede utilizar el método `shouldNotContainKey` para verificar que un elemento no está en la base de datos.
6. Implemente el método `remove` del objeto `GameDatabase` de forma que pase el test. *Hint: Utilice el método `MutableMap::remove(K)` para implementar el método.*
7. Considere la siguiente implementación del método `search`:

```

fun search(name: String, year: Int): Game? {
    return entries[Game(name, "", year, 0.0).hashCode()]
}

```

Escriba un test que verifique que se puede buscar un videojuego por nombre y año de lanzamiento.

Bibliografía

Dolan: Effective Kotlin

dolanEffectiveKotlinItem2021

Matthew Dolan. *Effective Kotlin: Item 11 — Always Override hashCode When You Override Equals*. Medium. 29 de oct. de 2021. URL: <https://appmattus.medium.com/effective-kotlin-item-11-always-override-hashcode-when-you-override-equals-608a090aeaed> (visitado 09-02-2023).

Effective Kotlin Item 43

EffectiveKotlinItem

Effective Kotlin Item 43: Respect the Contract of hashCode. URL: <https://kt.academy/article/ek-hashcode> (visitado 09-02-2023).

Equality | Kotlin

EqualityKotlin

Equality | Kotlin. Kotlin Help. URL: <https://kotlinlang.org/docs/equality.html> (visitado 09-02-2023).

Gradle Build Tool

GradleBuildTool2023

Gradle Build Tool. Gradle. 6 de feb. de 2023. URL: <https://gradle.org/> (visitado 09-02-2023).

Kotest | Kotest

KotestKotest

Kotest | Kotest. URL: <https://kotest.io/> (visitado 09-02-2023).

Maven Central Repository Search

MavenCentralRepository

Maven Central Repository Search. URL: <https://search.maven.org/> (visitado 09-02-2023).

Properties | Kotlin

PropertiesKotlin

Properties | Kotlin. Kotlin Help. URL: <https://kotlinlang.org/docs/properties.html> (visitado 09-02-2023).

Software Framework

SoftwareFramework2022

Software Framework. En: *Wikipedia*. 25 de dic. de 2022. URL: https://en.wikipedia.org/w/index.php?title=Software_framework&oldid=1129497874 (visitado 09-02-2023).

Anot.: Page Version ID: 1129497874.

Test Case

TestCase2022

Test Case. En: *Wikipedia*. 6 de sep. de 2022. URL: https://en.wikipedia.org/w/index.php?title=Test_case&oldid=1108908331 (visitado 08-02-2023).

Anot.: Page Version ID: 1108908331.

Test-Driven Development. En: *Wikipedia*. 2 de feb. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=1136984902 (visitado 08-02-2023).

Anot.: Page Version ID: 1136984902.

Chuan Zhang. *A Brief Introduction to Build Systems*. The Startup. 27 de ago. de 2020. URL: <https://medium.com/swlh/a-brief-introduction-to-build-systems-1e45cb1cf667> (visitado 09-02-2023).

Capítulo 6

Nuevo de fábrica: *factory method pattern*

Para este capítulo, seguiremos extendiendo el juego *Bakémon* para incluir tipos de *Bakémon*.

6.1. Modelando tipos

Consideremos que en el juego existen tres tipos de *Bakémon*: *Fuego*, *Agua* y *Planta*. Por ahora pensemos que cada *Bakémon* tiene un solo ataque de su mismo tipos. Esto significa que un *Bakémon* de tipo *Fuego* tiene un ataque de tipo *Fuego*, un *Bakémon* de tipo *Agua* tiene un ataque de tipo *Agua* y un *Bakémon* de tipo *Planta* tiene un ataque de tipo *Planta*.

Comencemos pensando en cómo modelar los tipos de *Bakémon*. Una idea es hacer que nuestra clase *Bakemon* sea abierta para extensión y así poder crear subclases de *Bakemon* para cada tipo de *Bakémon*. Esta parece una buena idea, así que comencemos por ahí.

Pero antes de implementar, debemos escribir tests para seguir el proceso TDD.

Crearemos tres clases: *FireBakemonTest*, *WaterBakemonTest* y *GrassBakemonTest*.

```
class FireBakemonTest : FunSpec({
  lateinit var bakemon: FireBakemon
  val bakemonName = "Karmander"
  val bakemonHealth = 25
  val bakemonLevel = 5
  val bakemonAttack = 4
  beforeTest {
    bakemon = FireBakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
  }
  test("Two objects created from the same class should be equal") {
    val b = FireBakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
    bakemon shouldBe b
  }
  test("Two objects created from the same class should have the same hashCode") {
    val b = FireBakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
    bakemon shouldHaveSameHashCodeAs b
  }
})
```

```

class WaterBakemonTest : FunSpec({
    lateinit var bakemon: WaterBakemon
    val bakemonName = "Kokodile"
    val bakemonHealth = 25
    val bakemonLevel = 5
    val bakemonAttack = 4
    beforeTest {
        bakemon = WaterBakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
    }
    test("Two objects created from the same class should be equal") {
        val b = WaterBakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
        bakemon shouldBe b
    }
    test("Two objects created from the same class should have the same hashCode") {
        val b = WaterBakemon(bakemonName, bakemonHealth, bakemonLevel, bakemonAttack)
        bakemon shouldHaveSameHashCodeAs b
    }
})

```

Nota. Omitiremos la implementación de algunas clases y métodos de la clase `GrassBakemon` para no extender demasiado el código.

Ahora que tenemos los tests, podemos implementar las clases de *Bakémon*:

```

class FireBakemon(name: String, health: Int, level: Int, attack: Int) :
    Bakemon(name, health, level, attack) {

    override fun equals(other: Any?) = when {
        this === other -> true
        other !is FireBakemon -> false
        else -> name == other.name &&
            health == other.health &&
            level == other.level &&
            attack == other.attack
    }

    override fun hashCode() =
        Objects.hash(FireBakemon::class, name, health, level, attack)

    override fun toString() =
        "FireBakemon(name='$name', health=$health, level=$level, attack=$attack)"
}

```

```

class WaterBakemon(name: String, health: Int, level: Int, attack: Int) :
    Bakemon(name, health, level, attack) {

    override fun equals(other: Any?) = when {
        this === other -> true

```



```

    other !is WaterBakemon -> false
  else -> name == other.name &&
    health == other.health &&
    level == other.level &&
    attack == other.attack
  }

  override fun hashCode() =
    Objects.hash(WaterBakemon::class, name, health, level, attack)

  override fun toString() =
    "WaterBakemon(name='$name', health=$health, level=$level, attack=$attack)"
}

```

Ahora podemos comprobar que los tests pasan, y hacer *commit* de los cambios.

```

git add .
git commit -m "FEAT Adds FireBakemon and WaterBakemon classes"

```

6.2. Factory method pattern

Ya hicimos tests e implementamos los tipos de *Bakémon* que queríamos, nos falta el último paso del TDD: refactorizar.

Comencemos con la siguiente pregunta: ¿Existen los Bakémon sin tipo? Podría ser, pero en la manera que estamos modelando el juego no tiene mucho sentido. ¿De qué nos sirve entonces la clase *Bakémon*? La verdad es que la estamos usando solamente para agrupar distintos tipos concretos de *Bakémon*. En otras palabras, Bakemon no tiene sentido como una clase.

Transformemos la clase Bakemon en una interfaz.

```

interface Bakemon {
    val name: String
    val health: Int
    val level: Int
    val attack: Int
}

```

Y ahora modifiquemos las clases concretas para que implementen la interfaz Bakemon.

```

class FireBakemon(
    override val name: String,
    override val health: Int,
    override val level: Int,
    override val attack: Int
) : Bakemon {...}

```

```
class WaterBakemon(
    override val name: String,
    override val health: Int,
    override val level: Int,
    override val attack: Int
) : Bakemon {...}
```

Noten que agregamos la palabra reservada `override` a los atributos de las clases concretas. Esto es porque los atributos de las interfaces son *abstractos* por defecto, y por lo tanto deben ser sobrescritos en las clases que implementan la interfaz.

Ahora, borremos el archivo `BakemonTest.kt` ya que nuestro cambio de diseño rompió los tests y no tiene sentido mantenerlos.

Probemos que los tests sigan pasando y hagamos *commit*.

```
git add .
git commit -m "REFACTOR Changes Bakemon class to an interface"
```

No hay mucho más que hacer para nuestros tipos de *Bakémon*, pero notarán que los tests que escribimos para las clases de *Bakémon* son muy similares. ¿Hay alguna forma de reusar ese código? La respuesta es que sí, pero primero debemos solucionar un problema importante.

Veamos qué es lo que cambia entre los tests de las clases de *Bakémon*. En todos los tests, creamos un objeto de la clase que estamos probando, y luego comparamos el objeto creado con otro objeto creado con los mismos parámetros. Noten que lo único que cambia entre los tests es el tipo de *Bakémon* que estamos probando. En otras palabras, estamos concretizando la interfaz *Bakemon* en cada test.

¿Cómo podemos hacer para mantener la abstracción de la interfaz *Bakemon* en los tests?

La respuesta nace del origen del problema, el constructor de las clases concretas. Cuando usamos un constructor estamos acoplando nuestro código a una implementación concreta en lugar de una abstracción. Nos gustaría tener una manera genérica de crear objetos de la interfaz *Bakemon*. ¿Cómo podemos hacer eso?

¡Vamos con nuestro primer patrón de diseño!

Factory method pattern

Problema: Necesito una manera de crear objetos de una interfaz sin acoplar mi código a una implementación concreta.

Solución: Definir una interfaz para crear objetos, pero dejar que las subclases decidan que clase concreta instanciar.

El patrón *factory method* es un patrón de diseño creacional que nos permite reusar los constructores de clases concretas mediante la abstracción del proceso de creación.

La figura 6.1 muestra la estructura del patrón de diseño *Abstract Factory*. Veamos como adaptar esto a nuestro problema. Pero primero: ¿Que es un patrón de diseño?

Patrón de diseño

Un patrón de diseño es una solución general a un problema recurrente en el diseño de software.

¿Así como un algoritmo?

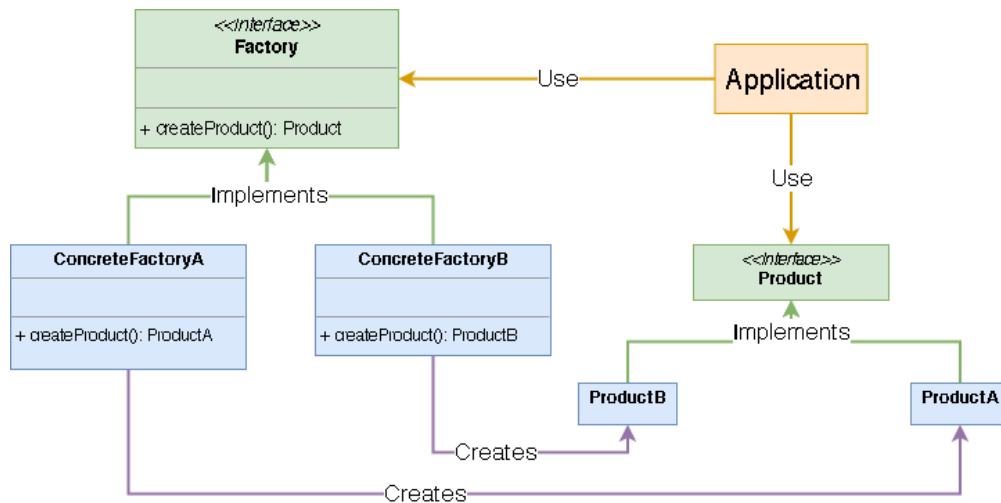


Figura 6.1: Estructura del patrón de diseño *Factory Method*

No, un patrón de diseño no es un algoritmo. Un algoritmo define una serie de pasos para solucionar un problema particular. Un patrón de diseño por otro lado define una estructura general para estructurar una solución. Esto significa que la implementación concreta de un patrón de diseño puede variar de acuerdo a las necesidades del problema.

Ahora sí, veamos como adaptar el patrón de diseño *Factory Method* a nuestro problema. Lo primero que debemos definir es cuál es la familia de objetos que queremos crear. En nuestro caso, la familia de objetos que queremos crear son los tipos de *Bakémon*. Luego, debemos definir la interfaz que va a proveer la creación de los objetos de la familia, en nuestro caso, una interfaz *BakemonFactory*. Por último, debemos definir cuáles son los métodos que va a tener la interfaz *BakemonFactory*, en nuestro caso, sólo necesitamos un método, ya que cada tipo de fábrica va a crear un tipo de *Bakémon*.

Veamos cómo implementar esto en código. Comencemos por definir la interfaz *BakemonFactory*, para esto, vamos a crear un nuevo paquete llamado `cl.ravenhill.bakemon.factory` y dentro de ese paquete vamos a crear una nueva interfaz llamada *BakemonFactory*.

```

interface BakemonFactory {
    fun createBakemon(name: String, health: Int, level: Int, attack: Int): Bakemon
}

```

Luego, debemos definir las clases concretas que implementan la interfaz *BakemonFactory*. Comencemos por los tests:

```

class FireBakemonFactoryTest : FunSpec({
    val name = "Karmander"
    val health = 25
    val level = 5
    val attack = 4

    lateinit var factory: BakemonFactory

    beforeTest {
        factory = FireBakemonFactory()
    }
}

```

```

    }

    test("A FireBakemon should be created") {
        val bakemon = factory.createBakemon(name, health, level, attack)
        bakemon.name shouldBe name
        bakemon.health shouldBe health
        bakemon.level shouldBe level
        bakemon.attack shouldBe attack
    }
})

```

```

class WaterBakemonFactoryTest : FunSpec({
    val name = "Kokodile"
    val health = 25
    val level = 5
    val attack = 4

    lateinit var factory: BakemonFactory

    beforeTest {
        factory = WaterBakemonFactory()
    }

    test("A WaterBakemon should be created") {
        val bakemon = factory.createBakemon(name, health, level, attack)
        bakemon.name shouldBe name
        bakemon.health shouldBe health
        bakemon.level shouldBe level
        bakemon.attack shouldBe attack
    }
})

```

Bien, ahora que tenemos los tests, podemos implementar las clases concretas que implementan la interfaz BakemonFactory.

```

class FireBakemonFactory : BakemonFactory {
    override fun createBakemon(name: String, health: Int, level: Int, attack: Int) =
        FireBakemon(name, health, level, attack)
}

```

```

class WaterBakemonFactory : BakemonFactory {
    override fun createBakemon(name: String, health: Int, level: Int, attack: Int) =
        WaterBakemon(name, health, level, attack)
}

```

Ahora corremos los tests y vemos que todo funciona como esperábamos. Con esto podemos hacer *commit* a nuestros cambios.

```
git add .
git commit -m "FEAT Implements Factory Method pattern"
```

6.3. Test factories

Ahora veamos cómo podemos utilizar el patrón de diseño *Abstract Factory* para reducir la repetición de código en los tests.

Kotest provee una funcionalidad llamada *test factories* que nos permite reusar tests de una manera muy sencilla. Para usarlas, crearemos una función que va a utilizar un método de *Kotest* llamado *funSpec*. Se entenderá mejor poniéndolo en práctica.

Primero, creemos un archivo en el paquete `cl.ravevnhill.bakemon` del directorio de tests haciendo click derecho en el paquete y seleccionando *New* → *Kotlin File/Class*, y en la ventana que se abre, seleccionaremos *File* y lo llamaremos *TestFactories*. Luego, agregaremos el siguiente código:

```
fun `Bakemon equality and hashCode test`() {
    factory: BakemonFactory,
    name: String,
    health: Int,
    level: Int,
    attack: Int
) = funSpec {
    lateinit var bakemon: Bakemon

    beforeTest {
        bakemon = factory.createBakemon(name, health, level, attack)
    }

    test("Two objects created from the same class should be equal") {
        val b = factory.createBakemon(name, health, level, attack)
        bakemon shouldBe b
    }
    test("Two objects created from the same class should have the same hashCode") {
        val b = factory.createBakemon(name, health, level, attack)
        bakemon shouldHaveSameHashCodeAs b
    }
}
```

Probablemente les llame la atención un detalle: el nombre de la función tiene espacios. Esto es porque *Kotlin* permite definir nombres de variables y funciones «literales», esto se hace encerrando el nombre entre *backticks* (```). Estos nombres sólo deben usarse en los tests¹ ya que abusar de ellos puede hacer que el código sea difícil de leer.

Ahora, veamos cómo podemos utilizar esta función para reducir la repetición de código en los tests. Para esto, utilizaremos la función *include* de *Kotest* que nos permite incluir tests definidos en otras funciones.

¹*Coding Conventions* | *Kotlin*.

```
class FireBakemonTest : FunSpec({
  include(
    `Bakemon equality and hashCode test`(FireBakemonFactory(), "Karmander", 25, 5, 4))
})
```

```
class WaterBakemonTest : FunSpec({
  include(
    `Bakemon equality and hashCode test`(WaterBakemonFactory(), "Kokodile", 25, 5, 4))
})
```

Importante. Cada test factory tiene su propio contexto, por lo que no puede acceder a las variables definidas fuera de ella, y no se puede acceder a las variables definidas dentro de ella desde fuera.

Esto implica que el método `beforeTest` es propio de cada test factory, por lo que no podemos «reusarlo» entre distintas test factories.

6.4. Lo bueno y lo malo

Todos los patrones de diseño tienen sus pros y sus contras, y el patrón de diseño *Factory Method* no es la excepción.

6.4.1. Lo bueno

- **Abstracción:** Podemos abstraer la creación de objetos, permitiendonos encapsular lo que cambia (!).
- **Desacoplamiento:** Podemos desacoplar la creación de objetos de su uso.
- **Single Responsibility:** Podemos encapsular la creación de objetos en una clase separada, lo que nos permite tener una única responsabilidad por clase.
- **Open/Closed:** Podemos agregar nuevas familias de objetos sin necesidad de modificar el código que los usa.

6.4.2. Lo malo

- **Overengineering:** Podemos estar agregando complejidad innecesaria a nuestro código.

Si bien menciono un único contra, es uno extremadamente importante y tiene que ver con la parte más difícil de diseñar software: poner en la balanza los pros y los contras de cada solución. En el diseño de software, todas las soluciones son correctas si se puede argumentar el porqué de cada decisión.

Aquí elegimos complicar la implementación de nuestro código para simplificar su uso. La pregunta que siempre debemos hacernos es: ¿qué tanto trabajo significa implementar una solución ahora y qué tanto trabajo nos ahorrará en el futuro? Si la respuesta es que nos ahorrará mucho trabajo en el futuro, entonces vale la pena implementar la solución. Si la respuesta es que no nos ahorrará mucho trabajo en el futuro, entonces podemos posponer la decisión de diseño para más adelante.

En este caso, elegimos esta solución pensando que crear nuevas instancias de Bakemon es algo que tendremos que hacer mucho en nuestro programa, y que es probable que en el futuro tengamos que agregar nuevos tipos de Bakemon.

En el próximo capítulo seguiremos extendiendo nuestro juego para poder implementar el combate con una nueva técnica de programación.

6.5. Ejercicios

Importante. Recuerde hacer commit después de cada pregunta.

Ejercicio 14 Lo que faltó

1. Implemente la clase `GrassBakemonFactoryTest` que pruebe la clase `GrassBakemonFactory`.
2. Implemente la clase `GrassBakemonFactory` para que pase las pruebas.
3. Implemente la clase `GrassBakemonTest` que pruebe la clase `GrassBakemon` y utilice la fábrica `GrassBakemonFactory`.
4. Implemente la clase `GrassBakemon` para que pase las pruebas.

Ejercicio 15 La Pizzería

La pizzería *Imma Cooka* necesita un sistema para manejar sus pedidos. Cada pedido tiene un nombre, una dirección y una lista de pizzas. Cada pizza tiene una lista de ingredientes y tipo de masa. Cada ingrediente tiene un nombre y un precio base.

El precio de una pizza se calcula como el precio base de la masa más la suma de los precios de los ingredientes. El precio de un pedido se calcula como la suma de los precios de cada pizza. El sistema debe permitir agregar pizzas a un pedido, agregar ingredientes a una pizza y calcular el precio de un pedido.

1. Cree una clase `IngredientTest` que verifique que un ingrediente se pueda crear con un nombre y un precio base. Para esto utilice correctamente el método `beforeTest`.

Asuma que tiene acceso a los métodos `Ingredient::equals(Any?): Boolean` e `Ingredient::hashCode(): Int`.

2. Implemente la clase `Ingredient` para que pase las pruebas.
3. Cree una clase `PizzaTest` que verifique que una pizza se pueda crear con un nombre, una lista de ingredientes y un precio base. Para esto utilice correctamente el método `beforeTest`.

Asuma que tiene acceso a los métodos `Pizza::equals(Any?): Boolean` e `Pizza::hashCode(): Int`.

4. Implemente la clase `Pizza` para que pase las pruebas.
5. Cree un test que verifique que el precio de una pizza se calcula correctamente.
6. Implemente el método `Pizza::price()` para que pase las pruebas.
7. Cree una clase `OrderTest` que verifique que un pedido se pueda crear con un nombre, una dirección y una lista de pizzas. Para esto utilice correctamente el método `beforeTest`.

Asuma que tiene acceso a los métodos `Order::equals(Any?): Boolean` y `Order::hashCode(): Int`.

8. Implemente la clase `Order` para que pase las pruebas.
9. Cree un test que verifique que el precio de un pedido se calcula correctamente.
10. Implemente el método `Order::price()` para que pase las pruebas.

Además de permitirle a sus clientes elegir los ingredientes que quieren en sus pizzas, la pizzería quiere ofrecerles una variedad de pizzas predefinidas, donde cada una tiene un porcentaje de descuento sobre el precio final de la pizza. Existen dos tipos de pizzas predefinidas: *Margarita* y *Napolitana*. Los descuentos son los siguientes:

- *Margarita*: 10 %.
- *Napolitana*: 15 %.

Los ingredientes de cada una de estas pizzas son los siguientes:

- *Margarita*: Tomate, Queso y Albahaca.
- *Napolitana*: Tomate, Queso y Aceitunas.

Los precios de cada ingrediente son los siguientes:

- Tomate: \$400.
- Queso: \$500.
- Albahaca: \$100.
- Aceitunas: \$200.

11. Cree la clase `MargaritaPizzaTest` que verifique que una pizza *Margarita* se pueda crear correctamente.
12. Considere la siguiente implementación de la clase `MargaritaPizza`:

```
class MargaritaPizza(basePrice: Int): Pizza(basePrice, listOf(
    Ingredient("Tomato", 100),
    Ingredient("Cheese", 200),
    Ingredient("Basil", 300)
)) {
    override fun price() = (super.price() * 0.9).toInt()
}
```

¿Qué significa la llamada a `super` en el método `price()`? ¿A qué objeto se está referenciando?

13. Considere la interfaz `PizzaFactory`:

```
interface PizzaFactory {
    fun create(basePrice: Int): Pizza
}
```

Escriba una clase `MargaritaPizzaFactoryTest` que verifique que la fábrica de pizzas *Margarita* crean pizzas *Margarita* correctamente.

14. Utilice *test factories* para reescribir la clase `MargaritaPizzaTest` utilizando la fábrica `MargaritaPizzaFactory`.
15. Cree la clase `NapolitanaPizzaFactoryTest` que verifique que la fábrica de pizzas *Napolitana* crean pizzas *Napolitana* correctamente.
16. Cree la clase `NapolitanaPizzaTest` que verifique que una pizza *Napolitana* se pueda crear correctamente utilizando *test factories*.

Bibliografía

Coding Conventions | Kotlin

CodingConventionsKotlin

Coding Conventions | Kotlin. Kotlin Help. URL: <https://kotlinlang.org/docs/coding-conventions.html> (visitado 11-02-2023).

Freeman y col.: Chapter 4: The Factory Pattern. Baking with OO Goodness **freemanChapterFactoryPattern2021**

Eric Freeman y Elisabeth Robson. «Chapter 4: The Factory Pattern. Baking with OO Goodness». En: *Head First Design Patterns*. Second edition. Head First Series. Beijing [China] ; Boston [MA]: O'Reilly, 2021, págs. 109-168. ISBN: 978-1-4920-7800-5.

Gamma y col.: Factory Method

gammaFactoryMethod1995

Erich Gamma, Richard Helm y col. «Factory Method». En: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley, 1995, págs. 107-116. ISBN: 978-0-201-63361-0.

Shvets: Factory Method

shvetsFactoryMethod2021

Alexander Shvets. «Factory Method». En: *Dive Into Design Patterns*. 2021, págs. 74-89.

Test Factories | Kotest

TestFactoriesKotest

Test Factories | Kotest. URL: <https://kotest.io/docs/framework/test-factories.html> (visitado 11-02-2023).

Capítulo 7

El capítulo ambiguo

En este capítulo seguiremos implementando nuestro juego de Bakémon. En el capítulo anterior, implementamos la clase *Bakemon* y sus subclases. En este capítulo implementaremos ataques y un sistema de combates simple.

Cada tipo de Bakémon tiene ventajas y debilidades contra otros tipos de Bakémon. Cuando un *Bakémon* ataca a otro *Bakémon* de un tipo que tiene ventaja sobre el tipo del *Bakémon* atacante, el ataque hace $\frac{3}{2} \cdot \text{attack}$ de daño. Cuando un *Bakémon* ataca a otro *Bakémon* de un tipo resistente, el ataque hace $\frac{1}{2} \cdot \text{attack}$ de daño. Si no es ninguno de los casos anteriores, el ataque hace attack de daño. Las fortalezas y debilidades de los tipos se muestran en la cuadro 7.1.

Atacante	Atacado		
	<i>Fuego</i>	<i>Agua</i>	<i>Planta</i>
<i>Fuego</i>	$\times 1$	$\times 1/2$	$\times 3/2$
<i>Agua</i>	$\times 3/2$	$\times 1$	$\times 1/2$
<i>Planta</i>	$\times 1/2$	$\times 3/2$	$\times 1$

Cuadro 7.1: Tipos de Bakémon y sus ventajas y resistencias

7.1. Ataques: primer intento

7.1.1. Recibiendo daño

Comencemos por lo más simple y digamos que los *Bakémon* pueden atacarse entre sí ignorando las fortalezas y debilidades de los tipos. Para esto, vamos a crear primero un método `Bakemon::takeDamage(Int)` que nos permitirá reducir la vida de un *Bakémon* en una cantidad determinada asegurándonos de que la vida no sea negativa.

```
interface Bakemon {  
    ...  
    fun takeDamage(damage: Int)  
}
```

Luego, podemos implementar los tests para este método, para esto podemos crear un *test factory* que nos permita crear los tests para cada tipo de *Bakémon* de forma sencilla:

```

fun `Bakemon can take damage`(
    factory: BakemonFactory,
    name: String,
    health: Int,
    level: Int,
    attack: Int
) = funSpec {
    lateinit var bakemon: Bakemon

    beforeTest {
        bakemon = factory.createBakemon(name, health, level, attack)
    }
    test("Taking damage should reduce the Bakemon's health") {
        bakemon.takeDamage(5)
        bakemon.health shouldBe 20
    }
    test(
        "Taking damage should reduce the Bakemon's health to 0 if the damage is " +
        "greater than its health"
    ) {
        bakemon.takeDamage(30)
        bakemon.health shouldBe 0
    }
}

```

Ahora podemos crear los tests para cada tipo de *Bakémon* de forma sencilla:

```

class FireBakemonTest : FunSpec({
    ...
    include(`Bakemon can take damage`(FireBakemonFactory(), "Karmander", 25, 5, 4))
})

```

```

class FireBakemonTest : FunSpec({
    ...
    include(`Bakemon can take damage`(WaterBakemonFactory(), "Kokodile", 25, 5, 4))
})

```

Ahora podemos implementar el método `Bakemon::takeDamage(Int)` en cada tipo concreto, pero antes,

```

class FireBakemon(...) : Bakemon {
    ...
    override fun takeDamage(damage: Int) {
        health -= damage
        if (health < 0) {
            health = 0
        }
    }
}

```

```
}
```

```
class WaterBakemon(...) : Bakemon {  
    ...  
    override fun takeDamage(damage: Int) {  
        health -= damage  
        if (health < 0) {  
            health = 0  
        }  
    }  
}
```

Ahora podemos correr los tests y ver que todos pasan :)

Nos quedaría mejorar nuestro diseño, y aquí debieran notar que el método `takeDamage` es idéntico para todos los tipos de *Bakémon*, por lo que podríamos usar una clase abstracta para implementarlo. Para esto primero creemos un nuevo paquete `cl.ravenhill.bakemon.types` tanto en la aplicación como en los tests, y movamos todas las clases de tipos de *Bakémon* a este nuevo paquete. Ahora, creemos una clase abstracta `AbstractBakemon` que implemente la interfaz `Bakemon` y que contenga el método `takeDamage`:

```
abstract class AbstractBakemon(  
    override val name: String,  
    override var health: Int,  
    override val level: Int,  
    override val attack: Int  
) : Bakemon {  
    health -= damage  
    if (health < 0) {  
        health = 0  
    }  
}
```

Ahora, podemos hacer que las clases de tipos de *Bakémon* hereden de esta clase abstracta y eliminar el método `takeDamage` de cada una de ellas:

```
class FireBakemon(name: String, health: Int, level: Int, attack: Int) :  
    AbstractBakemon(name, health, level, attack) {  
    override fun equals(other: Any?) = when {...}  
    override fun hashCode() = ...  
    override fun toString() = ...  
}
```

```
class WaterBakemon(name: String, health: Int, level: Int, attack: Int) :  
    AbstractBakemon(name, health, level, attack) {  
    override fun equals(other: Any?) = when {...}  
    override fun hashCode() = ...  
}
```

```
    override fun toString() = ...
}
```

Corramos los tests para ver que todo sigue funcionando y hagamos un commit:

```
git add .
git commit -m "FEAT Bakemon can take damage"
```

7.1.2. Atacando

Ahora que tenemos implementado el método

`Bakemon::takeDamage(Int)`, podemos implementar el método `Bakemon::attackBakemon(Bakemon)`. Como siempre, comencemos por los tests.

```
fun `Bakemon can attack Bakemon`(
    factory: BakemonFactory,
    name: String,
    health: Int,
    level: Int,
    attack: Int,
    enemyFactory: BakemonFactory,
    enemyName: String,
    enemyHealth: Int,
    enemyLevel: Int,
    enemyAttack: Int,
    expectedDamage: Int
) = funSpec {
    lateinit var bakemon: Bakemon
    lateinit var enemy: Bakemon

    beforeTest {
        bakemon = factory.createBakemon(name, health, level, attack)
        enemy = enemyFactory.createBakemon(enemyName, enemyHealth, enemyLevel, enemyAttack)
    }

    test("An attack should reduce the enemy's health") {
        bakemon.attackBakemon(enemy)
        enemy.health shouldBe enemyHealth - expectedDamage
    }

    test("An attack should not set the enemy's health below 0") {
        enemy.takeDamage(enemyHealth)
        bakemon.attackBakemon(enemy)
        enemy.health shouldBe 0
    }
}
```

Este método debería recibir un *Bakémon* y realizar el cálculo de daño correspondiente. Con esto, ya tenemos los tests

para los ataques entre *Bakémon* de distintos tipos. Noten que reusamos la función ``Bakemon can fight`` para definir peleas entre *Bakémon* de distintos tipos.

Ahora incluyamos el *test factory* en los tests de cada tipo de *Bakémon*:

```
class FireBakemonTest : FunSpec({
  ...
  include(
    "Fire Bakemon -> Fire Bakemon:",
    `Bakemon can attack Bakemon`(
      FireBakemonFactory(), "Karmander", 25, 5, 4,
      FireBakemonFactory(), "Karmander", 25, 5, 4, 4
    )
  )
  include(
    "Fire Bakemon -> Water Bakemon:",
    `Bakemon can attack Bakemon`(
      FireBakemonFactory(), "Karmander", 25, 5, 4,
      WaterBakemonFactory(), "Kokodile", 25, 5, 4, 2
    )
  )
})
```

```
class WaterBakemonTest : FunSpec({
  ...
  include(
    "Water Bakemon -> Fire Bakemon:",
    `Bakemon can attack Bakemon`(
      WaterBakemonFactory(), "Kokodile", 25, 5, 4,
      FireBakemonFactory(), "Karmander", 25, 5, 4, 8
    )
  )
  include(
    "Water Bakemon -> Water Bakemon:",
    `Bakemon can attack Bakemon`(
      WaterBakemonFactory(), "Kokodile", 25, 5, 4,
      WaterBakemonFactory(), "Kokodile", 25, 5, 4, 4
    )
  )
})
```

Noten que aquí le pasamos un parámetro extra a la función `include`, esto lo hacemos para que no se repitan los nombres de los tests.¹

Corramos los tests para ver cómo fallan.

Ahora, pasemos a implementar el método `Bakemon::attackBakemon(Bakemon)`.

Como los ataques no consideran fortalezas o debilidades, debieramos notar que el método va a ser idéntico para todos los tipos de *Bakémon*, así que podemos definirlo en la clase abstracta.

¹Esto no es necesario, pero es una buena práctica.

```
abstract class AbstractBakemon(...) : Bakemon {
    ...
    override fun attackBakemon(bakemon: Bakemon) {
        bakemon.takeDamage(attack)
    }
}
```

Corramos los tests y veamos que ahora pasan. Por último, podemos hacer *commit* de los cambios.

```
git add .
git commit -m "FEAT Adds attackBakemon method"
```

7.2. Ataques: segundo intento

Ahora intentemos agregar las fortalezas y debilidades de los tipos. Para esto comencemos por crear dos *test factories*, una para los ataques muy efectivos y otra para los ataques poco efectivos:

```
fun `effective attack test`(
    factory: BakemonFactory,
    name: String,
    health: Int,
    level: Int,
    attack: Int,
    enemyFactory: BakemonFactory,
    enemyName: String,
    enemyHealth: Int,
    enemyLevel: Int,
    enemyAttack: Int,
) = funSpec {
    lateinit var bakemon: Bakemon
    lateinit var enemy: Bakemon

    beforeTest {
        bakemon = factory.createBakemon(name, health, level, attack)
        enemy = enemyFactory.createBakemon(enemyName, enemyHealth, enemyLevel, enemyAttack)
    }

    test("An effective attack should deal 1.5 times the damage") {
        bakemon.attackBakemon(enemy)
        enemy.health shouldBe enemyHealth - (attack * 1.5).toInt()
    }
}
```

```
fun `ineffective attack test`(
    factory: BakemonFactory,
    name: String,
```



```

health: Int,
level: Int,
attack: Int,
enemyFactory: BakemonFactory,
enemyName: String,
enemyHealth: Int,
enemyLevel: Int,
enemyAttack: Int,
) = funSpec {
  lateinit var bakemon: Bakemon
  lateinit var enemy: Bakemon

  beforeTest {
    bakemon = factory.createBakemon(name, health, level, attack)
    enemy = enemyFactory.createBakemon(enemyName, enemyHealth, enemyLevel, enemyAttack)
  }

  test("An ineffective attack should deal 0.5 times the damage") {
    bakemon.attackBakemon(enemy)
    enemy.health shouldBe enemyHealth - attack / 2
  }
}

```

Ahora podemos usar estas funciones para definir los tests de los ataques entre *Bakémon* de distintos tipos:

```

class FireBakemonTest : FunSpec({
  ...
  include(
    "Fire Bakemon -> Fire Bakemon:",
    `Bakemon can attack Bakemon`(
      FireBakemonFactory(), "Karmander", 25, 5, 4,
      FireBakemonFactory(), "Karmander", 25, 5, 4, 4
    )
  )
  include(
    "Fire Bakemon -> Water Bakemon:",
    `ineffective attack test`(
      FireBakemonFactory(), "Karmander", 25, 5, 4,
      WaterBakemonFactory(), "Kokodile", 25, 5, 4
    )
  )
  include(
    "Fire Bakemon -> Grass Bakemon:",
    `effective attack test`(
      FireBakemonFactory(), "Karmander", 25, 5, 4,
      GrassBakemonFactory(), "Kriko", 25, 5, 4
    )
  )
})

```

```

class WaterBakemonTest : FunSpec({
  ...
  include(
    "Water Bakemon -> Water Bakemon:",
    `Bakemon can attack Bakemon`(
      WaterBakemonFactory(), "Kokodile", 25, 5, 4,
      WaterBakemonFactory(), "Kokodile", 25, 5, 4, 4
    )
  )
  include(
    "Water Bakemon -> Fire Bakemon:",
    `effective attack test`(
      WaterBakemonFactory(), "Kokodile", 25, 5, 4,
      FireBakemonFactory(), "Karmander", 25, 5, 4
    )
  )
  include("Water Bakemon -> Grass Bakemon:",
    `ineffective attack test`(
      WaterBakemonFactory(), "Kokodile", 25, 5, 4,
      GrassBakemonFactory(), "Kriko", 25, 5, 4
    )
  )
})

```

Corramos los tests para ver que fallan como esperamos.

Ahora pasemos a implementar las fortalezas y debilidades de los *Bakémon*.

La implementación más simple que podríamos pensar es simplemente preguntar por el tipo del *Bakémon* enemigo y actuar en consecuencia. Veamos cómo implementaríamos esto:

```

class FireBakemon(name: String, health: Int, level: Int, attack: Int) :
  AbstractBakemon(name, health, level, attack) {

  override fun attackBakemon(bakemon: Bakemon) = when(bakemon) {
    is WaterBakemon -> bakemon.takeDamage(attack / 2)
    is GrassBakemon -> bakemon.takeDamage((attack * 1.5).toInt())
    else -> bakemon.takeDamage(attack)
  }
  ...
}

```

```

class WaterBakemon(name: String, health: Int, level: Int, attack: Int) :
  AbstractBakemon(name, health, level, attack) {

  override fun attackBakemon(bakemon: Bakemon) = when(bakemon) {
    is GrassBakemon -> bakemon.takeDamage(attack / 2)
    is FireBakemon -> bakemon.takeDamage((attack * 1.5).toInt())
    else -> bakemon.takeDamage(attack)
  }
}

```

```

    }
    ...
}

```

Ahora corramos los tests y veamos que pasa.

Lo último que nos queda es mejorar nuestro diseño. Partamos por revisar nuestras *test factories*, debieran notar que hay mucho código repetido. El problema es que en realidad todas las *test factories* de los ataques hacen las mismas comprobaciones y solo cambia la cantidad de daño que se hace.

Para esto, primero agreguemos un nuevo parámetro a la función `Bakemon.canAttack(Bakemon)` para que reciba los nombres de los tests:

```

fun `Bakemon can attack Bakemon`(
    ...
    testNames: Pair<String, String> = "An attack should reduce the enemy's health" to
        "An attack should not set the enemy's health below 0"
) = funSpec {
    lateinit var bakemon: Bakemon
    lateinit var enemy: Bakemon

    beforeTest {
        bakemon = factory.createBakemon(name, health, level, attack)
        enemy = enemyFactory.createBakemon(enemyName, enemyHealth, enemyLevel, enemyAttack)
    }

    test(testNames.first) {
        bakemon.attackBakemon(enemy)
        enemy.health shouldBe enemyHealth - expectedDamage
    }

    test(testNames.second) {
        enemy.takeDamage(enemyHealth)
        bakemon.attackBakemon(enemy)
        enemy.health shouldBe 0
    }
}

```

Aquí tenemos un par (jaja) de cosas nuevas.

Primero, el tipo `Pair` es un tipo de dato que representa un par de elementos, en este caso, dos `Strings`. Luego tenemos la función `to` que nos permite crear un par de elementos de cualquier tipo. Por último, notemos que le pasamos un valor directamente a un parámetro de la función, esto se llama *parámetro por defecto*, y nos permite omitir el parámetro si no queremos cambiar su valor. Para entender mejor esto veamos un ejemplo:

```

fun jojoReference(name: String = "Giorno Giovanna") =
    println("My name is $name and I have a dream")

fun main() {
    jojoReference()
}

```

```
jojoReference("Jotaro Kujo")
}
```

En este caso, el programa imprimiría:

```
My name is Giorno Giovanna and I have a dream
My name is Jotaro Kujo and I have a dream
```

Volviendo a nuestro código, ahora podemos usar esta función para simplificar nuestras *test factories*:

```
fun `effective attack test`(...) = `Bakemon can attack Bakemon`(
    factory, name, health, level, attack,
    enemyFactory, enemyName, enemyHealth, enemyLevel, enemyAttack,
    (attack * 1.5).toInt(),
    "An effective attack should deal the damage" to
    "An effective attack should not set the enemy's health below 0"
)
```

```
fun `ineffective attack test`(...) = `Bakemon can attack Bakemon`(
    factory, name, health, level, attack,
    enemyFactory, enemyName, enemyHealth, enemyLevel, enemyAttack,
    attack / 2,
    "An ineffective attack should deal the damage" to
    "An ineffective attack should not set the enemy's health below 0"
)
```

Nos queda una última cosa que hacer, y es mejorar la forma en que decidimos si un ataque es efectivo o no. Aquí nos topamos con un problema que ya habíamos visto antes, estamos haciendo que cada *Bakémon* tenga que manejar el tipo de cada uno de sus enemigos. Esto es un problema porque si queremos agregar un nuevo tipo de *Bakémon* tenemos que modificar el método `attackBakemon` de todos los *Bakémon* existentes. Volvemos a tener el problema de encapsular el cambio.

Este problema es difícil de resolver, pero vamos a ver una solución que nos permitirá implementar nuestro sistema de tipos de una forma más flexible.

Pero antes, hagamos *commit* de nuestro código.

```
git add .
git commit -m "FEAT Adds effective and ineffective attacks"
```

7.3. Double Dispatch

Idealmente queremos que el daño que un *Bakémon* hace a otro dependa de su tipo y el tipo del *Bakémon* atacado, pero también nos gustaría una forma de hacer la comprobación de tipos sin tener que preguntar por el tipo de cada *Bakémon* en cada ataque.

Nos gustaría poder hacer algo como:

```
interface Bakemon {
    val name: String
    var health: Int
    val level: Int
    val attack: Int

    fun attackBakemon(bakemon: FireBakemon)
    fun attackBakemon(bakemon: WaterBakemon)
    fun attackBakemon(bakemon: GrassBakemon)
    fun takeDamage(damage: Int)
}
```

```
class FireBakemon(name: String, health: Int, level: Int, attack: Int) :
    AbstractBakemon(name, health, level, attack) {
    override fun attackBakemon(bakemon: FireBakemon) = bakemon.takeDamage(attack / 2)
    override fun attackBakemon(bakemon: WaterBakemon) =
        bakemon.takeDamage((attack * 1.5).toInt())
    override fun attackBakemon(bakemon: GrassBakemon) = bakemon.takeDamage(attack)
    ...
}
```

Pero si intentamos ejecutar esto obtendremos un error de compilación en la función `Bakemon` con `attack Bakemon` (figura 7.1) porque no existe una función `attackBakemon(Bakemon)`. Podemos agregar esa función, pero entonces cada vez que hagamos `attackBakemon` estaremos llamando al método `attackBakemon(Bakemon)` ya que `Bakemon` es supertipo de `FireBakemon`, `WaterBakemon` y `GrassBakemon`.

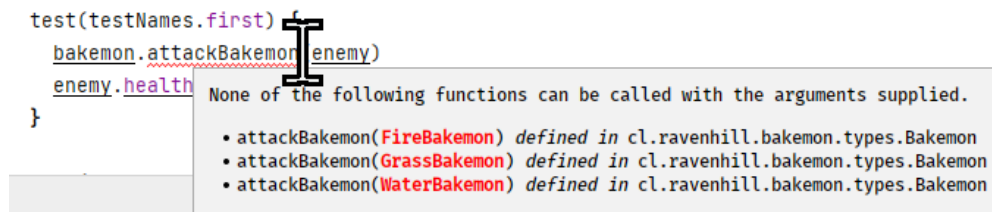


Figura 7.1: Error de compilación por ambigüedad

El proceso mediante el que se resuelve el método a llamar se llama *enlace dinámico* (*dynamic dispatch*). En el enlace dinámico, el compilador no puede saber qué método se va a llamar hasta que se ejecute el programa. Aquí entra en juego el concepto de *ambigüedad*.

Existen dos tipos de enlace dinámico: *single dispatch* y *multiple dispatch*.

En *single dispatch* el programa decide qué método llamar basándose en el tipo de la instancia sobre la que se llama al método. Es decir, se decide qué método llamar basándose en un sólo parámetro. Volvamos al ejemplo de la familia *Caninae* del capítulo 4 y veamos cómo se resuelve el enlace dinámico de la funcionalidad de mover la cola.

```
interface Canine {
    fun wagTail()
}
```

```
class Wolf : Canine {
    override fun wagTail() = println("Wolf wagging tail")
}
```

```
class Dog : Canine {
    override fun wagTail() = println("Dog wagging tail")
}
```

```
class Fox : Canine {
    override fun wagTail() = println("Fox wagging tail")
}
```

```
fun main() {
    val wolf = Wolf()
    val dog = Dog()
    val fox = Fox()
    wolf.wagTail()
    dog.wagTail()
    fox.wagTail()
}
```

En estos casos, el compilador sabe que el tipo de la instancia sobre la que se llama al método `wagTail()` es `Wolf`, `Dog` o `Fox` respectivamente. Al objeto que recibe el mensaje lo llamamos *receptor* del mensaje. En este caso, los receptores son `wolf`, `dog` y `fox`. Entonces, la decisión de qué método llamar se basa en el tipo del receptor.

En *multiple dispatch* el programa decide qué método llamar basándose en el tipo de la instancia sobre la que se llama al método y en los tipos de las instancias que se pasan como argumentos. Es decir, se decide qué método llamar basándose en dos o más parámetros. En el ejemplo de los *Bakémon* queremos que el daño que un *Bakémon* hace a otro dependa de su tipo y el tipo del *Bakémon* atacado.

Entonces lo que nos gustaría hacer en nuestro programa es *multiple dispatch*. El problema es que *Kotlin* (y la mayoría de los lenguajes) no soporta *multiple dispatch*. ¿Cómo podemos hacer entonces para que el daño que un *Bakémon* hace a otro dependa de su tipo y el tipo del *Bakémon* atacado? La solución es usar *double dispatch*.

Double Dispatch

Double dispatch es un mecanismo que permite implementar *multiple dispatch* en lenguajes que no lo soportan. En *double dispatch* se realizan dos llamados a métodos (*single dispatch*):

- El primer llamado desambigua el tipo del receptor del mensaje.
- El segundo llamado desambigua el tipo del argumento del mensaje.

Veamos esto con un ejemplo. El *cachipún* (piedra, papel o tijera) es un juego de manos en el que cada jugador simultáneamente muestra una de las tres posibilidades formando así un triángulo. El jugador que muestra la opción que gana es el ganador de la ronda. Las posibilidades son:

- Piedra gana a tijera.
- Tijera gana a papel.

- Papel gana a piedra.

Modelemos este juego con *Kotlin* usando *double dispatch*.

Lo primero que necesitaremos es una interfaz que abarque a las tres opciones del juego.

```
interface Hand {  
    fun playWith(hand: Hand)  
    ...  
}
```

Ahora definamos las tres opciones del juego.

```
class Rock : Hand {  
    override fun playWith(hand: Hand) = hand.playWithRock(this)  
    ...  
}
```

Veamos lo que pasa cuando llamamos al método `playWith(hand: Hand)` con un `Rock` como receptor y un `Paper` como argumento. Lo primero que hace el compilador es desambiguar el tipo del receptor del mensaje. Como el receptor es un `Rock`, el compilador sabe que debe llamar al método `Rock::playWith(Hand)`. Luego, el receptor del segundo mensaje es el `Paper` que se pasa como argumento, así que el programa sabe que debe llamar al método `Paper::playWithRock(Rock)`.

Por último podemos definir el método `Hand::playWithRock(Rock)`.

```
interface Hand {  
    fun playWithRock(rock: Rock)  
    ...  
}
```

```
class Paper : Hand {  
    override fun playWithRock(rock: Rock) = println("Paper wins")  
    ...  
}
```

7.4. Ataques: Tercer intento

Volvamos a nuestro proyecto y veamos cómo podemos implementar los ataques utilizando *double dispatch*. Para esto, primero agreguemos los métodos necesarios a la interfaz `Bakemon`:

```
interface Bakemon {  
    ...  
    fun attackBakemon(bakemon: Bakemon)  
    fun attackedByFireBakemon(bakemon: FireBakemon)  
    fun attackedByWaterBakemon(bakemon: WaterBakemon)  
    fun attackedByGrassBakemon(bakemon: GrassBakemon)
```

```
}
```

Ahora, implementemos estos métodos en cada una de las clases que implementan Bakemon:

```
class FireBakemon(name: String, health: Int, level: Int, attack: Int) :  
    AbstractBakemon(name, health, level, attack) {  
    override fun attackBakemon(bakemon: Bakemon) = bakemon.attackedByFireBakemon(this)  
  
    override fun attackedByFireBakemon(bakemon: FireBakemon) =  
        takeDamage(bakemon.attack)  
    override fun attackedByWaterBakemon(bakemon: WaterBakemon) =  
        takeDamage((bakemon.attack * 1.5).toInt())  
    override fun attackedByGrassBakemon(bakemon: GrassBakemon) =  
        takeDamage(bakemon.attack / 2)  
    ...  
}
```

```
class WaterBakemon(name: String, health: Int, level: Int, attack: Int) :  
    AbstractBakemon(name, health, level, attack) {  
    override fun attackBakemon(bakemon: Bakemon) = bakemon.attackedByWaterBakemon(this)  
  
    override fun attackedByFireBakemon(bakemon: FireBakemon) =  
        takeDamage(bakemon.attack / 2)  
  
    override fun attackedByWaterBakemon(bakemon: WaterBakemon) =  
        takeDamage(bakemon.attack)  
  
    override fun attackedByGrassBakemon(bakemon: GrassBakemon) =  
        takeDamage((bakemon.attack * 1.5).toInt())  
    ...  
}
```

Pero

Ya empezó ~-

tenemos un problema.

Estamos repitiendo los mismos cálculos de daño en cada una de las clases que implementan, esto podría no parecer tanto problema, pero si tenemos 10 tipos de Bakemon, entonces tendremos que implementar 10 veces el mismo código, pero más importante aún, si queremos cambiar el cálculo de daño, tendremos que hacerlo en 10 lugares distintos. Es decir que no encapsulamos el cambio.

Podemos resolver esto de manera simple creando métodos auxiliares en la clase abstracta. Crearemos dos métodos adicionales, uno para calcular el daño de una debilidad y otro para calcular el daño de una resistencia. Pero preguntémosnos primero. ¿Desde dónde quiero usar estos métodos?

Desde los métodos attackedBy? uwu

Asíes, entonces, si pesa lo mismo que un pato...

¡Es bruja!

Estos métodos los vamos a implementar en la clase abstracta y lo vamos a utilizar solo desde sus subclases, entonces, como por la propiedad de encapsulamiento queremos restringir el acceso a las componentes internas de un objeto lo más posible, nos gustaría que los métodos `takeDamage` fueran privadas dentro de jerarquía de clases. Por suerte, *Kotlin* tiene una pequeña palabra secreta que pareciera que nos leyó la mente: `protected`.

Modificador de privilegios `protected`

Un miembro `protected` es accesible desde la clase que lo define y desde las que heredan directa o indirectamente de ella.

Entonces, los métodos `takeDamage` podrían ser `protected`. Pero los miembros de una interfaz deben ser públicas, entonces eliminaremos al método `Bakemon::takeDamage(Int)` de la interfaz y lo implementaremos sólo en la clase abstracta.

```
abstract class AbstractBakemon(...) : Bakemon {
    protected fun takeDamage(damage: Int) {
        health -= damage
        if (health < 0) {
            health = 0
        }
    }
}

protected fun takeWeaknessDamage(damage: Int) = takeDamage((damage * 1.5).toInt())

protected fun takeResistanceDamage(damage: Int) = takeDamage(damage / 2)
}
```

Este cambio va a romper nuestros tests, ya que ahora los métodos `takeDamage` no son visibles desde ellos. Esto es esperado, lo primero que haremos será borrar la función ``Bakemon` can take damage`` del archivo `TestFactories.kt` y todos sus usos en los tests.

Para lo siguiente vamos a necesitar hacer un cambio un poco más grande. Vamos a modificar nuestros factories para hacerlos más flexibles.

Otra de las ventajas de utilizar *factory method* es que podemos crear objetos de manera dinámica, es decir, podemos reutilizar una misma factory para crear objetos que varían poco entre sí. La idea es permitir asignar valores a las factories después de que estas ya hayan sido creadas. Para esto partamos por modificar la interfaz `BakemonFactory` para que tenga variables de instancia que puedan ser modificadas y simplifiquemos el método `createBakemon` para que no reciba ningún parámetro.

Pero utilicemos los tests para tener una idea de cómo debería quedar la interfaz.

```
class FireBakemonFactoryTest : FunSpec({
    ...
    beforeTest {
        factory = FireBakemonFactory()
        factory.name = name
        factory.health = health
        factory.level = level
        factory.attack = attack
    }
})
```

```

test("A FireBakemon should be created") {
    val bakemon = factory.createBakemon()
    bakemon.name shouldBe name
    bakemon.health shouldBe health
    bakemon.level shouldBe level
    bakemon.attack shouldBe attack
}
})

```

```

class WaterBakemonFactoryTest : FunSpec({
    ...
    beforeTest {
        factory = WaterBakemonFactory()
        factory.name = name
        factory.health = health
        factory.level = level
        factory.attack = attack
    }

    test("A WaterBakemon should be created") {
        val bakemon = factory.createBakemon()
        bakemon.name shouldBe name
        bakemon.health shouldBe health
        bakemon.level shouldBe level
        bakemon.attack shouldBe attack
    }
})

```

Aquí notemos que en ambos tests tenemos el mismo código, entonces podemos pensar en una *test factory* que nos permita reusar el código común:

```

fun `a Bakemon factory can create Bakemon`(
    factory: BakemonFactory, name: String, health: Int, level: Int, attack: Int
) = funSpec {
    beforeTest {
        factory.name = name
        factory.health = health
        factory.level = level
        factory.attack = attack
    }

    test("A Bakemon should be created") {
        val bakemon = factory.createBakemon()
        bakemon.name shouldBe name
        bakemon.health shouldBe health
        bakemon.level shouldBe level
        bakemon.attack shouldBe attack
    }
}

```

```
}  
}
```

Ahora podemos reescribir nuestros tests de la siguiente manera:

```
class FireBakemonFactoryTest : FunSpec({  
    val name = "Karmander"  
    val health = 25  
    val level = 5  
    val attack = 4  
  
    include(  
        `a Bakemon factory can create Bakemon`(  
            FireBakemonFactory(), name, health, level, attack  
        )  
    )  
})
```

```
class WaterBakemonFactoryTest : FunSpec({  
    val name = "Kokodile"  
    val health = 25  
    val level = 5  
    val attack = 4  
  
    include(  
        `a Bakemon factory can create Bakemon`(  
            WaterBakemonFactory(), name, health, level, attack  
        )  
    )  
})
```

Ahora nos quedaría modificar las funciones `Bakemon can attack Bakemon` y `Bakemon equality and hashCode test` para que utilice la nueva interfaz de las factories.

```
fun `Bakemon equality and hashCode test`(...) = funSpec {  
    ...  
    beforeTest {  
        factory.name = name  
        factory.health = health  
        factory.level = level  
        factory.attack = attack  
        bakemon = factory.createBakemon()  
    }  
    ...  
}  
  
fun `Bakemon can attack Bakemon`(...) = funSpec {
```

```

...
beforeTest {
    factory.name = name
    factory.health = health
    factory.level = level
    factory.attack = attack
    bakemon = factory.createBakemon()
    enemyFactory.name = enemyName
    enemyFactory.health = enemyHealth
    enemyFactory.level = enemyLevel
    enemyFactory.attack = enemyAttack
    enemy = enemyFactory.createBakemon()
}
...
test(testNames.second) {
    factory.attack = enemyHealth * 2
    bakemon = factory.createBakemon()
    bakemon.attackBakemon(enemy)
    enemy.health shouldBe 0
}
}

```

Noten que en el test ahora podemos modificar el ataque del *Bakémon* enemigo antes de crearlo.

Naturalmente, si corremos los tests ahora, estos van a fallar.

Procedamos a modificar la interfaz `BakemonFactory` para que tenga variables de instancia que puedan ser modificadas.

```

interface BakemonFactory {
    var name: String
    var health: Int
    var level: Int
    var attack: Int
    fun createBakemon(): Bakemon
}

```

Y la implementación de las factories:

```

class FireBakemonFactory : BakemonFactory {
    override lateinit var name: String
    override var health: Int = 0
    override var level: Int = 0
    override var attack: Int = 0

    override fun createBakemon() = FireBakemon(name, health, level, attack)
}

```

```
class WaterBakemonFactory : BakemonFactory {
    override lateinit var name: String
    override var health: Int = 0
    override var level: Int = 0
    override var attack: Int = 0

    override fun createBakemon() = WaterBakemon(name, health, level, attack)
}
```

Nota. La palabra reservada `lateinit` no puede ser usada en variables de tipos numéricos, por lo que en este caso tenemos que inicializarlas con un valor por defecto.

Nos quedaría modificar los métodos `attackedBy` de las clases `FireBakemon` y `WaterBakemon` para que utilicen nuestras nuevas implementaciones de `takeDamage` de la siguiente manera:

```
class FireBakemon(name: String, health: Int, level: Int, attack: Int) :
    AbstractBakemon(name, health, level, attack) {
    override fun attackBakemon(bakemon: Bakemon) = bakemon.attackedByFireBakemon(this)

    override fun attackedByFireBakemon(bakemon: FireBakemon) =
        takeDamage(bakemon.attack)

    override fun attackedByWaterBakemon(bakemon: WaterBakemon) =
        takeWeaknessDamage(bakemon.attack)

    override fun attackedByGrassBakemon(bakemon: GrassBakemon) =
        takeResistanceDamage(bakemon.attack)
    ...
}
```

```
class WaterBakemon(
    name: String,
    health: Int,
    level: Int,
    attack: Int
) : AbstractBakemon(name, health, level, attack) {
    override fun attackBakemon(bakemon: Bakemon) = bakemon.attackedByWaterBakemon(this)

    override fun attackedByFireBakemon(bakemon: FireBakemon) =
        takeResistanceDamage(bakemon.attack)

    override fun attackedByWaterBakemon(bakemon: WaterBakemon) =
        takeDamage(bakemon.attack)

    override fun attackedByGrassBakemon(bakemon: GrassBakemon) =
        takeWeaknessDamage(bakemon.attack)
    ...
}
```

Con esto deberíamos poder correr los tests y ver que todos pasan. Nos queda hacer *commit* de los cambios.

```
git add .
git commit -m "REFACTOR Adapts code to use Double Dispatch on Bakemon attacks"
```

7.5. Ejercicios

Importante. Recuerde:

- Hacer *commit* de sus cambios luego de cada ejercicio.
- Definir el método `toString` en las clases que lo requieran.

Ejercicio 16

Smalltalk es un lenguaje de programación puramente orientado a objetos. Es un lenguaje de programación que se utilizó mucho para el desarrollo de la programación orientada a objetos. *Smalltalk* es considerado como el primer lenguaje de programación orientado a objetos y el primero con tipado dinámico. En *Smalltalk* todo es un objeto, incluso los números y los booleanos, además las instrucciones de control de flujo son métodos de los objetos.

Al ser completamente dinámico ningún objeto tiene un tipo definido en tiempo de compilación por lo que cuando se envía un mensaje, el receptor del mensaje será el que decida qué hacer con él.

1. Cree una interfaz `SmalltalkNumber` en el paquete `cl.ravenhill.lilvoices` con lo siguiente:

```
interface SmalltalkNumber {
    val value: Number

    fun plus(other: SmalltalkNumber): SmalltalkNumber
    fun minus(other: SmalltalkNumber): SmalltalkNumber
    fun times(other: SmalltalkNumber): SmalltalkNumber
    fun dividedBy(other: SmalltalkNumber): SmalltalkNumber
}
```

2. Cree una clase `SmalltalkIntegerTest` que pruebe que un número entero puede ser creado con un valor entero de *Kotlin* como parámetro.
3. Programe la clase `SmalltalkInteger` que implemente la interfaz `SmalltalkNumber` y que pueda ser creado con un valor entero de *Kotlin* como parámetro.
4. Cree una clase `SmalltalkFloatTest` que pruebe que un número flotante puede ser creado con un valor flotante de *Kotlin* como parámetro. Utilice el tipo `Float` de *Kotlin* para representar los números flotantes.
5. Programe la clase `SmalltalkFloat` que implemente la interfaz `SmalltalkNumber` y que pueda ser creado con un valor flotante de *Kotlin* como parámetro.
6. Escriba tests para probar que los números enteros pueden ser sumados, restados, multiplicados y divididos.
7. Implemente las operaciones de suma, resta, multiplicación, división, módulo y potencia entre números enteros.
8. Escriba tests para probar que los números flotantes pueden ser sumados, restados, multiplicados y divididos.
9. Implemente las operaciones de suma, resta, multiplicación y división entre números flotantes.

Para los siguientes ejercicios considere las siguientes reglas:

- Las operaciones entre números del mismo tipo retornan un número del mismo tipo.
- Las operaciones entre números de distinto tipo retornan un número del tipo más preciso (en este caso, el tipo `SmalltalkFloat`).

10. Escriba tests para probar que los números enteros y flotantes pueden ser sumados, restados, multiplicados y divididos.
11. Implemente las operaciones de suma, resta, multiplicación y división entre números enteros y flotantes utilizando *double dispatch* para desambiguar el tipo de los operandos.

Bibliografía

Double Dispatch

DoubleDispatch2022

Double Dispatch. En: *Wikipedia*. 8 de ago. de 2022. URL: https://en.wikipedia.org/w/index.php?title=Double_dispatch&oldid=1103211310 (visitado 12-02-2023).

Anot.: Page Version ID: 1103211310.

Multiple Dispatch

MultipleDispatch2023

Multiple Dispatch. En: *Wikipedia*. 9 de feb. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Multiple_dispatch&oldid=1138471332 (visitado 12-02-2023).

Anot.: Page Version ID: 1138471332.

Multiple Dispatch - Programación Avanzada Con Objetos

MultipleDispatchProgramacion

Multiple Dispatch - Programación Avanzada Con Objetos. URL: <https://sites.google.com/site/programacionhm/conceptos/multiple-dispatch> (visitado 12-02-2023).

Multiple Dispatch

MultipleDispatchFix

Multiple Dispatch: A Fix for the Problems of Single Dispatch Languages - DZone. dzone.com. URL: <https://dzone.com/articles/multiple-dispatch-fix-some> (visitado 12-02-2023).

Overload Resolution - Kotlin Language Specification

OverloadResolutionKotlin

Overload Resolution - Kotlin Language Specification. URL: <https://kotlinlang.org/spec/overload-resolution.html#overload-resolution> (visitado 08-02-2023).

Capítulo 8

Cuando todo male sal

Este capítulo podría llamarse algo así como «*Kotlin 2: La venganza*». Aquí seguiremos expandiendo el ejemplo de los capítulos anteriores, pero deteniendonos en ciertos detalles que pueden mejorarse de lo que ya hemos visto.

Comencemos implementando algunas funcionalidades nuevas para hacer más interesante nuestro juego.

Algo común que podríamos querer hacer es que un jugador pueda usar ítems en un combate. Por ejemplo, un jugador podría usar un ítem para curarse, o para aumentar su ataque.

Partamos creando una interfaz para los ítems, diremos que los ítems tienen un nombre, una descripción y una función que se ejecuta cuando el ítem es usado. Crearemos la interfaz en un paquete llamado `cl.ravenhill.items`.

```
interface Item {  
    var name: String  
    var description: String  
    fun useOn(bakemon: Bakemon)  
}
```

Ahora crearemos dos ítems, uno para curar y otro para aumentar el ataque. Los llamaremos Potion y RaiseAttack respectivamente. Una Potion cura 30 puntos de vida, y un RaiseAttack aumenta el ataque en un 10 %.

Comencemos creando los tests para los ítems.

```
class PotionTest : FunSpec({  
    lateinit var potion: Potion  
    lateinit var bakemon: FireBakemon  
  
    beforeTest {  
        potion = Potion()  
        bakemon = FireBakemon("Karmander", 100, 5, 4)  
    }  
  
    test("Potions heal 30 HP to Bakemon") {  
        potion.useOn(bakemon)  
        bakemon.health shouldBe 55  
    }  
})
```

```
})
```

```
class RaiseAttackTest : FunSpec({
  lateinit var raiseAttack: RaiseAttack
  lateinit var bakemon: FireBakemon

  beforeTest {
    raiseAttack = RaiseAttack()
    bakemon = FireBakemon("Karmander", 100, 10, 20)
  }

  test("RaiseAttack increase attack by 10%") {
    raiseAttack.useOn(bakemon)
    bakemon.attack shouldBe 22
  }
})
```

Como siempre, podemos correr los tests para ver que fallan.

Ahora veamos como implementar estos ítems.

```
class Potion : Item {
  override var name: String = "Potion"
  override var description: String = "Heals 30 HP to Bakemon"

  override fun useOn(bakemon: Bakemon) {
    bakemon.health += 30
  }
}
```

```
class RaiseAttack : Item {
  override var name: String = "Raise Attack"
  override var description: String = "Increases attack by 10%"

  override fun useOn(bakemon: Bakemon) {
    bakemon.attack = (bakemon.attack * 1.1).toInt()
  }
}
```

Notarán que ahora *IntelliJ* nos reclama attack es un **valor**, cambiémoslo a **var**.

Con esto ya deberíamos poder correr los tests y ver que pasan. Ahora hacemos *commit* y nos preparamos para el primer problema de este capítulo.

```
git add .
git commit -m "FEAT Adds items"
```

8.1. Setters

Al igual que cuando restringimos la vida mínima de un *Bakémon* a 0, también deberíamos restringir una vida máxima. Además, tendría sentido que el ataque no pueda ser negativo. Pero si se dan cuenta, estaremos haciendo todos esos chequeos cada vez que haya un método que quiera modificar la vida o el ataque de un *Bakémon*.

¡No encapsulamos el cambio!

En efecto, lo bueno es que *Kotlin* nos ofrece una forma sencilla de solucionar esto.

Setter

Un *setter* es un método que se ejecuta cada vez que se asigna un valor a una propiedad.

En esta parte voy a escribir los tests a posteriori, ya que primero tenemos que entender qué es un *setter* para entender cómo usarlo y probarlo.

En *Kotlin* podemos definir un *setter* para una propiedad utilizando la palabra reservada `set`. Puede que esto les suene familiar, ya que lo usamos antes para definir una variable que fuera accesible desde fuera de la clase pero que solo fuera modificable desde dentro de la clase; cuando usamos `private set`.

Por ejemplo, podemos definir un *setter* para el ataque de un *Bakémon* de la siguiente forma:

```
abstract class AbstractBakemon(  
    override val name: String,  
    override var health: Int,  
    override val level: Int,  
    attack: Int  
) : Bakemon {  
    override var attack: Int = attack  
        set(value) {  
            field = if (value < 0) 0 else value  
        }  
    ...  
}
```

Veamos lo que hicimos aquí. Primero, cambiamos `attack` en los parámetros del constructor a un parámetro «normal» (le quitamos el `override var`) y creamos una propiedad `attack` en el cuerpo de la clase. Luego definimos un setter con `set(value)`, donde `value` es el valor que se le asigna a la propiedad.¹ Finalmente, dentro del setter, asignamos el valor a la propiedad `field`, que es una variable que representa la propiedad (algo así como un `this` para las propiedades). En este caso, si el valor es negativo, asignamos 0, de lo contrario asignamos el valor que se le asignó a la propiedad.

De la misma forma, podemos definir un setter para la vida de un *Bakémon*, pero esta vez necesitamos que exista un límite superior para la vida. Para esto, primero definamos una propiedad que represente el límite superior de la vida en la interfaz *Bakemon*:

```
interface Bakemon {  
    val maxHealth: Int  
    ...  
}
```

¹Noten que podemos pasarle un valor inicial.

Y luego la sobrescribimos en la clase AbstractBakemon:

```
abstract class AbstractBakemon(  
    override val name: String,  
    final override val maxHealth: Int,  
    override val level: Int,  
    attack: Int  
) : Bakemon {  
    override var health: Int = maxHealth  
    set(value) {  
        field = when {  
            value > maxHealth -> maxHealth  
            value < 0 -> 0  
            else -> value  
        }  
    }  
    ...  
}
```

Ahora veamos lo que hicimos. Primero agregamos un parámetro `maxHealth` al constructor de la clase abstracta, y lo definimos como `final` para que no pueda ser sobrescrito. Luego, definimos la propiedad `health` y la inicializamos con el valor de `maxHealth`. Por último, definimos el setter de modo que si el valor es mayor que el límite superior de la vida, se asigna el límite superior de la vida, si es menor que 0, se asigna 0, y de lo contrario se asigna el valor que se le asignó a la propiedad.

Espera, ¿por qué queremos que la vida máxima no pueda ser sobrescrita?

Porque luego iniciamos la propiedad `health` con el valor de `maxHealth` y si luego sobrescribimos la propiedad en alguna de las subclases de `AbstractBakemon` el constructor podría tener comportamientos ineseperados. Para entender por qué esto es un problema veamos un ejemplo:

```
open class A {  
    open val x: Int = 0  
    init {  
        println("x = $x")  
    }  
}
```

```
class B : A() {  
    private val l = listOf(1, 2, 3)  
    override val x: Int = l.size  
}
```

```
fun main() {  
    B()  
}
```

¿Qué pasa si ejecutamos este código?

1. B llama al constructor de su superclase
2. A inicializa la propiedad x con el valor 0
3. A imprime x = 0
4. B inicializa la propiedad x con el tamaño de la lista

Esto es un problema ya que si bien definimos que x es un **valor**, vemos que tiene dos valores distintos.

En nuestro caso, el problema surge de que **todos** los miembros definidos en una interfaz son abiertos (ya que no tiene sentido que no lo sean) y al sobrescribir una componente abierta esta se mantiene abierta.

Ahora pasemos a los tests. Partamos por crear un *test factory* para el ataque de un *Bakémon*:

```
fun `Bakemon attack can be set within limits`(  
    factory: BakemonFactory,  
    name: String, health: Int, level: Int, attack: Int  
) = funSpec {  
    lateinit var bakemon: Bakemon  
  
    beforeTest {  
        factory.name = name  
        factory.health = health  
        factory.level = level  
        factory.attack = attack  
        bakemon = factory.createBakemon()  
    }  
  
    test("Bakemon attack can be set within limits") {  
        bakemon.attack shouldBe 4  
        bakemon.attack = 5  
        bakemon.attack shouldBe 5  
    }  
  
    test("Bakemon attack can't be set to a negative value") {  
        bakemon.attack shouldBe 4  
        bakemon.attack = -1  
        bakemon.attack shouldBe 0  
    }  
}
```

El test es bastante similar a lo que hemos visto hasta ahora. Y ahí está la gracia, a pesar de que definimos un método para asignar el ataque, nosotros ocupamos la propiedad `attack` como si fuera una variable cualquiera.

Lo que está sucediendo por detrás es que el compilador de *Kotlin* (al momento de traducir el programa a un lenguaje de nivel más bajo) genera un método `setAttack` y cambia todas las asignaciones a la propiedad `attack` por llamadas a este método, i.e. `bakemon.attack = 5` se traduce a `bakemon.setAttack(5)`.

Aprovechemos de agregar un *test factory* para la vida de un *Bakémon*:

```
fun `Bakemon hp is set within limits`(  
    factory: BakemonFactory,
```

```

    name: String, health: Int, level: Int, attack: Int
) = funSpec {
    lateinit var bakemon: Bakemon

    beforeTest {
        factory.name = name
        factory.health = health
        factory.level = level
        factory.attack = attack
        bakemon = factory.createBakemon()
    }

    test("Bakemon health can be set within limits") {
        bakemon.health shouldBe 25
        bakemon.health = 20
        bakemon.health shouldBe 20
    }

    test("Bakemon health can't be set outside limits") {
        bakemon.health shouldBe 25
        bakemon.health = 30
        bakemon.health shouldBe 25
        bakemon.health = -1
        bakemon.health shouldBe 0
    }
}

```

Por último, incluimos los *factories* en los tests:

```

class FireBakemonTest : FunSpec({
    ...
    include(
        `Bakemon hp is set within limits`(FireBakemonFactory(), "Karmander", 25, 5, 4)
    )
    include(
        `Bakemon attack can be set within limits`(
            FireBakemonFactory(), "Karmander", 25, 5, 4
        )
    )
})

```

```

class WaterBakemonTest : FunSpec({
    ...
    include(
        `Bakemon hp is set within limits`(WaterBakemonFactory(), "Kokodile", 25, 5, 4)
    )
    include(
        `Bakemon attack can be set within limits`(

```



```
        WaterBakemonFactory(), "Kokodile", 25, 5, 4
    )
}
})
```

Con esto, podemos correr los tests para ver que pasan todos menos el de las pociones, esto se debe a que agregamos una cantidad de vida máxima que no existía antes. Para solucionarlo basta cambiar el test de `PotionTest` por:

```
test("Potions heal 30 HP to Bakemon") {
    bakemon.health = 100
    potion.useOn(bakemon)
    bakemon.health shouldBe 130
}
```

Recordemos correr los tests para ver que todo está bien y luego hacer un commit:

```
git add .
git commit -m "FEAT Adds Bakemon attack and health bounds"
```

No sé si necesite decirles a estas alturas, pero tenemos un problema. Si bien los *Bakémon* pueden tener un ataque y vida máximos, no hay nada que nos impida usar el constructor de *Bakemon* para crear un *Bakémon* con vida y ataque negativos. ¿Qué podemos hacer para evitar esto? Existen varias formas de lidiar con esto, y la respuesta depende de cómo queremos que se ocupe nuestro código, pero una solución posible sería utilizar excepciones. Sin embargo, hay un par de cosas que deberíamos entender antes de continuar.

8.2. Java Virtual Machine

Una *Máquina Virtual de Java* (JVM) es un programa que ejecuta código escrito en *bytecode*² de *Java*. Pero, a pesar de que el *bytecode* es de *Java*, existen varios *compiladores* que pueden generar *bytecode* de *Java* para otros lenguajes. Por ejemplo, *Kotlin* puede ser compilado a *bytecode* de *Java* y ejecutado en una JVM, así como *Scala* y *Groovy*.

Cuando corremos un programa en *Kotlin* lo hacemos en dos grandes pasos.

- **Compilación:** El código fuente de *Kotlin* es compilado a *bytecode* de *Java*.
- **Ejecución:** El *bytecode* de *Java* es interpretado por una JVM.

Nota. El proceso de compilación es sumamente complejo,³ y no es el objetivo de este libro explicarlo.

De esta forma, la JVM funcionará como una capa de abstracción entre el código fuente de *Kotlin* y el hardware de la computadora, permitiendo que el código fuente de *Kotlin* sea ejecutado en cualquier computadora que tenga una JVM instalada. En otras palabras, a nuestro computador no le importa si escribimos el programa en *Kotlin* o en *Java*, lo que implica que podemos tener aplicaciones que tengan partes escritas en *Kotlin* y otras partes escritas en *Java*. Esto es sumamente útil si estamos tratando de migrar una aplicación de *Java* a *Kotlin*, ya que podemos ir migrando las partes más críticas primero, y luego ir migrando el resto del código poco a poco. Otra implicancia de esto es que podemos usar librerías escritas en *Java*⁴ en nuestro código de *Kotlin*, y viceversa.

²El *bytecode* o (código intermedio) es código generado por un compilador escrito en un lenguaje de programación diseñado para ser eficientemente interpretado por una máquina.

³<https://github.com/JetBrains/kotlin/tree/master/compiler>

⁴Para no tener que reinventar la rueda.

Nota. Dentro de nuestro proyecto, en la carpeta `build/classes` podemos encontrar los archivos `.class` generados por el compilador de Kotlin.

Una vez que el programa es compilado, la máquina virtual se encarga de correr el programa y manejar la memoria. La memoria de la JVM se divide en áreas llamadas *Run-Time Data Areas*, pero nos enfocaremos en dos de ellas: *Memory Heap* y *Runtime Stack*

Memory Heap

El *Memory Heap*^a (lit. montón de memoria) es un sector de memoria compartida por todo el programa y que se encarga de almacenar todos los objetos que creamos en nuestro programa.

^aEl *Memory Heap* no es un heap

Runtime Stack

El *Runtime Stack* (pila de ejecución) es un área de memoria que se encarga de almacenar los *frames* de ejecución de nuestro programa. Un *frame* es un área de memoria que se encarga de almacenar los datos locales de un método y resultados parciales de la ejecución de un método.^a Cada vez que llamamos a un método, se crea un nuevo *frame* en la pila de ejecución. Cuando el método termina, el *frame* es removido de la pila de ejecución. La pila de ejecución es una estructura de datos *LIFO* (Last In, First Out).

^aEl valor de retorno de una función es un ejemplo de un resultado parcial.

Para entender mejor cómo funciona la memoria veamos el siguiente ejemplo:

```
fun main() {
    val a = 1
    val b = "2"
    val c = sum(a, b)
}

fun sum(x: Int, y: String): Int {
    return x + y.toInt()
}
```

1. Agregamos el método `main` a la pila de ejecución.
2. Creamos las variables `a` y `b` en la pila de ejecución, como la variable `a` es de tipo `Int` ocupa una cantidad de memoria constante así que se almacena directamente en la pila de ejecución, mientras que la variable `b` es de tipo `String` y su tamaño depende de la cadena que se le asigne, por lo que se almacena en el *Memory Heap* y se guarda una referencia a la cadena en la pila de ejecución.
3. Llamamos al método `sum` y creamos un nuevo *frame* en la pila de ejecución.
4. Creamos las variables `x` y `y` en el nuevo *frame* de la pila de ejecución.
5. Asignamos los valores de las variables `a` y `b` a las variables `x` y `y` respectivamente.
6. El resultado intermedio de la suma se almacena en el *stack* y la función retorna el resultado de la suma.
7. El *frame* de la función `sum` es removido de la pila de ejecución.
8. El resultado de la suma se almacena en la variable `c`.
9. El *frame* de la función `main` es removido de la pila de ejecución.

En algunos lenguajes de programación, como C y C++, la memoria es administrada por el programador, lo que significa que es el programador quien se encarga de liberar la memoria que ya no se está usando. En *Kotlin* esto no es necesario, ya que la memoria es administrada por el *Garbage Collector* (GC).

Garbage collection

El *Garbage Collector* es un objeto que se encarga de liberar la memoria que ya no se está usando.

El algoritmo utilizado por el *Garbage Collector* es el *Mark and Sweep* (marcar y barrer). Este algoritmo funciona de la siguiente forma:

1. El *Garbage Collector* recorre la pila de ejecución y marca todos los objetos que están siendo utilizados.
2. El *Garbage Collector* recorre el *Memory Heap* y elimina todos los objetos que no están marcados.

Los detalles de cómo decide qué objetos marcar y qué objetos eliminar es un tema bastante complejo, pero en general el *Garbage Collector* es bastante bueno y no deberíamos tener que preocuparnos por la memoria.

8.3. Excepciones

En programación generalmente vamos a tener que tratar con situaciones en las que tengamos que decidir qué hacer cuando algo no sale como esperábamos. Por ejemplo, ¿qué sucede si intentamos abrir un archivo que no existe? Podríamos imprimir un mensaje de error en pantalla y no hacer nada más, pero en la práctica esto no es muy útil. De hecho, ¿cómo testearmos que el programa imprime algo en pantalla? Un mensaje en pantalla en general es información que deja de existir una vez que se imprime y no podemos hacer nada con ella. Existen formas de capturar la salida de un programa, pero en general es algo complejo de hacer.

Una forma más útil de tratar con situaciones en las que algo no sale como esperábamos es lanzar una excepción.

Excepción

Una excepción es un objeto que representa un comportamiento excepcional que ocurre durante la ejecución de un programa.

La principal ventaja de las excepciones es que estas pueden ser arrojadas y capturadas en cualquier parte del programa. Cuando una excepción es arrojada, el programa deja de ejecutarse y se salta a la parte del código que la captura, para esto, la máquina virtual busca en la pila de ejecución un *frame* que capture la excepción y pasa el control a ese *frame* (los otros *frames* son descartados). Si no se encuentra ningún *frame* que capture la excepción, se imprime un mensaje de error y el programa termina.

En *Kotlin* las excepciones son un tipo particular de objetos que heredan de la clase *Throwable*. Existen dos tipos de *Throwable*: *Exception* y *Error*.

Los *Error* son errores que ocurren en la máquina virtual y que no deben ser capturados. Por ejemplo, si se lanza una *StackOverflowError* es porque la pila de ejecución se ha llenado y no hay más espacio para seguir ejecutando el programa. El nombre de las clases que heredan de *Error* terminan en *Error* para indicar que no deben ser capturadas.

Las *Exception* por otro lado, son errores que ocurren dentro de nuestro programa y que pueden ser manejadas por el mismo. Las *Exception* pueden clasificarse en dos tipos: *excepciones propias* y *excepciones de la librería estándar*. Las *excepciones propias* son las que creamos nosotros para indicar que algo salió mal en nuestro programa. Las *excepciones de la librería estándar* son las que lanzan las funciones de la librería estándar de *Kotlin* cuando algo sale mal, este tipo de excepciones pueden ser capturadas, pero no es recomendable hacerlo ya que, al no ser parte de nuestro código, no tenemos seguridad de cuál es la razón por la que se lanzó la excepción.

8.3.1. Arrojando excepciones

Para lanzar una excepción, usamos la palabra reservada `throw` seguida de una expresión que evalúe a un objeto de tipo `Throwable`. Por ejemplo, si queremos lanzar una excepción de tipo `Exception` podemos hacerlo de la siguiente forma:

```
throw Exception("Error message")
```

Sin embargo, en la práctica es una mala práctica lanzar una excepción de tipo `Exception` ya que no es específica al problema. En su lugar, es mejor crear una excepción propia que herede de `Exception` y que sea más específica al problema que estamos tratando de resolver.

Para crear una excepción propia, debemos crear una clase que herede de `Exception`. Por ejemplo, si queremos crear una excepción que indique que un archivo no existe, podemos hacerlo de la siguiente forma:

```
class FileNotFoundException(filename: String) : Exception("File $filename not found")
```

Luego podríamos utilizar esta excepción de la siguiente forma:

```
fun openFile(filename: String): String {
    if (File(filename).exists()) {
        return File(filename).readText()
    } else {
        throw FileNotFoundException(filename)
    }
}

fun main() {
    openFile("nonexistent.txt")
}
```

Esto entregará un resultado como el siguiente:

```
Exception in thread "main" FileNotFoundException: File nonexistent.txt not found
    at MainKt.openFile(Main.kt:9)
    at MainKt.main(Main.kt:14)
    at MainKt.main(Main.kt)
```

Como podemos ver, la excepción se imprime en pantalla junto con la pila de ejecución. Esto nos permite saber en qué parte del programa se lanzó la excepción y por qué.

Adicionalmente, existen dos formas de documentar funciones que lanzan excepciones. La primera es agregando el tag `@throws` seguido del nombre de la excepción que puede ser lanzada por la función y su causa en la documentación de la función. Por ejemplo, si queremos documentar que la función `openFile` puede lanzar una excepción de tipo `FileNotFoundException`, podemos hacerlo de la siguiente forma:

```
/**
 * Opens a file and returns its contents.
 *
 * @throws FileNotFoundException if the file does not exist.
 */
```

```
fun openFile(filename: String): String {
    if (File(filename).exists()) {
        return File(filename).readText()
    } else {
        throw FileNotFoundException(filename)
    }
}
```

La segunda forma es utilizando el tag `@Throws` seguido de una lista de excepciones que puede lanzar la función. Por ejemplo, si queremos documentar que la función `openFile` puede lanzar una excepción de tipo `FileNotFoundException`, podemos hacerlo de la siguiente forma:

```
/**
 * Opens a file and returns its contents.
 */
@Throws(FileNotFoundException::class)
fun openFile(filename: String): String {
    if (File(filename).exists()) {
        return File(filename).readText()
    } else {
        throw FileNotFoundException(filename)
    }
}
```

Esta segunda forma es útil más que nada para poder utilizar estas funciones desde *Java*, ya que *Java* requiere que las funciones que lanzan excepciones lo declaren en la firma de la función. En *Kotlin* esto no es necesario así que la primera forma es la más recomendada.

8.3.2. Atrapando excepciones

En *Kotlin* podemos atrapar excepciones con la expresión `try ... catch ... finally`, esto se divide en tres partes:

- **try**: Contiene el código que puede lanzar una excepción.
- **catch**: Contiene el código que se ejecuta si se lanza una excepción.
- **finally**: Contiene el código que se ejecuta **siempre**⁵, ya sea que se lance una excepción o no.

Por ejemplo, si queremos abrir un archivo y mostrar su contenido en pantalla, podemos hacerlo de la siguiente forma:

```
fun main() {
    try {
        val file = openFile("file.txt")
        println(file)
    } catch (e: FileNotFoundException) {
        println(e.message)
    } finally {
        println("Finally")
    }
}
```

⁵SIEMPRE

```
}
```

Esto entregará un resultado como el siguiente:

```
File file.txt not found
Finally
```

Siempre que usemos una expresión `try` debemos tener una o más expresiones `catch` que capturan las excepciones que pueden ser lanzadas por el código dentro de la expresión `try` y/o una expresión `finally` que se ejecuta siempre, ya sea que se lance una excepción o no.

Nota. Como `try ... catch ... finally` es una expresión, podemos usarla en cualquier lugar donde se pueda usar una expresión. Esto significa que podemos usarla para asignar una variable de la misma forma que lo haríamos con una expresión `if` o `when`.

8.3.3. Usando excepciones

Volvamos a nuestro código de *Bakemon* y veamos como podemos usar las excepciones para manejar los casos de borde.

Partamos por crear un paquete `cl.ravenhill.bakemon.exceptions` en el directorio de tests y creemos una clase `InvalidStatValueExceptionTest` y probemos que una excepción puede ser creada correctamente:

```
class InvalidStatExceptionTest : FunSpec({
  test("An InvalidStatException can be created with a message") {
    val exception = InvalidStatException("health", -1)
    exception.message shouldBe "Invalid health: -1"
  }
})
```

Corramos el test para ver que falla y pasemos a implementar la clase `InvalidStatException`:

```
class InvalidStatException(statName: String, statValue: Int) :
  Exception("Invalid $statName: $statValue")
```

Ahora sí los tests deberían pasar.

Veamos como podemos usar esta excepción en nuestro código de *Bakemon*. Lo primero que necesitamos es un test que vea que la excepción es lanzada cuando se intenta crear un *Bakemon* con un stat inválido. Partamos por crear un *test factory* que verifique que la excepción es lanzada cuando se intenta crear un *Bakemon* con un stat inválido:

```
fun `a Bakemon health cannot be set to a non positive value`(
  factory: BakemonFactory, name: String, level: Int, attack: Int
) = funSpec {

  beforeTest {
    factory.name = name
    factory.health = -1
    factory.level = level
    factory.attack = attack
  }
}
```

```
test(
    "Creating a Bakemon with a non positive health should throw an exception"
) {
    shouldThrow<InvalidStatException> { factory.createBakemon() }
}
}
```

Aquí notarán la función *shouldThrow* que es una función de *Kotest* que nos permite verificar que una excepción es lanzada. La sintaxis de esta función es la siguiente:

```
shouldThrow<ExceptionType> {
    // Code that should throw an exception
}
```

Ahora, podemos usar nuestra factory en nuestros tests:

```
class FireBakemonTest : FunSpec({
    ...
    include(
        `a Bakemon health cannot be set to a non positive value`(
            FireBakemonFactory(), "Karmander", 10, 25
        )
    )
})
```

```
class WaterBakemonTest : FunSpec({
    ...
    include(
        `a Bakemon health cannot be set to a non positive value`(
            WaterBakemonFactory(), "Kokodile", 10, 25
        )
    )
})
```

Si corremos los tests ahora, veremos que fallan.

Ahora, podemos implementar esta funcionalidad en la clase *AbstractBakemon*:

```
abstract class AbstractBakemon(
    override val name: String,
    final override val maxHealth: Int,
    override val level: Int,
    attack: Int
) : Bakemon {
```

```

init {
    if (maxHealth < 0) {
        throw InvalidStateException("maxHealth", maxHealth)
    }
}
...
}

```

Ahora podemos correr los tests y ver que pasan. Con esto ya podemos hacer *commit* de nuestro código.

```

git add .
git commit -m "FEAT Adds checks for maxHealth in AbstractBakemon"

```

Ejercicio 8.1. Implemente la funcionalidad de manejo de excepciones para los stats de ataque y nivel.

8.4. Getters

Comencemos a implementar a los jugadores del juego. Llamaremos a estos jugadores *Bakemon trainers* (entrenadores de Bakemon).

Un entrenador tiene un nombre, un equipo de Bakemon, y un inventario de ítems.

Comencemos por crear una clase `TrainerTest` en el paquete `cl.ravenhill.bakemon`:

```

class TrainerTest : FunSpec({
    val team = mutableListOf<Bakemon>(
        FireBakemon("Karmander", 130, 5, 20),
        WaterBakemon("Kokodile", 130, 5, 20)
    )
    val items = mutableListOf(Potion(), RaiseAttack())
    lateinit var trainer: Trainer

    beforeTest {
        trainer = Trainer("Ack", team, items)
    }

    test("A Trainer can be created with a name, a team and items") {
        trainer.name shouldBe "Ack"
        trainer.team shouldBe team
        trainer.items shouldBe items
    }
})

```

Como siempre, podemos correr los tests para ver que fallan.

Ahora veamos como implementar a los entrenadores. Para esto crearemos una clase `Trainer` como sigue:


```
class Trainer(  
    val name: String,  
    val team: MutableList<Bakemon>,  
    val items: MutableList<Item>  
)
```

Ahora corramos los tests para ver que pasan.

Pero

¿Tenemos un problema?

Sip.

Tanto `team` como `items` son `val`ores, pero como ya vimos, eso no significa que no podamos modificarlos. Ya hemos dicho muchas veces que nos interesa *encapsular el cambio*, y algo esencial para lograrlo es que un objeto maneje su propio estado. Pero si podemos modificar `team` y `items` desde afuera, entonces el objeto no tiene control de lo que pasa con su estado. Pero nos gustaría poder agregar y quitar `Bakemon` e ítems de los entrenadores, por lo que necesitamos que las propiedades sean mutables...

¿Qué podemos hacer?

Getter

Un *getter* es un método que se ejecuta cada vez que se intenta acceder a una propiedad.

En *Kotlin*, podemos definir un *getter* utilizando la palabra reservada `get`. La sintaxis es muy similar a la de un *setter* con la diferencia de que un *getter* retorna un valor y no tiene parámetros.

Para nuestro ejemplo, usaremos una técnica llamada *backing field* (campo de respaldo), que consiste en crear una propiedad privada que almacena el valor de la propiedad pública. Luego, el *getter* de la propiedad pública retorna una copia del valor de la propiedad privada. De esta forma, el objeto tiene control sobre el estado de la propiedad, y no podemos modificarlo desde fuera sin que el objeto se entere. Veamos cómo hacemos esto:

```
class Trainer(  
    val name: String,  
    team: MutableList<Bakemon>,  
    items: MutableList<Item>  
) {  
    private val _team = team  
    private val _items = items  
    val team: List<Bakemon>  
        get() {  
            return _team.toList()  
        }  
    val items: List<Item>  
        get() {  
            return _items.toList()  
        }  
}
```

Veamos lo que estamos haciendo. Primero, creamos dos propiedades privadas llamadas `_team` y `_items` que almacenan los valores de las propiedades públicas `team` y `items`. Luego, creamos dos propiedades públicas llamadas `team` y `items`

que tienen un *getter* que retorna una copia inmutable (con `toList()`) de los valores de las propiedades privadas. Corramos los tests para ver que no hayamos roto nada y luego hagamos un commit.

```
git add .
git commit -m "FEAT Adds Trainer class"
```

8.5. Ejercicios

Ejercicio 17

Para los siguientes fragmentos de código indique qué se imprime en pantalla y por qué (basta con una idea general, no es necesario que sea exacto).

1.

```
fun fibonacci(n: Int): Int {
    return fibonacci(n - 1) + fibonacci(n - 2)
}
```

```
fun main() {
    println(fibonacci(5))
}
```

2.

```
fun foo(): String {
    try {
        return "foo"
    } finally {
        return "bar"
    }
}
```

```
fun main() {
    println(foo())
}
```

3.

```
fun foo(): String {
    try {
        return "foo"
    } finally {
        throw Exception()
    }
}
```

```
fun main() {
    println(foo())
}
```

4.

```
fun foo(): String {
    try {
        throw Exception()
    } finally {
        return "bar"
    }
}
```

```

    }
}

fun main() {
    println(foo())
}

5. fun openFile(filename: String) = try {
    File(filename)
} catch (e: FileNotFoundException) {
    throw Exception("File $filename not found")
} catch (e: Exception) {
    throw Exception("Unknown error")
}

fun main() {
    // Note: file.txt does not exist
    println(openFile("file.txt"))
}

6. fun foo(): String {
    try {
        throw IllegalStateException("Error message")
    } catch (e: Exception) {
        return "foo"
    } catch (e: IllegalStateException) {
        return "bar"
    }
}

fun main() {
    println(foo())
}

```


Bibliografía

baeldung: JVM Garbage Collectors Baeldung	baeldungJVMGarbageCollectors2017
---	----------------------------------

baeldung. *JVM Garbage Collectors | Baeldung*. 18 de abr. de 2017. URL: <https://www.baeldung.com/jvm-garbage-collectors> (visitado 15-02-2023).

Bytecode	Bytecode2022
----------	--------------

Bytecode. En: *Wikipedia*. 29 de nov. de 2022. URL: <https://en.wikipedia.org/w/index.php?title=Bytecode&oldid=1124581452> (visitado 15-02-2023).

Anot.: Page Version ID: 1124581452.

Chapter 2. The Structure of the Java Virtual Machine	ChapterStructureJava
--	----------------------

Chapter 2. *The Structure of the Java Virtual Machine*. URL: <https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5> (visitado 15-02-2023).

Exception Handling	ExceptionHandling2022
--------------------	-----------------------

Exception Handling. En: *Wikipedia*. 9 de dic. de 2022. URL: https://en.wikipedia.org/w/index.php?title=Exception_handling&oldid=1126399307 (visitado 15-02-2023).

Anot.: Page Version ID: 1126399307.

Exceptions - Kotlin Language Specification	ExceptionsKotlinLanguage
--	--------------------------

Exceptions - *Kotlin Language Specification*. URL: <https://kotlinlang.org/spec/exceptions.html> (visitado 15-02-2023).

Exceptions Kotlin	ExceptionsKotlin
---------------------	------------------

Exceptions | *Kotlin*. Kotlin Help. URL: <https://kotlinlang.org/docs/exceptions.html> (visitado 15-02-2023).

hotkey: Answer to "Kotlin Calling Non Final Function in Constructor Works"	hotkeyAnswerKotlinCalling2018
--	-------------------------------

hotkey. *Answer to "Kotlin Calling Non Final Function in Constructor Works"*. Stack Overflow. 7 de mayo de 2018. URL: <https://stackoverflow.com/a/50222496/6037217> (visitado 15-02-2023).

Java Bytecode	JavaBytecode2023
---------------	------------------

Java Bytecode. En: *Wikipedia*. 28 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Java_bytecode&oldid=1136125917 (visitado 15-02-2023).

Anot.: Page Version ID: 1136125917.

Java Virtual Machine. En: *Wikipedia*. 28 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Java_virtual_machine&oldid=1136126142 (visitado 15-02-2023).

Anot.: Page Version ID: 1136126142.

Properties | Kotlin. Kotlin Help. URL: <https://kotlinlang.org/docs/properties.html> (visitado 14-02-2023).

Capítulo 9

En efecto

En este capítulo seguiremos implementando *Kygo*, un proyecto que tenemos triste y abandonado desde hace varios capítulos. Hasta ahora tenemos tres tipos de cartas: *Monstruos*, *Mágicas* y *Trampas*. Pero tanto las cartas mágicas como las trampas no hacen nada. En este capítulo vamos a implementar efectos de modo que las cartas mágicas y trampas puedan hacer algo.

Para empezar, necesitamos agregar las dependencias de *Kotest* a nuestro proyecto. Para hacer esto necesitaremos transformar nuestro proyecto en un proyecto *Gradle*. Para hacer esto, iremos a *File* → *New* → *Project*.... Aquí, haremos un proyecto de tipo *Gradle* de la misma forma que antes, lo llamaremos *kygo* y lo ubicaremos en la misma carpeta que el proyecto del capítulo 4 (es importante que el proyecto se llame de la misma forma que el de ese capítulo ya que estamos intentando «sobrescribir» el proyecto).

Una vez que hayamos creado el proyecto, podemos configurarlo de la misma forma que hicimos en el apartado 5.2.

Con esto, de la misma forma que en ese capítulo, podemos hacer *commit* de lo que hicimos:

```
git add .
git commit -m "PROJECT Migration to Gradle"
```

9.1. Efectos: Primer intento

Para estos ejemplos consideremos que queremos implementar tres cartas mágicas:

- *Crimson Beverage*: Aumenta en 500 la vida de un jugador.
- *Indigo Potion*: Aumenta en 400 la vida de un jugador.
- *Saucepan of Avidity*: Roba 2 cartas del mazo del jugador.

Comencemos por crear tests para estas cartas. En el paquete `cl.ravenhill.kygo.cards` comencemos creando las clases `CrimsonBeverageTest` y `IndigoPotionTest`:

```
class CrimsonBeverageTest : FunSpec({
    val initHealth = 8000
    lateinit var player: Player
    lateinit var crimsonBeverage: CrimsonBeverage
```

```

beforeTest {
    player = Player("Jaiva", initHealth, mutableListOf())
    crimsonBeverage = CrimsonBeverage()
}

test("Crimson Beverage should increase 500 HP") {
    player.health shouldBe initHealth
    crimsonBeverage.useOn(player)
    player.health shouldBe initHealth + 500
}
})

```

```

class IndigoPotionTest : FunSpec({
    val initHealth = 8000
    lateinit var player: Player
    lateinit var indigoPotion: IndigoPotion

    beforeTest {
        player = Player("Jaiva", initHealth, mutableListOf())
        indigoPotion = IndigoPotion()
    }

    test("Indigo Potion should increase 400 HP") {
        player.health shouldBe initHealth
        indigoPotion.useOn(player)
        player.health shouldBe initHealth + 400
    }
})

```

Con los tests podemos empezar a pensar en la implementación concreta. Podríamos decir que cada tipo de carta mágica es una especialización de una carta mágica así que tiene sentido querer implementar las cartas que queremos como subclases de MagicCard. Primero, transformemos la clase MagicCard en una clase abstracta y renombrémosla AbstractMagicCard y agreguemos un método abstracto useOn(Player).

```

abstract class AbstractMagicCard(
    name: String, text: String, serializer: CardSerializer) :
    AbstractCard(name, text, serializer) {
    abstract fun useOn(player: Player)
}

```

Ahora, podemos implementar las cartas mágicas como subclases de AbstractMagicCard.

```

class CrimsonBeverage :
    AbstractMagicCard("Crimson Beverage", "Increase 500 health", XmlCardSerializer()) {
    override fun useOn(player: Player) {
        player.health += 500
    }
}

```



```
}
```

```
class IndigoPotion :  
    AbstractMagicCard("Indigo Potion", "Increase 400 health", XmlCardSerializer()) {  
    override fun useOn(player: Player) {  
        player.health += 400  
    }  
}
```

Ahora, adaptamos la clase Player para poder modificar su salud utilizando un *setter* y mejoremos un poco su diseño:

```
class Player(  
    val name: String,  
    health: Int,  
    val deck: MutableList<Card>  
) {  
    var health: Int =  
        if (health > 0) health else throw InvalidStatException("Health", health)  
    set(value) {  
        field = max(value, 0)  
    }  
  
    fun takeDamage(damage: Int) {  
        this.health -= damage  
    }  
}
```

Donde `InvalidStatException` está definida de la misma forma que en el capítulo anterior.

Ahora podemos ejecutar los tests y ver que todo funciona como esperábamos.

Antes de continuar con la implementación de la carta Saucepan of Avidity vamos a refactorizar un poco el código.

Primero notarán que las dos cartas mágicas que hemos implementado tienen un comportamiento muy similar: aumentan la vida del jugador en un valor fijo. Podríamos decir que la única diferencia entre ambas es el valor de la vida que se aumenta. Tiene sentido entonces agrupar el comportamiento común en una clase abstracta y dejar que las subclases implementen el valor de la vida que se aumenta.

```
abstract class AbstractHealthMagicCard(  
    name: String, text: String, serializer: CardSerializer, private val health: Int  
) : AbstractMagicCard(name, text, serializer) {  
    override fun useOn(player: Player) {  
        player.health += health  
    }  
}
```

Y reimplementamos las cartas mágicas como subclases de `AbstractHealthMagicCard`.

```
class CrimsonBeverage : AbstractHealthMagicCard(
    "Crimson Beverage",
    "Increase 500 health",
    XmlCardSerializer(),
    500
)
```

```
class IndigoPotion : AbstractHealthMagicCard(
    "Indigo Potion",
    "Increase 400 health",
    XmlCardSerializer(),
    400
)
```

Ahora podemos ejecutar los tests y ver que todo sigue funcionando como esperábamos.

Nos queda implementar la carta Saucepan of Avidity, como costumbre, primero escribimos el test.

```
class SaucepanOfAvidityTest : FunSpec({
    val initHealth = 8000
    lateinit var player: Player
    lateinit var saucepanOfAvidity: SaucepanOfAvidity

    beforeTest {
        player = Player(
            "Jaiva",
            initHealth,
            mutableListOf(CrimsonBeverage(), IndigoPotion())
        )
        saucepanOfAvidity = SaucepanOfAvidity()
    }

    test("Saucepan of Avidity should draw 2 cards") {
        player.hand shouldBe emptyList()
        saucepanOfAvidity.useOn(player)
        player.hand.size shouldBe 2
        player.deck.size shouldBe 0
    }
})
```

Corramos el test para ver que todo falla como esperamos y ahora implementemos la carta.

```
class SaucepanOfAvidity : AbstractMagicCard(
    "Saucepan of Avidity",
    "Draw 2 cards",
    XmlCardSerializer()
) {
```

```

    override fun useOn(player: Player) {
        player.draw(2)
    }
}

```

Y agregamos lo necesario a la clase Player para que el test pase.

```

class Player(...) {
    ...
    private val _hand: MutableList<Card> = mutableListOf()
    val hand: List<Card>
        get() = _hand.toList()

    fun draw(n: Int = 1) {
        for (i in 1..n) {
            if (deck.isEmpty()) {
                break
            }
            _hand.add(deck.removeAt(0))
        }
    }
}

```

Con esto debiera bastar para que los tests pasen. Hagamos *commit*, y pasemos a revisar nuestro diseño.

```

git add .
git commit -m "FEAT Adds Magic Cards"

```

9.2. Strategy Pattern

Ya deberían empezar a ver el problema en el que nos estamos metiendo. Cada vez que queramos agregar una nueva carta mágica, tendremos que crear una nueva subclase de `AbstractMagicCard` y agregar un nuevo método `useOn(Player)`. Muchas cartas tendrán efectos similares, así que vamos a poder aprovechar la herencia para no repetir tanto código, pero a medida que el problema se va complejizando, por ejemplo, si ahora queremos agregarle efecto a otros tipos de cartas, como las cartas trampa o monstruo, vamos a tener que los mismos efectos (o variaciones de los mismos) en distintos tipos de cartas.

Una manera de resolver este problema es utilizando el principio de *composition over inheritance* que dice que las clases debieran poder lograr un comportamiento polimórfico a través de la composición de otras clases en lugar de la herencia. En este caso, en lugar de crear una jerarquía de clases que represente las cartas, vamos a crear una jerarquía de clases que represente los efectos de las cartas.

Strategy Pattern

Problema: Tenemos clases relacionadas que se diferencian solamente en su comportamiento.

Solución: Extraer el comportamiento que varía a una clase separada y hacer que las clases que se diferencian se compongan de la clase que contiene el comportamiento que varía.

El patrón Strategy es un patrón de diseño de comportamiento que nos permite definir una familia de algoritmos, encapsular cada uno de ellos y hacer que los algoritmos sean intercambiables dentro de un objeto.

La figura 9.1 muestra la estructura del patrón Strategy.

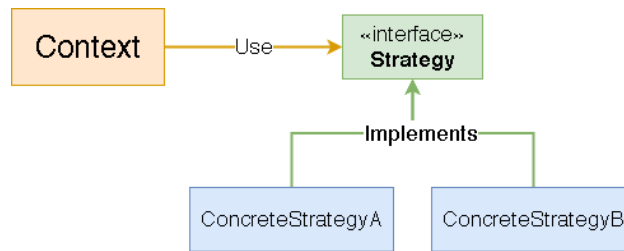


Figura 9.1: Estructura del patrón Strategy

9.3. Efectos: segundo intento

Veamos como podemos utilizar el patrón *Strategy* para implementar los efectos de las cartas.

Primero, creamos un paquete `cl.ravenhill.kygo.effects` dentro de la carpeta `src` y dentro de este paquete creamos una interfaz `Effect` con el método `Effect::useOn(Player)`.

```
interface Effect {
    fun useOn(player: Player)
}
```

Ahora, podemos crear un test para probar que un efecto que modifique la vida de un jugador funcione correctamente.

```
class ModifyHealthEffectTest : FunSpec({
    lateinit var modifyHealthEffect: ModifyHealthEffect

    beforeTest {
        modifyHealthEffect = ModifyHealthEffect(500)
    }

    test("ModifyHealthEffect should modify health by the given amount") {
        val player = Player("Jaiva", 8000, mutableListOf())
        player.health shouldBe 8000
        modifyHealthEffect.useOn(player)
        player.health shouldBe 8500
    }
})
```

Comprobemos que el test falla.

Ahora podemos implementar la clase `ModifyHealthEffect` de la siguiente manera:

```
class ModifyHealthEffect(private val amount: Int) : Effect {
    override fun useOn(player: Player) {
```

```

        player.health += amount
    }
}

```

Si corremos el test ahora, veremos que pasa. Nos queda implementar el efecto de robar cartas.

Partimos de un test que nos asegure que el efecto de robar cartas funciona correctamente.

```

class DrawCardsEffectTest : FunSpec({
    lateinit var drawCardsEffect: DrawCardsEffect

    beforeTest {
        drawCardsEffect = DrawCardsEffect(2)
    }

    test("DrawCardsEffect should draw the given amount of cards") {
        val player = Player(
            "Jaiva", 8000, mutableListOf(
                MagicCard(
                    "Saucepan of Avidity",
                    "Draw 2 cards",
                    DrawCardsEffect(2),
                    XmlCardSerializer()
                ),
                MagicCard(
                    "Crimson Beverage",
                    "Heal 500 HP",
                    ModifyHealthEffect(500),
                    XmlCardSerializer()
                )
            )
        )
        player.hand.size shouldBe 0
        drawCardsEffect.useOn(player)
        player.hand.size shouldBe 2
    }
})

```

Ahora corremos el test para ver que falla y pasamos a implementar el efecto.

```

class DrawCardsEffect(private val amount: Int) : Effect {
    override fun useOn(player: Player) = player.draw(amount)
}

```

Si corremos los tests ahora veremos que todo funciona como esperábamos. Aprovechemos de hacer *commit* de nuestro trabajo.

```
git add .
git commit -m "FEAT Adds effects"
```

9.4. *Data classes*

Actualizar la interfaz

```
interface Card {
    val name: String
    val text: String
    val effect: Effect
    val serializer: CardSerializer
}
```

Agregamos efectos a las cartas

```
abstract class AbstractCard(
    override val name: String,
    override val text: String,
    override val effect: Effect,
    override val serializer: CardSerializer
) : Card {

    fun toFile(filename: String) = serializer.toFile(this, filename)

    fun serialize() = serializer.serialize(this)

    fun useOn(player: Player) = effect.useOn(player)
}
```

Noten que ahora que tenemos los efectos definidos como clases separadas, la implementación de las cartas mágicas que definimos dejan de tener sentido. Por lo tanto, las vamos a eliminar. Pero si borramos esas clases también pierde sentido que las cartas mágicas sean clases abstractas, así que podríamos cambiarla por una clase concreta. Con esto, la clase `MagicCard` quedaría así:

```
class MagicCard(
    name: String,
    text: String,
    effect: Effect,
    serializer: CardSerializer
) : AbstractCard(name, text, effect, serializer)
```

También debemos actualizar las cartas de monstruos y trampas.

```

class MonsterCard(
    name: String,
    text: String,
    val attack: Int,
    effect: Effect,
    serializer: AbstractCardSerializer
) : AbstractCard(name, text, effect, serializer) {
    fun attack(player: Player) {
        player.takeDamage(this.attack)
    }
}

```

```

class TrapCard(
    name: String,
    text: String,
    effect: Effect,
    serializer: CardSerializer
) : AbstractCard(name, text, effect, serializer)

```

Ahora actualicemos hagamos un nuevo test para probar que las cartas funcionan correctamente. Para eso creemos una clase `MagicCardTest` y modelemos cada una de las cartas mágicas que habíamos definido como un test.

```

class MagicCardTest : FunSpec({
    lateinit var player: Player

    beforeTest {
        player = Player(
            "Jaiva", 8000, mutableListOf(
                MagicCard(
                    "Saucepan of Avidity",
                    "Draw 2 cards",
                    DrawCardsEffect(2),
                    XmlCardSerializer()
                ),
                MagicCard(
                    "Crimson Beverage",
                    "Heal 500 HP",
                    ModifyHealthEffect(500),
                    XmlCardSerializer()
                )
            )
        )
    }

    test("Crimson Beverage should heal 500 HP") {
        val crimsonBeverage = MagicCard(
            "Crimson Beverage",
            "Heal 500 HP",

```

```

        ModifyHealthEffect(500),
        XmlCardSerializer()
    )
    player.health shouldBe 8000
    crimsonBeverage.useOn(player)
    player.health shouldBe 8500
}

test("Indigo Potion should heal 400 HP") {
    val indigoPotion = MagicCard(
        "Indigo Potion",
        "Heal 400 HP",
        ModifyHealthEffect(400),
        XmlCardSerializer()
    )
    player.health shouldBe 8000
    indigoPotion.useOn(player)
    player.health shouldBe 8400
}

test("Saucepan of Avidity should draw 2 cards") {
    val saucepanOfAvidity = MagicCard(
        "Saucepan of Avidity",
        "Draw 2 cards",
        DrawCardsEffect(2),
        XmlCardSerializer()
    )
    player.hand shouldBe emptyList()
    saucepanOfAvidity.useOn(player)
    player.hand.size shouldBe 2
    player.deck.size shouldBe 0
}
}
})

```

Si corremos los tests, veremos que todos pasan. Veamos que podemos mejorar del diseño.

Notemos que ahora nuestras cartas se encargan principalmente de guardar información y de delegar sus funcionalidades a otras clases. Para estos casos, *Kotlin* nos provee de una mejor forma de modelar estas clases.

Data Class

Una *data class* es una clase que se encarga principalmente de guardar información. Estas clases tienen varios beneficios:

- *Kotlin* provee de una implementación por defecto de los métodos `equals()`, `hashCode()` y `toString()`.
- *Kotlin* provee de una implementación por defecto de los métodos `copy()` y `componentN()`.

Importante. Una *data class* no puede ser heredada, pero una *data class* si puede heredar de otra clase.

Para convertir una clase en una *data class* debemos agregar la palabra clave `data` antes de la palabra clave `class`. Por ejemplo, la clase `MagicCard` podría quedar así:


```
data class MagicCard(
    override val name: String,
    override val text: String,
    override val effect: Effect,
    override val serializer: CardSerializer
) : Card
```

Notarán que agregamos `override val` antes de los atributos de la clase. Esto es porque ahora que la clase es una *data class*, todos los parámetros que recibe el constructor deben ser propiedades de la clase.

Si corremos los tests, veremos que todos siguen pasando.

Con esto, podemos hacer *commit* de los cambios y pasar a lo último que nos falta.

```
git add .
git commit -m "REFACTOR Magic card is now a data class"
```

Ejercicio 9.1. Transforme las clases `MonsterCard` y `TrapCard` en *data classes*.

9.5. Coverage

Usualmente, cuando estamos escribiendo tests, nos gustaría saber qué funcionalidades estamos probando. El *coverage* es una métrica que nos permite saber qué porcentaje de nuestro código está siendo probado por los tests.

Importante. *Tener un buen coverage no es garantía de que nuestro código esté bien testado, es sólo una métrica que nos permite saber qué porcentaje de nuestro código está siendo probado.*

Kotest provee de una herramienta para medir el *coverage* de nuestros tests. Para utilizarlo, debemos dar click derecho sobre la carpeta `src/test/kotlin` y seleccionar 'More Run/Debug' y luego 'Run "Tests in 'kygo.test"' with Coverage' (figura 9.2).

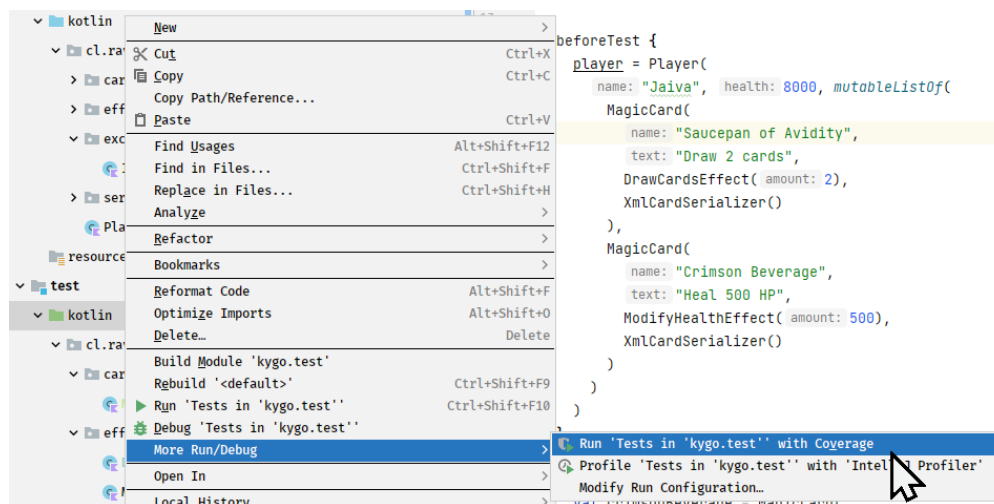
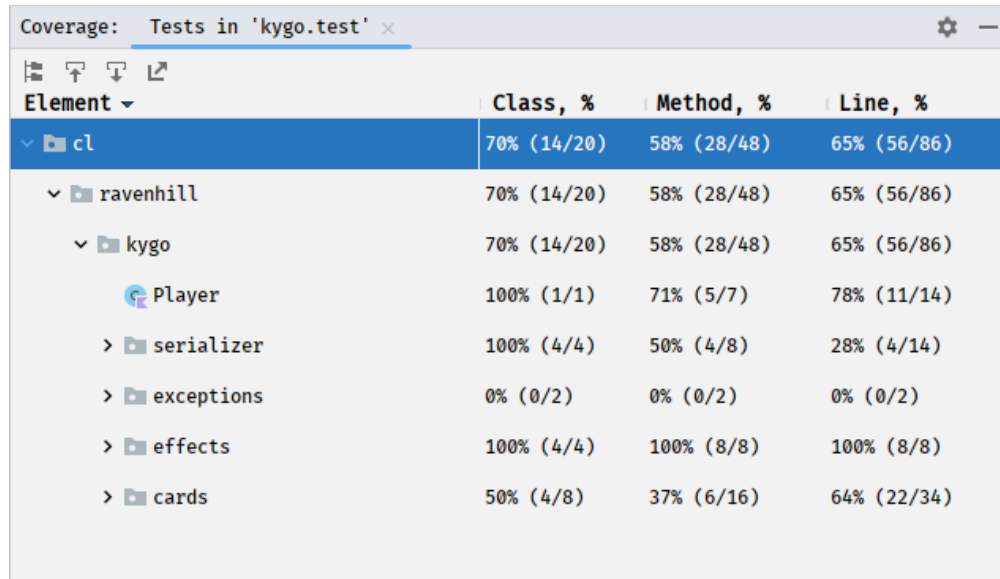


Figura 9.2: Correr tests con *coverage*

Una vez que los tests terminen de correr, se abrirá una ventana con el *coverage* de los tests (figura 9.3). Aquí veremos cuatro columnas:

- **Element:** Nombre del paquete o clase que se está probando.
- **Class:** Porcentaje de las clases que se están probando.
- **Method:** Porcentaje de los métodos que se están probando.
- **Line:** Porcentaje de las líneas que se están probando.

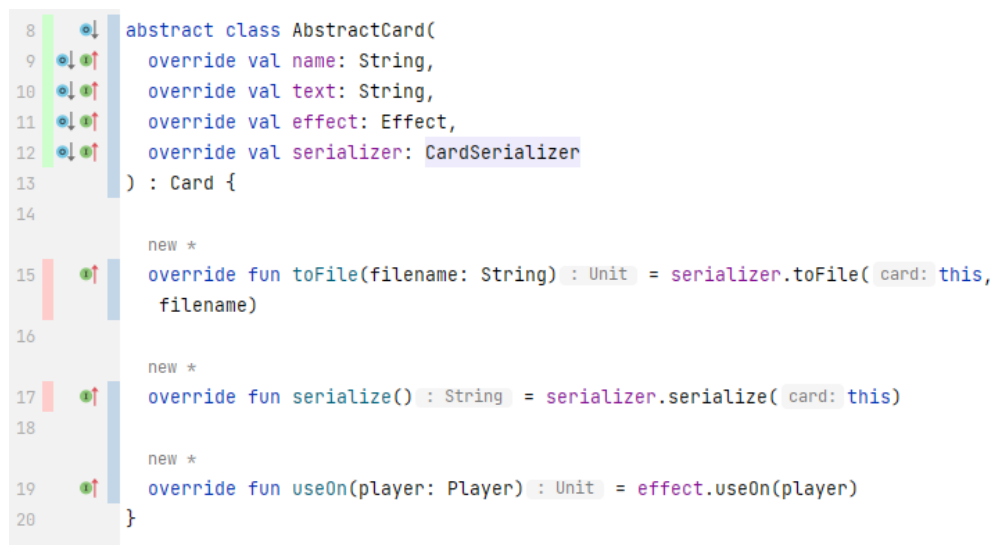


Coverage: Tests in 'kygo.test' x

Element	Class, %	Method, %	Line, %
cl	70% (14/20)	58% (28/48)	65% (56/86)
ravenhill	70% (14/20)	58% (28/48)	65% (56/86)
kygo	70% (14/20)	58% (28/48)	65% (56/86)
Player	100% (1/1)	71% (5/7)	78% (11/14)
> serializer	100% (4/4)	50% (4/8)	28% (4/14)
> exceptions	0% (0/2)	0% (0/2)	0% (0/2)
> effects	100% (4/4)	100% (8/8)	100% (8/8)
> cards	50% (4/8)	37% (6/16)	64% (22/34)

Figura 9.3: Reporte de *coverage*

En general, se recomienda tener un *coverage* de al menos 90 % de las líneas. Para saber qué líneas están siendo probadas, *Kotest* marcará con un color verde las líneas que están siendo probadas y con un color rojo las líneas que no están siendo probadas como se muestra en la figura 9.4.



```

8  abstract class AbstractCard(
9      override val name: String,
10     override val text: String,
11     override val effect: Effect,
12     override val serializer: CardSerializer
13 ) : Card {
14
15     new *
16     override fun toFile(filename: String) : Unit = serializer.toFile( card: this,
17         filename)
18
19     new *
20     override fun serialize() : String = serializer.serialize( card: this)
21
22     new *
23     override fun useOn(player: Player) : Unit = effect.useOn(player)
24 }

```

Figura 9.4: A la izquierda se marca con un color verde las líneas que están siendo probadas y con un color rojo las líneas que no están siendo probadas.

Notarán que las líneas que están marcadas con rojo son las de los métodos `toFile(String)` y `serialize` de la clase `AbstractCard`.

Escribamos los tests para estos métodos. Noten que como no podemos instanciar una clase abstracta, vamos a tener que probar el método en sus subclases. Como los métodos están definidos en la clase abstracta, sabemos que hacen lo mismo en todas las subclases, así que podemos crear un *test factory* para reusar el código de los tests.

```
fun `a card can be serialized to a string`(card: Card, serializedString: String) =
    funSpec {
        test("a card can be serialized to a string") {
            card.serialize() shouldBe serializedString
        }
    }
}
```

```
fun `a card can be serialized to a file`(
    card: Card, filename: String, serializedString: String
) = funSpec {
    lateinit var file: File

    beforeTest {
        file = File(filename)
    }

    afterTest {
        file.delete()
    }

    test("a card can be serialized to a file") {
        card.toFile(filename)
        file.readText() shouldBe serializedString
    }
}
```

Aquí tenemos algo nuevo, la función `afterTest`. Esta función es similar a la función `beforeTest`, pero se ejecuta después de cada test. En general se usa para «limpiar» el estado del sistema después de cada test. En este caso, estamos borrando el archivo que creamos para el test.

Ahora podemos crear los tests para las clases que concretizan a `Card`:

```
class MagicCardTest : FunSpec({
    ...
    include(`a card can be serialized to a string`(
        MagicCard(
            "Saucepan of Avidity",
            "Draw 2 cards",
            DrawCardsEffect(2),
            XmlCardSerializer()
        ),
        """
```

```

        |<Card>
        | <name>Saucepan of Avidity</name>
        | <text>Draw 2 cards</text>
        |</Card>
        """.trimMargin()
    ))
    include(`a card can be serialized to a file`(
        MagicCard(
            "Saucepan of Avidity",
            "Draw 2 cards",
            DrawCardsEffect(2),
            XmlCardSerializer()
        ),
        "saucepan_of_avidity.xml",
        """
        |<Card>
        | <name>Saucepan of Avidity</name>
        | <text>Draw 2 cards</text>
        |</Card>
        """.trimMargin()
    ))
})

```

TODO

- DDT

Bibliografía

Composition over Inheritance

CompositionInheritance2023

Composition over Inheritance. En: *Wikipedia*. 3 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Composition_over_inheritance&oldid=1131338651 (visitado 18-02-2023).

Anot.: Page Version ID: 1131338651.

Data Classes | Kotlin

DataClassesKotlin

Data Classes | Kotlin. Kotlin Help. URL: <https://kotlinlang.org/docs/data-classes.html> (visitado 18-02-2023).

Gamma y col.: Strategy

gammaStrategy1995

Erich Gamma, Richard Helm y col. «Strategy». En: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley, 1995, págs. 315-323. ISBN: 978-0-201-63361-0.

Martin: TEMPLATE METHOD & STRATEGY: Inheritance vs. Delegation

martinTEMPLATEMETHODSTRATEGY2014

Robert C. Martin. «TEMPLATE METHOD & STRATEGY: Inheritance vs. Delegation». En: *Agile Software Development, Principles, Patterns, and Practices*. First edition, Pearson new international edition. Harlow: Pearson, 2014, págs. 161-172. ISBN: 978-1-292-02594-0.

Shvets: Strategy

shvetsStrategy2021

Alexander Shvets. «Strategy». En: *Dive Into Design Patterns*. 2021, págs. 368-380.

Capítulo 10

Juguemos en el bosque

Index

.gitignore, 50, 92
Any::toString(): String, 69, 96
beforeTest, 101
lateinit, 101
git commit, 51

Build system, 88

Caso de prueba, 87
Chocolatey, 16
Clase abstracta, 72
Compilador, 14, 68
Composición, 60
consola REPL, 32
Constructor primario, 102
Control de flujo, 21

Double Dispatch, 134

Encapsulamiento, 67
Entorno de desarrollo integrado, 29

Factory method pattern, 114, 137
Firma, 67
Framework, 88
Funciones abiertas, 68
Funciones finales, 68
Función abstracta, 72

Git, 48
git add, 51
git init, 50
git status, 50
Gradle, 88

Homebrew, 17, 48

igualdad referencial, 97
IntelliJ IDEA, 14, 29, 30
Interprete, 68

Java, 13
Java Virtual Machine, 13
JetBrains Toolbox, 30

Kotest, 88, 92, 95
Kotlin, 13, 14

MacOS, 17
Mensaje, 67
Method-lookup, 67
Modificador de privilegios protected, 137

Oh-My-Posh, 15
Open-closed principle, 73

Paquete, 34
Parámetro por defecto, 131
PascalCase, 36
Patrón de diseño creacional, 114
Patrón de diseño, 114
Polimorfismo de subtipos, 73
Principio de sustitución de Liskov, 77

Refactor, 101

SDKMAN, 16
Search Everywhere, 33
Sistema de control de versiones, 47
Sistema de tipos, 18
Sobrecarga de funciones, 67
Sobrescritura de funciones, 68
super, 75

Test-Driven Development, 87, 88
Testing framework, 88
this, 75

Windows, 14
Windows Package Manager, 14, 48
Windows Terminal, 14