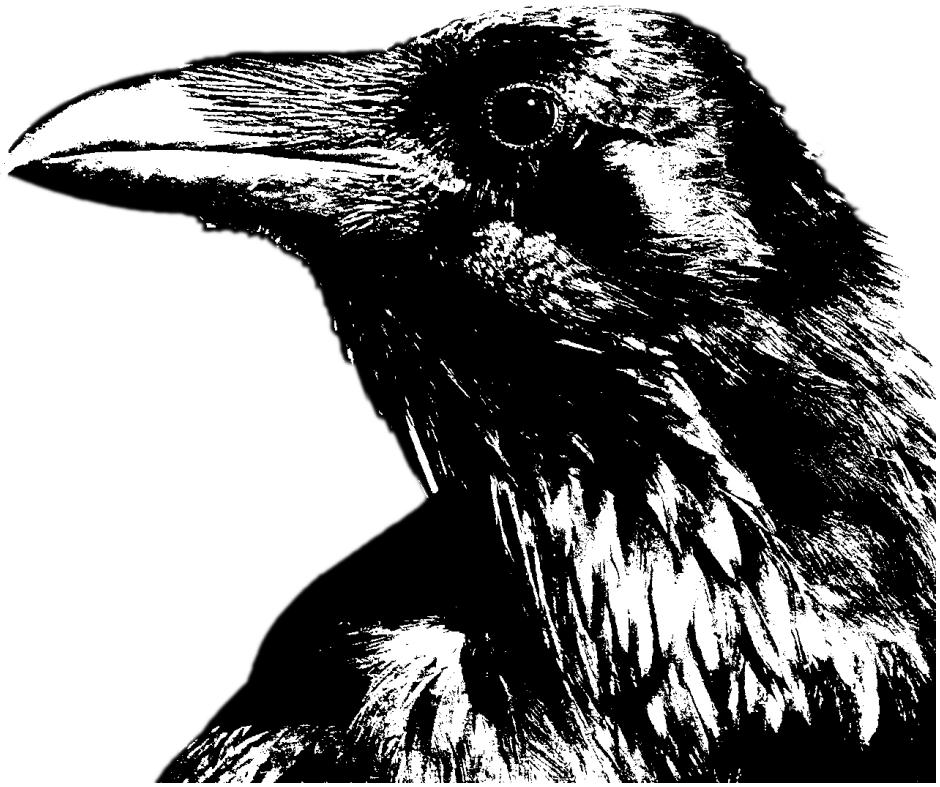
EL TIEMPO PASA Y EL SOFTWARE MUERE

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 **9**

1. ¡Kotlin! **11**
1.1. ¿Java? . . . 11
1.2. ¿Kotlin? . . . 11
1.2.1. Instalando Kotlin . . . 12
1.2.2. *Windows* . . . 12
1.2.3. *Linux* . . . 14
1.2.4. *MacOS* . . . 15
1.3. Tipos en *Kotlin* . . . 15
1.3.1. Tipos explícitos . . . 16
1.3.2. Tipos inferidos . . . 16
1.3.3. Funciones . . . 17
1.4. Complicando las cosas . . . 19
1.4.1. La función de Fibonacci . . . 19
1.4.2. Implementación básica (recursiva) . . . 19
1.4.3. Implementación iterativa . . . 20
1.4.4. Programación dinámica . . . 21

2. *IntelliJ*, tu nuevo *bff* **25**
2.1. No termina de convencerme, pero le voy a dar una oportunidad. ¿Cómo lo instalo? . . . 25
2.1.1. JetBrains Toolbox . . . 26
2.1.2. IntelliJ IDEA . . . 26
2.2. Conociendo *IntelliJ* . . . 27
2.2.1. Creando un proyecto . . . 27
2.2.2. Trabajando con *IntelliJ* . . . 28
2.3. Mi primera aplicación . . . 30
2.3.1. Paquetes . . . 30
2.3.2. Mi primer *Kotlin* . . . 31
2.3.3. Loop principal . . . 33
2.4. Ejercicios . . . 36

II OOP no es religión **41**

3. Para no olvidar **43**
3.1. Sistemas de control de versiones . . . 43
3.1.1. Git . . . 44
3.1.2. Instalando Git . . . 44
3.1.3. Configurando Git . . . 44

3.2. Creando el proyecto . . . 45
3.3. Objetos . . . 47
3.4. Clases . . . 49
3.4.1. Constructores primarios . . . 49

4. Principios de la programación orientada a objetos **53**
4.1. Composición . . . 53
4.2. Encapsulamiento . . . 55
4.3. Herencia . . . 57
4.3.1. Herencia en Kotlin . . . 59
4.3.2. Method-lookup . . . 60
4.4. Polimorfismo de subtipos . . . 64
4.5. Principio de Liskov . . . 71
4.6. Ejercicios . . . 73

5. Cuando todo no funciona **79**
5.1. Testing frameworks . . . 80
5.2. Gradle . . . 80
5.3. Conociendo a *Kotest* . . . 87
5.4. Segunda cita con *Kotest* . . . 90
5.5. Mejorando nuestro código . . . 92
Index . . . 99

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 Graboloza, Dimitri Svandich, Nancy Hitschfeld y, por supuesto, Alexandre Bergel y Juan-Pablo Silva.

9 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 ibamos 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.

[1] Freeman y Robson, «Chapter 1: Intro to Design Patterns».

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*

[4] Un 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).

[5] Es 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 nosotrxs, 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. 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*.

[6] En 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.

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.

```
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:

[7] https://github.com/islaterm/software-design-book-es/blob/master/install/windows/chocolatey.ps1
[8] El «!» es parte del nombre

```
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 **reasignada** 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
```

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.*

[9] Término propio

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: $i : Int \iff i \in \mathbb{Z} \land -2^{31} \leq i \leq 2^{31}-1$.
- `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: $d : Double \iff d \in \mathbb{R} \land 4{,}9 \times 10^{-324} \leq |d| \leq 1{,}8 \times 10^{308}$.
- `String`: El conjunto de todas las cadenas de caracteres. Formalmente: $s : String \iff s \in \mathbb{L}^*$, 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 trabajen 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:

Para ver cómo funciona la inferencia de tipos, podemos reescrbir 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.

[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

Ejercicio 1.1. Indique qué tipo de dato es cada una de las siguientes variables:

1. `var i = true`
2. `var i = 8000`
3. `var i = "8000"`
4. `var i = Unit`
5. `var i = 8000.0`

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:

$$FT(A_1, \ldots, A_n) \to R$$

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 escritoAsí, 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:

[12] Trad. Diversión

[13] El 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:

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

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:

```kotlin
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.

Ejercicio 1.2. Simplifique la implementación de la función de Fibonacci usando *if-else* como expresión. *Hint: utilice una expresión de la forma* `return if (condicion) {expresion1} else {expresion2}`.

Ejercicio 1.3. ¿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.

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 operandor `..`, 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:

```kotlin
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 constraste, 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:

```kotlin
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:

```kotlin
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.

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 (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 testear[1] 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 les lectores es bienvenido. En caso de querer colaborar pueden hacerlo mediante un *Pull Request* al repositorio de *GitHub* de este libro (`https://github.com/islaterm/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`) 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*.

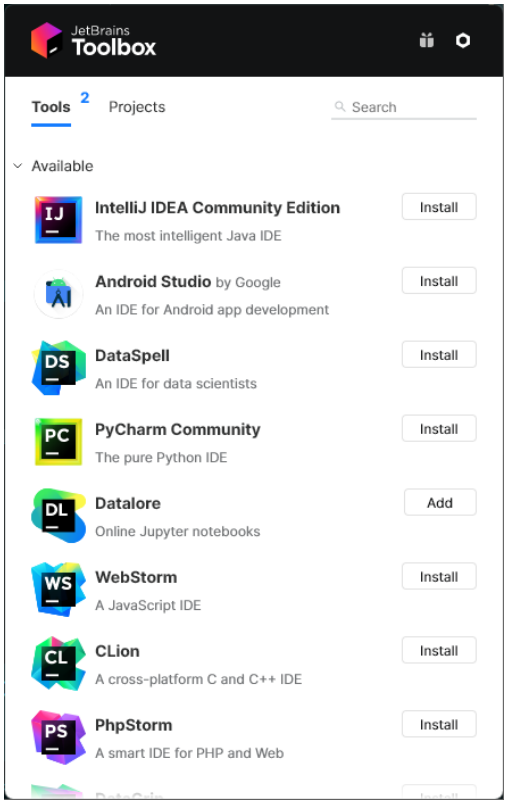


Figura 2.1: Interfaz de *JetBrains Toolbox*.

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*.

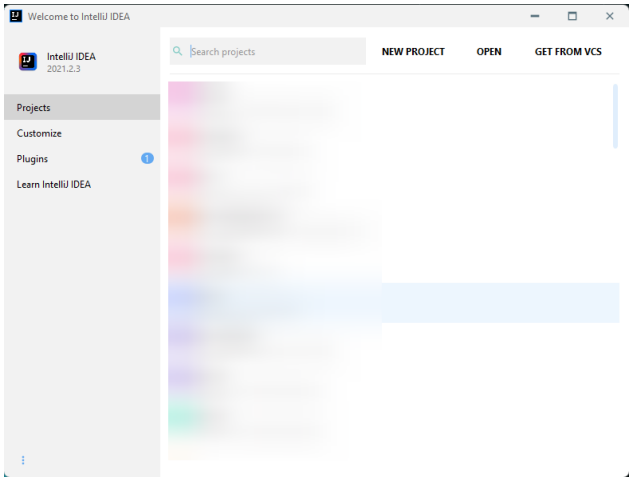


Figura 2.2: Ventana inicial de *IntelliJ*.

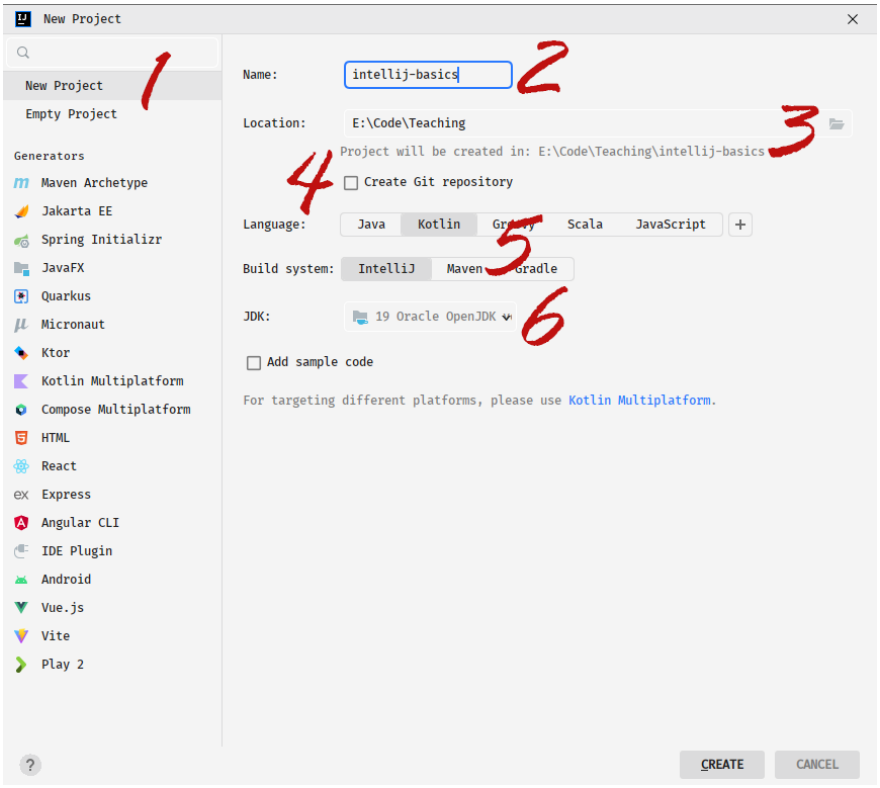


Figura 2.3: Ventana de creación de proyectos

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

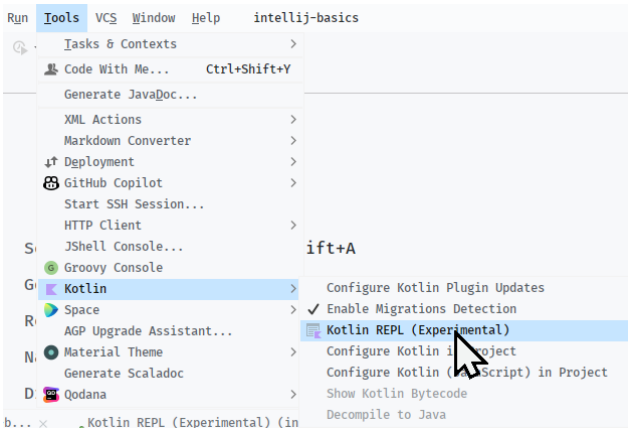


Figura 2.4: ubicación de la *consola REPL* en *IntelliJ*.

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

> *Search Everywere*
>
> 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`.

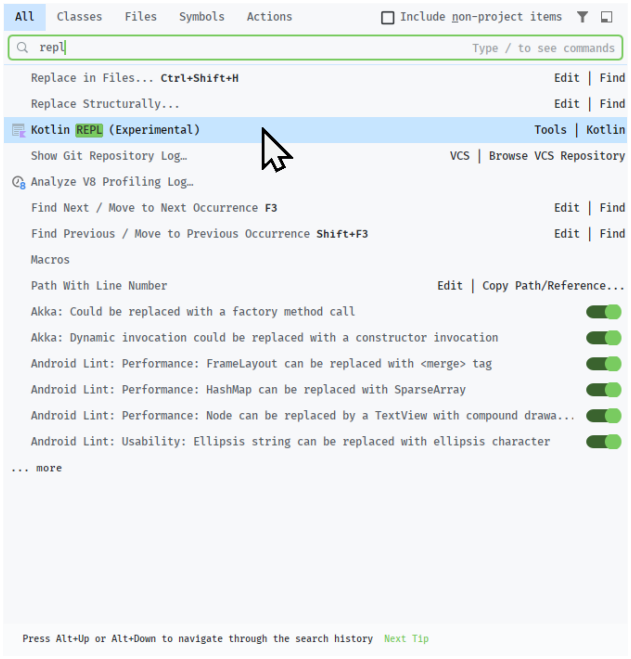


Figura 2.5: Resultado de *Search Everywhere* para la búsqueda de `repl`.

Ahora intenten reescribir el código del apartado 1.4 para ver cómo se siente trabajar con un *IDE* (reescríbanlo, 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.

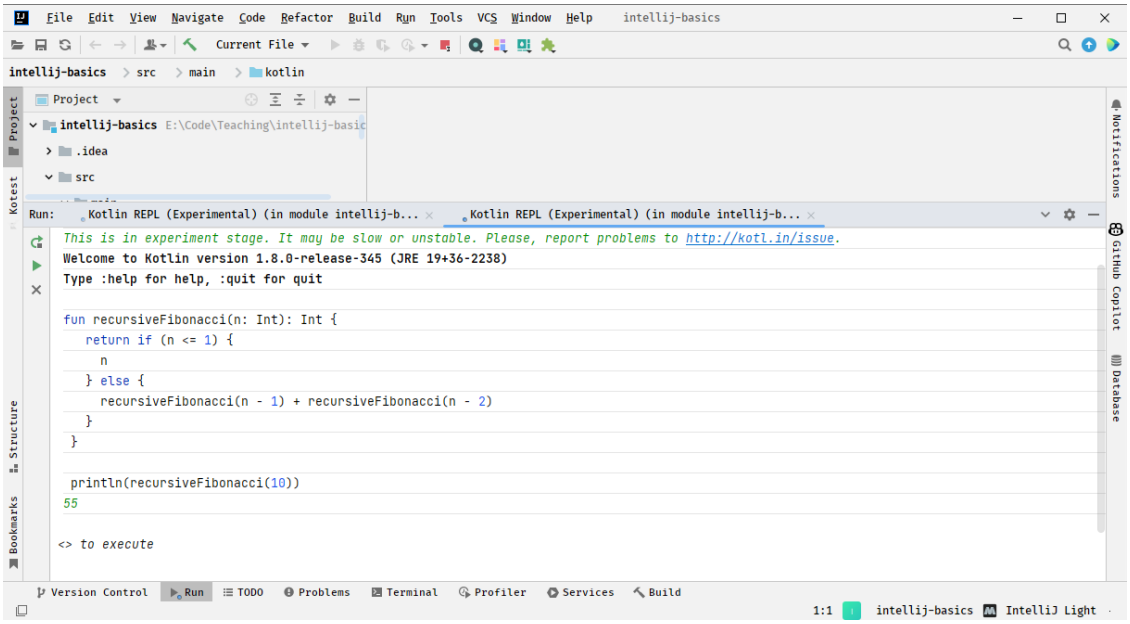


Figura 2.6: Ejemplo de uso de la *consola REPL* en *IntelliJ*.

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*:

[3] En 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*.

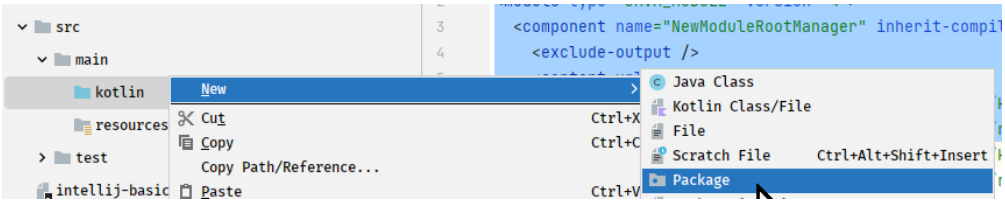


Figura 2.7: Creando un nuevo paquete

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).

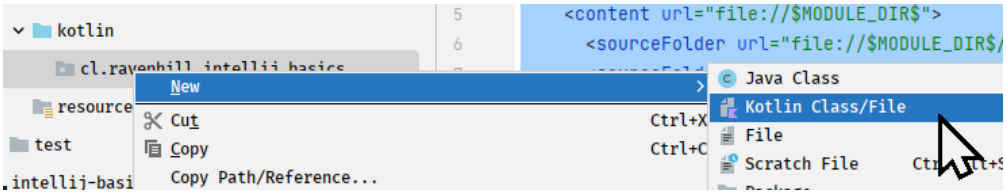


Figura 2.8: Creando un nuevo archivo *Kotlin*

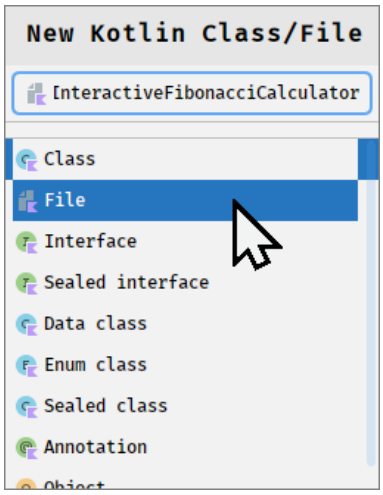


Figura 2.9: Creando un nuevo archivo *Kotlin*

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:

```kotlin
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).

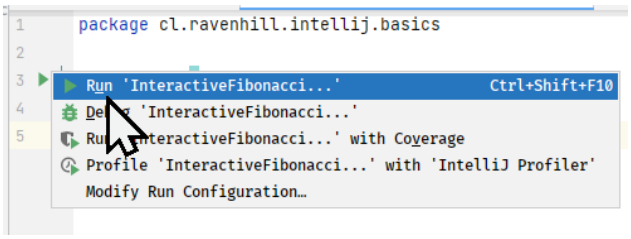


Figura 2.10: Ejecutando la aplicación

[4] *What 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 escenciales 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.

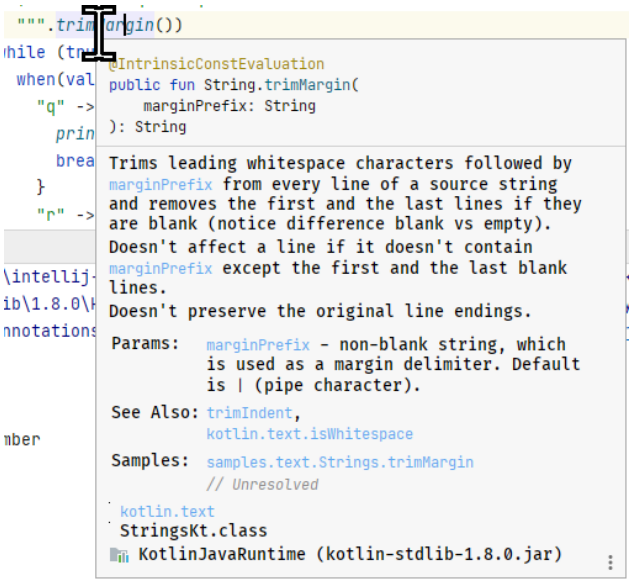


Figura 2.11: Documentación de la función `trimMargin()`.

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 1 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:

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

   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 2 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:

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

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:

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

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

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

```
}
```

```
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

Para no olvidar

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 mantener[1] 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 principal[2] 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.

[1] Y a veces perder.
[2] imho

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

Partamos por crear 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).

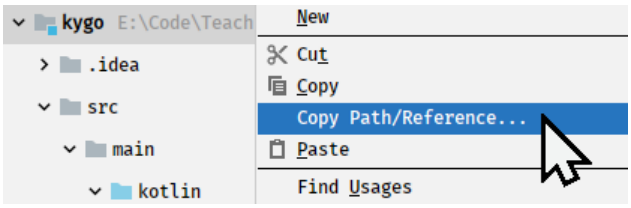


Figura 3.1: Copiando la ruta del proyecto

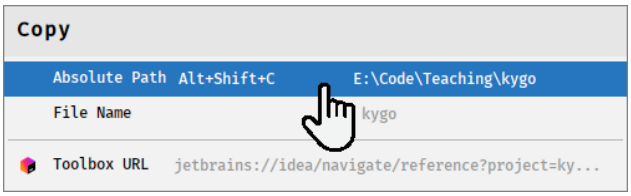


Figura 3.2: Copiando la ruta absoluta del proyecto

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][4]/`Set-Location`[5] 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` (nuestro otro 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 nunca[6] 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
```

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*.

[3] Change directory
[4] *Cd Man Page - Linux - SS64.Com*.
[5] sdwheeler, *Set-Location (Microsoft.PowerShell.Management) - PowerShell*.
[6] Existen formas de agregarlos de todos modos, pero por qué quieres hacer eso `D:`

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.[7]

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 abstraer[8] 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 (a esto se le conoce como transparencia) 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 {
}
```

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.

[7] Lo 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.

[8] Abstraer 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.

```
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 sintáxis 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"
```

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).

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 **Gitb**

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

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

Principios de la programación orientada a objetos

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,
  healthPoints: Int,
  deck: MutableList<String>,
  cardsText: MutableMap<String, String>,
  cardsAttacks: MutableMap<String, Int>
) {
  val name: String
  var healthPoints: Int
  val deck: MutableList<String>
  val cardsText: MutableMap<String, String>
  val cardsAttacks: MutableMap<String, Int>

  init {
    this.name = name
    this.healthPoints = healthPoints
    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("Yulian", 8000, deck, cardsText, cardsAttacks)
println(player.name)
println(player.healthPoints)
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 $f(x) = x + 1$ es una composición de las funciones $g(x) = x + 1$ y $h(x) = x$.

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,
  healthPoints: Int,
  deck: MutableList<Card>
) {
  val name: String
  var healthPoints: Int
  val deck: MutableList<Card>

  init {
    this.name = name
    this.healthPoints = healthPoints
    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 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 -= this.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)`. 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.

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.

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
```

[3] Que 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.

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

Noten que ahora nos deschicimos 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 enfoquémosnos 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 A hereda de otro objeto B si A es una especialización de B. A A se le conoce como **subclase** de B y a B como **superclase** de A.

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. Luego de los dos puntos, escribimos el nombre de la clase de la que queremos heredar
4. 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 clase padre puede tener múltiples subclases.*

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

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

Noten que agregamos paréntesis después del nombre de la clase padre, 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 clase padre, antes de ejecutar cualquier otra línea de código.

Importante. *Al no ser un método los constructores no se heredan, por lo que debemos llamar explícitamente al constructor de la clase padre.*

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

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 la 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 complejo[5] 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 debamos 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 búsqueda de una función a la que enviar 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 clase padre.
3. Si no se encuentra, se busca en la clase padre de la clase padre.
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.

[5] *Overload Resolution - Kotlin Language Specification*.

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 padre 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 padre?

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 clase padre. En Kotlin, la sobrecarga **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 y 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: ningúna 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 ${player.healthPoints} health points")
dragon.attack(player)
println("$player has ${player.healthPoints} health points")
reflect.attack(player)
println("$player has ${player.healthPoints} 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
}
```

Ejercicio 4.1. ¿Qué pasa si se ejecuta el siguiente código? ¿Por qué?

```
open class A {
  open fun f(): Unit {
    println("A.f()")
  }
}

class B : A() {
  override fun f(): String {
    return "B.f()"
  }
}

fun main() {
```

```
  val a: A = B()
  a.f()
}
```

Ejercicio 4.2. Implemente las clases `MagicCard` y `TrapCard` que hereden de `Card`.

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

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

```yaml
# 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 sobreescribir en la clase que hereda de ella? Como veran, 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.

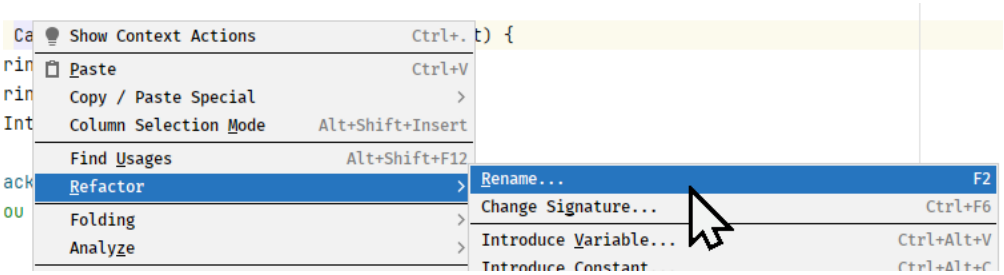


Figura 4.1: Renombrando la clase `Card` a `AbstractCard`.

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:

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

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

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 polimmorfísmo.

Polimorfísmo 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.

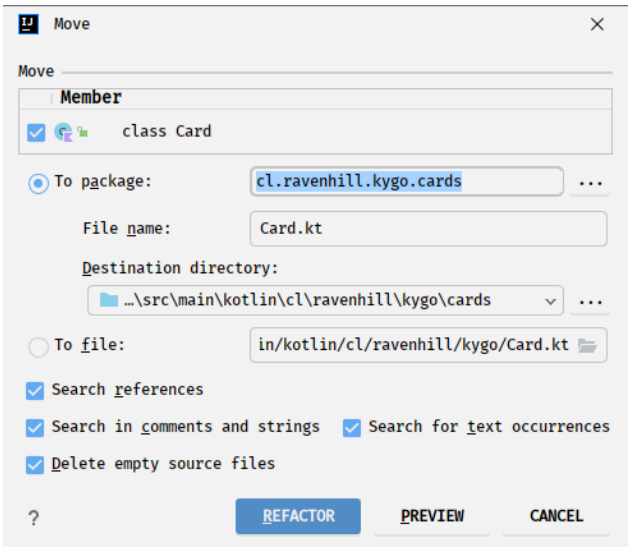


Figura 4.2: Moviendo las clases `Card` y `MonsterCard` al paquete `cards`.

Hagamos un paquete `cards` haciendo click derecho en el paquete `cl.ravenhill.kygo` y arrastremos los archivos `Card.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 `serialization` en el paquete `cl.ravenhill.kygo`.

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

```
package cl.ravenhill.kygo.serializer

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


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

  abstract fun serialize(card: Card): 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(card: Card) : AbstractCardSerializer() {
  private val card = card

  override fun serialize(): String {
    return """
      |<Card>
      |  <name>${card.name}</name>
      |  <text>${card.text}</text>
      |  <attack>${card.attack}</attack>
      |</Card>
    """.trimMargin()
  }
}
```

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

```
open class Card(
  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-looup* 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
) : Card(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 polimorfísmo 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(Card): String` y `toFile(Card, String)` a la interfaz:

```
interface CardSerializer {
  fun serialize(card: Card): String
  fun toFile(card: Card, 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: Card, filename: String) {
    val file = File(filename)
    file.writeText(serialize(card))
  }

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

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

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

Como siempre, terminamos con:

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

Ejercicio 4.3. 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 `attackPoints` 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:

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

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

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

Ejercicio 3 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`:

```kotlin
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 4 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:

$$\mathbf{v} \cdot \mathbf{w} = \mathbf{v}_x\mathbf{w}_x + \mathbf{v}_y\mathbf{w}_y + \mathbf{v}_z\mathbf{w}_z$$

   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:

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

    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**

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).

Encapsulation (Computer Programming) **EncapsulationComputerProgramming2023**

Encapsulation (Computer Programming). 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. *OOP Concepts for Beginners: What Is Composition?* Stackify. 16 de ene. de 2018. URL: https://stackify.com/oop-concepts-composition/ (visitado 08-02-2023).

MutableList - Kotlin Programming Language **MutableListKotlinProgramming**

MutableList - Kotlin Programming Language. Kotlin. URL: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/index.html (visitado 08-02-2023).

MutableMap - Kotlin Programming Language **MutableMapKotlinProgramming**

MutableMap - Kotlin Programming Language. 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**

Overload Resolution - Kotlin Language Specification. URL: https://kotlinlang.org/spec/overload-resolution.html#overload-resolution (visitado 08-02-2023).

Shvets: SOLID Principles **shvetsSOLIDPrinciples2021**

Alexander Shvets. «SOLID Principles». En: *Dive Into Design Patterns*. 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.

[1] O 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. Refactorizar[3] 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 importante:

- 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).

[2] Si escribimos tests que pasan antes de implementar la funcionalidad, hay algo sus en nuestros tests.
[3] Refactorizar es una técnica de desarrollo de software que consiste en reescribir el código sin cambiar su comportamiento.
[4] Actualmente propiedad de The Pokémon Company

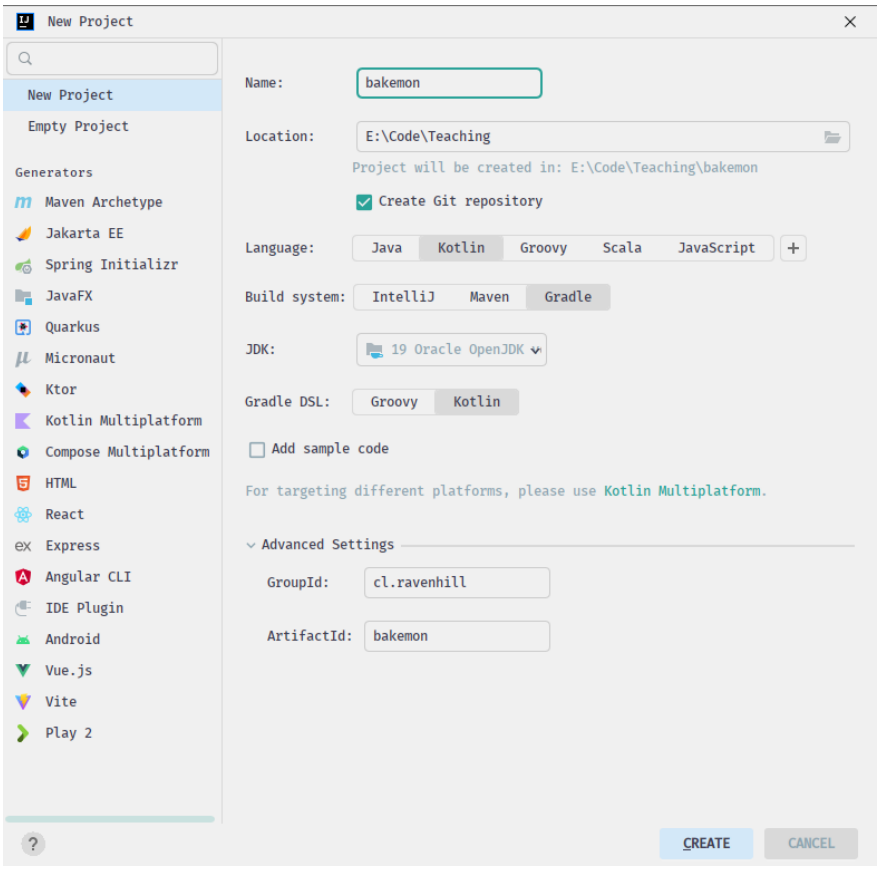


Figura 5.1: Creando un proyecto Gradle en IntelliJ

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.
- Un archivo que contiene la configuración 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 debiera 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…

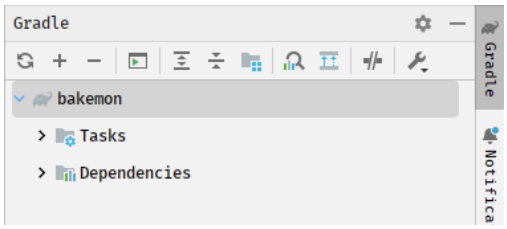


Figura 5.3: La ventana de Gradle

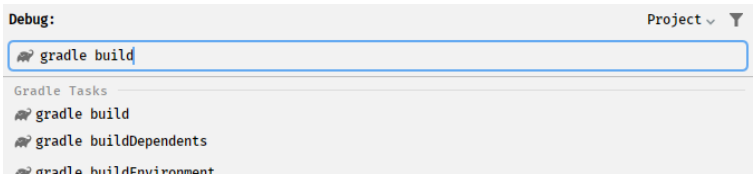


Figura 5.4: Ejecutando el comando `build`

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 necesitemos.

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

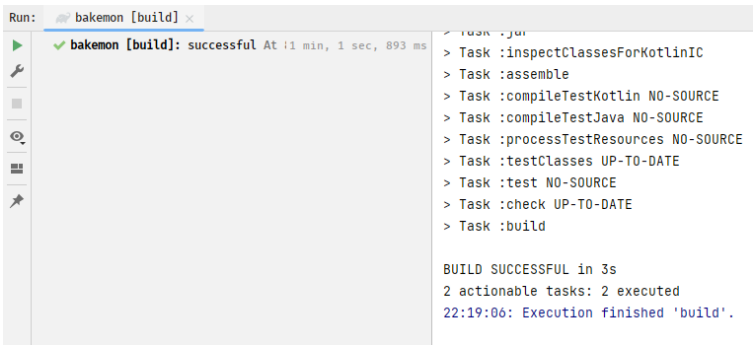


Figura 5.5: El comando `build` terminó exitosamente

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 el 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).

[7] Distinto a un plugin de *Gradle*.

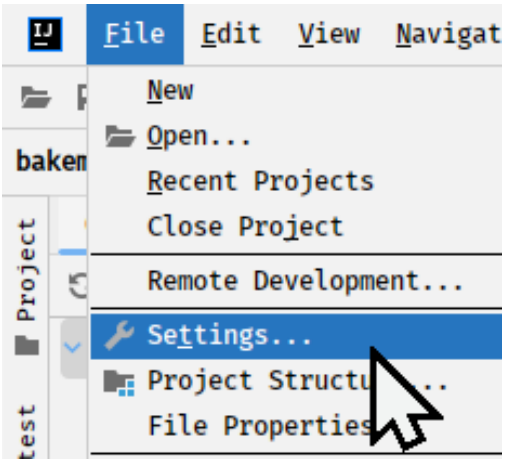


Figura 5.6: Menú de *Settings*

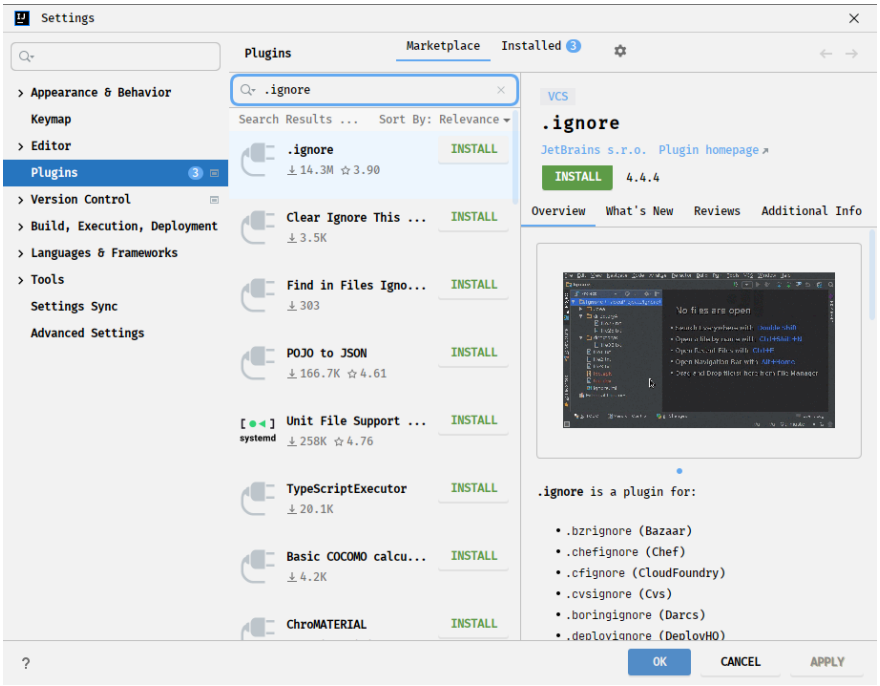


Figura 5.7: Instalación del plugin *.ignore*

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.

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

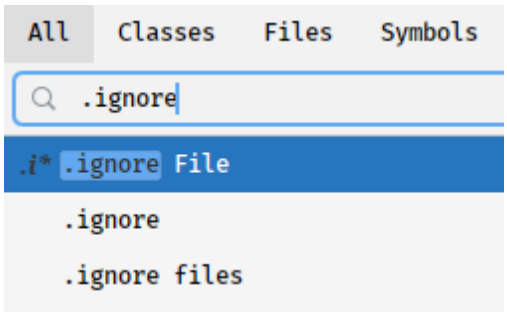


Figura 5.8: Opción *.ignore File*

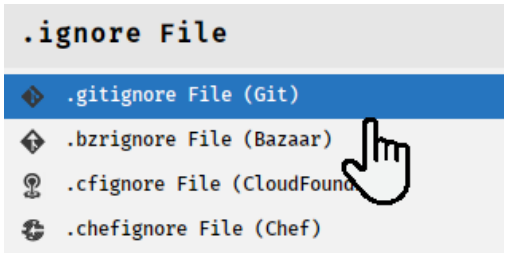


Figura 5.9: Opción *.gitignore File*

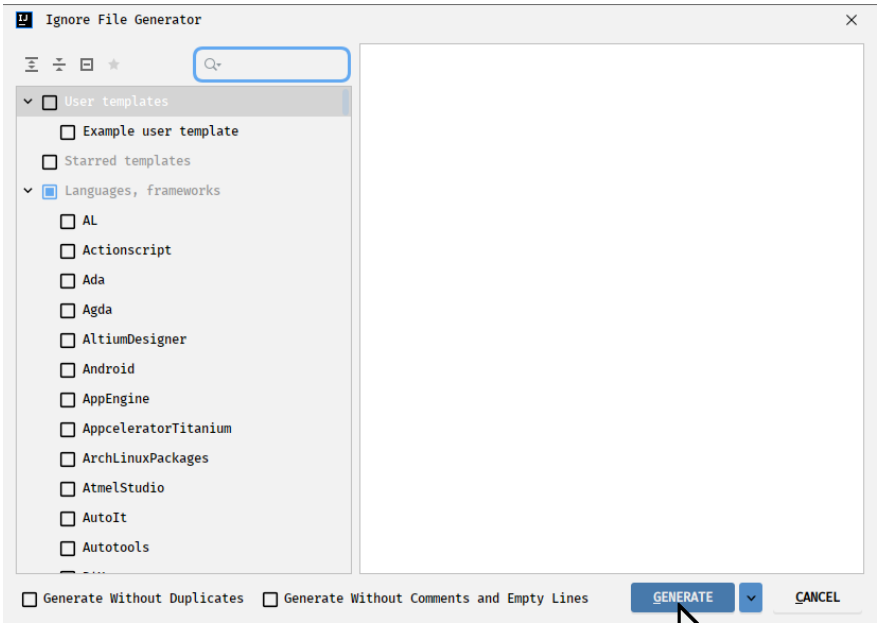


Figura 5.10: Generación del `.gitignore`

```
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 lo 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 nigún 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 querremos 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.

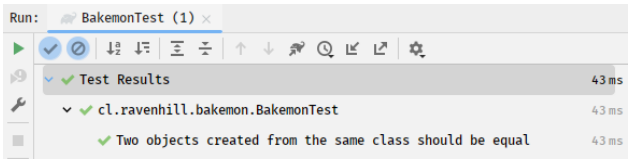


Figura 5.11: Primer test

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 sobreescrito 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 nosotrxs. 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`.

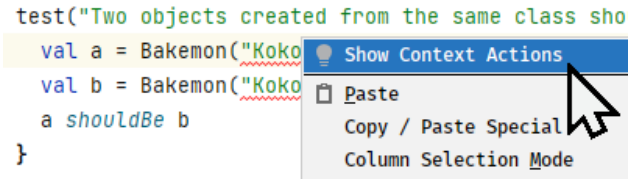


Figura 5.12: Mostrar acciones de contexto

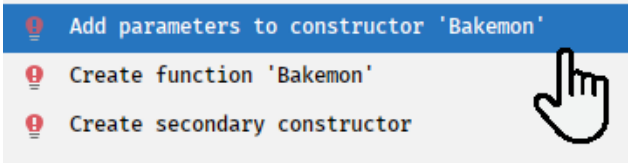


Figura 5.13: Agregar parámetros al constructor

Figura 5.14: Cambiar firma del 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()`.

```
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 si 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 sintáxis 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 valores[9]. Cuando una función en *Kotlin* solo tiene una expresión, podemos utilizar una sintáxis más compacta para definirla y es que podemos omitir el cuerpo de la función y escribir la expresión directamente después de los paréntesis de los parámetros como si estuvieramos 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.

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

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.

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 **TestdrivenDevelopment2023**

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.

Zhang: A Brief Introduction to Build Systems **zhangBriefIntroductionBuild2020**

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).

Index

`.gitignore`, 46, 84
`Any::toString(): String`, 63, 88
`beforeTest`, 93
`git init`, 46
`git status`, 46
`lateinit`, 93
`git commit`, 47
`super`, 69
`this`, 69

Build system, 80

Caso de prueba, 79
Chocolatey, 14
Clase abstracta, 66
Compilador, 12, 61
Composición, 54
consola REPL, 28
Constructor primario, 94
Control de flujo, 19

Encapsulamiento, 61
Entorno de desarrollo integrado, 25

Firma, 60
Framework, 80
Funciones abiertas, 62
Funciones finales, 62
Función abstracta, 66

Git, 44
git add, 46
Gradle, 80

Homebrew, 15

igualdad referencial, 89
IntelliJ IDEA, 12, 25, 26
Interprete, 61

Java, 11
Java Virtual Machine, 11
JetBrains Toolbox, 26

Kotest, 80, 84, 87
Kotlin, 11, 12

MacOS, 15
Mensaje, 61
Method-lookup, 61

Oh-My-Posh, 13
Open-closed principle, 67

Paquete, 30
PascalCase, 32
Polimorfísmo de subtipos, 67
Principio de sustitución de Liskov, 71

Refactor, 92

SDKMAN, 14
Search Everywhere, 29
Sistema de control de versiones, 43
Sistema de tipos, 16
Sobrecarga de funciones, 61
Sobrescritura de funciones, 62

Test-Driven Development, 79, 80
Testing framework, 80

Windows, 12
Windows Package Manager, 12
Windows Terminal, 12