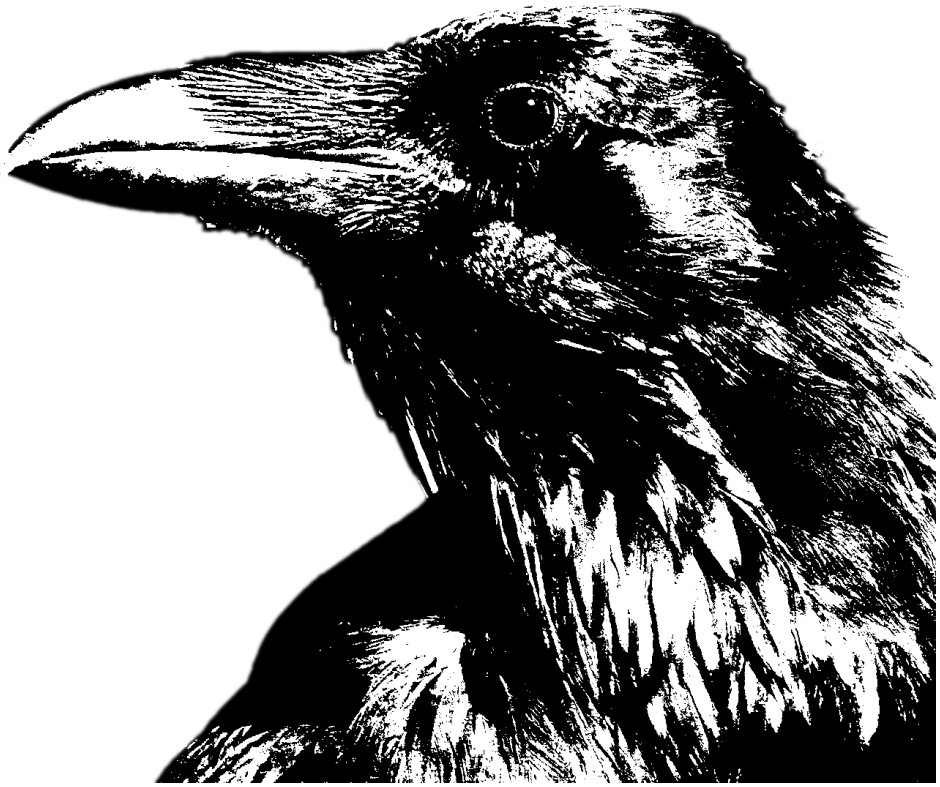
EL TIEMPO PASA Y EL SOFTWARE MUERE

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.

24 de septiembre de 2022, 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).

1.1.1. ¿Debería instalar *Java*?

> «Haz la w** que querai, pero ponte chaleco»
>
> (Anita Tijoux)

Esto es algo completamente opcional, no es necesario para este libro, pero es uno de los lenguajes más utilizados en el mundo y los contenidos de este libro son directamente aplicables a este lenguaje.

Si por alguna razón quieren instalar *Java*, puedes encontrar una guía de instalación al final de este capítulo.

[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.

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.

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.

1.3. 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.

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* y *Kotlin*, 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 Java?

La razón es simple, pero muy importante, como dije un poco más arriba, *IntelliJ* es una herramienta diseñada específicamente para desarrollar en *Java* y *Kotlin*, esto hace que sea muchísimo más completo que un editor de texto cualquiera 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 o lectora 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 lxs lectorxs 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"
```

2.2. Conociendo *IntelliJ*

Ahora sí, llegó el momento de la verdad, *Nostradamus* predijo que algún día instalarían *IntelliJ*, y ese día es hoy. Si eres estudiante, puedes obtener una licencia para utilizar la versión *Ultimate* de forma gratuita siguiendo las instrucciones que aparecen en el sitio oficial de *JetBrains*, esto les dará acceso a todas las herramientas de *JetBrains*.

Para esta parte usaremos como ejemplo la versión *Community*, pero las instrucciones son las mismas para la versión *Ultimate*.

En la figura 2.1 pueden ver la interfaz de *JetBrains Toolbox*, en este caso nos interesa instalar *IntelliJ IDEA Community Edition*.

Y eso es todo. *IntelliJ* está instalado, ahora vamos a usarlo.

2.2.1. Creando un proyecto

El primer paso para utilizar *IntelliJ* es abrirlo (amazing), esto puede tomar varios 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 *Java* (y posteriormente *Gradle*). Si quieren seguir los ejemplos en *Kotlin* deberán marcar la opción resaltada y seguir a la siguiente pantalla con «*Next*».

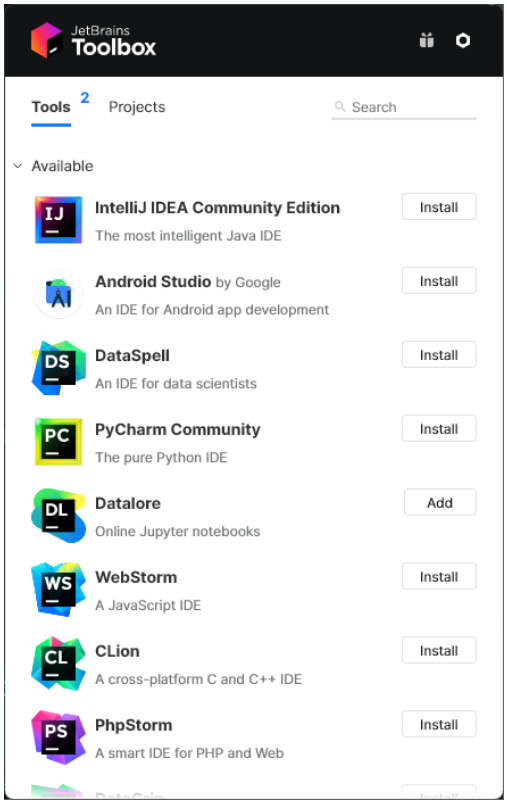


Figura 2.1: Interfaz de *JetBrains Toolbox*.

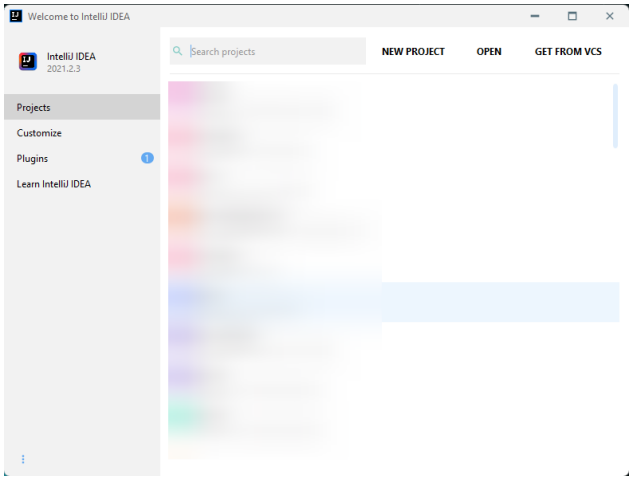


Figura 2.2: Ventana inicial de *IntelliJ*.

La pantalla siguiente es bastante más simple, aquí definiremos el nombre y la ubicación de nuestro proyecto. En este caso lo llamaremos «IntelliJBasics». Una vez que hayamos terminado, deberíamos finalmente ver la interfaz principal de *IntelliJ* (figura 2.4).

¡Y listo, tenemos nuestro primer proyecto!

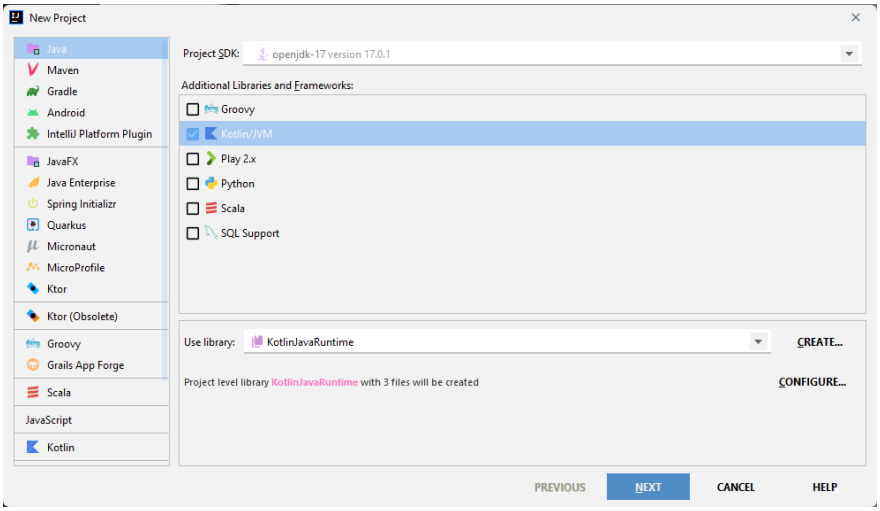


Figura 2.3: Ventana de creación de proyectos

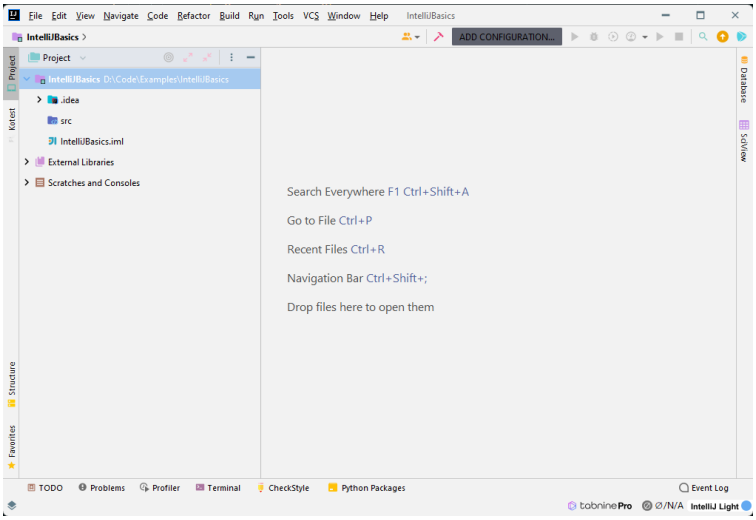


Figura 2.4: Pantalla principal de *IntelliJ*.

2.2.2. Trabajando con *IntelliJ*

En el **??** les dije que nos despediríamos de *jshell*, les mentí (parcialmente). Antes de comenzar a utilizar realmente *IntelliJ* para crear un programa veremos una pincelada de las funcionalidades de este *IDE* mediante la consola de *Jshell* de *IntelliJ*.

Pueden acceder a la consola desde la sección «Tools» como se muestra en la figura 2.5.

Search Everywere

Otra forma de abrir la consola de *Jshell* es abriendo el menú *Search Everywhere*[a] (ver figura 2.6), y en la barra de búsqueda escribir textttjshell.
Esta herramienta nos ayudará mucho a conocer y usar las herramientas de *IntelliJ*.

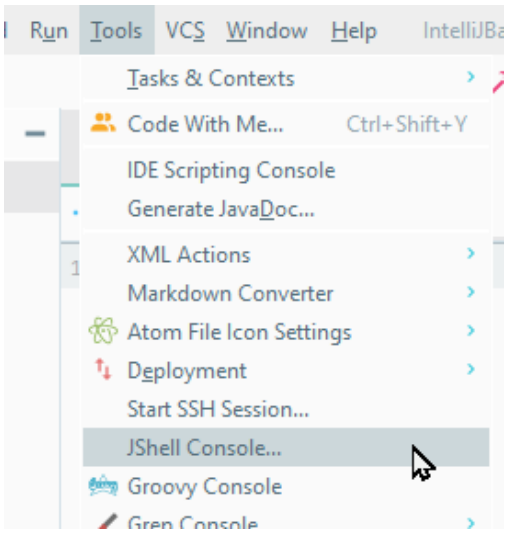


Figura 2.5: Acceso a la consola *JShell*.

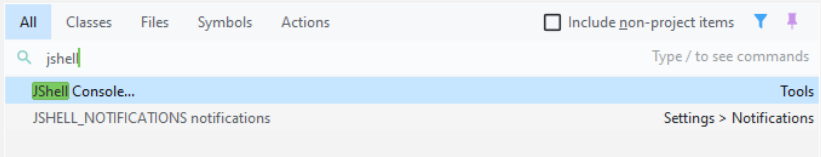


Figura 2.6: Resultado de *Search Everywhere* para la búsqueda de `jshell`.

[a] Disponible en todos los *IDEs* de *JetBrains*

Ahora intenten reescribir el código de *Java* del **??** para ver cómo se siente trabajar con un *IDE*. Deberían tener algo como en la figura 2.7. Luego pueden ejecutar el código y ver el resultado en la consola presionando el botón «*play*» en la esquina superior izquierda del script.

Noten que al definir la función `fibonacci` agregamos la *keyword* `static` al principio de la función, es complejo explicar la razón de eso en este momento, así que por ahora sólo créanme que se hace así.

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 `IntelliJBasics.iml`, y las carpetas `.idea` y `src`. Su utilidad es:

- `.idea`: contiene la configuración de IntelliJ.
- `src`: contiene el código fuente de nuestra aplicación.
- `IntelliJBasics.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`.

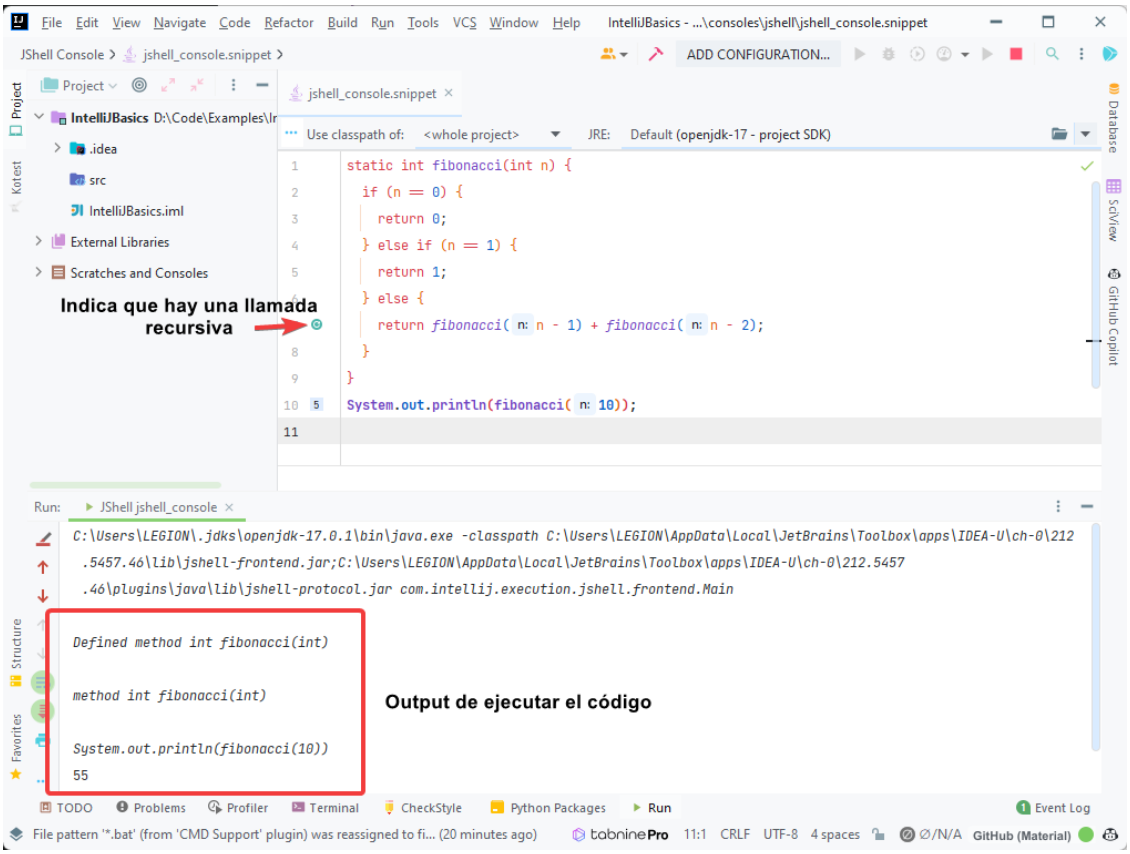


Figura 2.7: Ejemplo de uso de *JShell* en *IntelliJ*.

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.

Tanto *Java* como *Kotlin* 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* y *Kotlin*, en general es recomendado utilizar el nombre de algún sitio web para asegurarse de que el nombre del paquete es único (para evitar problemas de nombres duplicados al utilizar librerías externas). Veamos algunos ejemplos:

package cl.ravenhill.intellijbasics; // 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

[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.

2.3.2. Mi primer *Java*

2.3.3. Mi primer *Kotlin*

Capítulo 3

Git ¿Amigo o enemigo?

Una de las herramientas más importantes que deben conocer es *Git*, o algún otro *sistema de versionado*. Un *sistema de versionado* (VCS) es un programa para mantener un historial de todos los cambios realizados sobre un conjunto de archivos. Existen variados VCS como *Azure DevOps*, *CVS* o *Mercurial*, pero el más utilizado es *Git*.

Git fue creado por Linus Torvalds el aµo 2005 para poder desarrollar el núcleo de *Linux* junto a otros colaboradores. Esto deja un precedente de que *Git* fue ideado con la idea de poder trabajar de forma colaborativa en una aplicación, pero también es extremadamente útil para trabajar en un proyecto de forma individual.

Los beneficios de utilizar *Git*, y algunos otros detalles sobre cómo funciona, los iremos viendo a medida que ponemos en práctica esta herramienta.

3.1. Instalando *Git*

3.1.1. *Windows*

***Chocolatey* (Recomendado)**

Nuevamente, el método recomendado será utilizar *Chocolatey* para instalar *Git* en *Windows*.

El proceso de instalación es simple, solamente deben ejecutar *Powershell* como administrador y escribir:

```
choco install git
# O de forma equivalente:
# cinst git
```

Git for Windows

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

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

3.1.2. Linux

La forma más fácil de instalar *Git* es utilizando el gestor de paquetes del sistema operativo que estén usando, el problema de esto es que dependiendo de su distribución de *Linux* las instrucciones de instalación serán distintas, por esto se

mostrará como instalar en arquitecturas basadas en *Debian* (como *Ubuntu*), para instalar en otro SO deberán revisar las instrucciones de la documentación oficial.

Para instalar deben abrir una consola y ejecutar:

```
sudo apt install git-all
```

3.1.3. macOS

***Git* mediante *Xcode CLT* (Recomendado)**

La forma más simple de instalar *Git* en sistemas *macOS* es a través de *Xcode CLT* (¡No confundir con *Xcode*!), en este caso, lo primero que deberán hacer es instalar el *CLT* desde una terminal:

```
xcode-select --install
```

Una vez instalada esta herramienta[1] para instalar *Git* bastará ejecutar el siguiente comando en la terminal:

```
git --version
```

Git* mediante *Homebrew

Otra forma de instalar *Git* es utilizando *Homebrew*, esto también es simple de hacer. En una terminal ejecuten:

```
brew install git
```

Luego, para comprobar la instalación hacemos:

```
git --version
```

3.2. ¿Repositorios?

Uno de los conceptos claves de *Git* son los repositorios. En la práctica, un repositorio no es más que una carpeta en su computador, lo que lo hace especial es cómo usamos esa carpeta.

En el caso de *Git*, un repositorio va a ser una carpeta que contiene un directorio con una carpeta llamada `.git` con ciertos archivos de configuración especiales. Podemos crear un repositorio de forma simple usando el comando `git init`. El siguiente ejemplo crea una carpeta y luego la configura como un repositorio de *Git*.

```
mkdir git_example
cd git_example
git init
```

Ahora, podremos ver que se creó una carpeta `.git` dentro de `git_example`, sin embargo, esta carpeta en general está oculta. Para ver la carpeta desde la terminal podemos hacer:

[1] Pueden verificar que se haya instalaado de forma correcta ejecutando `xcode-select -p`.

Powershell

```powershell
# Desde `git_example`
Get-ChildItem . -Hidden
```

Bash

```bash
# Desde `git_example`
ls -a .
```

Ya tenemos nuestro repositorio, ahora hay que sacarle provecho. Lo primero que haremos será conocer a su nuevo comando favorito: `git status`. Este comando nos servirá para ver el estado del repositorio en cualquier momento, si ahora lo ejecutamos en nuestro repositorio debiera decir:

```
On branch master

No commits yet

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

Aca vemos muchos conceptos nuevos: *branch*, *master*, *commit*, etc. Vamos a comenzar con lo más básico y haremos nuestro primer *commit*.

3.2.1. *Commits*

Tomemos un pequeño desvío de la computación y pensemos en un libro de historia. Los libros de historia registran sucesos históricos para que luego puedan ser estudiados por otrxs que estén interesados en conocer y analizar estos sucesos. ¿Pero qué es un suceso historico? Podríamos argumentar que todo lo que sucede en nuestro día a día es un suceso histórico, pero no todo lo que sucede en el mundo será relatado en los libros de historia, alguien (historiadores principalmente) tiene que decidir qué sucesos serán registrados.

Podemos pensar nuestro repositorio como un libro de historia, cada cambio que hagamos dentro del repositorio puede ser un suceso histórico, pero no todos esos cambios debieran ser registrados. A la acción de registrar un «suceso» en nuestro repositorio le llamaremos *Commit*.

Veamos cómo poner esto en práctica. Primero, crearemos un archivo dentro del repositorio que se llame `README.md` con el siguiente contenido:

```markdown
# Mi pequeño gran repositorio

**SIEMPRE** deben incluir un _readme_ en sus repositorios.
```

Este archivo es bastante particular, pero también estándar. Una buena práctica es siempre agregar un archivo `README.md` en nuestros repositorios, esto le servirá a todxs lxs que tengan que trabajar después con nuestros repositorios. El formato `.md` indica que el archivo utiliza el lenguaje *Markdown*. No entraremos en detalles en la sintáxis de *Markdown*, para esto pueden referirse a las bibliografías.[2]

Ahora estamos listos para hacer un *Commit*, pero primero veamos el estado del repositorio con `git status`, esto debiera resultar en:

[2] **gh-markdown**.

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md

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

Como es esperable, el mensaje sigue diciendo que no hay *commits*, pero ahora aparece una nueva sección *Untracked files* con el archivo que creamos. Los *untracked files* son los archivos que no hemos registrado en nuestro libro de historia y, como veremos un poquito más adelante veremos, no siempre vamos a querer registrar todos nuestros cambios, pero es sumamente importante nunca dejar *untracked files* «sin resolver».

Ahora, pongámosle énfasis a la última línea del mensaje: «*nothing added to commit*»; esto significa que no le hemos dicho a *Git* qué cambios son los que debe registrar. Podemos agregar estos cambios con el comando `git add` de la siguiente forma:

```
git add README.md
```

De nuevo, podemos hacer `git status` para ver los cambios.

Por último, nos queda hacer *Commit* (omg!). Para esto usamos el comando `git commit` pasándole un mensaje descriptivo para identificar el *commit*:

```
git commit -m "doc(repo): Created README"
```

Index