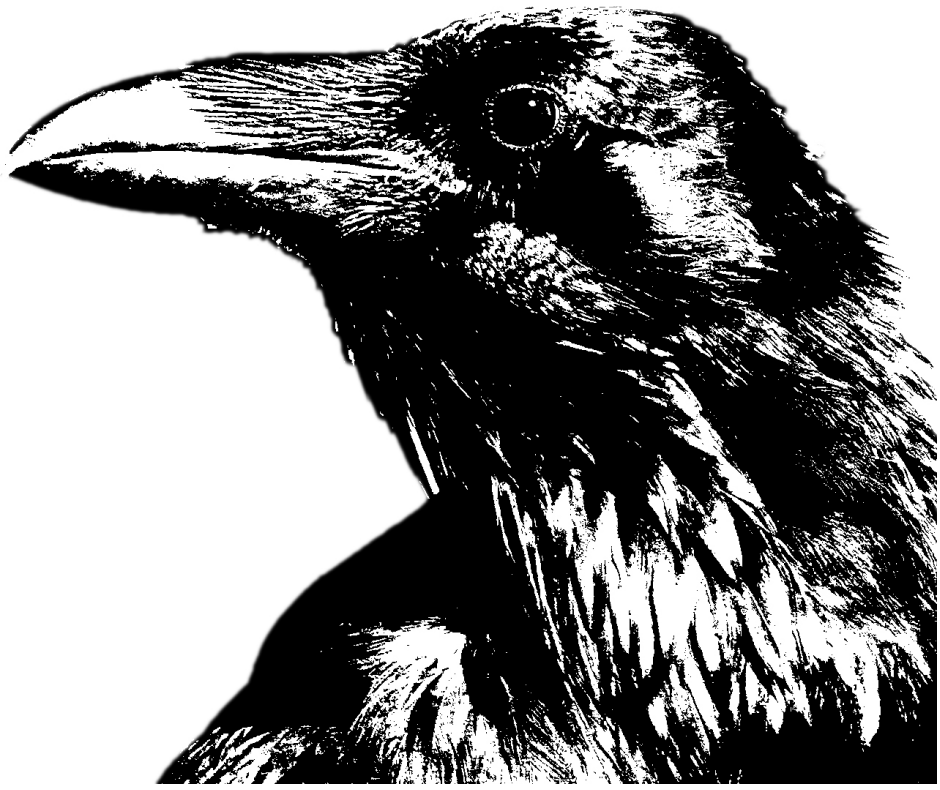
EL TIEMPO PASA Y EL SOFTWARE MUERE

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 de los lenguajes de programación *Java* y *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.

8 de abril 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] Eric Freeman y Elizabeth Freeman, *Head First Design Patterns*.

Parte I

Por algo se empieza

Capítulo 1

¿Qué es un 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).

No tiene mucho sentido seguir hablando de *Java* si no podemos poner en práctica lo que vayamos aprendiendo, así que es un buen momento para instalarlo.

1.1. Instalando el *JDK*

El *Java Development Kit* (*JDK*) es un conjunto de herramientas que incluyen todo lo necesario para desarrollar aplicaciones en *Java* (la *JVM*, el compilador, la librería estándar, etc.), esto es lo que generalmente instalaremos si queremos programar en *Java*.[3]

Si sólo quieren instalar todo sin entender nada pueden utilizar el código existente en `https://github.com/islaterm/software-design-book-installation-scripts`.

[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.

[3] Existen otras herramientas que incluyen los contenidos de el *JDK* de forma total o parcial, por ejemplo: *Java SE*, *JRE*, o incluso otros lenguajes de programación que usan la *JVM*.

1.1.1. Windows

OpenJDK

La primera opción es instalar la versión de código abierto del *JDK*.

Para esto tenemos dos opciones:

OpenJDK con Chocolatey (Recomendado)

Lo primero que necesitaremos para instalar las herramientas que usaremos será un gestor de
Para partir abran una ventana de *Powershell* como administrador. Una vez abierta, deben ejecutar las instrucciones:[a]

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

[a] https://github.com/islaterm/software-design-book-es/blob/master/install/windows/chocolatey.ps1

(Opcional) Cmder

Es sabido que las terminales por defecto de *Windows* dejan bastante que desear, por eso es una buena idea instalar una terminal externa (o más bien un emulador de una). Existen varias opciones, pero *Cmder* es una de las más completas.
Para instalar la terminal utilizaremos *Chocolatey*. En una terminal de *Powershell* con permisos de administrador ejecuten:

```
cinst cmder -y # Equivalente a 'choco install cmder -y'
```

Con esto basta para tener *Cmder* instalado, pero una de las principales ventajas de utilizar este emulador de consola es la capacidad de personalizarlo. A partir de aquí continuaremos desde *Cmder*.
Lo siguiente será instalar herramientas que ayudarán a entregar de mejor forma la información al momento de usar la consola. Para esto, deberán ejecutar los siguientes comandos:

```
Install-PackageProvider NuGet -MinimumVersion '2.8.5.201' `
  -Force
Set-PSRepository -Name PSGallery -InstallationPolicy Trusted
Install-Module -Name 'oh-my-posh'
Install-Module -Name 'Get-ChildItemColor' -AllowClobber
```

Lo último es configurar el perfil de *Powershell*, esto se hace en un archivo que es el equivalente a `.bashrc` de los sistemas *Linux*. Para abrir este archivo ejecuten:

```
# Primero verificamos si existe un perfil de PS, si no existe
# lo creamos
if (-not $(Test-Path $PROFILE -Type Leaf)) {
  New-Item $PROFILE
}
# Aquí pueden usar el editor de texto que quieran
notepad $PROFILE
```

Esto abrirá notepad `D:`[a] (si nunca han configurado la consola, entonces debiera estar vacío). Como último paso, escriban lo siguiente en el archivo de configuración y guarden los cambios.

```
# Helper function to set location to the User Profile directory
function cuserprofile { Set-Location ~ }
Set-Alias ~ cuserprofile -Option AllScope

Import-Module 'oh-my-posh' -DisableNameChecking

# CHOCOLATEY PROFILE
$ChocolateyProfile = `
  "$env:ChocolateyInstall\helpers\chocolateyProfile.psm1"
if (Test-Path($ChocolateyProfile)) {
  Import-Module "$ChocolateyProfile"
}

Set-PSReadlineOption -BellStyle None
Set-Theme Honukai
```

La última línea solamente define el *tema* de la consola, pueden ver una lista de *temas* disponibles en el repositorio de *Oh-My-Posh*

[a] Ojalá sea la última vez en su vida en la que tengan que hacerlo.

OpenJDK sin Chocolatey

Si no funciona, o no quieren usar el gestor de paquetes, también se puede instalar el *JDK* manualmente. Para esto pueden abrir *Powershell* como administrador y ejecutar lo siguiente (sujétense):

```
$originalLocation = Get-Location # Para poder regresar
$url = 'https://download.java.net/java/GA/jdk17.0.1/' `
  + '2a2082e5a09d4267845be086888add4f/12/GPL/' `
  + 'openjdk-17.0.1_windows-x64_bin.zip'
# Creamos una carpeta para instalar Java
New-Item -Path '$Env:ProgramFiles' -Name 'Java' -ItemType Directory
Set-Location '$Env:ProgramFiles\Java'
# Descargamos y descomprimimos el JDK
Invoke-WebRequest -Uri $url -OutFile 'openjdk-17.0.1.zip'
Expand-Archive .\openjdk-17.0.1.zip -DestinationPath .
Remove-Item 'openjdk-17.0.1.zip'
```

Lo siguiente será configurar las variables de entorno necesarias, esto puede hacerse desde la «Configuración avanzada del sistema», o si continuan en la terminal:

```
Set-Location 'jdk-17.0.1'
[Environment]::SetEnvironmentVariable("JAVA_HOME",
                                      "$(Get-Location)")
[Environment]::SetEnvironmentVariable(
  "Path",
  [Environment]::GetEnvironmentVariable('Path',
  [EnvironmentVariableTarget]::Machine) + ";$(Get-Location)\bin",
  [EnvironmentVariableTarget]::Machine)
Set-Location $originalLocation
Update-SessionEnvironment
```

Luego, pueden probar la instalación de la misma manera que en la opción anterior.

Microsoft OpenJDK

Microsoft cuenta con una distribución propia de *OpenJDK*, esta distribución no tiene ninguna diferencia respecto a la mostrada anteriormente, sólo que el número de versión puede variar un poco.

La primera opción de instalación (la más fácil) es utilizar el gestor de paquetes nativo de *Windows* (*Windows Package Manager*). Este gestor de paquetes viene incluido en las versiones más modernas de *Windows 10* y en todas las versiones de *Windows 11*. Pueden ver si su computador tiene esta herramienta ejecutando:

```
winget --version
```

En caso de que eso funcione correctamente, para instalar el *JDK* basta que ejecuten:

```
winget install Microsoft.OpenJDK.17
```

Si lo anterior no funciona, entonces desde la terminal:

```
$url = 'https://aka.ms/download-jdk/' `
  + 'microsoft-jdk-17.0.1.12.1-windows-x64.msi'
Invoke-WebRequest -Uri $url -OutFile 'microsoft-jdk-17.msi'
cmd /c 'msiexec /i microsoft-jdk-17.msi ' `
  + 'ADDLOCAL=FeatureMain,FeatureEnvironment,' `
  + 'FeatureJarFileRunWith,FeatureJavaHome ' `
  + 'INSTALLDIR="c:\Program Files\Java\"'
Remove-Item 'microsoft-jdk-17.msi'
```

1.1.2. Linux (x64)

***Open JDK* (Recomendado)**

Para cualquier distribución de *Linux x64*, desde la terminal:

```
wget https://bit.ly/3kvJ17B
tar xvf openjdk-15*_bin.tar.gz
sudo mv jdk-15 /usr/lib/jdk-15
```

Luego, para verificar que el binario se haya extraído correctamente:

```
export PATH=$PATH:/usr/lib/jdk-15/bin
java -version
```

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.

Si se instaló correctamente, entonces el último paso es agregar el *JDK* a las variables de entorno del usuario, para esto primero deben saber qué *shell* están ejecutando, pueden ver esto con:

```
echo $SHELL
```

En mi caso, esto retorna:

/usr/bin/zsh

Luego, deben editar el archivo de configuración de su *shell*, en mi caso ese sería `~/.zshrc` (en *bash* sería `.bashrc`) y agregar al final del archivo la línea:

```
export PATH=$PATH:/usr/lib/jdk-15/bin
```

Oracle Java SE

Primero deben descargar el *JDK* desde el sitio de *Oracle* (asumiremos que descargaron la versión `.tar.gz`). Luego, desde el directorio donde descargaron el archivo:

```
tar zxvf jdk-15.interim.update.patch_linux-x64_bin.tar.gz
sudo mv jdk-15.interim.update.patch /usr/lib/jdk-15.interim.update.patch
```

Después, de la misma forma que se hizo con la opción anterior:

```
export PATH=$PATH:/usr/lib/jdk-15.interim.update.patch/bin
java -version
```

Si este comando funciona, entonces deberán modificar el archivo de configuración de su *shell* para incluir la línea:

```
export PATH=$PATH:/usr/lib/jdk-15.interim.update.patch/bin
```

1.1.3. MacOS

***Open JDK* (Recomendado)**

Primero, deben descargar la versión apropiada para su sistema operativo desde la página del proyecto *AdoptOpenJDK*, en particular el archivo `.pkg`.

Luego, desde una terminal ubicada en la carpeta donde descargaron los binarios:[4]

```
installer -pkg OpenJDK15U-jdk_x64_mac_hotspot_15.0.2_7.pkg \
  -target /
```

En caso de haberse instalado correctamente debiera haberse creado un directorio en `/Library/Java/JavaVirtualMachines/`. Por último, ejecuten el comando `java -version` para comprobar que la instalación funcione.

1.2. Tipos en Java

Veamos ahora uno de los aspectos más importantes de la programación en *Java*, sobre todo viniendo de algún otro lenguaje de programación. *Java* es un lenguaje con tipos estáticos y fuertemente tipado, estos conceptos los entenderemos mejor viendo algunos ejemplos.

Primero tenemos que repasar el concepto de tipo. Todas las variables de un programa **tienen** un tipo definido, esto es importante ya que de esto dependerá la memoria que se necesite alocar para que la aplicación funcione. Ahora, si bien todas las variables tienen tipo, no siempre será el trabajo del programador definir el tipo de cada variable; aquí diferenciamos en dos sistemas de tipado:

[4] No uso Mac, por lo que no tengo como probar que las instrucciones funciones, si tuvieran problemas por favor díganme para solucionarlos y actualizar las instrucciones.

- **Tipado estático:** El tipo de todas las variables es chequeado al compilar el programa (i.e. antes de ejecutar). Esto tiene varias consecuencias, entre ellas, los tipos de las variables se definen junto a la variable[5] y que una vez que se define el tipo de una variable este no puede cambiar. Algunos ejemplos de lenguajes con tipado estático son *Java*, *C* y *Kotlin*.
- **Tipado dinámico:** El tipo de las variables se decide en tiempo de ejecución (i.e. cuando la aplicación está corriendo). Contrario al tipado estático, como los tipos de las variables no se chequean en un comienzo, no se tiene información sobre las variables hasta que son utilizadas, esto implica que una variable puede «cambiar su tipo» durante la ejecución. Algunos ejemplos de lenguajes con tipado dinámico serían *Python*, *Ruby* y *Javascript*.

Adicionalmente, también existen algunos lenguajes con *tipado mixto*, esto quiere decir que se pueden definir algunas variables con tipo estático y otras con tipo dinámico.

Por otro lado, existe el concepto de tipados fuerte y débil, es común que se confundan con las dos maneras de tipado que explicamos recién, pero son conceptos independientes entre sí. Más en detalle:

- **Tipado fuerte:** El programa exige *seguridad de tipos*, esto significa que una variable de tipo *A* necesariamente se utilizará como si fuera de ese tipo y, en caso de que eso no se cumpla, se arrojará un error señalando que hubo un error de tipo. Noten que esto no se ve afectado si el lenguaje es estático o dinámico ya que en ambos casos el programa puede arrojar un error si hay un error de tipo. Ejemplos de esta modalidad de tipado son lenguajes como *Java*, *Kotlin* y *Python*.
- **Tipado débil:** No existe garantía acerca de cómo utilizar cada tipo de variable. Un caso típico en el que se ve este comportamiento es en lenguajes que utilizan punteros (por ejemplo *C*), ya que estos permiten manejar las variables apuntando a su dirección de memoria ignorando el tipo de valor almacenado en esa dirección.[6]

La mayoría de los lenguajes modernos aseguran la *seguridad de tipos*, pero algunos lenguajes más antiguos que siguen ocupándose frecuentemente en la actualidad son de tipado débil (principalmente *C* y *Assembly*).

Veamos cómo se ve reflejado esto en *Java*. Para correr los ejemplos utilizaremos la consola interactiva de *Java*, para esto ejecuten el comando `jshell` en la terminal.

La sintaxis para definir una variable en java es `<type><id>= <value>`, por ejemplo, si queremos definir una variable que almacene un entero entonces lo podemos hacer como:

```
int i = 8000;
```

Aquí es importante notar que en *Java* **todas** las instrucciones deben terminar con un `;`. Luego, podemos modificar libremente el valor de `i` a otro entero (por ejemplo hacer `i = 420`), pero no podemos asignarle un valor de un tipo distinto (e.g. `6.9`) debido al tipado estático.

Ejercicio 1.1. Si un lenguaje de programación infiere el tipo de las variables por contexto, entonces el lenguaje tiene tipado dinámico. ¿Es esto correcto?

Ejercicio 1.2. ¿Por qué usarían un lenguaje estático? ¿Por qué usarían uno dinámico?

Ejercicio 1.3. ¿Qué caso de uso podría tener un lenguaje de tipado débil?

Ejercicio 1.4. En *Perl* se pueden operar algunos valores de distinto tipo como si fueran el mismo, por ejemplo, el siguiente código entrega como resultado `42`:

```
$a = 11;
$b = "31a";
$c = $a + $b;
print $c;
```

¿Qué clase de tipado se ve en el ejemplo?

[5] Dependiendo del lenguaje y del contexto puede que se necesite declarar explícitamente el tipo de las variables
[6] En *C* esto se puede apreciar en el uso de aritmética de punteros.

Ejercicio 1.5. En *Kotlin* existe el tipo `Any` que representa a todos los tipos, esto significa que podemos asignarle cualquier valor a una variable definida con ese tipo, por ejemplo:

```
var a: Any = 0
a = "Ola ceamos hamigos"
```

Esto podría interpretarse como que *Kotlin* tiene tipado dinámico, pero no es así. ¿Por qué el supertipo `Any` no es un ejemplo de tipado dinámico?

Ejercicio 1.6. En *Javascript* existe lo que suele llamarse *unholy trinity*. En términos simples, se puede entender como en la figura 1.1. En código tendríamos lo siguiente:

```
> false == []
  true
> false == "0"
  true
> [] == "0"
  false
```

¿Qué clase de tipado ejemplifica esto?

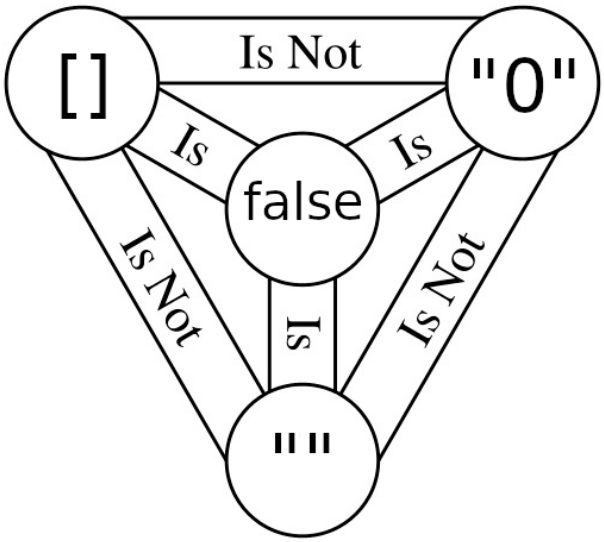


Figura 1.1: *Unholy trinity*

1.2.1. Tipos primitivos

Okay, ahora sabemos que *Java* tiene tipos, falta saber cuáles son esos tipos.

En *Java* los tipos se pueden agrupar en dos categorías: los **tipos primitivos** y los **objetos**. La mayor parte de este libro se enfocará en cómo utilizar y aprovechar las propiedades de los objetos pero primero debemos entender la diferencia que tienen estos de un tipo primitivo.

Los *tipos primitivos* son datos que ocupan una cantidad fija de espacio en la memoria y que pueden representarse directamente como valores numéricos. Esto hace que sean más eficientes de utilizar, ya que una vez que se les asigna un lugar en la memoria difícilmente tendrán que moverse a otra dirección (algo que sucederá mucho con los objetos).

La tabla 1.1 muestra los tipos primitivos de *Java* (en *Python* también existen los tipos primitivos pero la forma en que funcionan es más compleja).

Tipo	Uso de memoria	Rango	Uso	Valor por defecto
`byte`	8-bits	[-128, 127]	Representar arreglos masivos de números pequeños	`0`
`short`	16-bits	[-32.768, 32.767]	Representar arreglos grandes de números pequeños	`0`
`char`	16-bits	[0, 65.535]	Caracteres del estándar *ASCII*	`'\u0000'`
`int`	32-bits	$[-2^{31}, 2^{31}-1]$	Estándar para representar enteros	`0`
`long`	64-bit	$[-2^{63}, 2^{63}-1]$	Representar enteros que no quepan en 32-bits	`0L`
`float`	32-bit	Estándar *IEEE 754x32*	Representar números reales cuando la precisión no es importante	`0.0f`
`double`	64-bit	Estándar *IEEE 754x64*	Representar números reales con mediana precisión	`0.0`
`boolean`	No definido	`true` o `false`	Valores binarios	`false`

Cuadro 1.1: Tipos primitivos en *Java* (los valores por defecto son los que toma la variable si no se le entrega un valor explícitamente **y es un campo** de una clase)

Importante. *Noten que la sintaxis en Java para los valores boolean es* `true` y `false`*, mientras que en Python es* `True` y `False`.

Type promotion

Los tipos primitivos nos permitirán hacer las operaciones más básicas que vamos a necesitar: suma, resta, multiplicación, comparación, etc. Pero como *Java* es de tipado estático, todas las operaciones entre primitivos también deben tener un tipo definido. Un ejemplo de esto es que si sumamos dos números enteros el resultado también deberá ser un entero.

El «problema» surge cuando tenemos una expresión en la que no es totalmente evidente el tipo de retorno. Para solucionar esto *Java* implementa lo que se conoce como ***type promotion***, que se puede resumir en los siguientes pasos:

1. Si hay valores de tipo `byte`, `short`, o `char` en una operación el compilador los convierte en `int`
2. Si todos los valores de la operación son del mismo tipo, entonces retornan un valor de ese tipo; esto quiere decir que `double + double = double`, pero también que `int / int = int`.
3. Si quedan valores de distinto tipo, entonces se transforman al tipo «más grande», partiendo por `long`, luego `float` y, si no puede ser ninguno de los anteriores, `double`.

Wrappers

Todos los tipos primitivos en *Java* tienen un objeto «equivalente» para brindar operaciones que no se pueden realizar directamente con los tipos primitivos, más adelante veremos algunas de estas funcionalidades. A estos objetos equivalentes se les llama ***wrappers***. Algunos ejemplos de esto pueden ser `int` e *Integer*, o `char` y `Character`. Estos objetos, junto con los *strings* son tipos especiales que pueden definirse y utilizarse de una forma más simple que el resto de los objetos, algunos ejemplos:

```
// Podemos declarar un ``Integer`` de la misma forma que haríamos
// con un ``int``
int a = 1;
Integer b = 1;
// Podemos operar `a` y `b` como si fueran del mismo tipo
```

```java
System.out.println(a + b);
// Un caracter se declara entre comillas simples
char c1 = 'c';
// Mientras que un String con comillas dobles
String c2 = "c";  // Noten que c1 es un primitivo y c2 un objeto
// Podemos concatenar strings
System.out.println("Ola" + "comotai");
// Los strings pueden concatenarse con objetos de otros tipos; en
// este caso, el resultado de la concatenación siempre será un
// String
String s = 11 + ": entonces";
```

Ejercicio 1.7. Dentro de *jshell* pueden hacer `/vars` para obtener los datos de las variables que hayan creado, por ejemplo:

```
jshell> int a = 1;
jshell> double b = 2;
jshell> a + b;
jshell> /vars
    int a = 1
    double b = 2.0
    double $3 = 3.0
```

Para las siguientes preguntas utilicen *jshell* para verificar sus respuestas.

1. ¿Cuál es el tipo resultante de sumar un `int` con un `Integer`?
2. ¿Cuál es el resultado de hacer `'f' + 'c'`? ¿De qué tipo es el resultado? ¿Cambia el resultado si hacemos `char fc = 'f' + 'c'`?
3. ¿Qué resultado se obtiene de hacer `"f" + 'c'`? ¿Cuál es el tipo resultante?
4. `Integer.MAX_VALUE` es una constante igual a $2^{31} - 1$, que es el número más grande que se puede almacenar en un `int`. ¿Cuáles son el resultado y tipo de hacer `Integer.MAX_VALUE + 1`?

1.2.2. Arreglos

Además de los tipos primitivos tendremos otro tipo (o más bien una estructura) que debemos conocer. Los *arreglos* son un conjunto ordenado de valores.

Los arreglos son objetos que comparten la propiedad de los tipos primitivos de ocupar una posición fija y constante en la memoria. Con los tipos primitivos es fácil entender como un conjunto fijo de éstos ocupará siempre la misma cantidad de memoria pero cuando tenemos arreglos de objetos no es tan obvio y para entenderlo se necesita entender el concepto de punteros y referencias, pero esto se escapa del alcance de este libro.[7]

Dos conceptos que suelen confundirse son el de arreglo y lista. En *Python* no se hace diferencia entre lo que es un arreglo y lo que es una lista (en estricto rigor, *Python* no tiene arreglos), pero en otros lenguajes como *Java* y *C++* son fundamentalmente distintos y es importante saber cuándo usar uno u otro. Más adelante volveremos a retomar el concepto de lo que es una lista, pero ahora veamos cómo usar los arreglos. Existen contextos en los que se refiere a los arreglos como listas, pero eso es incorrecto; más adelante veremos más en detalle lo que es una lista y por qué es importante hacer la diferencia con los arreglos.

La sintaxis para crear un arreglo es `type[] name = new type[size]`. Un ejemplo podría ser:

```java
int[] integers = new int[3];
```

Lo que hace el código anterior es crear un arrreglo de 3 números enteros, o más bien, reservar la memoria para 3 números. Luego, podríamos «llenar» el arreglo de la siguiente forma:

[7] Evans y Flanagan, *Java in a Nutshell*.

```
integers[0] = 11;
integers[1] = 420;
integers[2] = 8000;
```

Pero se imaginarán que iniciar así uno a uno los valores terminaría en muchas líneas de código. Otra forma de hacer esto sería introduciendo un ciclo, pero para arreglos relativamente pequeños *Java* tiene una sintaxis más compacta:

```
int[] integers = new int[]{11, 420, 8000};
```

Noten que en esta segunda forma no especificamos el tamaño del arreglo ya que el compilador lo puede saber fácilmente.

Importante. *Una última cosa importante a tener en cuenta es que en Java* ***los arreglos son homogeneos****, esto significa que no podemos crear arreglos que tengan elementos de tipos distintos.*

1.2.3. Inferencia de tipos

Las últimas versiones de Java han añadido bastante *azúcar sintáctica*[8] al lenguaje, en gran parte para hacerse cargo de una de las críticas más comunes de *Java* que es lo verboso que es el lenguaje. En particular, cuando leemos un fragmento de código hay veces en las que el tipo de las variables nos puede ser evidente su tipo, tomemos el siguiente código en *Python* como ejemplo.

```
once = 11
apruebo = "entonces"
```

Del código anterior podemos ver claramente que `once` es un *entero* y `apruebo` es un *string*. Pero si usamos un lenguaje de tipado estático entonces necesitamos declarar el tipo de las variables, por lo que el mismo ejemplo en *Java* sería algo como:

```
int once = 11;
String apruebo = "entonces";
```

La inferencia de tipos le permite resolver el tipo de las variables al compilador cuando estos son «evidentes», evitando así que el programador tenga que especificarlos manualmente. En *Java* esto se puede hacer utilizando la keyword `var`, con lo que el ejemplo anterior podría reemplazarse por:

```
var once = 11;
var apruebo = "entonces";
```

Dejar que el compilador infiera los tipos puede hacer mucho más fácil escribir y leer el código, pero si se abusa de esta capacidad puede hacer el código muchísimo más difícil de mantener. La inferencia de tipos debe utilizarse solamente cuando el tipo de la variable se puede distinguir sin problemas cuando se le declara y que no de cabida a ambigüedades. En el ejemplo anterior la variable `once` no debiera declararse como `var` ya que, como mencionamos en la sección anterior, en *Java* existen dos representaciones para los números enteros, `int` e `Integer`, por lo que no es evidente cuál es su tipo.

Algunos ejemplos de usos correctos e incorrectos:

```
/* INCORRECTO */
// No queda claro si es un int o un Integer
var uno = 1;
// No sabemos a priori qué retorna el método
var ola = comotai();

/* CORRECTO */
// Esto sólo puede ser un string
var yes = "good";
```

[8] Sintáxis que facilita la lectura y/o escritura de un programa.

```java
// Se define claramente que es un arreglo de ``int''
var fib = new int[] {0, 1, 1, 2, 3, 5, 8};
```

Sin embargo, la inferencia de tipos en *Java* es muy limitada en comparación con otros lenguajes que comparten esta característica. Lo único que podremos declarar con tipos implícitos serán variables locales dentro de métodos. Esto quedará más claro a medida que vayamos utilizando esta característica a lo largo del libro.

1.2.4. Primeros pasos

Si programar fuera como salir a trotar (afortunadamente no lo es), podríamos pensar que todo hasta este punto fue la preparación previa, buscar las zapatillas, la ropa, poner su playlist más motivante. Pero todavía no podemos comenzar a hacer ejercicio, primero debemos calentar; eso es esta parte, un calentamiento para soltar las manos con *Java*.

Para los siguientes ejemplos puede serles útil tener dos terminales abiertas: una con *jshell* y otra con la consola interactiva de *Python*.[9]

Funciones

Lo más básico que nos gustaría poder hacer en un programa es crear una función. Partamos por algo simple, imprimir un mensaje en pantalla, en *Python* esto se hace con la función `print(str)` mientras que en *Java* tenemos que usar `System.out.println(String)`, vemos inmediatamente que el segundo es bastante más complejo, por ahora ignoren la parte `System.out` y asuman que esa instrucción escribe en la salida estándar.[10]

```python
# Python
def jojo_reference(nombre):
    print("Mi nombre es " + nombre + " y tengo tuto")
```

```java
// Java
void jojoReference(String nombre) {
  System.out.println("Mi nombre es " + nombre + " y tengo tuto");
}
```

Revisemos los ejemplos en detalle. La primera línea en ambos ejemplos es un comentario, así como en *Python* los comentarios comienzan con `#`, en *Java* comienzan con `//`.

La siguiente línea es la *firma*[11] de la función. En *Python* las funciones se definen como:

```python
def func_name(parameters):
```

sin necesidad de especificar los tipos por lo que explicamos en la sección anterior. En *Java* la sintaxis es bastante más explícita, teniendo que declarar el tipo de retorno de la función y el tipo de sus parámetros de la forma:

```java
returnType funcName(Param1Type param1, Param2Type param2, ...) {...}
```

donde: (1) `returnType` es el tipo del valor que retorna la función, (2) `funcName` es el nombre de la función y, (3) `paramXType` y `paramX` son el tipo y nombre de cada parámetro. Otro detalle que podemos ver es que en *Python* la función se llama `func_name` y en *Java* `funcName`, esto es por que las convenciones de ambos lenguajes son distintas; pueden revisar las convenciones de *Java* en la sección **??**.

Por último está el cuerpo de la función. En *Python* el comienzo del cuerpo de una función se marca con `:` y el final de ésta está dada por la sangría (o *indentación*). En *Java* los espacios en blanco no tienen importancia para que el programa funcione,[12] en cambio, el inicio y fin de una función lo marcan la apertura y cierre de las llaves «{}». En

[9] Los comandos para abrirlos son `jshell` y `python` respectivamente.

[10] Si les ayuda, lo que hace la instrucción es escribir un mensaje en el canal de salida (*out*) del sistema, que no necesariamente va a ser la salida estándar siempre.

[11] Esto lo veremos en detalle en la sección **??**.

[12] ¡Pero sí para hacer su código legible!

general podremos pensar que en todos los casos en los que *Python* ocupa `:` el equivalente en *Java* serán las llaves (e.g. `if`, `while`).

Recursión y control de flujo

Veamos algo un poquito más complicado, supongamos que queremos computar el $n-$ésimo número de la *sucesión de Fibonacci*, esta secuencia está definida por:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Esta definición es naturalmente recursiva, y podemos implementarla fácilmente de la siguiente forma:

```python
# Python
def fibonacci(n):
  if n < 2:
    return n
  return fibonacci(n - 1) + fibonacci(n - 2)
```

```java
// Java
int fibonacci(int n) {
  if (n < 2) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Con lo que hemos visto hasta ahora debiera ser fácil darse cuenta que ambos códigos son equivalentes.

Ahora, si bien la recursión puede ser una solución «elegante»,[13] requiere de más recursos que una solución iterativa ya que, como veremos más adelante, los llamados sucesivos a funciones comienzan a llenar la pila de ejecución y si no tenemos cuidado la podemos llenar (nuestro querido *stack overflow*).

Podemos implementar la función de Fibonacci de manera relativamente simple con el siguiente algoritmo:

```python
# Python
def fibonacci(n):
  a = 0
  b = 1
  for k in range(0, n + 1):
    c = b + a
    a = b
    b = c
  return a
```

```java
// Java
int fibonacci(int n) {
  int a = 0;
  int b = 1;
  for (int k = 0; k <= n; k++) {
    int c = b + a;
    a = b;
    b = c;
  }
```

[13] Al menos según Olmedo

```
  return a;
}
```

Aquí se comienza a notar un poco más la diferencia entre ambos lenguajes, en particular la sintaxis del ciclo `for`. Revisemos esto en detalle, el ciclo se define en base a 3 componentes: (1) un estado inicial (`initialize`), (2) una condición de repetición (`test`) y (3) una actualización del estado (`update`), con esto podríamos escribir un ciclo genérico como:

```
for (initialize; test; update) { ... }
```

En el ejemplo específico que vimos recién definimos el estado inicial de `k` como `int k = 0`. Luego, en cada iteración comprobamos la condición de repetición `k <= n`, esto quiere decir que si al final de una iteración se tiene que $k \leq n$, entonces se realizará otra pasada por el ciclo. El último componente, `update`, es una instrucción que se ejecuta *al final* de cada ciclo y que generalmente es una asignación (e.g. `i++`, `i += 2`, `i /= 10`). Todas las componentes del `for` son opcionales, lo que permite hacer un ciclo infinito con `for (;;)` (una muestra de maldad pura).

Veamos ahora un algoritmo más eficiente (y complicado) para generar la secuencia:[14]

```
# Python
def fibonacci(n):
  if n <= 0:
    return 0
  i = n - 1
  aux_one = 0
  aux_two = 1
  a, b = aux_two, aux_one
  c, d = aux_one, aux_two
  while i > 0:
    if i % 2 == 1:  # si `i` es impar
      aux_one = d * b + c * a
      aux_two = d * (b + a) + c * b
      a, b = aux_one, aux_two
    aux_one = c ** 2 + d ** 2
    aux_two = d * (2 * c + d)
    c, d = aux_one, aux_two
    i /= 2
  return a + b
```

```
// Java
int fibonacci(int n) {
  if (n <= 0) {
    return 0;
  }
  i = n - 1;
  auxOne = 0;
  auxTwo = 1;
  int a, b, c, d;
  a = c = auxOne;
  b = d = auxTwo;
  while (i > 0) {
    if (i % 2 == 1) {
      auxOne = d * b + c * a;
      auxTwo = d * (b + a) + c * b;
      a = auxOne;
```

[14] Tanto la versión recursiva como la iterativa con `for` son algoritmos con complejidad $O(n)$, mientras que utilizando un algoritmo del tipo *divide-and-conquer* se obtiene una complejidad de $O(\log(n))$.

```
      b = auxTwo;
    }
    auxOne = c * c + d * d;
    auxTwo = d * (2 * c + d);
    c = auxOne;
    d = auxTwo;
    i /= 2;
  }
  return a + b;
}
```

No entraremos en detalles sobre los segmentos de código recién mostrados ya que no es el enfoque del libro, pero es una buena idea leer ambos con detenimiento para ver las diferencias entre *Python* y *Java*.

Con esto ya nos familiarizamos con la sintáxis básica de *Java*, en el capítulo siguiente tomaremos un desvío y nos despediremos de *jshell*

uwu

pero no estén tristes, porque a partir de aquí es donde se pone bueno.

Ejercicio 1.8. Considere la función:

```
void countdown(int n) {
  System.out.println(n--);
}
```

¿Qué imprime?

¿Qué sucede si cambio la instrucción `n--` por `--n`? ¿Por qué?

1.3. Ejercicios

Ejercicio 1

Referencias

Evans y col.: Java in a Nutshell **java-nutshell-syntax-arrays**

Benjamin J. Evans y David Flanagan. *Java in a Nutshell*. Ed. por Mike Loukides y Meghan Blanchette. 6th ed. O'Reilly Media, Inc, 2015. Cap. 2. Java Syntax from the Ground Up, págs. 77-88.

Anot.: Arreglos y referencias.

Capítulo 2

¡Kotlin!

Este capítulo busca presentar *Kotlin* como una alternativa a *Java*, no es necesario, pero se recomienda leer el capítulo 1 antes de continuar este ya que muchos de los principios que se explican ahí también se cumplen para *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 ya voy a 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.

2.1. Instalando Kotlin

De la misma forma que hicimos con *Java*, lo primero que necesitaremos para trabajar en *Kotlin* es el compilador.

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).

2.1.1. *Windows*

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
```

2.1.2. *Linux*

Para instalar *Kotlin* en sistemas *Linux* se recomienda utilizar *SDKMAN!*[1]. Instalar *SDKMAN!* es relativamente simple, solo deben correr los comandos que presentamos a continuación y seguir los pasos que aparezcan en pantalla.

```
curl -s "https://get.sdkman.io" | bash
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:

[1] El «!» es parte del nombre

```
sdk install kotlin
```

Y comprobamos la instalación haciendo:

```
kotlinc -version
```

2.1.3. *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
```

2.2. 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 3

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]

3.1. 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* .

3.1.1. 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"
```

3.2. 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 3.1 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.

3.2.1. 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 3.2. Aquí deberán seleccionar la opción «*New Project*».

Luego deberíamos estar en la ventana principal de creación de proyectos (figura 3.3), aquí tendremos muchas opciones disponibles pero en este libro las ignoraremos y sólo le pondremos atención a proyectos de *Java* (y posteriormente

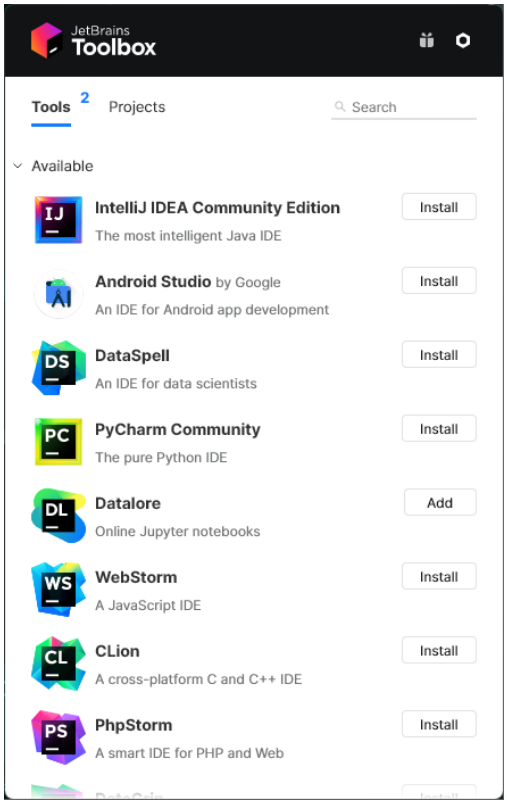


Figura 3.1: Interfaz de *JetBrains Toolbox*.

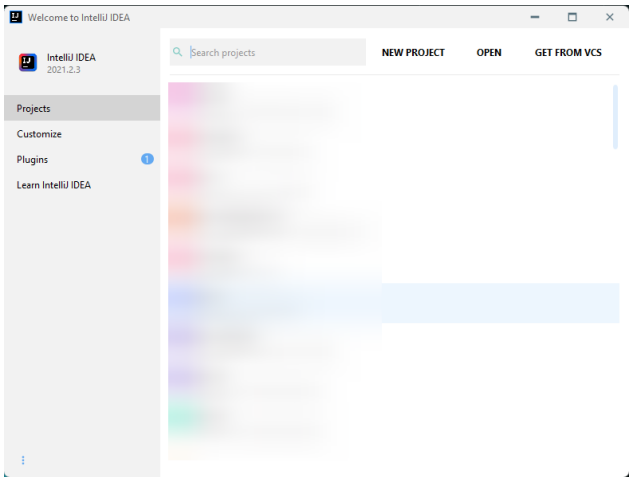


Figura 3.2: Ventana inicial de *IntelliJ*.

Gradle). Si quieren seguir los ejemplos en *Kotlin* deberán marcar la opción resaltada y seguir a la siguiente pantalla con «*Next*».

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 3.4).

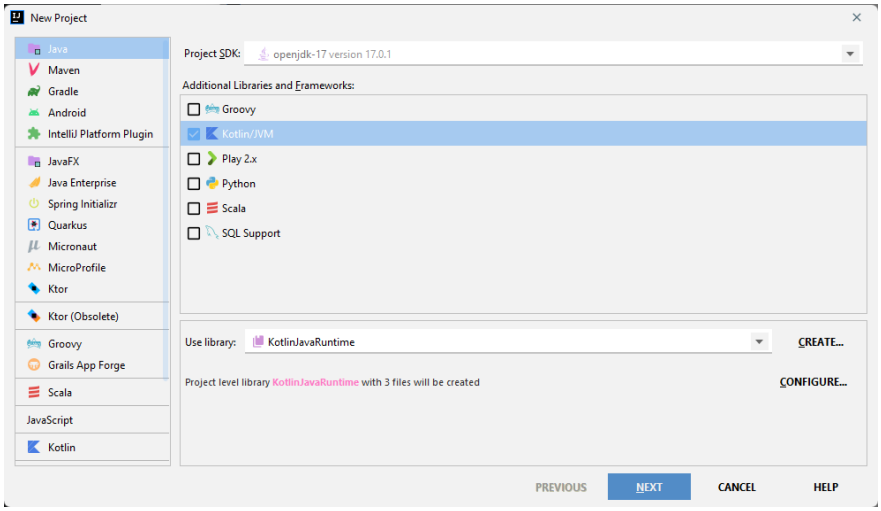


Figura 3.3: Ventana de creación de proyectos

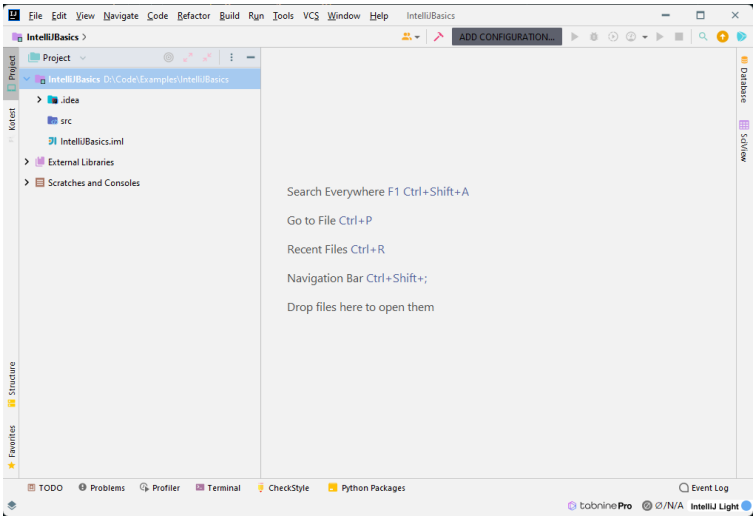


Figura 3.4: Pantalla principal de *IntelliJ*.

¡Y listo, tenemos nuestro primer proyecto!

3.2.2. Trabajando con *IntelliJ*

En el capítulo 1 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 3.5.

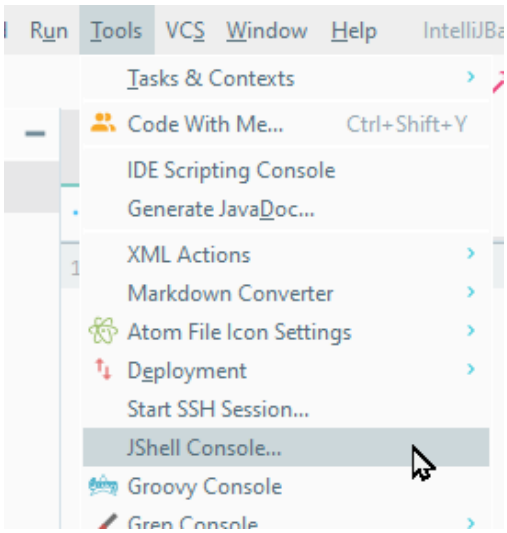


Figura 3.5: Acceso a la consola *JShell*.

Search Everywere

Otra forma de abrir la consola de *Jshell* es abriendo el menú *Search Everywhere*[a] (ver figura 3.6), y en la barra de búsqueda escribir textttjshell.
Esta herramienta nos ayudará mucho a conocer y usar las herramientas de *IntelliJ*.

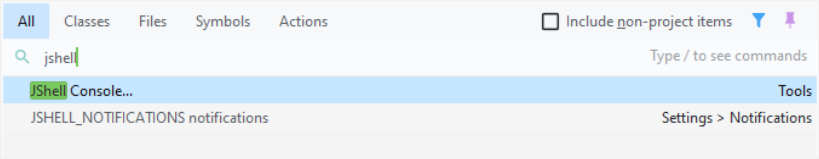


Figura 3.6: Resultado de *Search Everywhere* para la búsqueda de `jshell`.

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

Ahora intenten reescribir el código de *Java* del capítulo 1 para ver cómo se siente trabajar con un *IDE*. Deberían tener algo como en la figura 3.7. 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.

3.3. 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.

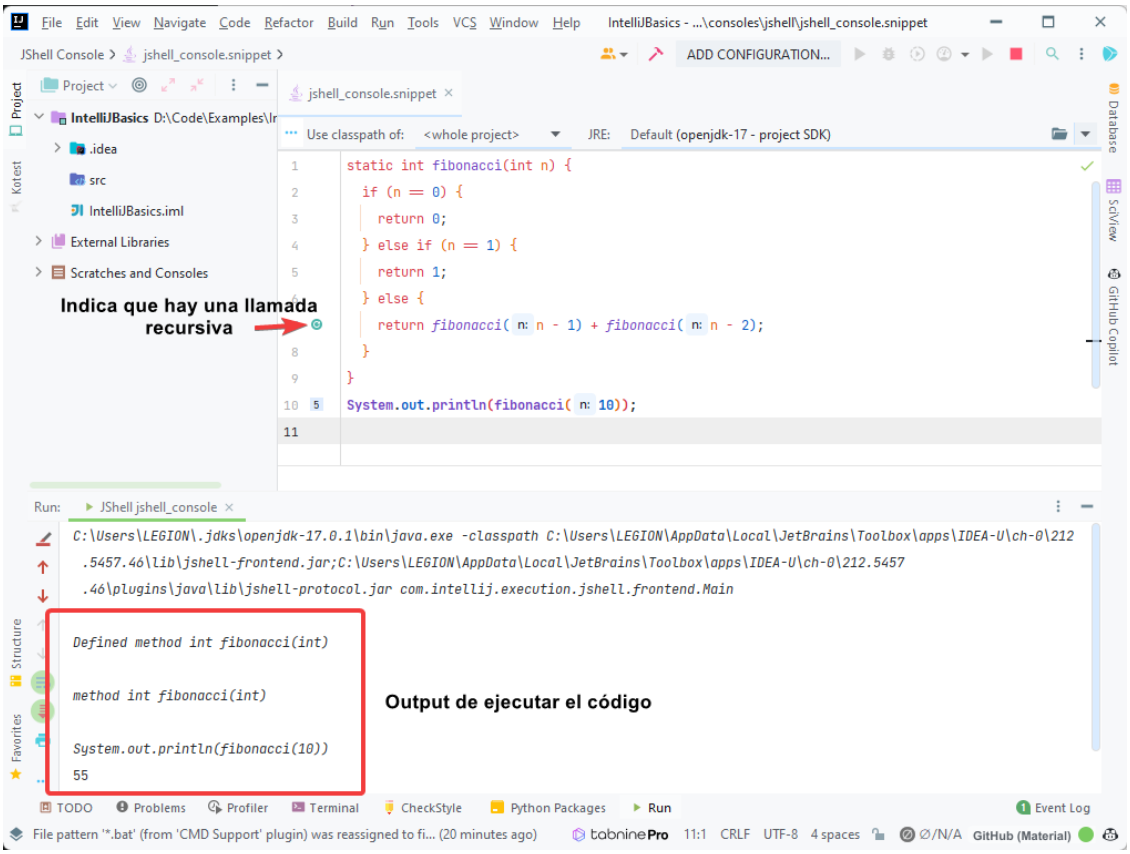


Figura 3.7: Ejemplo de uso de *JShell* en *IntelliJ*.

- `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`.

3.3.1. 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

[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.

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

3.3.2. Mi primer *Java*

3.3.3. Mi primer *Kotlin*

Referencias

JetBrains: IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains **jb-idea**

JetBrains. *IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains*. URL: https://www.jetbrains.com/idea/ (visitado 25-12-2021).

JetBrains: JetBrains Toolbox App: Manage Your Tools with Ease **jb-toolbox**

JetBrains. *JetBrains Toolbox App: Manage Your Tools with Ease*. URL: https://www.jetbrains.com/toolbox-app/ (visitado 25-12-2021).

Capítulo 4

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.

4.1. Instalando *Git*

4.1.1. *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.

4.1.2. 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
```

4.1.3. 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
```

4.2. ¿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*.

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

```
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:

Powershell

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

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*.

4.2.1. *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:

```
# 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:

```
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"
```

Referencias

Chacon y col.: Git **git-scm**

Scott Chacon, Jason Long y GitHub Community. *Git*. URL: https://git-scm.com (visitado 20-04-2020).

Anot.: Documentación oficial de Git.

[2] GitHub, *Basic writing and formatting syntax*.

Chacon y col.: gitignore **gitignore**

Scott Chacon, Jason Long y GitHub Community. *Git - gitignore Documentation*. URL: https://git-scm.com/docs/gitignore (visitado 20-04-2020).

Dudler: git - the simple guide **git-simple**

Roger Dudler. *git - the simple guide - no deep shit!* URL: https://rogerdudler.github.io/git-guide/ (visitado 20-04-2020).

GitHub: Basic writing and formatting syntax **gh-markdown**

GitHub. *Basic writing and formatting syntax*. URL: https://docs.github.com/en/github/writing-on-github/basic-writing-and-formatting-syntax (visitado 16-09-2020).

GitHub: Documenting your projects on GitHub **gh-wikis**

GitHub. *Documenting your projects on GitHub*. 2016. URL: https://guides.github.com/features/wikis/ (visitado 06-09-2020).

GitHub: Forking Projects **gh-forks**

GitHub. *Forking Projects*. 2017. URL: https://guides.github.com/activities/forking/ (visitado 06-09-2020).

GitHub: Git Handbook **git-handbook**

GitHub. *Git Handbook - GitHub Guides*. 2017. URL: https://guides.github.com/introduction/git-handbook/.

Resumen: Guía de los comandos básicos de *Git* y cómo se relacionan con *GitHub*.

GitHub: GitHub Training & Guides **gh-youtube**

GitHub. *GitHub Training & Guides*. URL: https://www.youtube.com/githubguides (visitado 06-09-2020).

GitHub: Mastering Issues **gh-issues**

GitHub. *Mastering Issues*. 2020. URL: https://guides.github.com/features/issues/ (visitado 06-09-2020).

GitHub: Mastering Markdown **markdown**

GitHub. *Mastering Markdown - GitHub Guides*. 15 de ene. de 2014. URL: https://guides.github.com/features/mastering-markdown/.

GitHub: Understanding the GitHub flow **gitflow**

GitHub. *Understanding the GitHub flow - GitHub Guides*. 30 de nov. de 2017. URL: https://guides.github.com/introduction/flow/ (visitado 24-04-2020).

Anot.: Les recomendamos encarecidamente que utilicen esta modalidad al trabajar individualmente y, en especial, en las tareas de este ramo.

Resumen: La modalidad (flujo) de trabajo recomendada por *GitHub* para trabajar en equipos.

Github: Hello World **github-hello-world**

Github. *Hello World - GitHub Guides*. URL: https://guides.github.com/activities/hello-world/ (visitado 20-04-2020).

Resumen: Tutorial básico de cómo crear y manejar un repositorio en *GitHub*.

GitKraken **gitkraken**

GitKraken. URL: https://www.gitkraken.com (visitado 06-09-2020).

Anot.: ***GitKraken*:** interfaz gráfica para manejar repositorios de *Git*. Pueden obtener una licencia gratis postulando a los beneficios de *GitHub Education*.

Kehoe: How to Install Xcode Command Line Tools on a Mac **xcode-clt**

Daniel Kehoe. *How to Install Xcode Command Line Tools on a Mac*. 19 de jul. de 2021. URL: https://www.freecodecamp.org/news/install-xcode-command-line-tools/ (visitado 02-01-2022).

LaunchCode: Using Git in IntelliJ **yt-git-intellij**

LaunchCode. *Using Git in IntelliJ*. 2017. URL: https://www.youtube.com/watch?v=uUzRMOCBorg (visitado 06-09-2020).

mercurial **mercurial**

mercurial. URL: https://www.mercurial-scm.org (visitado 06-09-2020).

Anot.: Alternativa a *Git* que también es ampliamente usada. *SourceForge* es uno de los ejemplos más importantes de *host* de repositorios de *Mercurial* (el equivalente a *GitHub*).

Microsoft: Using Version Control in VS Code **vscode-git**

Microsoft. *Using Version Control in VS Code*. 2020. URL: https://code.visualstudio.com/docs/editor/versioncontrol (visitado 06-09-2020).

Index