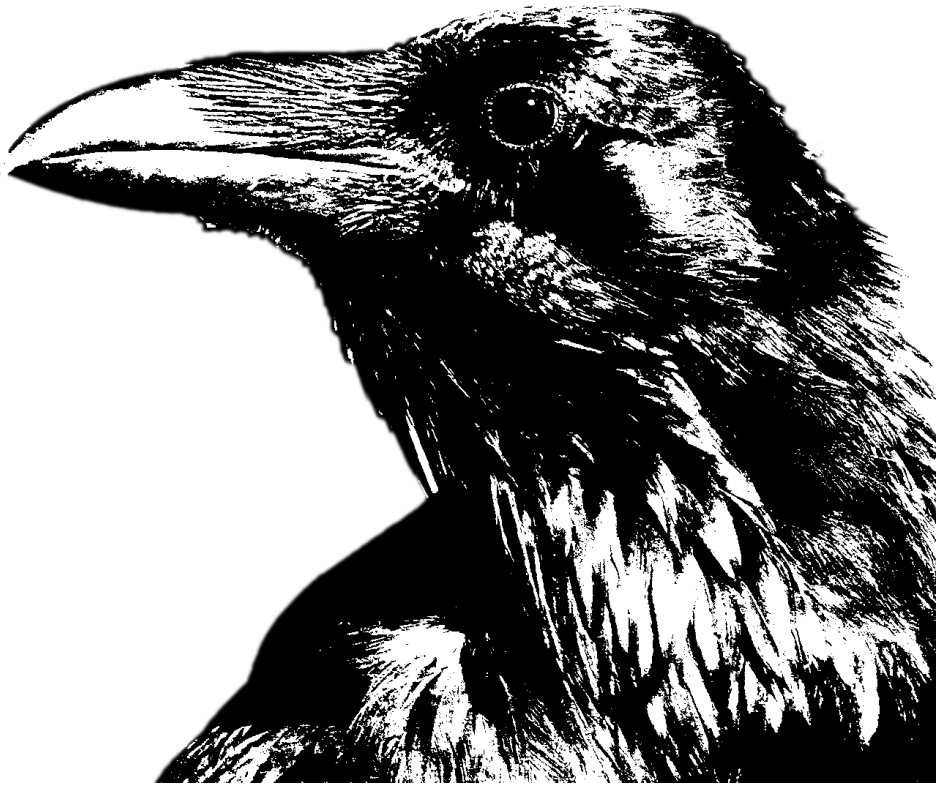
EL TIEMPO PASA Y EL SOFTWARE MUERE

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

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*. Se asume que los lectores tienen nociones básicas de programación, conocimiento básico de *Python* y, en menor medida, de *C*.

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.

6 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] **head-first-intro**.

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 (de ahí la necesidad de enseñarles éste y no otro lenguaje), 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*). 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 *JVM* 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.[2] 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] 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.[3]

Sorprendente

La forma más fácil de instalar el compilador de *Kotlin* es instalando *IntelliJ* (como veremos en el capítulo **??**), 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*. *Windosw Package Manager*[4] 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.

¿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][6] (WT). *WT* viene instalada en las versiones más nuevas de *Windows 10* y en todas las versiones de *Windows 11*. Pueden usar el *Buscador*

[3] 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).

[4] KevinLaMS, *Use the Winget Tool to Install and Manage Applications*.

[5] Cinnamon-msft, *An Overview on Windows Terminal*.

[6] 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,[7] 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[a] 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, 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
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*.

[7] 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.

[a] *Home | Oh My Posh*.

Chocolatey

Lo primero que necesitaremos para instalar las herramientas que usaremos será un gestor de paquetes, utilizaremos *Chocolatey*.[8]

Para partir abran una ventana de *Powershell* como administrador. Una vez abierta, deben ejecutar las instrucciones:[9]

```
[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 -?
```

Con chocolatey esto es simple, solamente deben ejecutar:

```
cinst 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.

Nuevamente, para instalar el compilador confiaremos en nuestro amigo *Chocolatey*, si no saben de lo que hablo, probablemente no leyeron la sección **??**, no voy a repetir esas instrucciones, así que vayan a leer esa sección y luego vuelvan aquí; lxs espero.

Una vez que tengan *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!*[10]. Instalar *SDKMAN!* es relativamente simple, solo deben correr los comandos que presentamos a continuación y seguir los pasos que aparezcan en pantalla.

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

```
curl -s "https://get.sdkman.io" | bash
source "/home/roac/.sdkman/bin/sdkman-init.sh"
```

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

```
sdk version
```

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

```
sdk install kotlin
```

Y comprobamos la instalación haciendo:

```
kotlinc -version
```

1.2.4. *MacOS*

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

```
brew update
brew install kotlin
```

Por último comprobamos la instalación:

```
kotlinc -version
```

1.3. Tipos en *Kotlin*

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

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

Un **valor**[11] (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:

[11] Término propio

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

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

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

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

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

1.3.1. Tipos explícitos

Todas las variables en *Kotlin* deben tener un tipo definido, y una vez definido no puede ser cambiado.[12]

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` (es decir,).

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: $i : Int \iff i \in \mathbb{Z} \wedge -2^{31} \leq i \leq 2^{31} - 1$.
- `Double`: El conjunto de todos los números reales de *doble precisión*.[13] En este caso, la definición formal es un poco más complicada, pero es cercana a: $d : Double \iff d \in \mathbb{R} \wedge 4{,}9 \times 10^{-324} \leq |d| \leq 1{,}8 \times 10^{308}$.
- `String`: El conjunto de todas las cadenas de caracteres.
- `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*.

Es importante notar 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 no es necesario definir explícitamente el tipo de una variable en la mayoría de los casos. 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:

[12] Este tipo de atrocidades sólo pueden suceder en lenguajes de tipado dinámico como *Python* o *Ruby*.
[13] https://en.wikipedia.org/wiki/Double-precision_floating-point_format

```kotlin
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.

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

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

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. En *Kotlin* tenemos dos formas de definir funciones: utilizando la palabra reservada `fun`[14] 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:

```kotlin
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`[15]).

[14] Trad. Diversión

[15] 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.

- 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:

```
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 expresión de control de flujo. Una expresión de control de flujo es una expresión 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 instrucción un número determinado de veces.
- *while*: es una **instrucción** que ejecuta una instrucción mientras una condición sea verdadera.

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

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

Veamos lo que está pasando:

- La función *recursiveFibonacci* recibe un número entero y devuelve un número entero.
- La función evalúa si el número recibido es menor o igual a 1.
- Si el número es menor o igual a 1, la función devuelve el número recibido.
- Si el número es mayor a 1, la función devuelve la suma de la función llamada con el número recibido menos 1 y la función llamada con el número recibido menos 2.

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

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 `resul` 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, si calculamos el valor de `f(5)` sabemos que para calcular el valor de `f(6)` necesitamos calcular el valor de `f(5)` y `f(4)`. Si guardamos el valor de `f(5)` podemos calcular el valor de `f(6)` sin tener que calcular nuevamente el valor de `f(5)`. De esta manera, podemos calcular el valor de `f(6)` sin tener que calcular el valor de `f(5)` ni el valor de `f(4)`.

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

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

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

Ahora volvemos al tema de mutabilidad. Un arreglo es un tipo de dato mutable, es decir, podemos cambiar la información que contiene.

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

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

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

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

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

Con esto ya estamos listxs para enfrentar al mundo.

Bibliografía

Cinnamon-msft: An Overview on Windows Terminal **cinnamon-msftOverviewWindowsTerminal**

cinnamon-msft. *An Overview on Windows Terminal*. URL: https://docs.microsoft.com/en-us/windows/terminal/ (visitado 08-08-2022).

Home | Oh My Posh **HomeOhMy**

Home | Oh My Posh. URL: https://ohmyposh.dev/ (visitado 08-08-2022).

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

KevinLaMS. *Use the Winget Tool to Install and Manage Applications*. URL: https://docs.microsoft.com/en-us/windows/package-manager/winget/ (visitado 08-08-2022).

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

```
choco install jetbrainstoolbox
```

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*.

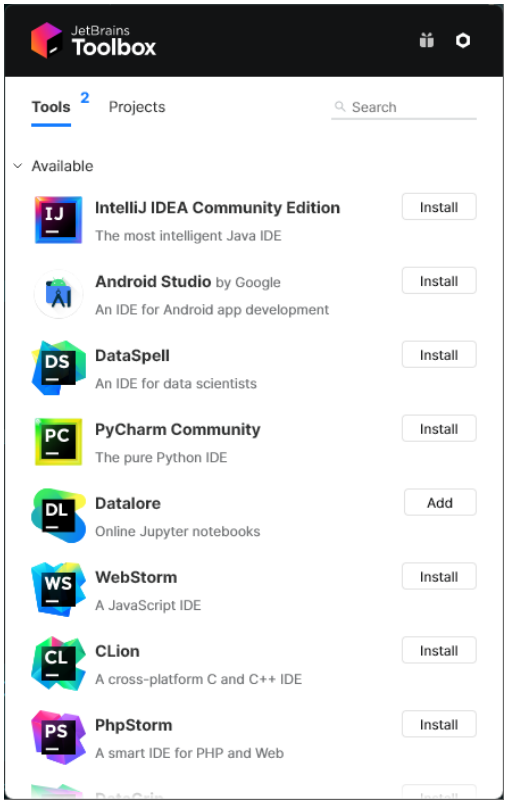


Figura 2.1: Interfaz de *JetBrains Toolbox*.

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*.

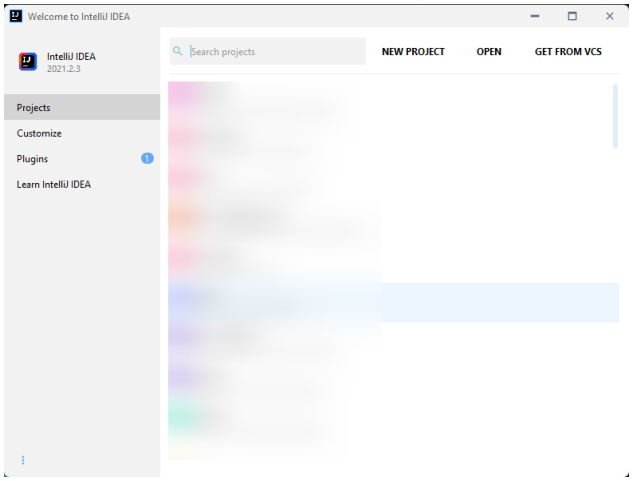


Figura 2.2: Ventana inicial de *IntelliJ*.

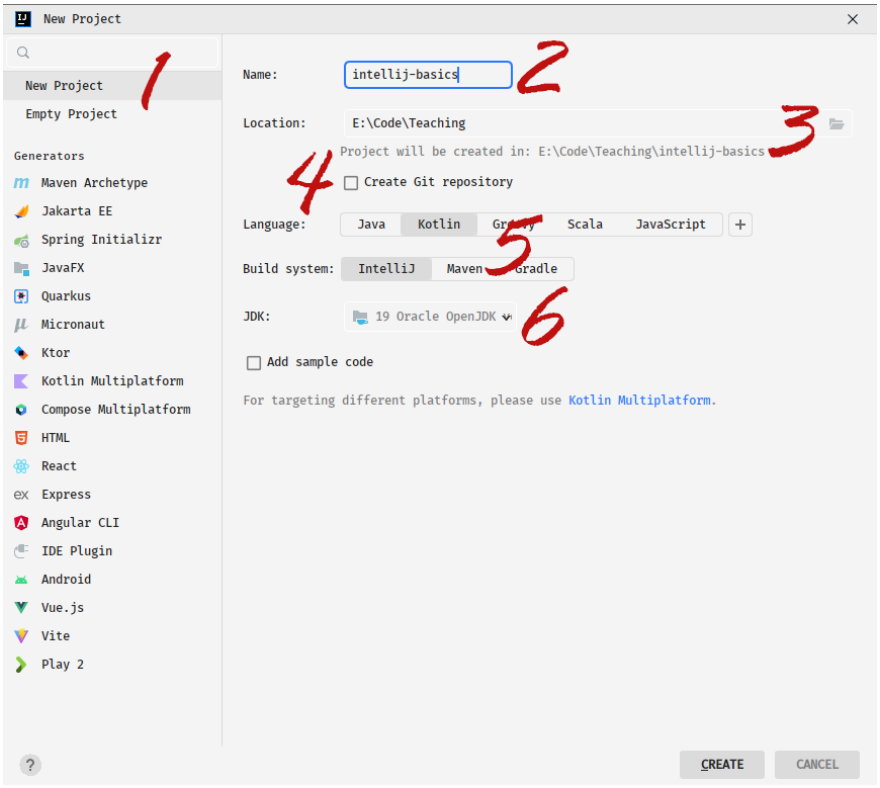


Figura 2.3: Ventana de creación de proyectos

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

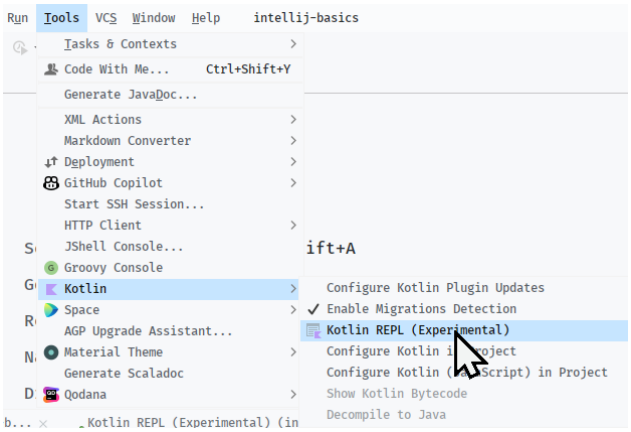


Figura 2.4: ubicación de la *consola REPL* en *IntelliJ*.

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`.

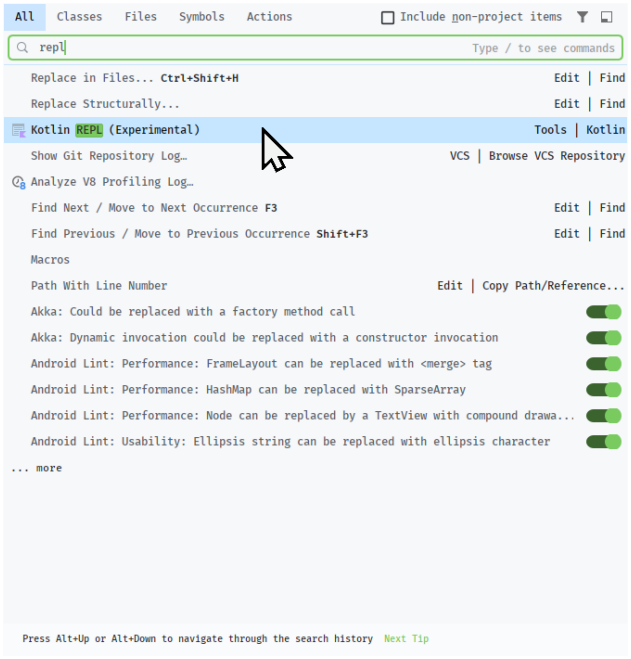


Figura 2.5: Resultado de *Search Everywhere* para la búsqueda de `repl`.

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.

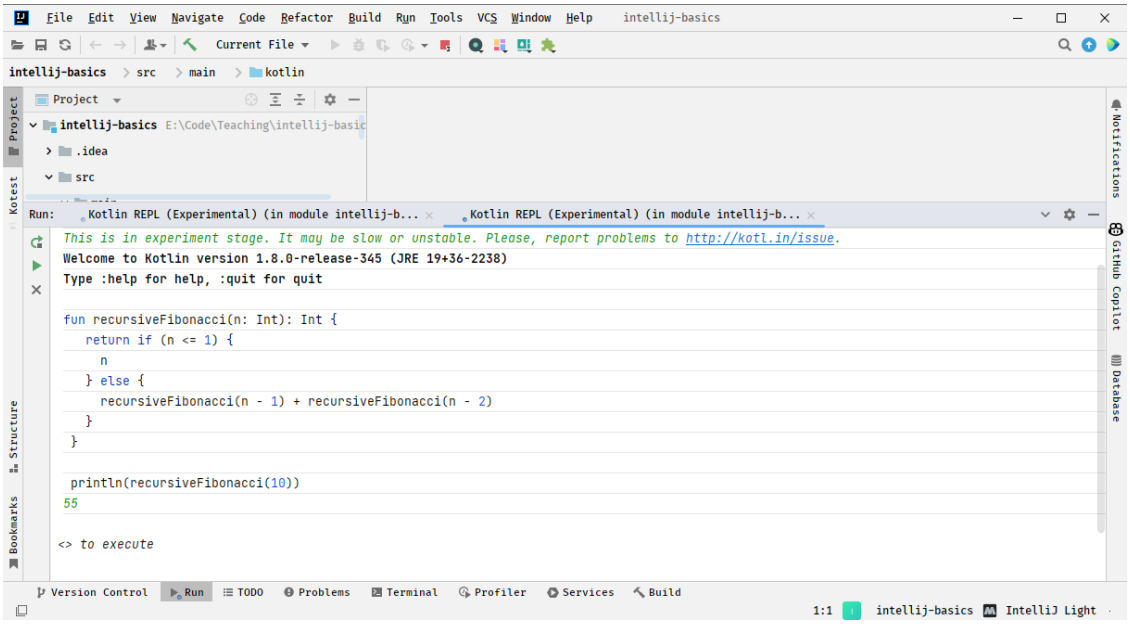


Figura 2.6: Ejemplo de uso de la *consola REPL* en *IntelliJ*.

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*.

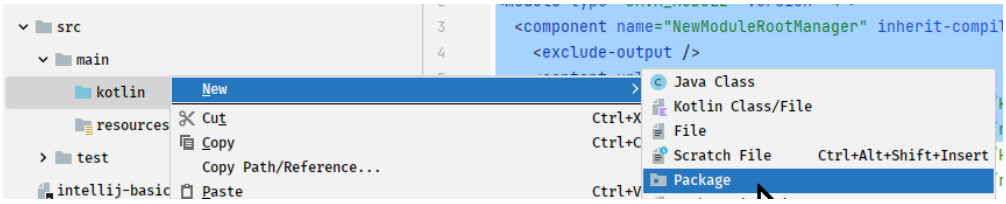


Figura 2.7: Creando un nuevo paquete

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).

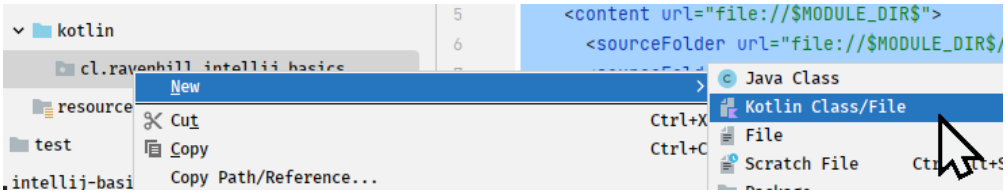


Figura 2.8: Creando un nuevo archivo *Kotlin*

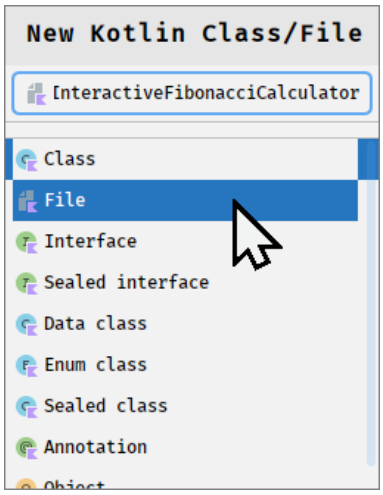


Figura 2.9: Creando un nuevo archivo *Kotlin*

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

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() {
  println("Welcome to the Fibonacci Calculator!")
}
```

¡Y listo! Tenemos nuestra primera aplicación en *Kotlin*. Para ejecutarla, buscaremos el botón *Run* ubicado al lado derecho 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).

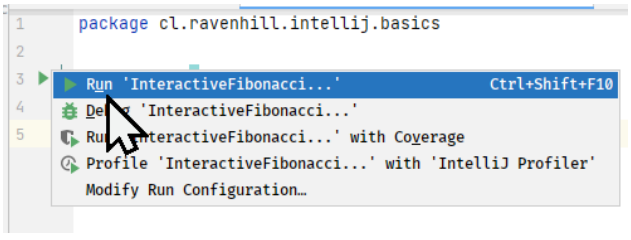


Figura 2.10: Ejecutando la aplicación

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.

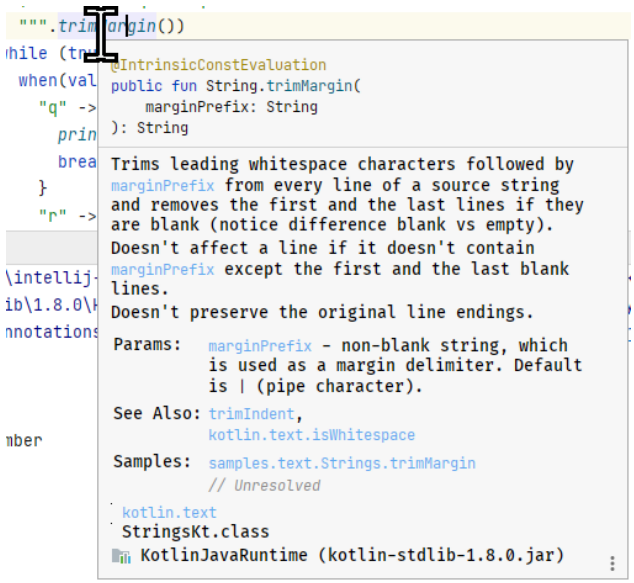


Figura 2.11: Documentación de la función `trimMargin()`.

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)`[4].

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))
```

[4] `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

Harwani: Convert Decimal to Binary in C **harwaniConvertDecimalBinary2021**

Pawan Harwani. *Convert Decimal to Binary in C*. Scaler Topics. 24 de nov. de 2021. URL: https://www.scaler.com/topics/decimal-to-binary-in-c/ (visitado 06-02-2023).

Parte II

OOP no es religión

Capítulo 3

Programación orientada a 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[1] problemas complejos en objetos que interactúan entre sí. 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.

3.1. Objetos

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 -> 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

[1] Abstraer es el proceso de eliminar detalles innecesarios de un objeto o sistema para enfocarnos en sus características esenciales *OOP Concept for Beginners: What is Abstraction?* Este será un concepto clave en todo lo que resta del libro.

el tipo de dato ya que *Kotlin* lo infiere automáticamente y es obvio por contexto. Digamos que nuestro dado tiene 6 caras, por lo que nuestro campo *sides* tendrá el valor por defecto de 6.

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

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

```
/**
 * Object that represents an n-sided dice.
 *
 * @property sides Number of sides of the dice.
 */
object Dice {
  var sides = 6

  /**
   * Rolls the dice.
   *
   * @return A random number between 1 and the number of sides of the dice.
   */
  fun roll(): Int {
    return (1..sides).random()
  }
}
```

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

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

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

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

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

3.2. 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 *Person* que represente a una persona.

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.oop.essential` y seleccionar la opción *New -> Kotlin Class/File* y seleccionar la opción *Class*. Esto creará el siguiente archivo:

```
package cl.ravenhill.oop.essential

class Person {
}
```

Ahora podemos agregar campos a la clase:

```
class Person {
  var name: String = ""
  var age: 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 *Person*:

```
fun main() {
  val person = Person()
  person.name = "Pepe"
  person.age = 30
  println("Name: ${person.name}")
  println("Age: ${person.age}")

  val person2 = Person()
  person2.name = "Pepa"
  person2.age = 25
  println("Name: ${person2.name}")
  println("Age: ${person2.age}")
}
```

3.2.1. 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 Person(name: String, age: Int) {
  val name: String
  val age: Int

  init {
    this.name = name
    this.age = age
  }
}
```

Vamos viendo lo que está pasando:

- Primero definimos los campos de la clase, los cuales son *name* y *age*.
- 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.age`?
- ¿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 person = Person("Pepe", 30)
  println("Name: ${person.name}")
  println("Age: ${person.age}")

  val person2 = Person("Pepa", 25)
  println("Name: ${person2.name}")
  println("Age: ${person2.age}")
}
```

3.3. Composición

Para las siguientes secciones utilizaremos como ejemplos un juego de cartas inspirado en *Yu-Gi-Oh!*.

Comencemos modelando al jugador. Un jugador tiene un nombre, una cantidad de puntos de vida y un mazo de cartas. Un primer intento 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* y *MutableMap*. 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 **composicion** 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
  }
}
```

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.

3.4. 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.healthPoints -= 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),
  Card("Light Magician", "The ultimate sorcerer with a powerful attack", 2500)
)
val player = Player("Yulian", 8000, deck)
player.deck[0].attack(player)
println(player.healthPoints)
```

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.healthPoints) {
    player.healthPoints -= attack
  } else {
    player.healthPoints = 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 < healthPoints) {
    healthPoints -= damage
```

```
  } else {
    healthPoints = 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**.[2]

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 *healthPoints* de la clase *Player* sea privado de la siguiente forma:

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

Ahora el campo *healthPoints* solo puede ser accedido desde la clase *Player*.

Yay

Pero introducimos otro problema.

Bro

Hicimos que el campo *healthPoints* 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.healthPoints)`. Podemos solucionar esto creando un método *getHealthPoints* 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 healthPoints: 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.

[2] 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.

3.5. 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

  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.

3.5.1. 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

```
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*, 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.

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.

3.5.2. Method-lookup

Antes de continuar debemos entender algunos conceptos nuevos.

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

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

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

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

Sobrecarga de funciones

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

Lo siguiente que debemos entender es el concepto de **mensaje**. Pola 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.

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 los 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 (overloading) 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 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("Yulian", 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.oop.essential.Player@cc34f4d has 8000 health points
cl.ravenhill.oop.essential.Player@cc34f4d has 5000 health points
You can't attack with this card
cl.ravenhill.oop.essential.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 `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 `toString()` de *Any* 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:

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

Ejercicio 3.1. ¿Qué pasa si se ejecuta el siguiente código? ¿Por qué?

```kotlin
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 3.2. Implemente las clases `MagicCard` y `TrapCard` que hereden de `Card`.

3.6. Polimorfismo

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`:[3]. 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 3.1, esto cambiará el nombre de la clase en todos los archivos donde se use además de cambiar el nombre del archivo.

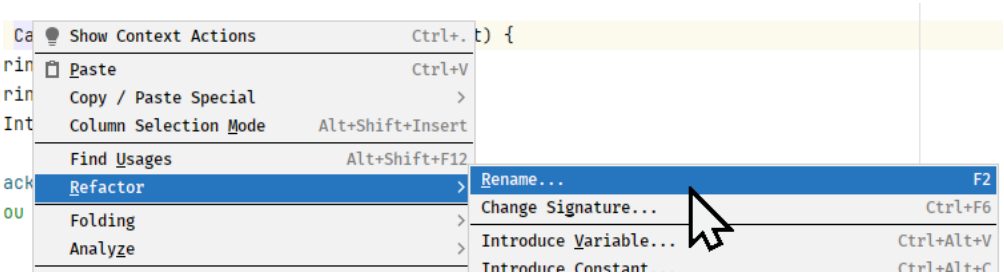


Figura 3.1: Renombrando la clase `Card` a `AbstractCard`.

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

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

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

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

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

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.

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

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 primero 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.

Hagamos un paquete `cards` haciendo click derecho en el paquete `cl.ravenhill.oop.essential` y arrastremos los archivos `Card.kt` y `MonsterCard.kt` dentro de él, esto desplegará una ventana como la de la figura 3.2 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.oop.essential`.

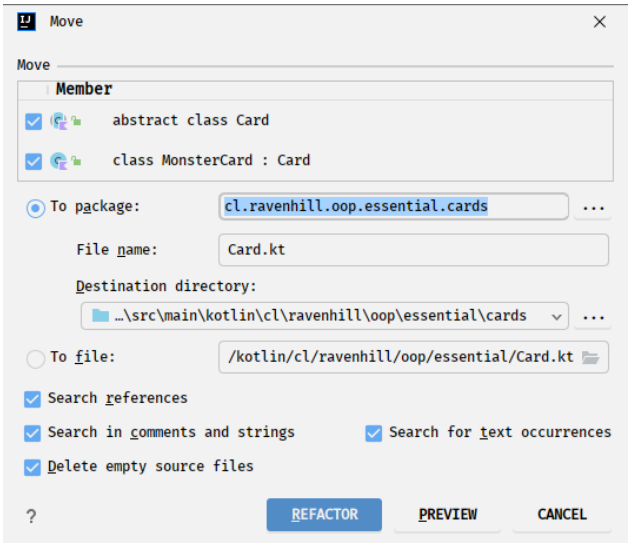


Figura 3.2: Moviendo las clases `Card` y `MonsterCard` al paquete `cards`.

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

```
package cl.ravenhill.oop.essential.serializer

import java.io.File
```

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

  abstract fun serialize(): 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 `AbstractCard` para que use la clase `AbstractCardSerializer`:

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

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

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

Y nuestra clase `MonsterCard` quedaría 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 usar cualquier objeto que cumpla con ese contrato.

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.oop.essential.serializer

interface CardSerializer {
}
```

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

```
interface CardSerializer {
  fun serialize(): String
  fun toFile(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(filename: String) {
```

```
    val file = File(filename)
    file.writeText(serialize())
  }

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

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

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

3.7. 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: CardSerializer) {
  val name: String = name
  val text: String = text
  var serializer = serializer

  fun toFile(filename: String) {
    serializer.toFile(filename)
```

```
  }

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

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

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

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

Bibliografía

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).

Inheritance (Object-Oriented Programming) **InheritanceObjectorientedProgramming2023**

Inheritance (Object-Oriented Programming). En: *Wikipedia*. 3 de feb. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Inheritance_(object-oriented_programming)&oldid=1137190825 (visitado 05-02-2023).

Liskov: Keynote Address - Data Abstraction and Hierarchy **liskovKeynoteAddressData1988**

Barbara Liskov. «Keynote Address - Data Abstraction and Hierarchy». En: *ACM SIGPLAN Notices* 23.5 (mayo de 1988), págs. 17-34. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/62139.62141. URL: https://dl.acm.org/doi/10.1145/62139.62141 (visitado 06-02-2023).

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).

OOP Concept for Beginners: What is Abstraction? **ObjectorientedProgramming2023**

OOP Concept for Beginners: What is Abstraction? 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.

Serialization **Serialization2023**

Serialization. En: *Wikipedia*. 6 de ene. de 2023. URL: https://en.wikipedia.org/w/index.php?title=Serialization&oldid=1132012847 (visitado 06-02-2023).

Shvets: SOLID Principles **shvetsSOLIDPrinciples2021**

Alexander Shvets. «SOLID Principles». En: *Dive Into Design Patterns*. 2021, págs. 51-70.

Index