

The background of the entire cover is a close-up, high-resolution photograph of dark brown, roasted coffee beans. The beans are densely packed and show natural variations in color and texture, with some appearing slightly lighter and others darker. A semi-transparent dark teal band runs horizontally across the middle of the image, serving as a backdrop for the text.

CHARLY CIMINO

# PROGRAMACIÓN ESTRUCTURADA CON JAVA

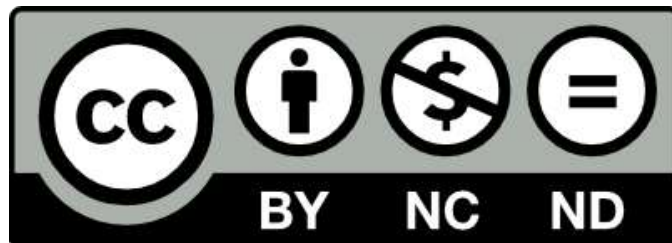
UN PRIMER ACERCAMIENTO AL LENGUAJE

# Programación estructurada con Java

Charly Cimino



Este documento se encuentra bajo Licencia Creative Commons 4.0 Internacional (CC BY-NC-ND 4.0).



Usted es libre para:

- **Compartir** — copiar y redistribuir el material en cualquier medio o formato.

Bajo los siguientes términos:

- **Atribución** — Usted debe darle crédito a esta obra de manera adecuada, proporcionando un enlace a la licencia, e indicando si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo del licenciante.
- **No Comercial** — Usted no puede hacer uso del material con fines comerciales.
- **Sin Derivar** — Si usted mezcla, transforma o crea nuevo material a partir de esta obra, usted no podrá distribuir el material modificado.

Versión 2023-03-17

## Tabla de contenido

|  |    |
|--|----|
| Introducción .....                         | 6  |
| Acerca de Java .....                       | 7  |
| Conceptos básicos.....                     | 7  |
| Breve historia.....                        | 7  |
| Arquitectura .....                         | 7  |
| Instalación .....                          | 8  |
| Primeros pasos .....                       | 9  |
| Escribiendo tu primer programa .....       | 9  |
| Compilando tu primer programa .....        | 10 |
| Ejecutando tu primer programa .....        | 12 |
| Creación de un proyecto en NetBeans .....  | 13 |
| Ejecución de un proyecto en NetBeans ..... | 14 |
| Guardar un proyecto de NetBeans .....      | 15 |
| Trasladar un proyecto de NetBeans .....    | 15 |
| Cerrar todos los proyectos abiertos .....  | 16 |
| Abrir un proyecto de NetBeans.....         | 16 |
| Flujo secuencial .....                     | 18 |
| Comentarios .....                          | 19 |
| Instrucción de salida .....                | 19 |
| Método println() .....                     | 19 |
| Método print().....                        | 20 |
| Caracteres de escape .....                 | 21 |
| Tipos de datos primitivos.....             | 22 |
| El tipo de dato String .....               | 23 |
| Expresiones aritméticas.....               | 23 |
| Operadores aritméticos .....               | 23 |
| Operaciones con enteros.....               | 24 |
| Variables .....                            | 25 |
| Definición .....                           | 25 |
| Reglas de nomenclatura .....               | 25 |
| Declaración .....                          | 25 |
| Operador de asignación .....               | 26 |
| Operador de concatenación .....            | 27 |
| Convención de los tipos de datos.....      | 28 |



|   |    |
|---|----|
| Actualizar una variable .....                 | 28 |
| Operadores incrementales .....                | 29 |
| Otros operadores de asignación .....          | 29 |
| Promoción de tipos (casting) .....            | 30 |
| Promoción por ensanchamiento .....            | 30 |
| Promoción por estrechamiento.....             | 31 |
| Promociones posibles .....                    | 31 |
| Casos de ambigüedad.....                      | 32 |
| Instrucción de entrada.....                   | 32 |
| La clase Scanner .....                        | 32 |
| Leer datos numéricos .....                    | 33 |
| Leer datos alfanuméricos con next().....      | 34 |
| Leer datos alfanuméricos con nextLine() ..... | 35 |
| El problema con nextLine().....               | 35 |
| Uso de funciones.....                         | 37 |
| Abstracción .....                             | 37 |
| La clase Math.....                            | 37 |
| Flujo de selección.....                       | 40 |
| Expresiones booleanas .....                   | 40 |
| Operadores relacionales.....                  | 40 |
| Bloque de selección simple if .....           | 41 |
| Bloque de selección doble if-else .....       | 42 |
| Bloques de selección anidados .....           | 44 |
| Primera forma: Bloques dentro de otros .....  | 44 |
| Segunda forma: Bloques anidados .....         | 46 |
| Operadores lógicos.....                       | 47 |
| Operador NOT.....                             | 47 |
| Operador OR.....                              | 48 |
| Operador AND .....                            | 48 |
| Ejemplo de aplicación.....                    | 49 |
| Jerarquía de operadores .....                 | 50 |
| Bloque de selección múltiple switch .....     | 50 |
| Flujo de repetición.....                      | 54 |
| Bloque de repetición for .....                | 54 |
| Bloque de repetición while.....               | 56 |
| Bloque de repetición do..while .....          | 57 |

|  |    |
|--|----|
| Ámbito de las variables.....                           | 59 |
| Modularización.....                                    | 61 |
| Procedimientos .....                                   | 61 |
| Definición de un procedimiento .....                   | 62 |
| Invocación de un procedimiento .....                   | 64 |
| Procedimiento con parámetros.....                      | 66 |
| Funciones.....   | 69 |
| Definición e invocación de una función .....           | 69 |
| Variables locales .....                                | 71 |
| Diferencia entre procedimiento, función y método ..... | 72 |
| Acerca de main .....                                   | 73 |
| Pasaje de parámetros por valor .....                   | 74 |
| Cuestiones de diseño .....                             | 75 |
| Cohesión .....   | 75 |
| Acoplamiento.....                                      | 76 |
| Arreglos.....  | 78 |
| Declaración .....                                      | 78 |
| Sin inicializar .....                                  | 78 |
| Inicializado por defecto .....                         | 79 |
| Inicializado con valores.....                          | 80 |
| Operaciones básicas .....                              | 80 |
| Obtener un valor.....                                  | 80 |
| Obtener longitud del arreglo.....                      | 81 |
| Obtener todos los valores .....                        | 82 |
| Establecer un valor.....                               | 83 |
| Establecer todos los valores .....                     | 84 |
| Arreglos como argumentos.....                          | 85 |
| Cadenas de caracteres.....                             | 88 |
| Breve introducción al concepto de clase y objeto ..... | 88 |
| La clase String .....                                  | 88 |
| Concatenar cadenas .....                               | 89 |
| Comparar cadenas .....                                 | 89 |
| Procesar cadenas.....                                  | 90 |
| Conocer la longitud de una cadena .....                | 90 |
| Obtener un caracter de una cadena.....                 | 91 |
| Extraer una porción de la cadena.....                  | 91 |

|  |    |
|--|----|
| Otras operaciones .....                    | 91 |
| Conversiones .....                         | 93 |
| Clases envoltorio (wrapper) .....          | 93 |
| Convertir un primitivo en una cadena ..... | 94 |
| Convertir una cadena en un primitivo ..... | 94 |
| Tabla de ilustraciones .....               | 95 |
| Tabla de códigos .....                     | 96 |

Charly Cimino

## Introducción

Lo primero que uno tiende a pensar cuando se topa con un libro de programación estructurada con Java es que el autor no tiene idea de lo que está haciendo. **Java es un lenguaje de programación orientada a objetos**, es decir, persigue otro paradigma.

El motivo de esta publicación es abordar los fundamentos de la programación estructurada mediante la plataforma Java, porque así está previsto en el temario de muchos cursos o asignaturas: se intenta enseñar a programar desde cero (entrada/salida, condicionales, bucles, funciones) usando a Java. Yo no lo recomendaría, pero es una realidad.

Si Java es un lenguaje orientado a objetos, ¿cómo se comportará ante el paradigma estructurado? La realidad es que, en determinados momentos, lo forzaré a comportarse como un lenguaje estructurado, como digo, "a lo C" (en referencia al lenguaje C).

En este libro mostraré conceptos básicos como entrada y salida por consola, estructuras de selección, estructuras de repetición y operadores aritméticos, relacionales y lógicos. No haré un tratamiento profundo sobre el tema, pues para ello existe otra publicación de mi autoría llamada **INTRODUCCION A LA PROGRAMACION**, donde abordo los fundamentos de la programación de manera genérica con la ayuda de una herramienta llamada PSeInt.

Además, te daré a conocer otros temas como definir e invocar tus propias **funciones** junto a una noción breve del concepto de **recursividad**, el tratamiento de una de las primeras y más básicas estructuras de datos llamadas **arreglos**, con sus algoritmos de **búsqueda** y **ordenamiento**, y, por último, a manejar **cadenas de caracteres**, donde indefectiblemente realizarás tus primeras operaciones con **objetos**.

No te preocupes si aparecen palabras que no comprendés del todo, recordá que estás utilizando el lenguaje de una forma para la cual no fue pensado. Es probable que tengas que toparte con objetos más de una vez. Hay cosas que quizá explique al pasar para que no te quedes con la intriga y otras donde con toda sinceridad te diré: "Esto tendrá sentido cuando sepas sobre objetos, por ahora hacelo así y punto".

Si ya tenés idea de los fundamentos de programación y la sintaxis básica de Java y querés entender el paradigma orientado a objetos, te dejo este enlace a mi playlist de Youtube sobre el tema: <https://www.youtube.com/playlist?list=PLOw7b-NX043aSC7ZNtEuVfY8xZoNzVqdJ>

Hechas las aclaraciones pertinentes, es momento de comenzar a preparar lo necesario para realizar con éxito los algoritmos descritos en el libro.

**¡Te doy la bienvenida!**

# Acerca de Java

## Conceptos básicos

Java es un lenguaje de programación de propósito general, orientado a objetos.

La intención de este es lograr una independencia de la plataforma de ejecución, es decir, que los programadores desarrollen un programa por única vez pudiendo ejecutarse en múltiples dispositivos con distinta arquitectura. En inglés, se conoce como **WORA**, de "Write Once, Run Anywhere" (*Escribilo una vez, correlo donde sea*). Con esto resuelve el problema de la portabilidad que presentan muchos lenguajes, donde la ejecución de determinado programa en dispositivos diferentes requiere que el código se recompile.

Desde hace unos años, Java es uno de los lenguajes de programación más populares, en especial para aplicaciones web cliente-servidor. Está basado en el lenguaje **C++**, y, por ende, en **C**, aunque tiene menos instrucciones de bajo nivel que ellos (en Java, por ejemplo, no existe el acceso directo a punteros).

## Breve historia

En 1991, un equipo dirigido por James Gosling, miembro de Sun Microsystems (adquirida en 2009 por la empresa Oracle), comenzó el desarrollo de Java, que finalmente vio la luz en 1995.

La filosofía del lenguaje radicaba en cumplir con las siguientes premisas:

- Utilizar el paradigma orientado a objetos.
- Ejecutar un mismo programa en diferentes sistemas operativos.
- Fácil de usar.
- Tomar lo mejor de otros lenguajes orientados a objetos, como **C++**.

El proyecto inicialmente se llamó **Oak**, en referencia a un roble que había fuera de la oficina de Gosling, pero al ver que tal nombre era una marca registrada, paso a llamarse **Green**.

Finalmente adquirió el nombre **Java**, cuyo origen no se sabe del todo, aunque hay ciertas hipótesis, como las iniciales de los diseñadores o un acrónimo cuyo resultado es "*Just Another Vague Acronym*" (*Sólo otro acrónimo ambiguo*). La más aceptada es la que aboga por que el nombre fue tomado de un tipo de café disponible en un bar cercano, de allí que el ícono de Java sea una taza de café caliente.

Desde la versión **1.0**, Java ha sufrido diversos cambios y agregado más características. Actualmente, la última versión es la **10**.

## Arquitectura

Alguna vez te habrás topado con alguna página web cuyo contenido estuviera basado en Java, por ejemplo, un chat o un juego. El requisito para ejecutar dicha aplicación es que "descargues Java", es decir un agregado externo al navegador (en inglés, "**plugin**") que lo permita.

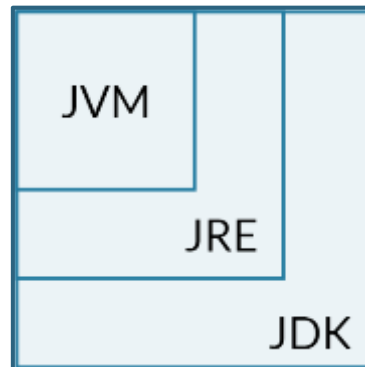
Para formalizar un poco las cosas, lo que un usuario promedio hace para correr programas en Java es descargar la **JRE**, de "Java Runtime Environment" (entorno de ejecución de Java). Este entorno incluye los componentes mínimos necesarios para ejecutar programas desarrollados en Java,



donde, entre otras utilidades, se encuentra la **JVM**, de "Java Virtual Machine" (máquina virtual de Java).

Cada sistema operativo (Windows, Linux, Mac, entre otros) tiene su propia implementación de la **JVM**, habiendo una abstracción con el programa Java y, por ende, permitiendo la ejecución de este en cualquiera de ellos.

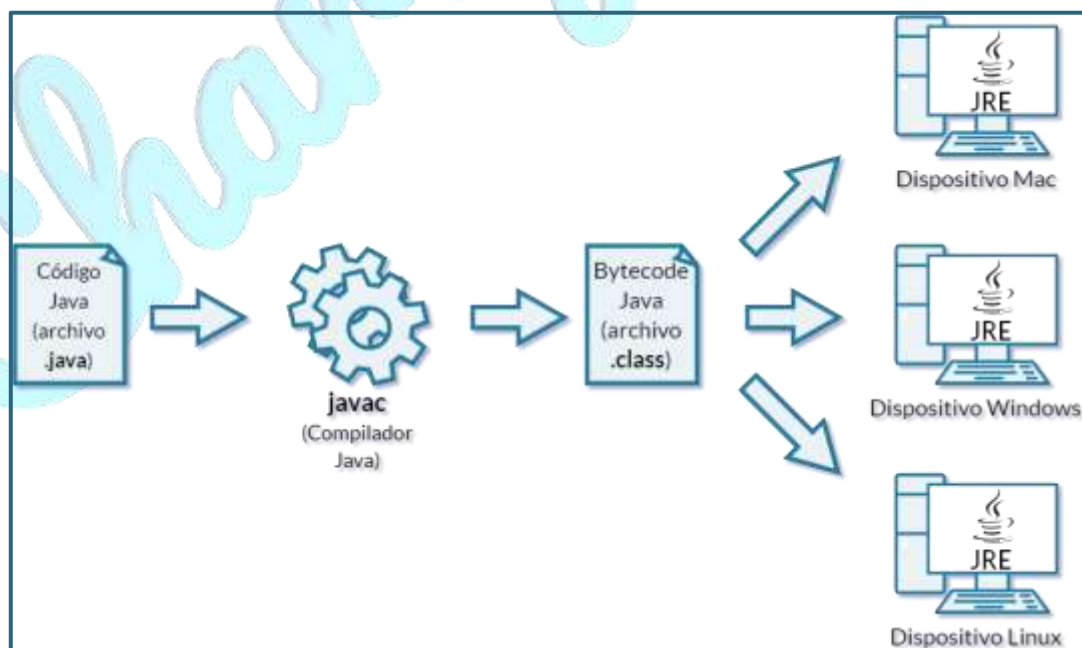
Los programadores no se bastan con correr programas en Java, sino que su tarea es desarrollarlos. Para ello, deben obtener herramientas adicionales que se empaquetan y se descargan bajo el nombre de **JDK**, de "Java Development Kit" (kit de desarrollo de Java). El **JDK** incluye al **JRE** y agrega herramientas para el desarrollo de aplicaciones.



**Ilustración 1: Diferencia entre JVM, JRE y JDK.**

Básicamente, se escribe código en Java en un archivo cuya extensión será **.java**. Una vez que el código está listo, se lo compila mediante un compilador llamado **javac**, a un código máquina intermedio denominado **bytecode**. Este bytecode, cuya extensión es **.class**, será interpretado y ejecutado por la JVM correspondiente al sistema operativo donde se encuentre.

Si alguna vez te topás con un archivo de extensión **.jar**, se trata de un **Java Archive** (archivo Java), que en realidad es un archivo **.zip** con la extensión modificada. La única función de los **.jar** es agrupar archivos **.class**. Los creadores de Java intentaron hacer coincidir la extensión con la palabra inglesa "jar" (tarro).



**Ilustración 2: Conversión de un código a un programa**

## Instalación

En este apartado mostraré los pasos necesarios para la instalación de Java, bajo el sistema operativo **Windows**, debido a que es el más popular. De todas maneras, la instalación en **Linux**, **Mac** u otros sistemas operativos es muy similar.

De manera de tener siempre la versión más actual, te compartiré el video donde enseño como instalar el JDK y NetBeans, a través de este enlace: <https://youtu.be/2Et13pH2484>

## Primeros pasos

Antes de crear tu primer programa en Java utilizando la gran ayuda de NetBeans, primero quiero mostrarte qué pasos tendrías que realizar en caso de no contar con ningún IDE. No es necesario que los realices. La idea es mostrar la conveniencia de usar entornos de desarrollo integrados.

## Escribiendo tu primer programa

En primer lugar, necesitás un editor de texto. Con el bloc de notas que trae Windows es suficiente. Escribí en el editor el siguiente código tal cual se describe:

```
1 public class HolaMundo {
2     public static void main(String[] args) {
3         System.out.println("Hola Mundo!");
4     }
5 }
```

### Código 1: Primer programa en Java.

Lo único que hace este programa es mostrarle un mensaje al usuario a través de la consola. La instrucción que lo permite es:

```
System.out.println("Hola Mundo!");
```

Más adelante verás en detalle cómo hacer salidas por consola. Lo demás no tiene sentido explicarlo aún.

El siguiente paso es guardar el archivo de texto. Para ello, seleccioná la opción de "**Guardar**" (en inglés, "**Save**") en el editor que estés utilizando. Es importante que tengas en cuenta lo siguiente:

- El nombre del archivo debe ser lo que en el código aparece luego de la palabra **class** (la llave de apertura no se tiene en cuenta). Por lo tanto, el nombre debe ser **HolaMundo**. La extensión debe ser **.java**.

- El directorio donde se guarda este archivo debe ser fácil de acceder, pues luego tendrás que direccionar la consola hacia ese lugar. En este caso voy a elegir a la carpeta **"Escritorio"** (en inglés, **"Desktop"**).

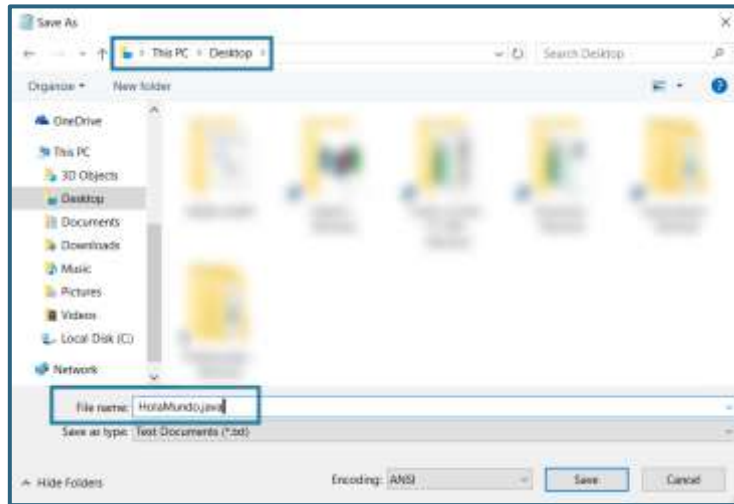


Ilustración 3: Guardando el código Java.

## Compilando tu primer programa

Lo siguiente ahora sería compilar el código. Pero antes, si estás en Windows, tenés que configurar la variable de entorno Path. Este libro no trata sobre sistemas operativos, pero para darte una idea, debés especificarle a Windows por única vez en qué directorio buscar el compilador de Java para no tener que direccionar la consola hacia el lugar donde se encuentra instalado el compilador.

Para configurar la variable de entorno Path en Windows 10, debés ingresar al **"Panel de control"**, elegir **"Sistema y seguridad"**, y luego hacer clic en **"Sistema"**. Ubicá a la izquierda la leyenda **"Configuración avanzada del sistema"** y hacé click allí. Tocá el botón **"Variables de entorno..."**. En el primer cuadro, hacé doble click en la variable **"Path"**.

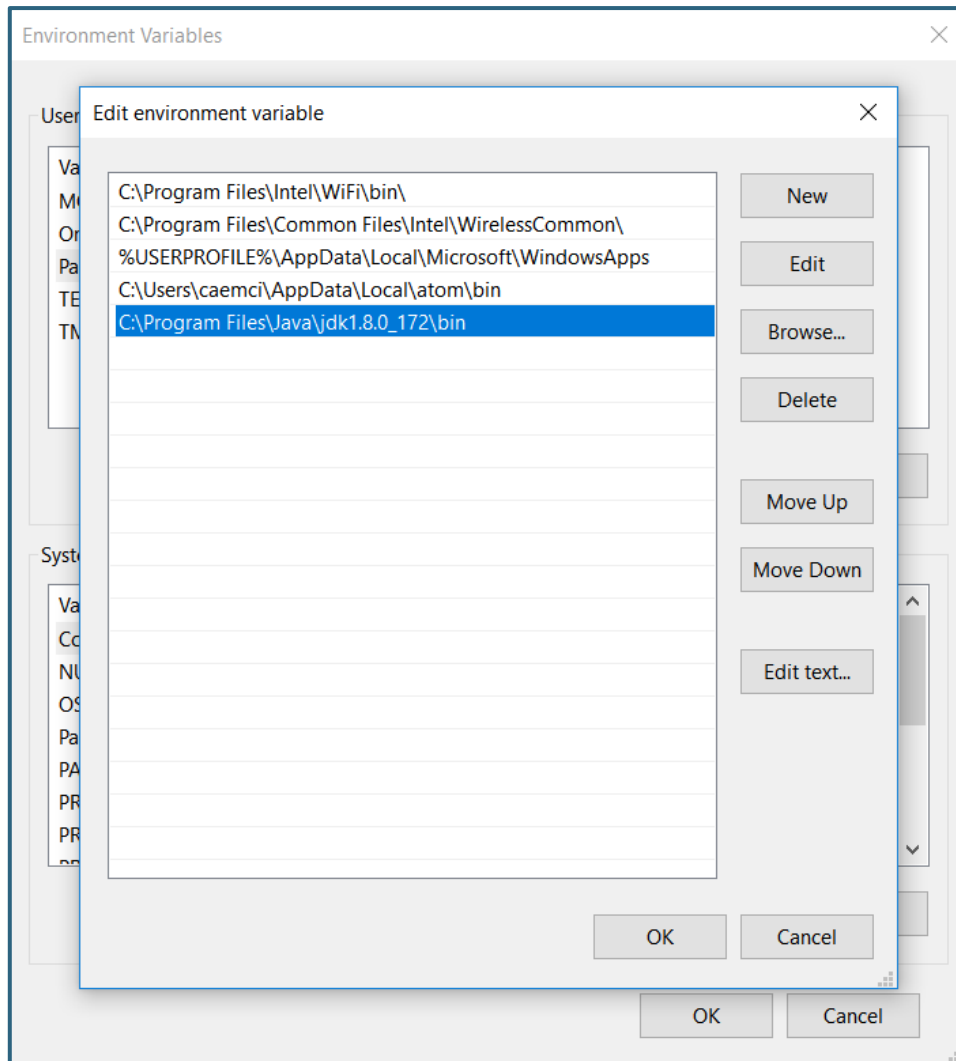
Se abre una nueva ventana con una lista de directorios donde por defecto Windows busca cuando se ejecuta un programa en la consola. Tocá el botón **"Nuevo"**.

La nueva ruta será la **dirección donde se encuentra instalada la carpeta "bin" del JDK**. Se supone que dejaste por defecto el directorio cuando instalaste el JDK, por lo tanto, la ruta es:

**C:\Program Files\Java\jdk1.8.0\_172\bin**

Lo que está marcado en rojo puede variar según la versión del JDK instalada. Revisalo en tu computadora.

El cuadro debería quedarte así:



**Ilustración 4: Modificando la variable de entorno Path.**

Aceptá todas las ventanas. Ya está todo listo para compilar tu primer programa.

Lo siguiente es abrir una consola. En Windows 10, la manera más fácil es buscar "cmd" o ir al menú "Inicio", ubicar la carpeta "Sistema de Windows" y seleccionar "Símbolo del sistema".

La consola se direcciona por defecto siempre en la carpeta del usuario, en mi caso:

**C:\Users\caemci**

Lo que está marcado en rojo puede variar según tu nombre de usuario.

Debés direccionar la consola en la carpeta "Escritorio", y, como la misma se encuentra dentro de la carpeta de usuario, podés hacer referencia directamente a ella. Recordá que tu archivo Java está en "Escritorio". Para direccionar la consola, se usa la instrucción **cd**. Ingresá lo siguiente y luego presioná **Enter**:

**cd Escritorio**

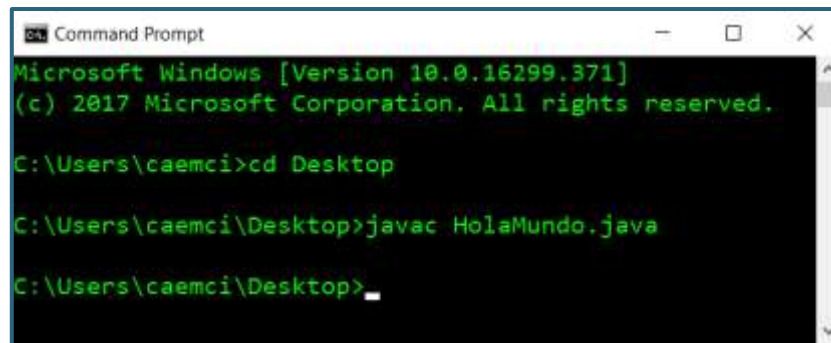
La consola ya está direccionada hacia el lugar donde se encuentra el archivo. El último paso es invocar al compilador de Java, llamado **javac** e indicarle el nombre del archivo a compilar. Tu archivo se llama **HolaMundo.java**, por lo tanto, ingresá lo siguiente y luego presioná **Enter**:

**javac HolaMundo.java**



Si no hay errores de compilación, simplemente no se muestra nada, de lo contrario, el compilador muestra por consola un mensaje de error.

La siguiente imagen ilustra los pasos que realizaste (utilizo Windows en inglés, por lo que puede lucir algo diferente):



```
Microsoft Windows [Version 10.0.16299.371]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\caemci>cd Desktop

C:\Users\caemci\Desktop>javac HolaMundo.java

C:\Users\caemci\Desktop>_
```

Ilustración 5: Compilando un código Java.

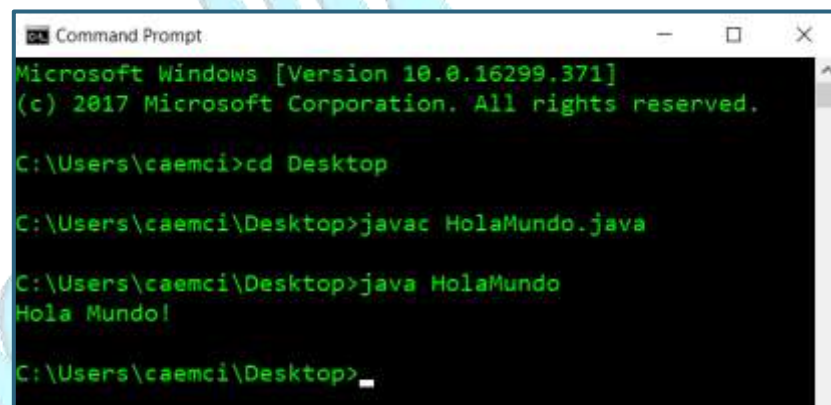
## Ejecutando tu primer programa

Tras haber compilado el código, fíjate que en **Escritorio** se acaba de crear un nuevo archivo, cuyo nombre es el mismo (**HolaMundo**) pero con extensión **.class**. Se trata del programa traducido a **bytecode**, el cual debe ser **interpretado por la JVM**.

Espero no hayas cerrado la consola. Lo único que te queda por hacer es ejecutar tu programa. Ingresá lo siguiente y luego presioná **Enter**:

**java HolaMundo**

Si todo va bien, el programa se ejecuta y muestra el mensaje en la consola:



```
Microsoft Windows [Version 10.0.16299.371]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\caemci>cd Desktop

C:\Users\caemci\Desktop>javac HolaMundo.java

C:\Users\caemci\Desktop>java HolaMundo
Hola Mundo!

C:\Users\caemci\Desktop>_
```

Ilustración 6: Ejecutando un código Java.

La configuración de la variable de entorno Path en Windows no se realiza cada vez que se deba compilar. Solo por única vez al principio.

Habrás visto que es bastante engorroso tener que escribir un código en un editor no preparado para programar como el Bloc de notas. Por cada cambio, guardar el archivo, abrir la consola, redireccionarla, compilar y ejecutar. Demasiados pasos, ¿verdad?

Para facilitar este proceso, te hice instalar **NetBeans**.

## Creación de un proyecto en NetBeans

Había mencionado anteriormente que un programa en Java no consta únicamente de un solo archivo con código sino también de librerías, clases auxiliares, metadatos y otros recursos. En este libro de todas maneras, se harán códigos sencillos, que bastan con un único archivo, ya que no se explica el paradigma orientado a objetos.

Aun así, para realizar un simple programa que muestre una salida por consola como lo hiciste antes, se requiere previamente crear un proyecto en NetBeans.

Abrió el menú **"File"** y clickeá la opción **"New Project"** (nuevo proyecto) o el botón  para comenzar la creación:

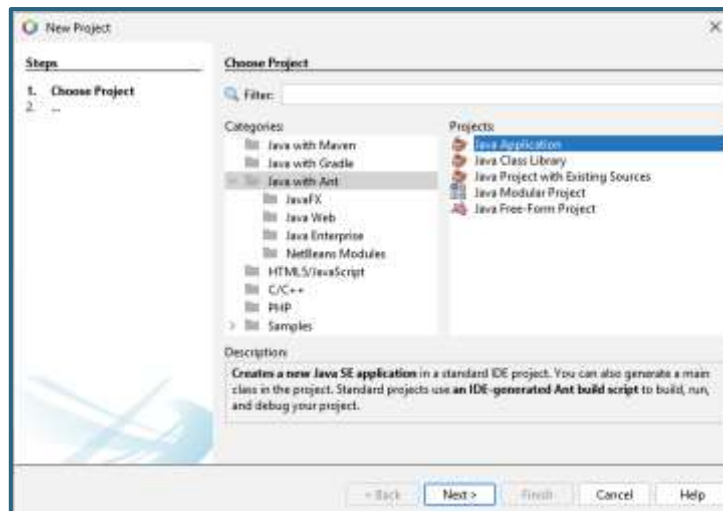


Ilustración 7: Creando un proyecto en NetBeans.

Asegurate que esté seleccionada la opción **"Java Application"** de la categoría **"Java with Ant"** y tocá el botón **"Next"**.

A continuación, se muestra la ventana de configuración del proyecto:

En el cuadro **"Project Name"**, escribí un nombre para el proyecto **que no lleve espacios**.

En el cuadro **"Project Location"**, se muestra el directorio donde se guardará el nuevo proyecto. Por defecto, NetBeans guarda los proyectos en la carpeta **"NetBeansProjects"** dentro de la carpeta **"Documentos"** del usuario. Tocá el botón **"Finish"** para crear el proyecto

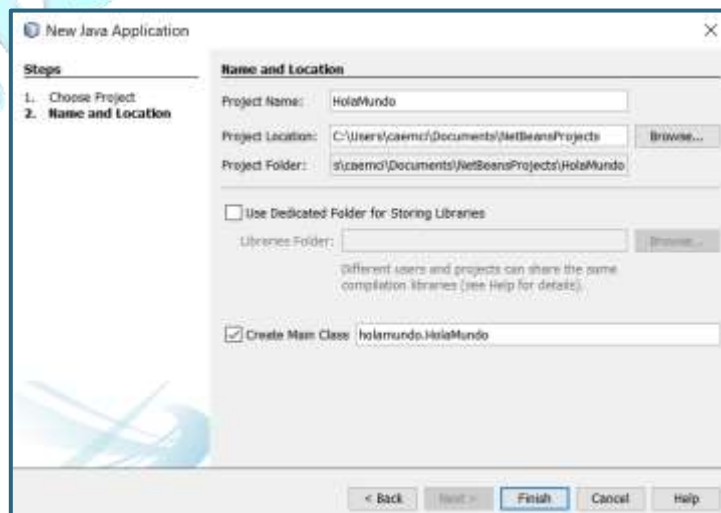


Ilustración 8: Configurando el nuevo proyecto en NetBeans.

## Ejecución de un proyecto en NetBeans

El IDE acaba de crear un proyecto llamado **HolaMundo** que además de algunas cosas que no merecen analizarse, cuenta con un archivo llamado igual que el proyecto, cuya extensión es **.java**.

En el panel izquierdo se muestran los archivos que componen el proyecto y en el cuadro grande de la derecha se observa el editor de código.

NetBeans abre por defecto el archivo **HolaMundo.java** que ya cuenta con el "esqueleto" básico para comenzar a programar. Las líneas que aparecen coloreadas en gris son comentarios. Podés borrarlos o dejarlos, da igual.

Dentro del par de llaves de más dentro simplemente escribí la instrucción para mostrar una salida por consola:

```
System.out.println("Hola Mundo!");
```

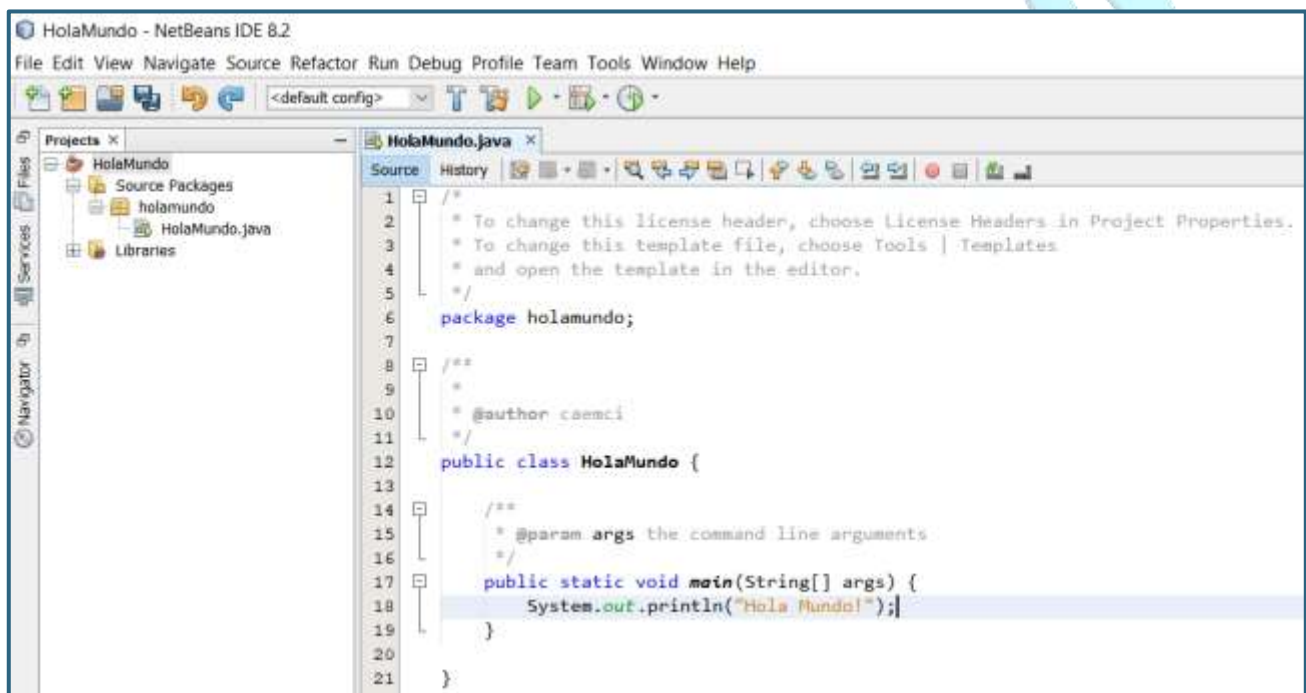



Ilustración 9: Escribiendo código Java en NetBeans.

Para compilar y ejecutar tu programa, ubicá el menú "Run" y tocá "Run Project" o el botón .

Si todo va bien, el propio NetBeans mostrará los resultados a través de su consola integrada. El mensaje "BUILD SUCCESSFUL" indica que la compilación del programa fue satisfactoria.

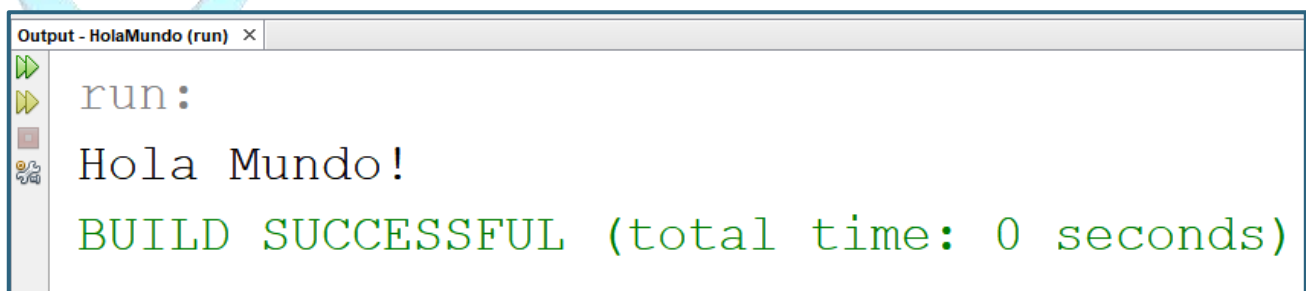



Ilustración 10: Resultado del programa en NetBeans.

## Guardar un proyecto de NetBeans

Cada vez que se ejecuta un programa en NetBeans con el botón , los archivos que componen el proyecto se guardan automáticamente.

Si se desean guardar los cambios, sin ejecutar el programa, abrí el menú **"File"** y clickeá la opción **"Save All"** (guardar todo) o el botón .

## Trasladar un proyecto de NetBeans

Los proyectos de NetBeans se guardan por defecto en la carpeta **"NetBeansProjects"**, dentro de **"Documentos"**. Si querés estar seguro, podés fijarte en las propiedades del proyecto y ver la ruta de este, haciendo **click derecho sobre el nombre del proyecto** y eligiendo la opción **"Properties"** (propiedades), de la siguiente manera:

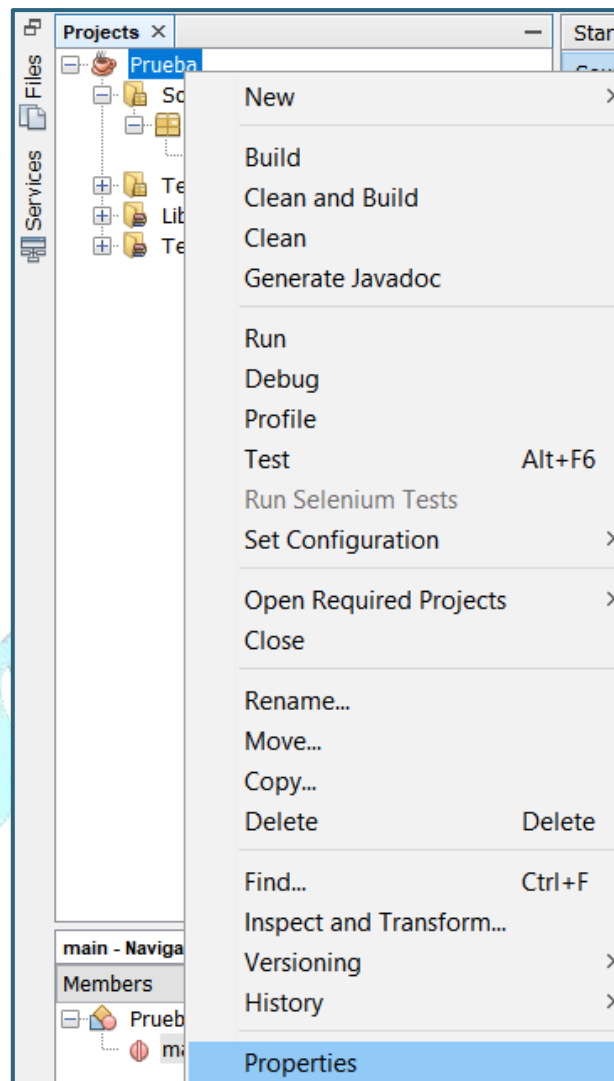


Ilustración 11: Accediendo a las propiedades del proyecto.



En el cuadro "Project Folder" (carpeta del proyecto), se muestra la ruta de acceso al mismo:

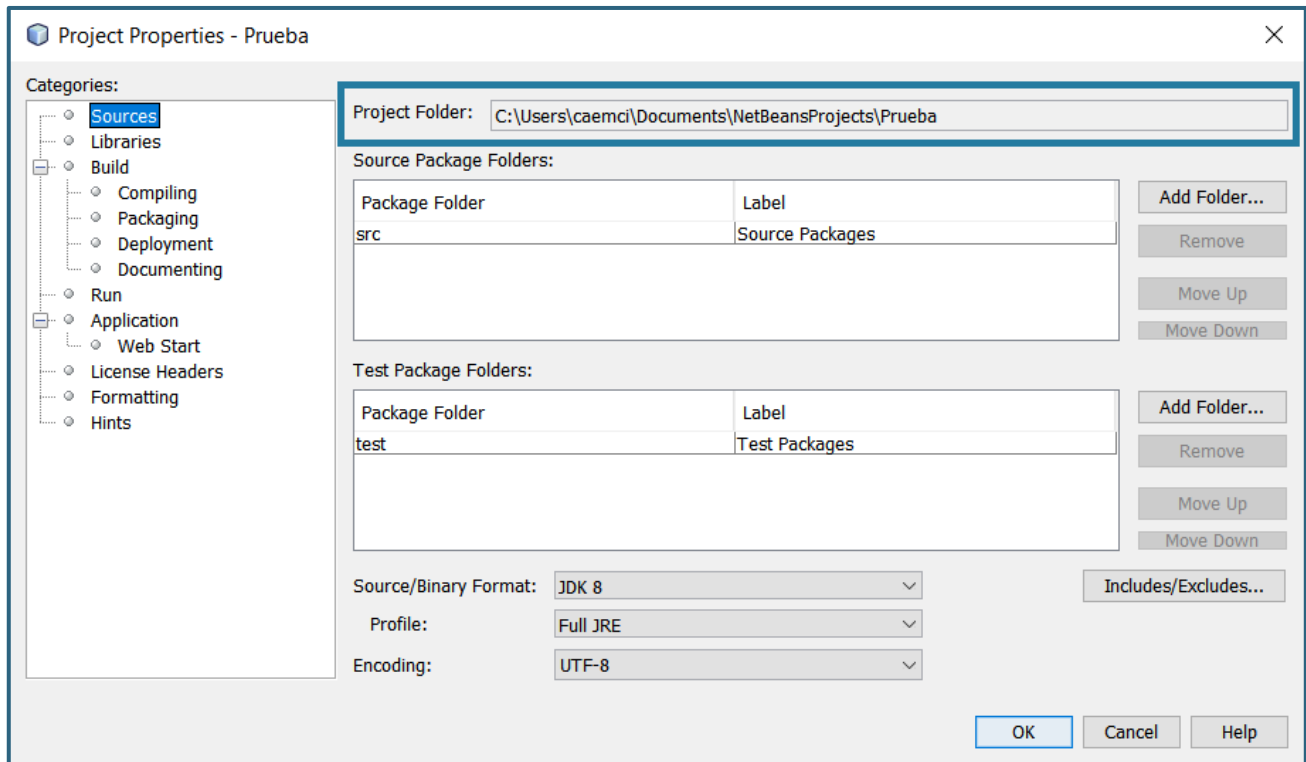


Ilustración 12: Ventana de propiedades del proyecto.


Un proyecto de NetBeans se guarda en una carpeta con su mismo nombre. Por lo tanto, la carpeta "Prueba" se puede copiar a un pendrive o subir a la nube para ser usada posteriormente en otra computadora.

## Cerrar todos los proyectos abiertos

Antes de abrir un proyecto de NetBeans, asegurate, sobre todo en máquinas públicas, de no tener otros proyectos abiertos, para evitar resultados inesperados.

Para cerrar todos los proyectos de NetBeans en un paso, abrí el menú "File" y clickeá la opción "Close All Projects" (cerrar todos los proyectos).

## Abrir un proyecto de NetBeans

Para abrir un proyecto de NetBeans, abrí el menú "File" y clickeá la opción "Open Project" (abrir proyecto) o el botón .

Buscá la carpeta con el proyecto de NetBeans. El IDE "detecta" qué carpeta es en particular un proyecto y le asigna un ícono de taza de café anaranjada. Es un buen indicio. **Seleccioná el proyecto con un click** y tocá el botón "Open Project":

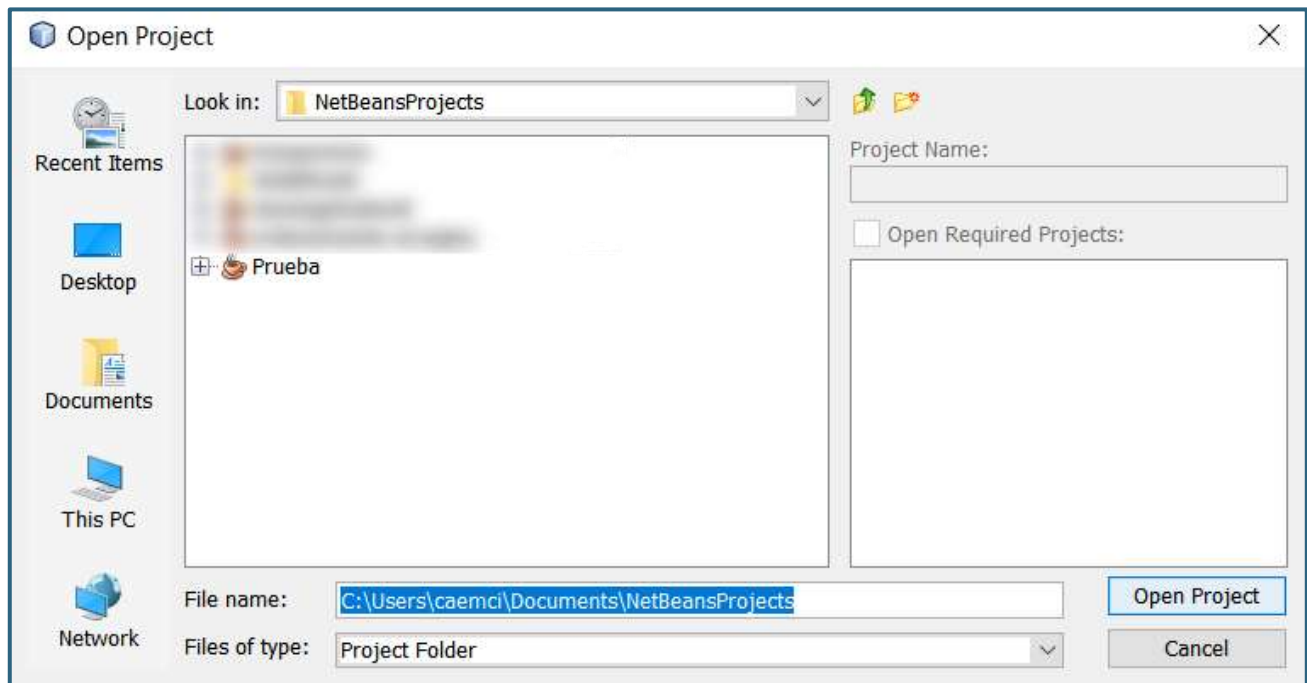


Ilustración 13: Ventana de apertura de proyecto.

## Flujo secuencial

En primer lugar, creá un nuevo proyecto en NetBeans que se llame "**Prueba**". Si no modificás ningún otro parámetro del proyecto, NetBeans creará un archivo llamado **Prueba.java**, cuyo contenido es similar al siguiente (recordá que podés borrar todos los comentarios):

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         // Tu primer código va aquí
6     }
7 }
```

### Código 2: Código mínimo para cualquier programa en Java.

A continuación, te explicaré la sintaxis mínima que sigue todo programa en Java.

La sentencia:

```
package prueba;
```

Indica que la clase se encuentra dentro de un **paquete** llamado **prueba** (notar que los paquetes se nombran con minúscula). Un **paquete** no es más que una carpeta que agrupa archivos. NetBeans crea un paquete por defecto del mismo nombre que el proyecto e introduce allí el archivo **Prueba.java**.

El bloque:

```
public class Prueba {
}
```

Indica que estás trabajando dentro de una **clase** que, en este ejemplo, se llama **Prueba**. El concepto de **clase** pertenece al paradigma orientado a objetos, por lo que por ahora solo tenés que tener en cuenta que **todo tu código Java debe ir dentro de las llaves que delimitan la clase**.

Además, el nombre de la clase debe ser idéntico al nombre del archivo. En este caso, tendrías la clase **Prueba** cuyo archivo se llama **Prueba.java**. No respetar esta similitud provocará un error de compilación.

El bloque:

```
public static void main(String[] args) {
}
```

Indica que estás trabajando dentro de un **método** llamado **main**. El concepto de **método** también pertenece al paradigma orientado a objetos, aunque en el contexto de la programación estructurada que se aborda en este libro, se trataría de una **función**. En este libro verás más adelante el concepto de **función**, por lo que por ahora solo tenés que tener en cuenta que por ahora **todo tu código Java debe ir dentro de las llaves que delimitan la función main**, que es el punto de entrada inicial de todo programa en Java.

Al igual que la gran mayoría de los lenguajes, las instrucciones se ejecutan de manera natural una tras otra, en el orden en que fueron escritas. Este flujo se denomina secuencial y es el más sencillo de todos. La máquina ejecutará paso a paso las líneas, desde la primera hasta la última, a no ser que se especifique lo contrario.

## Comentarios

Una muy buena práctica a la hora de programar es comentar el código. Esto significa añadir notas que ayuden a entender alguna instrucción compleja o para describir tareas pendientes. No son tenidos en cuenta por el compilador, solo sirven para el programador.

Un **comentario de línea** se inserta con una doble barra, sin espacios, de esta manera. Desde un `//` en adelante, se trata de un comentario.

Un **comentario de bloque** permite ocupar más de una línea. Se lo utiliza para evitar tener que poner `//` por cada línea a comentar. Para realizar un comentario de bloque, se debe encerrar el contenido entre los delimitadores `/*` y `*/`.

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         // Esto es un comentario en una línea completa
6         System.out.println("Hola Mundo!"); // Otro comentario
7         /* Esto es un comentario de bloque.
8            Todo lo que escriba entre estos delimitadores
9            no será evaluado por el compilador */
10    }
11 }
```

Código 3: Ejemplo de código comentado.

## Instrucción de salida

Lo primero que vas a aprender es a mostrarles mensajes al usuario por intermedio de la consola.

A continuación, pueden seguir apareciendo palabras que no comprendas, como **método**, **flujo (stream)** y **clase**. No te preocupes. Son términos de la programación orientada a objetos de los que no puedo despegarme si mi deseo es hablar con propiedad, pero cuyo significado no es necesario que conozcas para el desarrollo de este libro.

## Método `println()`

El método que permite generar una salida por consola, **posicionando luego el cursor en la línea siguiente** es `println()`, que se encuentra dentro del flujo **out** de la clase **System**.

La sintaxis es la siguiente:



```
System.out.println( <expresión> );
```

El término **<expresión>** debe reemplazarse por una verdadera expresión, como una cadena, una expresión aritmética o booleana, el contenido de una variable, etc. La expresión se escribe dentro de los paréntesis **()**, que son obligatorios.

**Toda instrucción en Java termina obligatoriamente con un punto y coma.**

El siguiente código realiza algunas salidas con **println()**:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         System.out.println("Una cadena");
6         System.out.println(123); // 123 como entero
7         System.out.println("123"); // "123" como cadena
8         System.out.println(2 + 3); // Evalúa y muestra 5
9     }
10 }
```

Código 4: Salidas con **println()**.

## Método **print()**

El método que permite generar una salida por consola, **posicionando luego el cursor en la misma línea** es **print()**, que se encuentra dentro del flujo **out** de la clase **System**.

La sintaxis es la siguiente:

```
System.out.print( <expresión> );
```

El término **<expresión>** debe reemplazarse por una verdadera expresión, como una cadena, una expresión aritmética, el contenido de una variable, etc. La expresión se escribe dentro de los paréntesis **()**, que son obligatorios.

**Toda instrucción en Java termina obligatoriamente con un punto y coma.**

El siguiente código realiza algunas salidas con **print()**:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         System.out.print("Una cadena");
6         System.out.print(123); // 123 como entero
7         System.out.print("123"); // "123" como cadena
```

```

8      System.out.print(2 + 3); // Evalúa y muestra 5
9      }
10     }

```

### Código 5: Salidas con print().

El no uso de comillas para una cadena o algún caso de ambigüedad (que abran comillas y que no cierren, o que no abran y sí cierren) derivará en un error en tiempo de compilación. Lo mismo ocurre con el uso de los paréntesis.

## Caracteres de escape

Los caracteres de escape permiten indicar cierto comportamiento especial, por ejemplo, el salto de línea o una tabulación. También permiten escribir caracteres por la consola que son utilizados por el lenguaje, como la doble comilla. ¿Cuáles son las comillas que abren y cierran una cadena y cuáles son las que se quieren mostrar en la consola de forma literal?

Los caracteres de escape lo solucionan:

| Carácter de escape | Nombre                   | Descripción   |
|--------------------|--------------------------|---|
| <code>\n</code>    | Nueva línea              | Posiciona el cursor en la próxima línea.            |
| <code>\t</code>    | Tabulador                | Posiciona el cursor una tabulación hacia adelante.  |
| <code>\r</code>    | Retorno de carro         | Posiciona el cursor al comienzo de la línea actual. |
| <code>\b</code>    | Borrado a la izquierda   | Borra el carácter anterior.                         |
| <code>\\</code>    | Carácter barra invertida | Imprime literalmente una barra invertida.           |
| <code>\'</code>    | Carácter comilla simple  | Imprime literalmente una comilla simple.            |
| <code>\"</code>    | Carácter comilla doble   | Imprime literalmente una comilla doble.             |

Realizá la siguiente prueba para comprobar el efecto:

```

1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         System.out.println("Texto\nen\nlíneas\nseparadas");
6         System.out.println("Texto\tseparado\tpor\ttabulaciones");
7         System.out.println("Esto no se ve\rCarro retornado");
8         System.out.println("Borro espacio \ba la izquierda");

```

```

9      System.out.println("Imprimo una barra invertida: \\");
10     System.out.println("Imprimo una comilla simple: \'");
11     System.out.println("Imprimo una comilla doble: \");
12     }
13 }
```

Código 6: Caracteres de escape.

## Tipos de datos primitivos

Java es un **lenguaje fuertemente tipado**, por lo que pone especial énfasis en los tipos de los datos. Las unidades mínimas de información se denominan **tipos de datos primitivos**. Son los siguientes:

| Tipo           | Descripción                     | Espacio en memoria | Rango de valores                                       |
|----------------|---------------------------------|--------------------|--|
| <b>byte</b>    | Entero mínimo                   | 1 byte             | -128 a 127   |
| <b>short</b>   | Entero corto                    | 2 bytes            | -32768 a 32767   |
| <b>int</b>     | Entero normal                   | 4 bytes            | -2147483648 a 2147483647                               |
| <b>long</b>    | Entero largo                    | 8 bytes            | -9223372036854775808 a 9223372036854775807             |
| <b>float</b>   | Número real de precisión simple | 4 bytes            | $\pm 3.4 \times 10^{-38}$ a $\pm 3.4 \times 10^{38}$   |
| <b>double</b>  | Número real de precisión doble  | 8 bytes            | $\pm 1.8 \times 10^{-308}$ a $\pm 1.8 \times 10^{308}$ |
| <b>char</b>    | Caracter simple Unicode         | 2 bytes            | \u0000 a \uFFFF  |
| <b>boolean</b> | Lógico                          | 1 byte             | true o false   |

Los tipos **byte**, **short**, **int** y **long** son enteros, donde la diferencia entre cada uno de ellos radica en el espacio ocupado en memoria, y, por ende, el rango de valores que permiten alojar.

Los tipos **float** y **double** permiten alojar números reales. La diferencia entre ambos es la precisión, es decir, la cantidad de decimales que puede albergar cada uno. Un **double** permite guardar más cifras decimales que un **float** a costa de ocupar el doble de espacio en memoria.

El tipo **char** permite guardar caracteres simples en formato **Unicode**<sup>1</sup>. En realidad, se trata de números, pues internamente la máquina los trata como tales, aunque la representación luego será como un símbolo.

<sup>1</sup> Unicode es un estándar de codificación de caracteres que define a cada uno de ellos mediante un número. En el formato UTF-16 que utiliza Java se cuenta con 65536 caracteres diferentes.

El tipo **boolean** permite alojar un valor lógico de verdad. Los únicos valores posibles son **true** (verdadero) o **false** (falso).

## El tipo de dato String

A la hora de trabajar con palabras, frases, texto o cualquier cosa que ocupe más de un carácter se necesita un tipo de dato capaz de alojar dichas posibilidades.

Un tipo de dato con el que cuenta Java entre sus utilidades es el **String**, que **no es un tipo de dato primitivo, sino una clase**. Por lo tanto, su tratamiento no se da de igual manera que con los tipos primitivos, aunque Java permite abreviar algunas cosas para que se parezca lo más posible a un primitivo.

Como primera diferencia sustancial, el tipo de dato **String** se escribe con la primera letra mayúscula.

Como convención, las clases se escriben con la primera letra en mayúsculas.

El tipo de dato **string** con la primera letra minúscula no existe. Intentar utilizarlo como tipo de dato generará un error de compilación.

## Expresiones aritméticas

Una expresión aritmética es aquella que se compone de números y operadores aritméticos. Toda expresión aritmética es evaluada por el compilador devolviendo un resultado numérico. Para poder formular expresiones aritméticas, es necesario conocer los operadores aritméticos de Java.

### Operadores aritméticos

Consta de los operadores fundamentales de la aritmética con el agregado del módulo o residuo, que devuelve el resto que se produce al realizar un cociente entre dos números.

| Operador | Nombre           | Ejemplo | Resultado | Descripción   |
|----------|------------------|---------|-----------|---|
| +        | Suma             | 12 + 3  | 15        | Devuelve la suma de dos expresiones.                  |
| -        | Resta            | 12 - 3  | 9         | Devuelve la resta de dos expresiones.                 |
| *        | Multiplicación   | 12 * 3  | 36        | Devuelve el producto de dos expresiones.              |
| /        | División         | 12 / 3  | 4         | Devuelve el cociente de dos expresiones.              |
| %        | Módulo o Residuo | 12 % 3  | 0         | Devuelve el resto del cociente entre dos expresiones. |

Siguiendo los fundamentos de la aritmética, los operadores de multiplicación, división y módulo tienen mayor prioridad que la suma y resta. Así, por ejemplo, la expresión **2 + 3 \* 4** devuelve como resultado **14**, pues primero se evalúa el término **3 \* 4** que resulta **12**, y luego se realiza la suma entre **2** y **12**.



## Operaciones con enteros

En Java, toda operación entre números enteros devolverá un resultado entero.

Esto puede parecer irrelevante, pero el caso donde no tenerlo en cuenta puede resultar peligroso es en la división.

Hacé la siguiente prueba:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         System.out.print(10 / 4); // Evalúa y muestra 2
6     }
7 }
```

### Código 7: Prueba de división entre enteros.

Java evalúa la expresión y muestra el valor **2**, es decir, el cociente entero de la división.

Tal efecto a veces puede resultar ventajoso, pero otras veces se necesita el resultado con decimales. Para que Java pueda devolver un valor con decimales (por defecto, en **double**), es necesario que haya al menos un operando de tipo **double**.

Probá ahora el siguiente programa:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         System.out.println(10.0 / 4.0); // Evalúa y muestra 2.5
6         System.out.println(10.0 / 4); // Evalúa y muestra 2.5
7         System.out.println(10 / 4.0); // Evalúa y muestra 2.5
8     }
9 }
```

### Código 8: Prueba de división con valores de tipo double.

En todos los casos, la máquina evalúa y muestra el valor **2.5**.

Si en cierta expresión aparecen dos variables de tipo entera, al menos una de ellas se puede convertir a **double**, mediante lo que se denomina "**casteo**". Lo verás después.

# Variables

## Definición

Una variable es un espacio reservado en la memoria de la máquina que permite alojar información. Tal como su nombre lo indica, esta información puede cambiar en el transcurso del programa, a contraposición de lo que sería una constante.

## Reglas de nomenclatura

Si bien la elección de un identificador para la variable es a elección del programador, deben tenerse en cuenta ciertas consideraciones:

- Un identificador debe comenzar con una letra.
- Puede contener letras, números y guiones bajos, pero no espacios, ni operadores.
- No puede coincidir con alguna palabra reservada del lenguaje.
- No debe contener caracteres "extraños" como eñes, acentos o diéresis.

Otra cuestión importante es que no puede haber identificadores repetidos, es decir, si ya declaraste la variable `sueldo` y necesitas guardar otro, deberás declarar una segunda variable usando algún identificador alternativo como `sueldo2` o `sueldoB`.

Es muy importante para la legibilidad y depuración del código, tanto para vos como para un tercero, que utilices nombres para las variables que describan lo más claro posible el contenido de estas. Nombrar a todas las variables como `p1`, `x` o `abcd` no es muy conveniente.

Hay una convención bastante extendida y que te recomiendo fehacientemente que adoptes: **los nombres de las variables siempre van en minúsculas, inclusive su primera letra.**

No hay grandes problemas con nombres cortos, como `precio` o `apellido`, pero la lectura se dificulta con nombres compuestos por más de una palabra. Para ellos existe un estilo de escritura llamado **lowerCamelCase** (**lower** por comenzar con minúscula y **CamelCase** por la similitud de la notación con las jorobas de un camello) que consiste en comenzar el nombre de la variable con minúscula y a cada nueva palabra (sin espacio, no lo olvides) comenzarla con mayúscula. Con este estilo, se hace mucho más legible, por ejemplo, nombrar a una variable como `estadoDeCuenta` antes que `estadodecuenta`.

A partir de aquí, todas las variables que use en los ejemplos adoptarán la notación **lowerCamelCase**.

## Declaración

Java requiere que se declaren las variables que van a ser utilizadas a lo largo del programa.

La sintaxis para definir una variable en Java es la siguiente:

```
<tipo> <identificador>;
```

Por ejemplo:

```
int edad;
```

Al hacer esto, la computadora reservará un espacio disponible en la memoria que permitirá alojar un dato de tipo entero. A partir de este momento, cualquier referencia a este dato, ya sea para cargarlo, modificarlo o leerlo es mediante el identificador que escogí, llamado **edad**, que, por ahora, **no tiene valor**.

No se puede volver a declarar una variable que ya lo había sido.

Intentar utilizar una variable sin declarar, derivará en un error de compilación.

Como buena práctica te sugiero declarar todas las variables al principio del programa, por más que haya alguna que vayas a utilizar a lo último. Esto permite una mejor legibilidad del programa.

En el momento en que se declara una variable, esta tiene un valor indefinido. Cualquier intento de operar con este valor derivará en un error de compilación. A este caso se lo llama como

## Operador de asignación

Recordá que hasta ahora **las variables no están inicializadas**, es decir, su valor es **indefinido**.

Hay dos maneras de establecer un valor a una variable. La primera que verás es a través del operador de asignación que en Java es el `=`.

Tanto en Java como en muchos otros lenguajes que adoptan al `=` como operador de asignación, es fácil confundir su función con el símbolo de igualdad usado en matemática. Por ello muchos lenguajes formales usan otros símbolos, como `<-` o `:=`, dejando al `=` para hacer comparaciones. En Java las comparaciones se hacen con el operador `==`.

Java permite hacer la declaración e inicialización de la variable en la misma línea, de la siguiente manera:

```
<tipo> <identificador> = <valor_inicial>;
```

Por ejemplo:

```
int edad = 25;
```

Si por alguna circunstancia debiera declararse una variable pero que aún no debiera ser inicializada, se puede realizar la asignación en una línea aparte.

Por ejemplo:

```
int edad;  
edad = 25;
```

No se puede asignar a una variable un valor que no se corresponde con el tipo de dato con la que fue definida.

El siguiente programa declara algunas variables y realiza algunas operaciones con ellas:

```
1 package prueba;  
2  
3 public class Prueba {
```

```
4    public static void main(String[] args) {  
5        int a;  
6        int b;  
7        String cad;  
8        double z = 3.45;  
9        double res;  
10       a = 10;  
11       b = 5;  
12       res = z / (a + b);  
13       cad = "El resultado de z / (a + b) es ";  
14       System.out.println(cad);  
15       System.out.println(res);  
16   }  
17 }
```

Código 9: Declaración, inicialización, operación y muestra de variables.

Es importante que asimiles que las asignaciones se leen de **derecha a izquierda**. El valor de la expresión o variable que está a la **derecha** del operador = se copia y se guarda en la variable que está a la **izquierda** del operador =. Si te acostumbrás a esto desde temprano, no tendrás problema más adelante en comprender cómo actualizar variables según su resultado anterior.

## Operador de concatenación

Para hacer más atractivas las salidas, se utiliza el **operador de concatenación**, que en Java se trata del +. Este permite unir cadenas con otras o con contenidos de variables.

Es indistinto dejarlo pegado o con espacios entre las partes. Donde sí hay diferencia es dentro de las comillas que delimitan una cadena.

Como el operador + también se utiliza para hacer sumas, es importante que reconozcas el contexto donde es utilizado, para entender si se está comportando como el operador algebraico de suma o el de concatenación.

Si reformulo el código anterior aplicando el operador de concatenación, se obtiene una salida más elegante:

```
1    package prueba;  
2  
3    public class Prueba {  
4        public static void main(String[] args) {  
5            int a;  
6            int b;
```

```
7      String cad;
8      double z = 3.45;
9      double res;
10     a = 10;
11     b = 5;
12     res = z / (a + b);
13     cad = "El resultado de z / (a + b) es " + res;
14     System.out.println(cad);
15 }
16 }
```

Código 10: Declaración, inicialización, operación y muestra de variables con el operador de concatenación.

## Convención de los tipos de datos

Si bien Java cuenta con ocho tipos de datos primitivos, más la clase **String** que es algo especial, en este libro adoptaré la siguiente convención:

- Para datos de tipo **entero**, utilizaré el tipo de dato **int**.
- Para datos de tipo **real**, utilizaré el tipo de dato **double**.
- Para datos de tipo **lógico**, utilizaré el tipo de dato **boolean**.
- Para datos de tipo **caracter**, utilizaré el tipo de dato **char**.
- Para datos de tipo **cadena de caracteres**, utilizaré el tipo de dato **String**.

En aplicaciones a gran escala, donde se hace un uso intenso de la memoria, los programadores más avanzados tienen en cuenta los demás tipos de datos. Por ejemplo, para guardar la edad de una persona, es más que suficiente con el tipo **byte**, que permite alojar números enteros hasta el valor **128**. Pero como la mayoría de las funciones de las librerías de Java devuelven datos de tipo **int** o **double**, además de que los números por defecto se representan como tales tipos, no conviene meterse en problemas de conversión de tipos tan temprano.

## Actualizar una variable

Muchas veces te verás en la necesidad de establecer el valor de una variable según su resultado actual. Eso se llama **actualizar una variable**. Es usado comúnmente en estructuras de repetición, en las que necesitarás contadores y acumuladores.

El siguiente programa demuestra cómo se actualiza una variable llamada **num**:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         int num;
```



```

6      num = 5; // num vale 5
7      System.out.println("num vale " + num); // Muestra 5
8      num = num + 4; // num ahora vale 9
9      System.out.println("num vale " + num); // Muestra 9
10     }
11 }

```

Código 11: Actualización de una variable.

## Operadores incrementales

A la hora de usar contadores, donde en general se incrementa o decrementa el valor de una variable de a una unidad, Java incorpora dos operadores que simplifican la instrucción.

Tomando como ejemplo una variable **x** con valor **12**, los operadores actúan de la siguiente manera:

| Operador  | Nombre     | Ejemplo    | Nuevo valor de x | Similar a        |
|-----------|------------|------------|------------------|------------------|
| <b>++</b> | Incremento | <b>x++</b> | <b>13</b>        | <b>x = x + 1</b> |
| <b>--</b> | Decremento | <b>x--</b> | <b>11</b>        | <b>x = x - 1</b> |

## Otros operadores de asignación

A la hora de usar acumuladores, donde en general se actualiza el valor de una variable respecto de otra, Java incorpora cinco operadores que simplifican la instrucción.

Tomando como ejemplo una variable **x** con valor **12** y una variable **a** con valor **3**, los operadores actúan de la siguiente manera:

| Operador  | Nombre                        | Ejemplo       | Nuevo valor de x | Similar a        |
|-----------|-------------------------------|---------------|------------------|------------------|
| <b>+=</b> | Suma y reasignación           | <b>x += a</b> | <b>15</b>        | <b>x = x + a</b> |
| <b>-=</b> | Resta y reasignación          | <b>x -= a</b> | <b>9</b>         | <b>x = x - a</b> |
| <b>*=</b> | Multiplicación y reasignación | <b>x *= a</b> | <b>36</b>        | <b>x = x * a</b> |
| <b>/=</b> | División y reasignación       | <b>x /= a</b> | <b>4</b>         | <b>x = x / a</b> |
| <b>%=</b> | Módulo y reasignación         | <b>x %= a</b> | <b>0</b>         | <b>x = x % a</b> |

El siguiente programa demuestra cómo se actualiza una variable llamada **num** utilizando distintos operadores:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         int num;
6         num = 5; // num vale 5
7         System.out.println("num vale " + num); // Muestra 5
8         num = num + 1; // num ahora vale 6
9         System.out.println("num vale " + num); // Muestra 6
10        num++; // num ahora vale 7
11        System.out.println("num vale " + num); // Muestra 7
12        num = num * 2; // num ahora vale 14
13        System.out.println("num vale " + num); // Muestra 14
14        num *= 2; // num ahora vale 28
15        System.out.println("num vale " + num); // Muestra 28
16    }
17 }
```

Código 12: Demostración de operadores incrementales.

## Promoción de tipos (casting)

El casting, también llamado promoción o conversión de tipos consiste en almacenar o representar un valor de un determinado tipo como si se tratara de otro.

No todas las conversiones entre los distintos tipos de dato son posibles. Por ejemplo, en Java no es posible convertir valores booleanos a otro tipo de dato y viceversa.

### Promoción por ensanchamiento

El **casting o promoción por ensanchamiento** trata de guardar un valor de determinado tipo en una variable cuyo tipo sea compatible y de mayor rango con respecto al valor a guardar. Se da de forma implícita, sin necesidad de una aclaración por parte del programador.

Por ejemplo, en una variable de tipo **double** es posible guardar un valor de tipo **int**:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         double numero;
6         numero = 3;
```

```

7      System.out.println(numero); // Muestra 3.0
8      }
9  }
```

Código 13: Demostración de casting implícito por ensanchamiento.

## Promoción por estrechamiento

El **casting o promoción por estrechamiento** trata de guardar un valor de determinado tipo en una variable cuyo tipo sea compatible y de menor rango con respecto al valor a guardar, asumiendo que habrá pérdida de información. Para tal fin, es necesario hacer un casting explícito, anteponiendo al valor a guardar el nuevo tipo de dato requerido entre paréntesis.

Por ejemplo, en una variable de tipo **int** es posible guardar un valor de tipo **double**, a costa de perder los decimales, haciendo casting explícito:

```

1  package prueba;
2
3  public class Prueba {
4      public static void main(String[] args) {
5          int numero;
6          numero = (int) 3.123456;
7          System.out.println(numero); // Muestra 3
8      }
9  }
```

Código 14: Demostración de casting explícito por estrechamiento.

## Promociones posibles

El siguiente cuadro sintetiza cuáles y cómo son las conversiones posibles entre los datos:

| Promoción de tipos en Java |                 |         |         |         |         |         |         |        |
|----------------------------|-----------------|---------|---------|---------|---------|---------|---------|--------|
| Dato de origen             | Dato de destino |         |         |         |         |         |         |        |
|                            | boolean         | byte    | short   | char    | int     | long    | float   | double |
| boolean                    |                 | X       | X       | X       | X       | X       | X       | X      |
| byte                       | X               |         | ✓       | casting | ✓       | ✓       | ✓       | ✓      |
| short                      | X               | casting |         | casting | ✓       | ✓       | ✓       | ✓      |
| char                       | X               | casting | casting |         | ✓       | ✓       | ✓       | ✓      |
| int                        | X               | casting | casting | casting |         | ✓       | ✓*      | ✓*     |
| long                       | X               | casting | casting | casting | casting |         | ✓*      | ✓*     |
| float                      | X               | casting | casting | casting | casting | casting |         | ✓      |
| double                     | X               | casting | casting | casting | casting | casting | casting |        |

Ilustración 14: Cuadro completo de promoción de tipos.

- Una **X** indica que no es posible la promoción.
- Una **✓** indica que la promoción es posible e implícita.
- Una **✓\*** indica que la promoción es posible e implícita, pero con posible pérdida de información.
- Un **casting** indica que la promoción es posible, pero con casting explícito.

## Casos de ambigüedad

Hay casos donde la representación de un valor puede causar cierta ambigüedad, como, por ejemplo, tener expresado el valor **2.5**. ¿Se trata de un **float** o de un **double**?

Con respecto a los reales, todo número se expresa en formato **double**. Si se requiere representar un valor como **float**, se tiene que escribir una **f** o **F** junto a él. Por lo tanto, un **2.5** a secas es un **double** (también se puede representar como **2.5D**) y **2.5F** es un valor **float**.

Con respecto a los enteros, todo número se expresa en formato **int**. Si se requiere representar un valor como **long**, se tiene que escribir una **l** o **L** junto a él. Por lo tanto, un **183** a secas es un **int** y un **183L** es un **long**. No hay representaciones para los tipos **byte** y **short**.

## Instrucción de entrada

### La clase Scanner

La entrada de datos por intermedio del usuario a través del teclado no es tan intuitiva. Al igual que en el caso de la salida, hay que hacer uso de un método, pero esta vez importando una clase auxiliar. Se necesita una clase llamada **Scanner**, que viene incluida en las librerías de Java, dentro del paquete **util**.

Lo primero a realizar es la importación de la clase **Scanner** al código, de la siguiente manera:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7
8     }
9 }
```

#### Código 15: Importación de la clase Scanner.

A continuación, se debe declarar una variable de tipo **Scanner** cuyo nombre será **entrada**. Al ser una variable, el nombre puede ser cualquiera que siga las reglas de nombrado.

A la variable **entrada** se le asigna una instancia de la clase **Scanner**, pasando como argumento al constructor el flujo de entrada estándar del sistema:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8     }
9 }
```

**Código 16: Declaración e inicialización del objeto entrada.**

No te preocupes si no entendés lo que estás haciendo. Por ahora tan solo copió el código como figura y funcionará.

## Leer datos numéricos

Para proceder a pausar la ejecución del programa y esperar que el usuario ingrese un valor, se debe hacer uso de los métodos que provee el objeto **entrada**.

El método a elegir depende del tipo de dato esperado. Para los tipos numéricos, existen los métodos **nextByte()**, **nextShort()**, **nextInt()**, **nextLong()**, **nextFloat()** y **nextDouble()**, que devuelven el dato del usuario en el formato que sus propios nombres describen (siempre y cuando lo que el usuario haya ingresado sea compatible con el tipo de dato esperado).

El siguiente programa lee dos números enteros y muestra la suma de ellos:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8         int a;
9         int b;
10        System.out.print("Ingresa un número: ");
11        a = entrada.nextInt();
12        System.out.print("Ingresa otro número: ");
13        b = entrada.nextInt();
14        System.out.println("La suma de ambos es " + (a + b));
15    }
```



16 }

**Código 17: Suma de dos números ingresados por el usuario.**

El separador de decimales en Java es el punto. Para expresar un valor **double** en el código Java, debés tener ello en cuenta:

```
double x = 3.57; // Correcto
```

```
double y = 3,57; // Incorrecto
```

Sin embargo, a la hora de leer números reales desde la consola, a través de los métodos **nextFloat()** o **nextDouble()**, hay que tomar ciertos recaudos.

El separador de miles de la consola depende de la configuración del sistema operativo. Generalmente, **si Windows está en español**, toma como separador de decimales la **coma**, por lo que los valores deben ingresarse de dicha manera.

**Leer datos alfanuméricos con next()**

El método **next()** de la clase **Scanner** lee una línea una línea hasta que se detecta un espacio y la devuelve en formato **String**, por lo tanto, un simple nombre compuesto como **"Juan Carlos"** solo sería guardado como **"Juan"**.

Como no existe un método para leer caracteres simples (de tipo **char**), la manera de obtenerlos es leyendo una línea hasta que se detecta el espacio con el método **next()** y quedarse con el primer carácter gracias al método **charAt()** que incorpora la clase **String**, con el valor **0** como argumento (indica que querés el carácter ubicado en la primera posición).

El siguiente programa lee un carácter a través de la consola y lo muestra (podés probar ingresando cualquier texto, solo se leerá el primer carácter):

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8         char caracter;
9         System.out.print("Ingresá un texto: ");
10        caracter = entrada.next().charAt(0);
11        System.out.println("Se detectó el caracter " + caracter);
12    }
13 }
```

**Código 18: Ingreso de un caracter simple.**

## Leer datos alfanuméricos con `nextLine()`

La clase `Scanner` permite leer una línea completa incluyendo los espacios gracias al método `nextLine()`, que devuelve el texto ingresado en formato **`String`**.

El siguiente programa lee un nombre a través de la consola y realiza un saludo:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8         String nombre;
9         System.out.print("Ingresa tu nombre: ");
10        nombre = entrada.nextLine();
11        System.out.println("Hola " + nombre + "!");
12    }
13 }
```

Código 19: Ingreso de una cadena.

## El problema con `nextLine()`

Cuando se utiliza el método `nextLine()`, se lee una línea completa del buffer de entrada, inclusive aquellos caracteres no imprimibles, como la tecla **Enter**. La lectura de esta manera puede evidenciar un problema cuando se ejecuta tras haber leído un dato numérico.

Cada vez que invocás a un método que lea datos numéricos como `nextInt()` o `nextDouble()`, éste se queda con el dato entero y deja en el buffer el carácter no imprimible de salto de línea (la tecla **Enter**).

Si a continuación se ejecuta un método `nextLine()`, leerá la línea completa, que consta de un carácter no imprimible de salto de línea que quedó en el buffer, lo que se interpreta como que finalizó la carga y la cadena queda vacía. Si hubieras esperado una cadena no vacía y quisieras trabajar con ella, probablemente te encontrarías con resultados inesperados o un error.

Probá el siguiente ejemplo:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
```

```
7      Scanner entrada = new Scanner(System.in);
8      int edad;
9      String nombre;
10     System.out.println("Ingresa tu edad: ");
11     edad = entrada.nextInt();
12     System.out.println("Ingresa tu nombre: ");
13     nombre = entrada.nextLine(); // No se Lee
14     System.out.println("Hola " + nombre + ". Tenés " + edad + " años.");
15 }
16 }
```

#### Código 20: Programa que explicita el problema de usar nextLine().

La variable **nombre** quedó cargada con un carácter no imprimible de salto de línea, cuestión que es evidenciada en la salida posterior.

Una solución para este problema es limpiar el buffer leyendo el carácter no imprimible de salto de línea, pero sin asignarlo en alguna variable (simplemente, se pierde). A partir de allí se puede usar el método **nextLine()** de manera segura. Debés repetir este proceso luego de cada instrucción de ingreso de datos numérico:

```
1  package prueba;
2
3  import java.util.Scanner;
4
5  public class Prueba {
6      public static void main(String[] args) {
7          Scanner entrada = new Scanner(System.in);
8          int edad;
9          String nombre;
10         System.out.println("Ingresa tu edad: ");
11         edad = entrada.nextInt();
12         entrada.nextLine(); // Limpia el buffer
13         System.out.println("Ingresa tu nombre: ");
14         nombre = entrada.nextLine();
15         System.out.println("Hola " + nombre + ". Tenés " + edad + " años.");
16     }
17 }
```

#### Código 21: Solución al problema de nextLine().

## Uso de funciones

Una **función** es un trozo de código que resuelve un pequeño problema en particular. No es necesario reinventar la rueda. La misma ya ha sido inventada y solo requerís saber usarla.

En este libro verás más adelante cómo definir tus propias funciones. Por ahora, lo que interesa es que sepas hacer uso de algunas de las que trae Java.

Las primeras funciones que utilizaste son aquellas que provee la clase **Scanner** para leer datos por la consola. En realidad, se trata de **métodos**, pues están dentro de un objeto, pero el funcionamiento es similar. Más adelante verás la diferencia entre una **función** y un **método**.

### Abstracción

Realizar un simple programa que pida al usuario dos números que representan catetos y mostrar el resultado de la hipotenusa demanda utilizar una operación matemática compleja cuyo operador no existe en Java: **la raíz cuadrada**. Por ende, el problema inicial ahora supone otro, que es pensar y codificar la lógica para que, dado cierto número, la computadora halle su raíz cuadrada. Como verás, son dos problemas completamente independientes entre sí.

Hay muchos problemas resueltos cuyas soluciones ya son provistas por las librerías de Java, no debés preocuparte. Tan solo debés invocar a cierta función, que, repito, no es más que un trozo de código que realiza una tarea en particular, enviándole como dato extra el número al cual se quiere calcular la raíz. La función realizará su tarea y devolverá el resultado correcto.

¿**Quién** realizó la función? ¿**Cómo** funciona por dentro? ¿**Por qué** funciona bien? La respuesta es: **no interesa**. Bienvenido al concepto de **abstracción** en la informática.

Cuando se realizan programas complejos, los programadores aprovechan que muchas cuestiones ya se encuentran solucionadas. La abstracción permite concentrarse en **qué** hace cierta función, **qué** datos necesita, **qué** datos retorna, sin importar el **cómo**.

### La clase Math

Para realizar operaciones matemáticas complejas, como por ejemplo una raíz cuadrada, Java cuenta con una clase llamada **Math** que incorpora métodos para realizarlas. No es necesario importarla, sino que basta con escribir su nombre seguido de un punto y el nombre del método a invocar.

Los métodos de **Math** están testeados y preparados para devolver el resultado correcto, siempre y cuando se respeten las precondiciones establecidas. Por ejemplo, el método **Math.sqrt(num)** calcula y devuelve la raíz cuadrada de la variable **num** siempre y cuando la misma no sea negativa.

Los datos que se envían a la función para que esta los procese se denominan **argumentos**. Hay funciones que esperan recibir un argumento, otras dos, otras ninguno. Depende de la función. **Los argumentos van siempre entre paréntesis** luego del nombre de la función o método. Si hay más de uno, se separan con comas. Si una función no espera recibir argumentos, los paréntesis deben escribirse igual, obviamente sin nada dentro.

Una función **devuelve un único valor** de determinado tipo.

Los métodos más importantes, junto a sus argumentos y el tipo de dato que devuelven, se detallan a continuación:

| Método                | Descripción  | Argumentos                                    | Ejemplo                      | Devuelve     |
|-----------------------|--|---|------------------------------|--------------|
| <code>abs()</code>    | Devuelve el valor absoluto en <b>double</b> .  | Un número <b>double</b> .                     | <code>Math.abs(-4.7)</code>  | <b>4.7</b>   |
| <code>ceil()</code>   | Devuelve el valor redondeado hacia arriba en <b>double</b> .                               | Un número <b>double</b> .                     | <code>Math.ceil(4.1)</code>  | <b>5.0</b>   |
| <code>cos()</code>    | Devuelve el coseno en <b>double</b> .  | Un número <b>double</b> .                     | <code>Math.cos(0)</code>     | <b>1.0</b>   |
| <code>pow()</code>    | Devuelve la potencia en <b>double</b> .  | Dos números <b>double</b> (base y exponente). | <code>Math.pow(2,3)</code>   | <b>8.0</b>   |
| <code>round()</code>  | Devuelve el valor redondeado en formato <b>long</b> .                                      | Un número <b>double</b> .                     | <code>Math.round(4.7)</code> | <b>5L</b>    |
| <code>sin()</code>    | Devuelve el seno en <b>double</b> .  | Un número <b>double</b> .                     | <code>Math.sin(0)</code>     | <b>0.0</b>   |
| <code>tan()</code>    | Devuelve la tangente en <b>double</b> .  | Un número <b>double</b> .                     | <code>Math.tan(0)</code>     | <b>0.0</b>   |
| <code>floor()</code>  | Devuelve el valor redondeado hacia abajo en <b>double</b> .                                | Un número <b>double</b> .                     | <code>Math.floor(4.7)</code> | <b>4.0</b>   |
| <code>random()</code> | Devuelve un número aleatorio entre <b>0</b> y <b>1</b> (sin incluirlos) en <b>double</b> . | -   | <code>Math.random()</code>   | <b>(0;1)</b> |
| <code>sqrt()</code>   | Devuelve la raíz cuadrada en <b>double</b> .   | Un número <b>double</b> .                     | <code>Math.sqrt(81)</code>   | <b>9.0</b>   |

El siguiente programa muestra los resultados de los ejemplos de la tabla anterior para que los compruebes:

```

1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         System.out.println("Math.abs(-4.7) vale " + Math.abs(-4.7) );
6         System.out.println("Math.ceil(4.1) vale " + Math.ceil(4.1) );
7         System.out.println("Math.cos(0) vale " + Math.cos(0) );
    }
}

```



```
8      System.out.println("Math.pow(2,3) vale " + Math.pow(2,3) );
9      System.out.println("Math.round(4.7) vale " + Math.round(4.7) );
10     System.out.println("Math.sin(0) vale " + Math.sin(0) );
11     System.out.println("Math.tan(0) vale " + Math.tan(0) );
12     System.out.println("Math.floor(4.7) vale " + Math.floor(4.7) );
13     System.out.println("Math.random() vale " + Math.random() );
14     System.out.println("Math.sqrt(81) vale " + Math.sqrt(81) );
15 }
16 }
```

Código 22: Demostración de algunos métodos de la clase Math.

## Flujo de selección

A la hora de resolver problemas más complejos, es necesario poder contar con alguna manera de poder discriminar entre la ejecución de ciertas instrucciones u otras en base a cierto criterio. Esto abre un nuevo abanico de posibilidades para la resolución de problemas, ya que la máquina será capaz de tomar uno u otro camino dependiendo de una condición planteada de antemano por el programador.

## Expresiones booleanas

Para poder establecer condiciones y que la máquina las evalúe para saber qué camino tomar, se debe hacer uso de **expresiones booleanas**, es decir, aquellas que **únicamente devuelven un valor verdadero o falso**.

En Java, los literales que representan los valores de verdad son **true** (para verdadero) y **false** (para falso). Son palabras reservadas del lenguaje.

El tipo de dato asociado a los valores booleanos es el **boolean**.

Las palabras **true** y **false** se escriben sin comillas. "**true**" y "**false**" no representarían valores de verdad, sino que se interpretarían como cadenas de caracteres (**String**), produciéndose un error de tipos.

## Operadores relacionales

Son aquellos que permiten comparar expresiones aritméticas. Si la evaluación es correcta, se retorna **true**, de lo contrario, se retorna **false**.

| Operador | Nombre            | Ejemplo | Resultado |
|----------|-------------------|---------|-----------|
| <        | Menor que         | 4 < 5   | true      |
| <=       | Menor o igual que | 5 <= 5  | true      |
| >        | Mayor que         | 4 > 5   | false     |
| >=       | Mayor o igual que | 5 >= 5  | true      |
| ==       | Igual que         | 4 == 5  | false     |
| !=       | Distinto que      | 4 != 5  | true      |

Para comprobar cómo Java evalúa las expresiones booleanas, te invito a que pruebes el siguiente programa, que cuenta con unas cuantas de ellas, de menor a mayor complejidad. Tu tarea no tan solo es ver los resultados de la ejecución del programa, sino que evalúes las expresiones de manera manual para comprobar que el resultado es el mismo.

Es muy común al principio confundir el operador de asignación (=) con el de comparación (==). Si cometes el error de intentar mostrar la siguiente comparación utilizando el operador de asignación:

```
System.out.println(2 = 2);
```

Java dará un error, indicando que no se puede asignar un **2** a un valor constante **2**.

Comenzá desde temprano a prestar atención y notar el contexto del código, para saber si se trata de una asignación (con =) o una comparación (con ==).

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         int a = 2;
8         int b = 3;
9         System.out.println(a == b); // false
10        System.out.println(a != b); // true
11        System.out.println(a < b); // true
12        System.out.println(a > b); // false
13        System.out.println(a >= b); // false
14        System.out.println(a <= b); // true
15        System.out.println(5 + 4 < b - a); // false
16        System.out.println(a < b - 2 * b * 5); // false
17        System.out.println(5 - b * 0 >= Math.sqrt(3 + a - 1)); // true
18    }
19 }
```

Código 23: Verificación de expresiones booleanas con operadores relacionales entre números.

## Bloque de selección simple if

La primera y más sencilla manera de alterar el flujo secuencial por defecto de cualquier programa estructurado es utilizar una estructura de selección simple.

La sintaxis es la siguiente:

```
if ( <expresión_booleana> ) {
    <instrucciones>
}
```

Donde:

- La palabra **if** es obligatoria. Indica el comienzo de un bloque de selección.
- **<expresión\_booleana>** es la condición que la computadora evaluará. Es obligatorio que se encuentre encerrada entre paréntesis.
- Las llaves de apertura y cierre delimitan las instrucciones del bloque de selección.
- **<instrucciones>** son las instrucciones que se ejecutarán. Puede ser solo una, varias o inclusive otros bloques de selección o repetición, siguiendo los conceptos vistos hasta aquí.

En cierto momento, definido por el programador, la computadora evalúa una condición, es decir, una expresión booleana. Si la expresión devuelve **true**, la computadora ejecuta las instrucciones dentro del bloque **if**, de lo contrario, continúa con las instrucciones tras la llave de cierre.

Un programa sencillo que facilita la comprensión podría ser el siguiente: pedirle al usuario que ingrese su edad. En caso de que tenga menos de 18 años, mostrarle un mensaje que diga **"No podés ingresar"**. Sea cual fuere la edad, el programa termina diciendo **"Suerte"**.

El código sería el siguiente:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8         int edad;
9         System.out.print("Ingresa tu edad: ");
10        edad = entrada.nextInt();
11        if (edad < 18) {
12            // Se escribe según La edad
13            System.out.println("No podés ingresar");
14        }
15        System.out.println("Suerte"); // Se escribe siempre
16    }
17 }
```

Código 24: Algoritmo que valida un pase según la edad.

## Bloque de selección doble if-else

La estructura anterior permite realizar una serie de instrucciones en caso de que una condición sea **true**, pero no especifica nada en caso de que sea **false**. De hecho, si la condición resulta **false**, el programa continúa su ejecución como si el bloque **if** no existiera.

Para contemplar y hacer una serie de instrucciones específicas en caso de que una condición resulte **false**, sin dejar de lado lo que se hacía cuando resultaba **true**, es necesario utilizar una estructura de selección doble.

La sintaxis es la siguiente:

```
if ( <expresión_booleana> ) {  
    <instrucciones>  
}  
else {  
    <instrucciones>  
}
```

Donde:

- La palabra **if** es obligatoria. Indica el comienzo de un bloque de selección.
- **<expresión\_booleana>** es la condición que la computadora evaluará. Es obligatorio que se encuentre encerrada entre paréntesis.
- Las llaves de apertura y cierre delimitan las instrucciones del bloque de selección. Un par de llaves para **if** y otro para **else**.
- **<instrucciones>** son las instrucciones que se ejecutarán. Puede ser solo una, varias o inclusive otros bloques de selección o repetición, siguiendo los conceptos vistos hasta aquí.

En cierto momento, definido por el programador, la computadora evalúa una condición, es decir, una expresión booleana. Si la expresión devuelve **true**, la computadora ejecuta las instrucciones dentro del bloque **if**, de lo contrario, ejecuta las instrucciones dentro del bloque **else**.

Continuando con el programa sencillo de ejemplo anterior, voy a pedirle al usuario que ingrese su edad. En caso de que tenga menos de 18 años, mostrarle un mensaje que diga "**No podés ingresar**", en caso contrario, dirá "**Te damos la bienvenida, ¡pasá!**". Sea cual fuere la edad, el programa termina diciendo "**Suerte**".

Las llaves que delimitan el bloque **if** se pueden obviar si se debe ejecutar solo una instrucción. En caso de haber más de una, el uso de llaves es necesario, ya que permiten resolver cualquier ambigüedad.

El código sería el siguiente:

```
1 package prueba;  
2  
3 import java.util.Scanner;  
4  
5 public class Prueba {  
6     public static void main(String[] args) {  
7         Scanner entrada = new Scanner(System.in);  
8         int edad;
```



```
9      System.out.print("Ingresá tu edad: ");
10     edad = entrada.nextInt();
11     if (edad < 18) {
12         System.out.println("No podés ingresar");
13     }
14     else {
15         System.out.println("Te damos la bienvenida, ¡pasá!");
16     }
17     System.out.println("Suerte"); // Se escribe siempre
18 }
19 }
```

Código 25: Algoritmo que valida o no un pase según la edad.

## Bloques de selección anidados

Hay casos donde cierto problema requiere más de dos posibilidades, como, por ejemplo, un algoritmo que pida al usuario ingresar un valor numérico entero positivo, e informe a qué momento

No puede haber instrucciones entre medio de los bloques `if {}` y `else {}`.

del día pertenece. Los números deben estar comprendidos en el rango de 0 a 23, ambos inclusive. El momento del día se muestra según el siguiente criterio:

- 0 a 11: Mañana
- 12: Mediodía
- 13 a 19: Tarde
- 20 a 23: Noche

Cualquier valor fuera de ese rango debe mostrarle al usuario en la pantalla un mensaje de error.

Es evidente que es necesaria más de una condición para resolver el problema.

### Primera forma: Bloques dentro de otros

Una de las maneras es ir acomodando bloques de selección dentro de otros, como sigue:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner (System.in);
```

```
8      int hora;
9      System.out.print("Ingresa una hora del día: ");
10     hora = entrada.nextInt();
11     if (hora < 0) {
12         System.out.println("La hora debe ser mayor que 0");
13     }
14     else {
15         if (hora <= 11) {
16             System.out.println("Es la mañana");
17         }
18         else {
19             if (hora == 12) {
20                 System.out.println("Es el mediodía");
21             }
22             else {
23                 if (hora <= 19) {
24                     System.out.println("Es la tarde");
25                 }
26                 else {
27                     if (hora <= 23) {
28                         System.out.println("Es la noche");
29                     }
30                     else {
31                         System.out.println("Hora debe ser menor que 23");
32                     }
33                 }
34             }
35         }
36     }
37 }
38 }
```

**Código 26:** Algoritmo que determina la hora del día utilizando bloques de selección dentro de otros.

Verás que a medida que avanza el número de condiciones, el código se va corriendo cada vez más a la derecha, tornándolo más difícil de leer. Además, se debe tener especial cuidado en que todas las llaves abran y cierren correctamente cada bloque.

## Segunda forma: Bloques anidados

Muchos programadores aprovechan que Java permite abrir un nuevo bloque `if` inmediatamente después de un `else`, haciendo el código más compacto:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner (System.in);
8         int hora;
9         System.out.print("Ingresá una hora del día: ");
10        hora = entrada.nextInt();
11        if (hora < 0) {
12            System.out.println("La hora debe ser mayor que 0");
13        }
14        else if (hora <= 11) {
15            System.out.println("Es la mañana");
16        }
17        else if (hora == 12) {
18            System.out.println("Es el mediodía");
19        }
20        else if (hora <= 19) {
21            System.out.println("Es la tarde");
22        }
23        else if (hora <= 23) {
24            System.out.println("Es la noche");
25        }
26        else {
27            System.out.println("La hora debe ser menor que 23");
28        }
29    }
30 }
```

Código 27: Algoritmo que determina la hora del día utilizando bloques de selección anidados.

Si hubiera dejado la expresión sin los paréntesis, la computadora intentará primero evaluar el término **!2**. ¿Qué es lo contrario de un dos? No tiene sentido. Es un error.

## Operadores lógicos

Son aquellos que permiten operar con expresiones booleanas. A través de ellos, pueden realizarse condiciones más complejas que no requieran un anidamiento de estructuras de selección.

Así como los **operadores aritméticos** tienen como operandos a **números**, los **operadores lógicos** tienen como operandos a **valores lógicos** (**true** o **false**).

Así como los **operadores aritméticos** devuelven como resultado un **número**, los **operadores lógicos** devuelven como resultado un **valor lógico** (**true** o **false**).

| Operador | Nombre          |
|----------|-----------------|
| !        | NO lógico (NOT) |
| &&       | Y lógico (AND)  |
|          | O lógico (OR)   |

Cada uno de estos operadores tiene su tabla de verdad, es decir, una representación de todas las combinaciones posibles con sus correspondientes resultados. A continuación, te muestro en detalle cada operador.

### Operador NOT

El operador **NOT**, también llamado **NO** o **negación lógica**, es un operador **monádico**. Esto significa que trabaja con un solo operando. Lo que hace este operador es negar la expresión que se encuentra a su derecha. Se representa con el símbolo **!**.

Probá el siguiente código:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         System.out.println(2 < 3); Devuelve true
6         System.out.println(!(2 < 3)); Devuelve false
7         System.out.println(!2 < 3); // Error: No puede negar el 2
8     }
9 }
```

#### Código 28: Demostración del operador NOT.

La tabla de verdad del operador **NOT** es la siguiente:

| Expresión booleana  | Resultado          |
|---------------------|--------------------|
| <code>!true</code>  | <code>false</code> |
| <code>!false</code> | <code>true</code>  |

## Operador OR

El operador **OR**, también llamado **O** o **disyunción lógica**, trabaja con dos operandos. Lo que hace este operador es devolver un valor `true`, si **al menos un operando** es `true`, en caso contrario devuelve `false`. Se representa con el símbolo `||`.

Es importante que uses el operador `||`. En Java, usar un solo `|` implica querer realizar un **OR de bajo nivel**, es decir bit a bit, cuestión que no trataré, pero cuyo resultado no es el mismo.

Para formalizar, la tabla de verdad del operador **OR** es la siguiente:

| Expresión booleana          | Resultado          |
|-----------------------------|--------------------|
| <code>true    true</code>   | <code>true</code>  |
| <code>true    false</code>  | <code>true</code>  |
| <code>false    true</code>  | <code>true</code>  |
| <code>false    false</code> | <code>false</code> |

## Operador AND

El operador **AND**, también llamado **Y** o **conjunción lógica**, trabaja con dos operandos. Lo que hace

Es importante que uses el operador `&&`. En Java, usar un solo `&` implica querer realizar un **AND de bajo nivel**, es decir bit a bit, cuestión que no trataré, pero cuyo resultado no es el mismo.

este operador es devolver un valor `true`, si **todos los operandos** resultan `true`, en caso contrario devuelve `false`. Se representa con el símbolo `&&`.

Para formalizar, la tabla de verdad del operador **AND** es la siguiente:

| Expresión booleana                  | Resultado          |
|-------------------------------------|--------------------|
| <code>true &amp;&amp; true</code>   | <code>true</code>  |
| <code>true &amp;&amp; false</code>  | <code>false</code> |
| <code>false &amp;&amp; true</code>  | <code>false</code> |
| <code>false &amp;&amp; false</code> | <code>false</code> |



## Ejemplo de aplicación

El algoritmo que informa el momento del día según un número puede ser realizado con operadores lógicos, quedando aún más compacto:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner (System.in);
8         int hora;
9         System.out.print("Ingresá una hora del día: ");
10        hora = entrada.nextInt();
11        if (hora < 0 || hora > 23) {
12            System.out.println("La hora es incorrecta");
13        }
14        else if (hora <= 11) {
15            System.out.println("Es la mañana");
16        }
17        else if (hora == 12) {
18            System.out.println("Es el mediodía");
19        }
20        else if (hora <= 19) {
21            System.out.println("Es la tarde");
22        }
23        else if (hora <= 23) {
24            System.out.println("Es la noche");
25        }
26    }
27 }
```

Código 29: Algoritmo que determina la hora del día con operadores lógicos.

El bloque que sigue permite saber si la hora ingresada es correcta o no utilizando un operador `||`.

```
if (hora < 0 || hora > 23) {
    System.out.println("La hora es incorrecta");
}
```

También pudo haberse reescrito utilizando un operador `&&` y un operador `!`, de la siguiente manera:

```
if ( !(hora > 0 && hora < 23) ) {
    System.out.println("La hora es incorrecta");
}
```

## Jerarquía de operadores

Ya que han sido analizados todos los operadores que usaré en este libro, es importante que notes el orden en que se evalúan cada uno de ellos. Recordá que, para alterar este orden, podés hacer uso de paréntesis en cualquier expresión, en caso contrario, se evaluará de la siguiente manera:

| Operadores                         | Nombres  |
|------------------------------------|--|
| <code>()</code>                    | Paréntesis.  |
| <code>++ -- !</code>               | Incremento, decremento y NOT lógico.   |
| <code>* / %</code>                 | Multiplicación, división y módulo.   |
| <code>+ -</code>                   | Suma y resta.  |
| <code>&lt; &lt;= &gt; &gt;=</code> | Menor, menor o igual, mayor y mayor o igual.   |
| <code>== !=</code>                 | Igual y distinto.  |
| <code>&amp;&amp;</code>            | AND lógico.  |
| <code>  </code>                    | OR lógico.   |
| <code>= += -= *= /= %=</code>      | Asignación, suma y asignación, resta y asignación, multiplicación y asignación, división y asignación y módulo y asignación. |

## Bloque de selección múltiple switch

Suponé que deseás realizar un menú de opciones, como cuando llamás por teléfono y la voz del otro lado te dice: "Para ventas, presione 1. Para pagos, presione 2. Para servicio técnico, presione 3...". Realizar un menú con, por ejemplo, diez opciones diferentes, requiere, como habrás visto, realizar un anidamiento de estructuras de selección que puede tornarse inmanejable, engorroso y poco legible.

Para estos casos, se utiliza una estructura de selección múltiple, pero con ciertas restricciones: solo es posible su uso en los casos donde el dato a evaluar sea de un tipo entero. Un tipo `double` no sería válido, pues entre un valor `double` y otro, hay infinitos valores intermedios. Además, la evaluación se realiza únicamente por comparación (es decir, cada opción tiene un número concreto, no un rango de valores).

La sintaxis es la siguiente:

```
switch (<variable_tipo_entera>) {  
    case 1:  
        <instrucciones>  
        break;  
    case 2:  
        <instrucciones>  
        break;  
    ...  
    case n:  
        <instrucciones>  
        break;  
    default:  
        <instrucciones>  
}
```

Donde:

- La palabra **switch** es obligatoria. Indica el comienzo de un bloque de selección múltiple.
- **<variable\_tipo\_ordinal>** es el número entero que la máquina evaluará. La variable puede ser de tipo **byte**, **short**, **int** o **char**.
- Las llaves de apertura y cierre delimitan las instrucciones del bloque de selección múltiple.
- La palabra **case** es una etiqueta que indica en qué punto se deben comenzar a ejecutar instrucciones para un caso particular. Toda palabra **case** va seguida de un valor del mismo tipo que la variable evaluada, que el programador espera que pueda llegar a tener. A continuación, se escribe el carácter **:** y se listan las instrucciones.
- **<instrucciones>** son las instrucciones que se ejecutarán para cada caso. Puede ser solo una, varias o inclusive otros bloques de selección o repetición, siguiendo los conceptos vistos hasta aquí.
- La instrucción **break** permite terminar el bloque de selección de manera prematura, haciendo que el flujo continúe con las instrucciones que hubiera por debajo de la llave de cierre del **switch**. Sin la instrucción **break**, se intentarían ejecutar secuencialmente todos los **case** listados a continuación de la línea actual.
- Cada bloque **case**, junto al valor y las instrucciones pueden repetirse indefinidamente por cada caso diferente que se desee evaluar.
- La etiqueta **default** es opcional y siempre se escribe al final. Permite realizar instrucciones en caso de que el valor de la variable evaluada no coincida con ningún **case**. Al ser el último caso, el uso de **break** no es necesario.

A continuación, te muestro cómo realizar el menú, con las siguientes opciones:

1. Ventas.

2. Pagos.
3. Servicio técnico.
4. Gerencia.

El código:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner (System.in);
8         int num;
9         System.out.println("MENU DE OPCIONES");
10        System.out.println("[1] Ventas");
11        System.out.println("[2] Pagos");
12        System.out.println("[3] Servicio técnico");
13        System.out.println("[4] Gerencia");
14        System.out.print("Elegí tu opción: ");
15        num = entrada.nextInt();
16        switch (num) {
17            case 1:
18                System.out.println("Elegiste ventas");
19                break;
20            case 2:
21                System.out.println("Elegiste pagos");
22                break;
23            case 3:
24                System.out.println("Elegiste servicio técnico");
25                break;
26            case 4:
27                System.out.println("Elegiste gerencia");
28                break;
29            default:
30                System.out.println("Incorrecto");
31        }
```

```
32     }  
33 }
```

### Código 30: Menú de opciones con switch.

Como verás, queda mucho más elegante que usar estructuras de selección doble anidadas.

Como la etiqueta default es opcional, no haberla usado implicaría que, si el usuario ingresa un número distinto de 1, 2, 3 y 4, sencillamente continúan las instrucciones por debajo de la llave de cierre del **switch**.

Poner una variable no entera (como un **double** o un **boolean**) para ser evaluada por un bloque **switch** derivará en un error de compilación.

## Flujo de repetición

Tanto el flujo secuencial como el flujo de selección tienen en común que las instrucciones siempre siguen un curso hacia adelante, es decir, una vez que una instrucción fue ejecutada, jamás volverá a ejecutarse, a no ser, claro está, que el programador la repita más adelante.

Hay muchas situaciones donde se requiere repetir una o más instrucciones un número definido o no definido de veces, pero queriendo evitar que las instrucciones pertinentes sean repetidas en el código.

Las estructuras de repetición nos permiten solucionar esto. Se denominan **ciclos**, **bucles** o **loops**. Un ciclo puede tener una o más repeticiones, las cuales normalmente se denominan **vuelatas** o **iteraciones**.

Hay tres componentes que deben ser correctamente analizados para poder efectuar ciclos:

- **Inicialización:** Cómo será el valor inicial de la(s) variable(s) en la condición.
- **Condición:** Qué tiene que ocurrir para que el ciclo continúe su ejecución. Debe tener al menos una variable involucrada, de lo contrario, el resultado sería constante.
- **Actualización:** La(s) variable(s) en la condición debe(n) cambiar su valor por cada iteración para que el resultado de la **condición** no sea siempre constante.

Java cuenta con tres formas de realizar repeticiones, entre otras:

- **for**
- **while**
- **do...while**

En general, el bloque **for** para ciclos controlados por contador y los bloques **while** y **do...while** para ciclos controlados por bandera.

### Bloque de repetición for

El bloque de repetición **for** permite formular ciclos controlados por contador de manera elegante y compacta. En la cabecera del bloque, se establecen la **inicialización**, la **condición** y la **actualización**, permitiendo que dentro del bloque solo queden las instrucciones a ser repetidas.

La sintaxis es la siguiente:

```
for (<inicializacion> ; <condición> ; <actualizacion>) {  
    <instrucciones>  
}
```

Donde:

- La palabra **for** es obligatoria. Indica el comienzo de un bloque de repetición.
- **<inicializacion>** es la sentencia que establece una variable que oficie de contadora y le dé un valor inicial.
- **<condición>** es la expresión booleana que la computadora evaluará. Mientras resulte **true**, se ejecutarán las instrucciones dentro del **for**.



- **<actualización>** es la sentencia que establece el valor de paso, es decir, cómo se cuenta. Por cada fin de iteración, se ejecuta la sentencia de actualización.
- Las llaves de apertura y cierre delimitan las instrucciones del bloque de repetición.
- **<instrucciones>** son las instrucciones que se ejecutarán por cada iteración. Puede ser solo una, varias o inclusive otros bloques de selección o repetición, siguiendo los conceptos vistos hasta aquí.

El siguiente ejemplo muestra un mensaje diez veces, junto al número de iteración correspondiente:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         for (int i = 1; i <= 10; i++) {
6             System.out.println("Mensaje " + i + " de " + 10);
7         }
8     }
9 }
```

Código 31: Salida por consola repetida 10 veces.

Es muy recomendable que te acostumbres a inicializar los contadores con el valor 0. En próximos temas verás **arreglos**, cuyos **índices** siempre se cuentan desde 0.

El siguiente código realiza la misma salida, pero utilizando un contador desde 0 hasta 9:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         for (int i = 0; i < 10; i++) {
6             System.out.println("Mensaje " + (i+1) + " de " + 10);
7             /* (i+1) no actualiza, sino que le muestra al usuario el
8                contador incrementado. Gracias a eso, el usuario observa
9                valores de 1 a 10, pero la máquina los cuenta de 0 a 9.
10            */
11         }
12     }
13 }
```

Código 32: Salida por consola repetida 10 veces, comenzando a contar desde 0.

## Bloque de repetición while

El bloque de repetición **while** permite formular cualquier tipo de ciclo, aunque generalmente se lo utiliza para realizar un número de iteraciones indefinidas, es decir, por bandera.

En un bloque **while**, la condición se evalúa el principio.

La sintaxis es la siguiente:

```
while (<condición>) {  
    <instrucciones>  
}
```

Donde:

- La palabra **while** es obligatoria. Indica el comienzo de un bloque de repetición.
- **<condición>** es la expresión booleana que la computadora evaluará. Mientras resulte **true**, se ejecutarán las instrucciones dentro del **while**.
- Las llaves de apertura y cierre delimitan las instrucciones del bloque de repetición.
- **<instrucciones>** son las instrucciones que se ejecutarán por cada iteración. Puede ser solo una, varias o inclusive otros bloques de selección o repetición, siguiendo los conceptos vistos hasta aquí.

El siguiente ejemplo pide números al usuario mientras no se ingrese el valor de corte, que es el 0. Finalmente, el programa muestra la suma de los números ingresados:

```
1 package prueba;  
2  
3 import java.util.Scanner;  
4  
5 public class Prueba {  
6     public static void main(String[] args) {  
7         Scanner entrada = new Scanner(System.in);  
8         int acumulador;  
9         int bandera;  
10        int num;  
11        acumulador = 0; // La inicialización del acumulador  
12        bandera = 0; // Valor de corte  
13        num = 1; // Puede ser cualquiera distinto del corte  
14        while (num != bandera) { // La condición  
15            System.out.print("Número (Para terminar, un 0): ");  
16            num = entrada.nextInt();  
17            acumulador = acumulador + num; // Act. del acumulador
```

```
18     }
19     System.out.println("La suma es " + acumulador);
20 }
21 }
```

Código 33: Sumatoria de números ingresados por el usuario con **while**.

La siguiente línea:

```
num = 1; // Puede ser cualquiera distinto del corte
```

Es necesaria, pues como la condición se evalúa al principio, **num** debe tener un valor definido para poder realizar la primera comparación.

## Bloque de repetición **do...while**

El bloque de repetición **do...while** permite formular cualquier tipo de ciclo, aunque generalmente se lo utiliza para realizar un número de iteraciones indefinidas, es decir, por bandera.

En un bloque **do...while**, la condición se evalúa al final.

La sintaxis es la siguiente:

```
do {
    <instrucciones>
} while (<condición>;
```

Donde:

- La palabra **do** es obligatoria. Indica el comienzo de un bloque de repetición.
- **<instrucciones>** son las instrucciones que se ejecutarán por cada iteración. Puede ser solo una, varias o inclusive otros bloques de selección o repetición, siguiendo los conceptos vistos hasta aquí.
- Las llaves de apertura y cierre delimitan las instrucciones del bloque de repetición.
- La palabra **while** es obligatoria. Indica el final del bloque de repetición.
- **<condición>** es la expresión booleana que la computadora evaluará. Mientras resulte **true**, se ejecutarán las instrucciones dentro del **while**. El **;** es necesario, para resolver ambigüedades.

No poner el **;** tras escribir la condición, implica para el compilador que se está intentado definir el inicio de un bloque **while** y no el final de un bloque **do...while**, produciéndose un error de compilación.

Muchas veces, un código realizado con **while** termina siendo casi idéntico que haberlo realizado con **do...while**. Como en este bloque, la condición se evalúa al principio, no necesito inicializar el valor **num** con un valor distinto del corte. En los bloques **do...while**, se asegura que el ciclo va a iterar al menos una vez:

```
1 package prueba;
```

```
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8         int acumulador;
9         int bandera;
10        int num;
11        acumulador = 0; // La inicialización del acumulador
12        bandera = 0; // Valor de corte
13        do {
14            System.out.print("Número (Para terminar, un 0): ");
15            num = entrada.nextInt();
16            acumulador = acumulador + num; // Act. del acumulador
17        } while (num != bandera); // La condición
18        System.out.println("La suma es " + acumulador);
19    }
20 }
```

Código 34: Sumatoria de números ingresados por el usuario con do...while.

## Ámbito de las variables

En Java, no todas las variables son accesibles desde cualquier lugar, sino que depende del contexto en el cual han sido declaradas. A esto se lo conoce como **ámbito** (en inglés, "scope").

Voy a ilustrar esto con un ejemplo. El siguiente programa muestra lo que ocurre cuando se intenta acceder a una variable declarada dentro de un bloque de selección:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8         int num;
9         System.out.print("Ingresa un número: ");
10        num = entrada.nextInt();
11        if (num > 0) {
12            String cadena = "Es positivo";
13        }
14        System.out.println(cadena); // No compila
15    }
16 }
```

**Código 35: Intento de acceso a una variable declarada dentro de un bloque.**

Esto ocurre porque **cadena** fue declarada dentro del bloque **if**, por lo que solo es accesible desde dentro del bloque en cuestión y otros que pudiera haber dentro. Esto aplica para variables que se declaren dentro de cualquier bloque **if**, **else**, **switch**, **for**, **while** o **do...while**.

Para solucionar el programa anterior, tendría que haber declarado la variable **cadena** fuera del bloque **if**:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
```

```
8      int num;
9      String cadena = ""; // La inicializo porque puede no entrar al if
10     System.out.print("Ingresa un número: ");
11     num = entrada.nextInt();
12     if (num > 0) {
13         cadena = "Es positivo";
14     }
15     System.out.println(cadena);
16 }
17 }
```

### Código 36: Declaración de la variable fuera del bloque.

Al haber sido declarada al principio de la función `main`, la variable `cadena` es accesible desde cualquier lugar dentro del método.



## Modularización

El concepto de **modularización** es uno de los más importantes de la programación.

Existen programas complejos, que realizan muchas y variadas operaciones de cómputo, quizá varias veces con diferentes datos. Es mucho más fácil diseñar, codificar, depurar y mantener un programa creado a través de la unión de pequeñas piezas de código independientes entre sí que intentar tratarlo como un todo.

La idea a partir de aquí es realizar programas de manera modularizada, es decir, separando cada proceso en bloques independientes que luego "unirán fuerzas" para lograr el objetivo inicial.

Conocer y comprender esta metodología, te permitirá no tan solo tener un código mucho más legible y ordenado, sino también ahorrar mucho tiempo y esfuerzo mental, dado que muchas veces se pueden reutilizar bloques ya testeados de otros programas creados con anterioridad, sin tener que volver a pensarlos.

Como habrás visto, Java incorpora la clase **Math** para abstraer al programador de los complejos pasos necesarios para realizar operaciones matemáticas más allá de las fundamentales. Pero no es la única: podrás encontrar clases y librerías con funcionalidades para manejar fechas y horas, cadenas de caracteres, archivos, bases de datos, interfaces gráficas y otras posibilidades.

## Procedimientos

Un procedimiento es un bloque de código que puede o no recibir datos de entrada, pero que **no retorna resultados**. Su objetivo es permitir "sacar factor común" de instrucciones que se repitan frecuentemente en el código, trasladando en cierto momento el flujo hacia el procedimiento, y volviendo al punto desde donde se invocó cuando se terminan de ejecutar las instrucciones dentro de él.

Los procedimientos se dividen en dos partes: la **definición** y la **invocación**.

La **definición de un procedimiento** es justamente crearlo. Ponerle nombre, establecer qué datos espera recibir y escribir las instrucciones que deberían realizarse. **La definición de un procedimiento se realiza una sola vez.**

El código que se haya definido en un procedimiento no va a ser ejecutado hasta que no se haga un pedido exclusivo.

La **invocación de un procedimiento** permite enviarle los datos necesarios al mismo, para que ejecute las operaciones que hay en su definición y, por tanto, provea el resultado esperado. **La invocación de un procedimiento se realiza cada vez que sea necesario.**

Voy a mostrarte un ejemplo concreto con poca utilidad práctica pero que ilustra perfectamente el concepto anterior. El siguiente programa imprime por consola la letra original de la canción "We Will Rock You", de **Queen**:

```
1 package prueba;  
2  
3 public class Prueba {  
4     public static void main(String[] args) {  
5         System.out.println("Buddy, you're a boy, make a big noise");  
6         System.out.println("Playing in the street, gonna be a big man some day");
```

```
7      System.out.println("You got mud on your face, you big disgrace");
8      System.out.println("Kicking your can all over the place");
9      System.out.println(""); // Línea en blanco
10     System.out.println("We will, we will rock you");
11     System.out.println("We will, we will rock you");
12     System.out.println(""); // Línea en blanco
13     System.out.println("Buddy, you're a young man, hard man");
14     System.out.println("Shouting in the street, gonna take on the world some day");
15     System.out.println("You got blood on your face, you big disgrace");
16     System.out.println("Waving your banner all over the place");
17     System.out.println(""); // Línea en blanco
18     System.out.println("We will, we will rock you");
19     System.out.println("We will, we will rock you");
20     System.out.println(""); // Línea en blanco
21     System.out.println("Buddy, you're an old man, poor man");
22     System.out.println("Pleading with your eyes, gonna make you some peace some day");
23     System.out.println("You got mud on your face, you big disgrace");
24     System.out.println("Somebody better put you back into your place");
25     System.out.println(""); // Línea en blanco
26     System.out.println("We will, we will rock you");
27     System.out.println("We will, we will rock you");
28     System.out.println(""); // Línea en blanco
29     System.out.println("We will, we will rock you");
30     System.out.println("We will, we will rock you");
31 }
32 }
```

### Código 37: Programa que imprime la letra de una canción.

Habrás notado que tanto en esta como en varias otras canciones, hay algo que siempre se repite: el estribillo.

La idea entonces es no repetir código de forma redundante, sino buscar la manera de definir por única vez cómo mostrar el estribillo y luego reemplazar las ocurrencias de este por una invocación a un procedimiento llamado `mostrarEstribillo()`.

## Definición de un procedimiento

Las definiciones de los procedimientos se escriben fuera de la función `main()` pero dentro de la clase **Prueba**, como ilustra el siguiente código:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         // Todo el código hasta ahora iba obligatoriamente aquí
6     } // Llave de cierre de main
7     // Aquí se insertan las definiciones de los procedimientos
8 } // Llave de cierre de la clase
```

### Código 38: Dónde se escribe la definición de un procedimiento.

La sintaxis básica para definir un procedimiento es la siguiente:

```
static void <nombre_procedimiento> (<tipo> <nom>, <tipo> <nom>, ...) {
    <instrucciones>
}
```

Donde:

- La palabra **static** es obligatoria. Es una palabra que cobra sentido cuando se habla del paradigma orientado a objetos. Como en este caso estoy intentando emular el paradigma estructurado, tan solo escribirla aunque no la comprendas.
- **void** es un tipo de dato especial que significa **vacío**. Indica que el procedimiento no devuelve ningún valor.
- **<nombre\_procedimiento>** es el nombre que se le dará al procedimiento, siguiendo las mismas reglas de nomenclatura que las variables.
- Los paréntesis son obligatorios.
- El par **<tipo>** y **<nom>** declaran que se recibe un dato de nombre **<nom>** cuyo tipo de dato es **<tipo>**. A ese dato se lo denomina **parámetro**. Si un procedimiento recibe más de un parámetro, se separa cada par con comas. Si un procedimiento no recibe parámetros, los paréntesis quedan vacíos.
- La cantidad, orden y tipos de los parámetros dependen del criterio del programador.
- Las llaves de apertura y cierre delimitan las instrucciones del procedimiento.
- **<instrucciones>** son las instrucciones que se ejecutarán cuando se invoque al procedimiento.

Por convención, los procedimientos se suelen nombrar de manera tal que describan su funcionamiento. La utilización de verbos es bastante conveniente, pues los procedimientos realizan acciones.

Voy a crear un procedimiento llamado **mostrarEstribillo()**, sin parámetros, que establezca lo que su propio nombre indica:

```
1 package prueba;
```

```
2
3 public class Prueba {
4     public static void main(String[] args) {
5         // Toda la canción va escrita aquí
6     }
7
8     static void mostrarEstribillo () {
9         for (int i = 0; i < 2; i++) {
10             System.out.println("We will, we will rock you");
11         }
12     }
13 }
```

Código 39: Definición de un procedimiento.

## Invocación de un procedimiento

Para realizar las instrucciones listadas en un procedimiento, es necesario invocarlo.

La sintaxis básica para invocar un procedimiento es la siguiente:

```
<nombre_procedimiento>(<arg>, <arg>, ...)
```

Donde:

- **<nombre\_procedimiento>** es el nombre del procedimiento que se desea invocar. **Debe haber sido definido previamente.**
- Los paréntesis son obligatorios.
- **<arg>** es el dato requerido por el procedimiento que se desea invocar. En este contexto, se denomina **argumento**. Si un procedimiento requiere más de un argumento, se separan con comas. Si un procedimiento no requiere argumentos, los paréntesis quedan vacíos.

El siguiente código muestra la letra de la canción, utilizando invocaciones a **mostrarEstribillo()** donde corresponda:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         System.out.println("Buddy, you're a boy, make a big noise");
6         System.out.println("Playing in the street, gonna be a big man some day");
7         System.out.println("You got mud on your face, you big disgrace");
8         System.out.println("Kicking your can all over the place");
9         System.out.println(""); // Línea en blanco
```

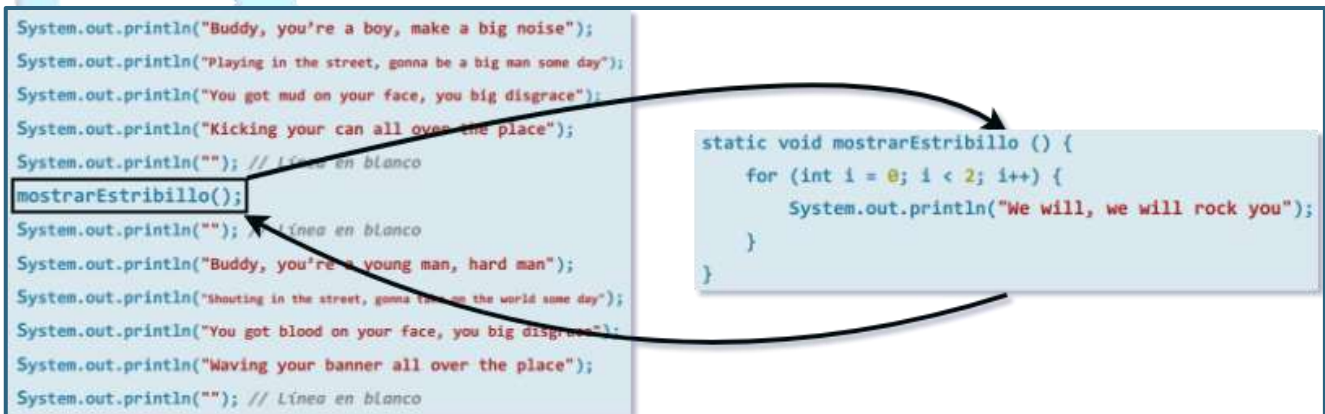
```

10  mostrarEstribillo();
11  System.out.println(""); // Línea en blanco
12  System.out.println("Buddy, you're a young man, hard man");
13  System.out.println("Shouting in the street, gonna take on the world some day");
14  System.out.println("You got blood on your face, you big disgrace");
15  System.out.println("Waving your banner all over the place");
16  System.out.println(""); // Línea en blanco
17  mostrarEstribillo();
18  System.out.println(""); // Línea en blanco
19  System.out.println("Buddy, you're an old man, poor man");
20  System.out.println("Pleading with your eyes, gonna make you some peace some day");
21  System.out.println("You got mud on your face, you big disgrace");
22  System.out.println("Somebody better put you back into your place");
23  System.out.println(""); // Línea en blanco
24  mostrarEstribillo();
25  System.out.println(""); // Línea en blanco
26  mostrarEstribillo();
27  }
28
29  static void mostrarEstribillo () {
30      for (int i = 0; i < 2; i++) {
31          System.out.println("We will, we will rock you");
32      }
33  }
34  }

```

**Código 40:** Programa que imprime la letra de una canción utilizando un procedimiento.

La siguiente imagen ilustra lo que el programa realiza:



**Ilustración 15:** Flujo del programa al invocar un procedimiento.

La salida del programa muestra exactamente lo mismo, pero el código es más mantenible. Suponé que deseás agregar un punto final a cada uno de los "We will, we will rock you". Solo tenés que hacer el cambio en el procedimiento `mostrarEstrillo()`. De la manera anterior (sin usar procedimientos) tendrías que haber puesto un punto por cada una de las ocho veces que se repite la salida de la cadena.

## Procedimiento con parámetros

El siguiente ejemplo muestra cómo realizar un procedimiento que reciba parámetros. El mismo elabora y escribe una presentación en la consola de acuerdo con los datos del usuario que provienen como parámetros, los cuales son el nombre, la edad, el género y si tiene o no hijos. Una vez definido el procedimiento, a través de la función `main()` haré algunas pruebas con datos constantes, para ver los resultados:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         presentar("Carlos", 25, 'M', false);
6         presentar("María", 46, 'f', true);
7         presentar("Luis", 14, 'X', false);
8     }
9
10    static void presentar (String nom, int edad, char gen, boolean tieneHijos)
11    {
12        System.out.print("Hola, mi nombre es " + nom + ". ");
13        if (gen == 'f' || gen == 'F') {
14            System.out.print("Soy una mujer. ");
15        }
16        else if (gen == 'm' || gen == 'M') {
17            System.out.print("Soy un hombre. ");
18        }
19        System.out.print("Tengo " + edad + " años y ");
20        if (!tieneHijos) { // Pregunto lo contrario: ¿no tiene hijos?
21            System.out.print("no ");
22        }
23        System.out.println("tengo hijos.");
24    }
25 }
```

Código 41: Procedimiento que presenta a un usuario según sus parámetros.



La siguiente imagen ilustra lo que el programa realiza:

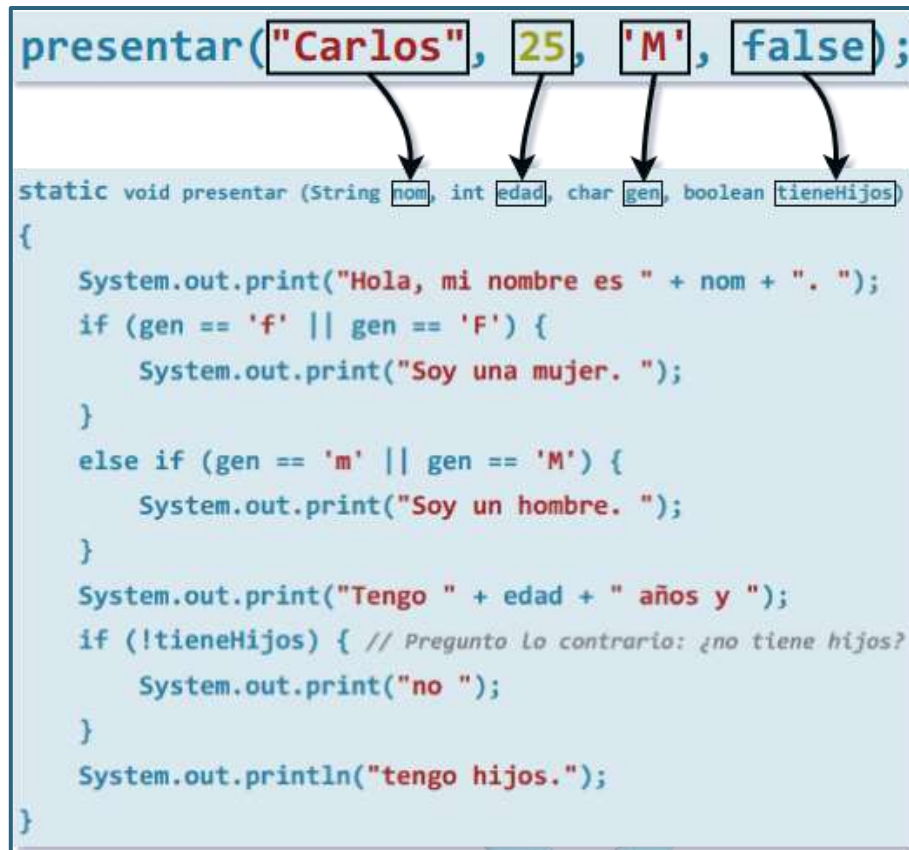


Ilustración 16: Los argumentos de una invocación se convierten en los parámetros de la definición.

El procedimiento `presentar()` no tiene que conocer el origen de los valores de sus parámetros. Pudieron provenir de variables, ser constantes o introducidos por el usuario. No es relevante. El procedimiento tan solo captura los datos que se enviaron cuando se lo invocó, les asigna sus propios nombres y trabaja con ellos.

Es muy importante que a la hora de invocar un procedimiento se respete su "firma", es decir, el orden, el tipo y la cantidad de los argumentos que espera recibir. Así, en el ejemplo anterior, el procedimiento `presentar()` esperaba recibir los siguientes parámetros (los nombres para cada uno quedan a criterio del programador):

`(String nom, int edad, char gen, boolean tieneHijos)`

Por lo tanto, cuando se invoque al procedimiento, se debe respetar tal firma:

`("Carlos", 25, 'M', false)`

`("María", 46, 'f', true)`

`("Luis", 14, 'X', false)`

Cualquier error, ya sea por tipos incompatibles, orden no respetado o cantidad no coincidente, produce que el programa no compile.

El siguiente nivel es permitir que el usuario introduzca sus datos y la computadora imprima la presentación correspondiente:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8         String nombre;
9         int edad;
10        char sexo;
11        int numeroDeHijos;
12        System.out.print("Ingresa tu nombre: ");
13        nombre = entrada.nextLine();
14        System.out.print("Ingresa tu edad: ");
15        edad = entrada.nextInt();
16        entrada.nextLine(); // Limpia el buffer
17        System.out.print("Ingresa tu sexo [m/f]: ");
18        sexo = entrada.next().charAt(0); // Lee el primer caracter
19        System.out.print("Ingresa tu número de hijos: ");
20        numeroDeHijos = entrada.nextInt();
21        if (numeroDeHijos == 0) {
22            presentar(nombre,edad,sexo,false);
23        }
24        else if (numeroDeHijos > 0) {
25            presentar(nombre,edad,sexo,true);
26        }
27        else {
28            System.out.println("Número de hijos inválido");
29        }
30    }
31
32    static void presentar (String nom, int edad, char gen, boolean tieneHijos)
33    {
34        System.out.print("Hola, mi nombre es " + nom + ". ");
35        if (gen == 'f' || gen == 'F') {
36            System.out.print("Soy una mujer. ");
```

```
37     }
38     else if (gen == 'm' || gen == 'M') {
39         System.out.print("Soy un hombre. ");
40     }
41     System.out.print("Tengo " + edad + " años y ");
42     if (!tieneHijos) { // Pregunto lo contrario: ¿no tiene hijos?
43         System.out.print("no ");
44     }
45     System.out.println("tengo hijos.");
46 }
47 }
```

Código 42: Procedimiento que presenta al usuario según datos ingresados por consola.

## Funciones

Una función es un bloque de código que puede o no recibir datos de entrada, pero que **retorna un resultado**. Su objetivo es permitir hacer cálculos precisos mediante sus parámetros de entrada que resultarán en un dato de retorno.

### Definición e invocación de una función

```
static <tipo> <nombre_función> (<tipo> <nom>, <tipo> <nom>, ...) {
    <instrucciones>
    return <valor>;
}
```

La sintaxis básica para definir una función es la siguiente:

Donde:

- La palabra **static** es obligatoria. Es una palabra que cobra sentido cuando se habla del paradigma orientado a objetos. Como en este caso estoy intentando programar emulando el paradigma estructurado, tan solo escribirla aunque no la comprendas.
- **<tipo>** es el de tipo de dato que retornará la función.
- **<nombre\_función>** es el nombre que se le dará a la función, siguiendo las mismas reglas de nomenclatura que las variables.
- Los paréntesis son obligatorios.
- El par **<tipo>** y **<nom>** declaran que se recibe un dato de nombre **<nom>** cuyo tipo de dato es **<tipo>**. A ese dato se lo denomina **parámetro**. Si una función recibe más de un parámetro, se separa cada par con comas. Si una función no recibe parámetros, los paréntesis quedan vacíos.
- La cantidad, orden y tipos de los parámetros dependen del criterio del programador.
- Las llaves de apertura y cierre delimitan las instrucciones de la función.

- **<instrucciones>** son las instrucciones que se ejecutarán cuando se invoque a la función.
- La palabra **return** indica que la función retornará un dato. En cuanto se ejecute la instrucción **return**, la función finaliza y el código retorna al punto desde donde se la invocó. Generalmente es la última instrucción del bloque.
- **<valor>** es el valor que la función devolverá, cuyo tipo debe coincidir con el **<tipo>** a devolver en la definición.

Por convención, las funciones se suelen nombrar de manera tal que describan su funcionamiento. La utilización de verbos es bastante conveniente, pues las funciones realizan acciones para hacer sus cálculos.

Una función que no lleve la instrucción **return** o que tenga un camino donde la ejecución pudiera finalizar sin pasar por un **return** genera un error.

En el siguiente ejemplo, creo la función **calcularFactorial()**, que recibe un número entero como parámetro y devuelve el factorial del número:

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         System.out.println( "3! = " + calcularFactorial(3) );
8         System.out.println( "4! = " + calcularFactorial(4) );
9         System.out.println( "5! = " + calcularFactorial(5) );
10        int num;
11        System.out.println("Ingrese un número: ");
12        num = entrada.nextInt();
13        if (num > 0) {
14            int f = calcularFactorial(num);
15            System.out.println( num + "! = " + f);
16        }
17    }
18
19    static int calcularFactorial (int numero) {
20        int fact = 1;
21        for (int i = 1; i <= numero; i++) {
22            fact *= i;
23        }
24    }
25 }
```

```

24     return fact;
25 }
26 }

```

#### Código 43: Definición e invocación de una función que calcula el factorial.

En las líneas 7, 8 y 9 se invoca a la función `calcularFactorial()` con valores constantes. En la línea 14 se invoca con un número entero introducido por el usuario, el cual debe ser mayor que 0, pues no existe el factorial de un número negativo.

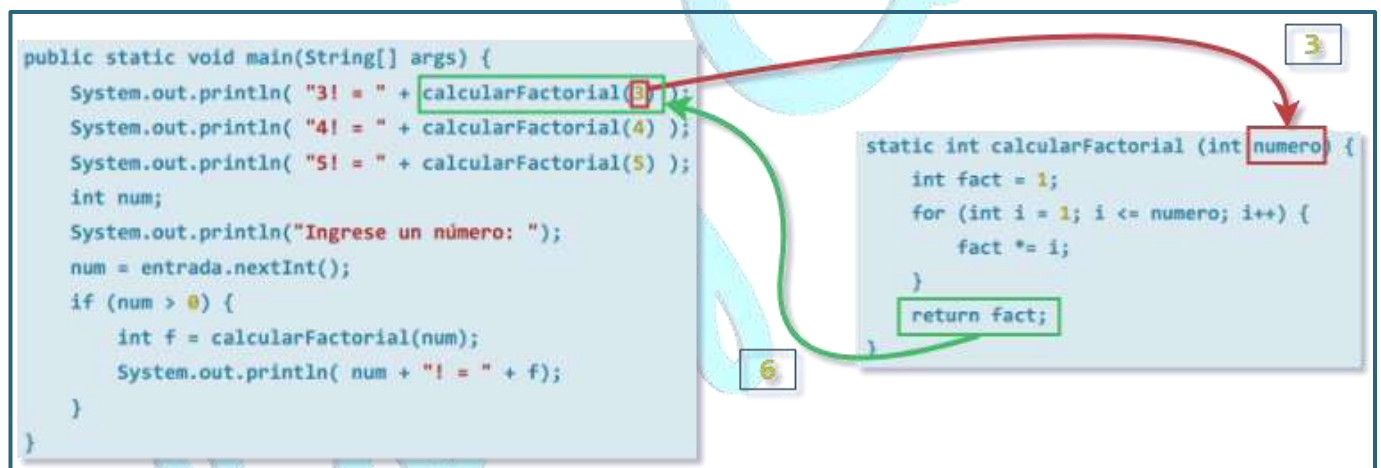
Los nombres de las variables que son argumentos enviados en la invocación de una función no tienen por qué coincidir con los nombres declarados como parámetros en la definición de una función.

En la línea 14 del ejemplo anterior, se envía como argumento el valor de la variable `num`.

En la línea 19 se recibe ese valor como parámetro, el cual pasa a denominarse `numero`.

La función `calcularFactorial()` no tiene que conocer si el valor del parámetro provino de una variable, constante o un valor introducido por el usuario. Tan solo captura los datos que se enviaron en la invocación y les asigna sus propios nombres.

La siguiente imagen ilustra lo que el programa realiza:



#### Ilustración 17: Las funciones se invocan y devuelven un valor.

Es muy importante que a la hora de invocar una función se respete su "firma", es decir, el orden, el tipo y la cantidad de los parámetros que espera recibir, al igual que en los procedimientos.

A partir de aquí, apelaré a la palabra **función** para referirme tanto a aquellos módulos que devuelven datos como aquellos que no.

Podría decir que todo procedimiento en realidad es una función, que retorna `void`.

## Variables locales

Una función no deja de ser una pequeña parte de un programa, completamente independiente, que será invocada por otra que la necesite. En toda función se pueden declarar variables, leer y/o escribir datos en la consola, utilizar estructuras de control de flujo, etc.

Se trata de una "caja negra" independiente de su entorno, por lo tanto, todas las variables adicionales que se pudieran llegar a declarar dentro de ella son **variables locales** a la función. Solo pertenecen dentro del contexto de la función (delimitado por las llaves de apertura y cierre) y no están disponibles fuera.

La siguiente función `calcularFactorial()`:

```
static int calcularFactorial (int numero) {  
    int fact = 1;  
    for (int i = 1; i <= numero; i++) {  
        fact *= i;  
    }  
    return fact;  
}
```

Hace uso de una variable adicional de tipo `int` llamada `fact`. Dicha variable es **local** de la función y no es accesible fuera de ella.

Una variable local no puede tener el mismo nombre que un parámetro.

## Diferencia entre procedimiento, función y método

Como habrás visto, un procedimiento se maneja igual que una función, con la diferencia de que un procedimiento no devuelve nada (tipo de retorno `void`) y una función devuelve (instrucción `return`) un dato. Aunque puedo generalizar y decir que todo procedimiento es una función, con la particularidad de que no devuelve datos.

En la programación estructurada queda bien claro el concepto de función. Pero como Java es un lenguaje orientado a objetos, en dicho paradigma tanto los datos como las funciones se encuentran dentro de una unidad llamada **objeto**, el cual se instancia a partir de una **clase**.

Si bien las funciones dentro de un objeto siguen los mismos conceptos que en el paradigma estructurado, en tal contexto se las llama **métodos**.

Para resumir:

- Una **función** en el paradigma estructurado existe por sí sola y es independiente de los datos.
- Un **método** en el paradigma orientado a objetos pertenece a un objeto en particular.

Por ejemplo, `nextInt()` es un **método** del objeto `entrada` de tipo `Scanner`, que permite devolver un entero ingresado en la consola. En la clase `Math`, existe un **método** llamado `sqrt()` que permite realizar una raíz cuadrada.

Para convenir, a las operaciones que se invocan provistas por Java a través de sus clases y objetos las llamaré como corresponde: **métodos**. A las operaciones que sean definidas por mí las llamaré **funciones**, aunque en realidad se trate de métodos estáticos de la clase que has creado (por defecto, los ejemplos que vengo trabajando se aplican sobre una clase llamada `Prueba`).



## Acerca de main

Habiendo visto algunos conceptos básicos, ahora podrás entender qué significa tener definido por obligación el siguiente bloque dentro de la clase, en los ejemplos, llamada **Prueba**:

```
public class Prueba {  
    public static void main(String[] args) {  
        // Tu código va aquí  
    }  
}
```

Dentro de la clase **Prueba**, excepto por la palabra **public**, que cobra sentido en la programación orientada a objetos, el resto luce como una definición de una función. De hecho, lo es.

Desde el primer programa que imprimía por consola un **"Hola Mundo"** hasta los últimos ejemplos vistos, todo tu código debía ir dentro de **main()**.

En realidad, **main()** (en inglés, **"principal"**) es un **método** de la clase **Prueba**, aunque yo la llamaré **función**, para adecuarme al paradigma estructurado que estoy intentando emular en Java.

Pero, si el bloque anterior es simplemente la definición de lo que debe hacer la función **main()**, y sabiendo que una función no realiza sus acciones hasta que sea invocada, ¿cómo es que el código se ejecuta?

Es la **JVM** quien invoca a la función **main()** automáticamente al hacer ejecución del programa. Es el punto de entrada de toda aplicación Java, desde la más simple hasta la más compleja. Todo nace allí.

Se la declara con tipo de retorno **void** porque no devuelve datos. Recibe un único parámetro, que se trata de un arreglo de **String** llamado por defecto **args** (podrías cambiar el nombre sin problemas). Aún no conoces el concepto de arreglo, por lo que me guardo esa explicación para más adelante.

Hasta aquí nunca has usado el parámetro **args**, pues no lo necesitas. Eso comprueba que una función puede definir que recibe ciertos parámetros, pero nunca hacer uso de ellos. Por supuesto, en tus propias definiciones de funciones esto no tendría sentido práctico. El caso de **main()** es particular.

Toda aplicación comienza y termina en la función **main()**. Ahora que conoces cómo modularizar un programa en partes más pequeñas puede que pierdas un poco el seguimiento del flujo de las instrucciones. Vale la aclaración.

El siguiente código aclara el panorama:

```
1 package prueba;  
2  
3 public class Prueba {  
4     public static void main(String[] args) {  
5         System.out.println("En main, justo antes de invocar");  
6         System.out.println( "5! = " + calcularFactorial(5) );  
7         System.out.println("En main, luego de la invocación");
```

```
8      }
9
10     static int calcularFactorial (int numero) {
11         int fact = 1;
12         for (int i = 1; i <= numero; i++) {
13             fact *= i;
14         }
15         return fact;
16     }
17 }
```

Código 44: Programa que comprueba que toda ejecución comienza y termina en la función main().

## Pasaje de parámetros por valor

Habrás visto que hay casos en los que ciertas funciones o procedimientos requieren de parámetros para funcionar, por lo tanto, cuando se los invoca es necesario respetar esa "firma" y enviar los argumentos correspondientes.

Lo que es importante aclarar es que, a la hora de invocar una función, enviando como argumento el valor de una variable, lo que hace Java es copiar el valor de dicha variable en el parámetro correspondiente en la definición de la función. Si dentro de la definición de la función, el parámetro cambia su valor, tal cambio no se ve reflejado fuera de la misma, porque se está trabajando con una copia. Este comportamiento se denomina **pasaje de parámetros por valor** y es de la única manera en la que Java trabaja.

Otros lenguajes de menor nivel, como C, permiten hacer pasaje de argumentos por referencia. En vez de enviar una copia del valor de la variable, se envía la dirección de memoria de la variable (conocida como puntero o apuntador), por lo que la función podría acceder directamente a la variable y modificar su valor para todo el programa.

El siguiente código te ayudará a entender este concepto:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         int x = 5;
6         System.out.println("Muestro x antes de invocar: " + x);
7         duplicarValor(x);
8         System.out.println("Muestro x luego de invocar: " + x);
9     }
10 }
```

```
11 static void duplicarValor(int numero) {  
12     numero *= 2;  
13 }  
14 }
```

#### Código 45: Demostración del pasaje de parámetros por valor.

Ambas salidas muestran el valor **5**. Por lo que queda demostrado que cuando se invoca en la línea **7** a la función **duplicarValor()** con el argumento **x**, se está enviando una copia del valor de **x**. En la definición de la función **duplicarValor()**, se recibe tal valor como parámetro con el nombre **numero** (pudo también haberse llamado **x**, son independientes, pero no quiero confundir). A pesar de que en la línea **12**, se reemplaza a la variable **numero** con el doble de su valor, tal parámetro es independiente de la variable local **x** en la función **main()**.

## Cuestiones de diseño

### Cohesión

Se denomina **cohesión** a la medición de cuán mínimo es el proceso en un módulo de software. En un buen diseño de soluciones, las funciones deben ser altamente cohesivas, es decir, sus procesos deben estar altamente relacionados entre sí y enfocarse en resolver un problema bien determinado. Por el contrario, una función con baja cohesión intenta resolver un problema más global utilizando instrucciones necesarias entre sí pero no relacionadas.

Una función es altamente cohesiva cuando sus acciones se pueden leer como una oración con un solo verbo. En cuanto se note que se realiza más de una acción, entonces la cohesión es muy baja.

Un claro ejemplo sería una función que permita calcular un promedio de tres notas enteras:

```
static void calcularPromedio () {  
    Scanner entrada = new Scanner(System.in);  
    System.out.print("Ingresa el primer número: ");  
    int n1 = entrada.nextInt();  
    System.out.print("Ingresa el segundo número: ");  
    int n2 = entrada.nextInt();  
    System.out.print("Ingresa el tercer número: ");  
    int n3 = entrada.nextInt();  
    double promedio = (n1 + n2 + n3) / 3.0;  
    System.out.println("El promedio es " + promedio);  
}
```

La función anterior posee **baja cohesión**, pues hay partes del código que se encargan de pedirle datos al usuario, otras que realizan el cálculo del promedio y otras que se encargan de mostrar los resultados por consola.

Una función `calcularPromedio()` altamente cohesiva simplemente recibe los números como parámetros, los procesa y devuelve el resultado a quien la invocó:

```
static double calcularPromedio (int n1, int n2, int n3) {  
    return (n1 + n2 + n3) / 3.0;  
}
```

Las responsabilidades de pedirle los números al usuario y de mostrar el resultado en pantalla deben ser abordadas por otras funciones.

## Acoplamiento

El concepto de **acoplamiento** está muy relacionado con la cohesión, y tiene que ver con la interdependencia entre funciones.

Lo que se busca es tener un programa con **bajo acoplamiento**, es decir, que las funciones componentes sean lo más independientes entre sí. Por el contrario, un programa con alto acoplamiento posee partes muy dependientes de otras. Lograr un bajo acoplamiento de las partes y que las mismas posean alta cohesión, es el ideal de diseño del software.

Continuando con el programa que permite pedir tres números al usuario y calcular su promedio, te muestro un diseño poco conveniente, donde las partes están muy acopladas:

```
1 package prueba;  
2  
3 import java.util.Scanner;  
4  
5 public class Prueba {  
6  
7     public static void main(String[] args) {  
8         Scanner entrada = new Scanner(System.in);  
9         double promedio;  
10        System.out.print("Ingresa el primer número: ");  
11        double n1 = entrada.nextDouble();  
12        System.out.print("Ingresa el segundo número: ");  
13        double n2 = entrada.nextDouble();  
14        System.out.print("Ingresa el tercer número: ");  
15        double n3 = entrada.nextDouble();  
16        promedio = calcularPromedio( (int) n1, (int) n2, (int) n3);  
17        System.out.println("El promedio es " + promedio);  
18    }  
19  
20    static double calcularPromedio(int a, int b, int c) {  
21        return (a + b + c) / 3.0;
```

```
22     }
23 }
```

#### Código 46: Programa demasiado acoplado y con poca cohesión.

La función `main()` está demasiado acoplada con la función `calcularPromedio()`, pues para que `calcularPromedio()` funcione bien, `main()` debe castear los números ingresados en `double` como `int`.

Además, `main()` es muy poco cohesivo: se encarga de pedir los números, castearlos, enviarlos como argumentos a la función `calcularPromedio()` y luego mostrar el resultado.

Un rediseño del programa anterior agrega más funciones que se encarguen de pequeñas soluciones. La función `main()` se encarga de unir todas las partes para cumplir con el objetivo:

```
1  package prueba;
2
3  import java.util.Scanner;
4
5  public class Prueba {
6
7      public static void main(String[] args) {
8          int n1 = obtenerEnteroDesdeConsola("Ingresá el primer número: ");
9          int n2 = obtenerEnteroDesdeConsola("Ingresá el segundo número: ");
10         int n3 = obtenerEnteroDesdeConsola("Ingresá el tercer número: ");
11         double promedio = calcularPromedio(n1, n2, n3);
12         System.out.println("El promedio es " + promedio);
13     }
14
15     static double calcularPromedio(int a, int b, int c) {
16         return (a + b + c) / 3.0;
17     }
18
19     static int obtenerEnteroDesdeConsola (String mensaje) {
20         Scanner entrada = new Scanner(System.in);
21         System.out.print(mensaje);
22         double num = entrada.nextDouble();
23         return (int) num;
24     }
25 }
```

#### Código 47: Programa con funciones altamente cohesivas y poco acopladas.

## Arreglos

Suponé que tenés que realizar un programa que le pida al usuario seis tiempos, en segundos, correspondientes a las vueltas de cierta carrera para que la máquina calcule cuántas vueltas superaron el tiempo promedio de vueltas.

Con lo que conocés, no queda más alternativa que declarar seis variables enteras, cuyos nombres pueden ser **vuelta1**, **vuelta2**, **vuelta3**, **vuelta4**, **vuelta5** y **vuelta6**. Operaciones simples como leer un dato desde consola y asignar a cada variable deberían repetirse seis veces. Necesitás una manera de agrupar en una única variable datos relacionados del mismo tipo.

Entre múltiples y variadas estructuras de datos existentes que permitirían guardarlos, verás la más sencilla: **un arreglo unidimensional**.

Un **arreglo unidimensional** (en inglés, "**array**"), también llamado **vector**, es una estructura de datos que consiste en agrupar elementos del mismo tipo en celdas contiguas, permitiendo acceder a tales celdas, y, por ende, a los datos, mediante un **número de índice**, cuyo primer valor es el **0**.

Es unidimensional porque solo consta de una dimensión, por lo tanto, me referiré a los arreglos unidimensionales directamente como **arreglos**. Existen también los **arreglos bidimensionales** (dos dimensiones), a los cuales se les suele denominar **matrices**.

Para el caso anterior, podrías tener una variable de tipo arreglo de enteros llamada **vuelatas** que guarde cada uno de los tiempos en una celda, como muestra la ilustración:

En Java, los arreglos no son tipos de datos primitivos, sino que son **objetos**.

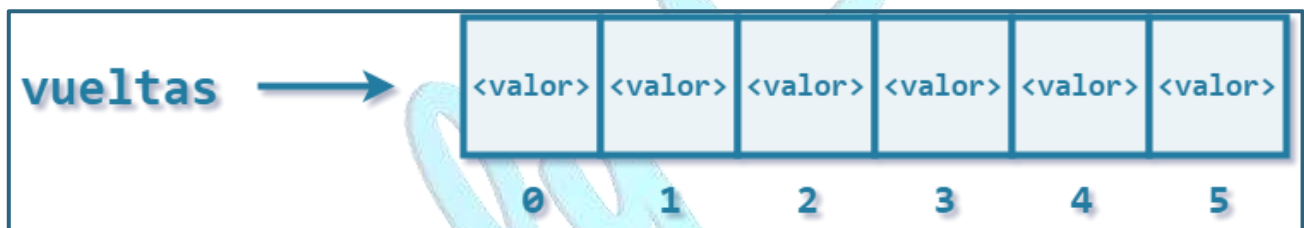


Ilustración 18: Representación abstracta de un arreglo.

A las **variables de tipo objeto** también se las llama de **referencia**, pues en realidad, una variable de tipo objeto no tiene como valor a un objeto en sí sino una referencia a él.

A continuación, te muestro diferentes formas de declarar una variable de tipo arreglo.

## Declaración

### Sin inicializar

La sintaxis para declarar un arreglo es la siguiente:

```
<tipo>[] <identificador>;
```

Donde **<tipo>** es el tipo de dato que junto a los corchetes de apertura y cierre indican que se trata de un conjunto de datos de ese tipo. Como ya sabés, **<identificador>** es el nombre del arreglo (siguiendo el mismo criterio de nomenclatura para cualquier variable).

Por lo general, los arreglos se llaman con un sustantivo plural, pues permiten guardar varios datos similares.

La siguiente instrucción declara un arreglo de enteros llamado **vuelatas**:



```
int[] vueltas;
```

Hasta aquí, **vueltas** está sin inicializar.

## Inicializado por defecto

Para que **vueltas** tenga una referencia a un arreglo, es necesario crear el mismo y asignarlo.

La sintaxis es la siguiente:

```
<tipo>[] <identificador> = new <tipo>[<longitud>];
```

El operador de asignación permite asignar una nueva referencia a un arreglo, que se crea mediante la palabra **new**, seguida del tipo **<tipo>** (debe coincidir con el de la declaración) y un par de corchetes. Dentro de los corchetes se escribe un número **<longitud>** que representa la dimensión del arreglo, es decir, la cantidad de celdas contiguas que poseerá.

Continuando con el ejemplo anterior, voy a crear un arreglo de seis enteros, que será referenciado por la variable **vueltas**.

Puedo declarar y asignar:

```
int[] vueltas = new int[6];
```

También puedo primero declarar en una línea y asignar después:

```
int[] vueltas;  
vueltas = new int[6];
```

Bien, **vueltas** apunta hacia un arreglo de seis valores. Pero ¿cuánto valen esos valores por defecto? La respuesta depende del tipo de dato del cual se definió el arreglo:

- Para arreglos de tipos **int**, **char** y **double** (junto a los otros tipos numéricos que no utilizo), el valor por defecto de cada una de las celdas es **0**.
- Para arreglos de tipo **boolean**, el valor por defecto de cada una de las celdas es **false**.
- Para arreglos de tipos por referencia (por ejemplo, un arreglo de cadenas **String**), el valor por defecto de cada una de las celdas es **null**.

El valor **null** es especial. Indica que la referencia es nula, es decir, no existe. Cualquier intento de hacer operaciones con un valor **null** genera un error.

Por lo tanto, al haber hecho la siguiente declaración y asignación:

```
int[] vueltas = new int[6];
```

El arreglo al que apunta **vueltas** queda así:

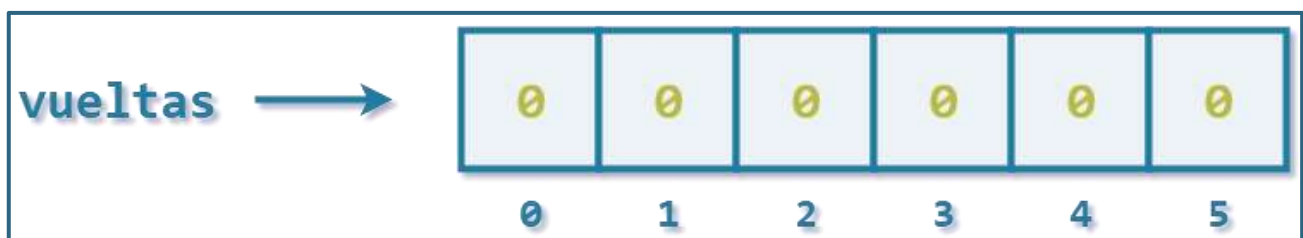


Ilustración 19: Arreglo con valores por defecto.

## Inicializado con valores

Existe una tercera manera de crear un arreglo con valores constantes definidos en el propio código. Esta forma es útil para hacer algunas pruebas sobre el funcionamiento de los arreglos, pero tené en cuenta que, en un programa de uso práctico, los valores debería introducirlos el usuario y no estar escritos estáticamente en el código. Ya verás cómo.

Por ahora, voy a crear un arreglo llamado **vuel**tas con los valores de tiempos **64, 62, 65, 68, 63 y 65**:

```
int[] vuel
```

tas = {64,62,65,68,63,65};

Como habrás observado, en un par de llaves se listan separados por comas los valores deseados. Java asume que se trata de un arreglo de **6** elementos, por lo que implícitamente lo crea con dicha longitud.

El arreglo al que apunta **vuel**tas queda así:

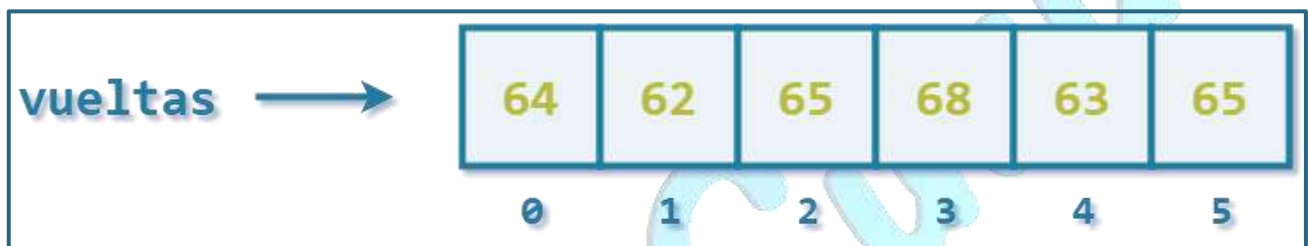


Ilustración 20: Arreglo inicializado con valores.

Crear un arreglo inicializado con valores tiene una particularidad.

Solo funciona si se declara y asigna en la misma línea:

```
int[] vuel
```

tas = {64,62,65,68,63,65};

La siguiente forma no funciona:

```
int[] vuel
```

tas;

```
vuel
```

tas = {64,62,65,68,63,65};

## Operaciones básicas

### Obtener un valor

Para obtener un valor de un arreglo, se debe escribir el nombre de este seguido de un par de corchetes. Dentro de los corchetes, se escribe el número de índice del dato que se desea mostrar (recordando que la cuenta empieza desde **0**).

El siguiente programa muestra los tiempos de la primera, la cuarta y la última vuelta del arreglo **vuel**tas:

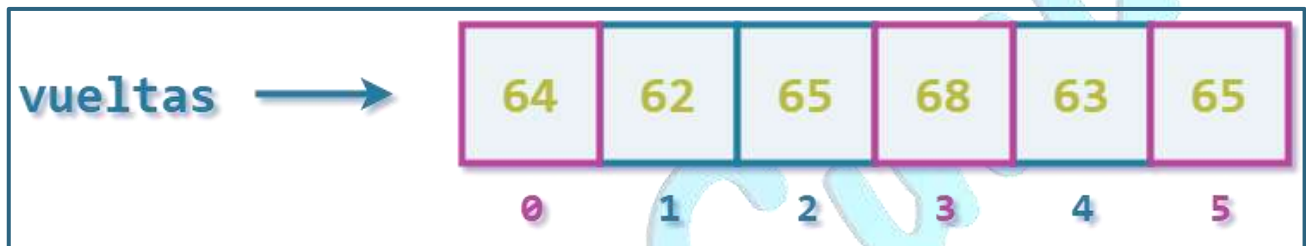
```
1 package prueba;  
2  
3 public class Prueba {  
4     public static void main(String[] args) {
```

```
5      int[] vueltas = {64,62,65,68,63,65};
6      System.out.println("1ª vuelta: " + vueltas[0] );
7      System.out.println("4ª vuelta: " + vueltas[3] );
8      System.out.println("Última vuelta: " + vueltas[5] );
9  }
10 }
```

#### Código 48: Obtención de valores individuales de un arreglo.

Intentar obtener el valor de un arreglo escribiendo un índice fuera de rango, como un valor negativo o mayor que el último índice, genera un error.

Una representación abstracta de lo que acabo de hacer sería la siguiente:



#### Ilustración 21: Obtención de valores individuales de un arreglo.

Hay algo de lo anterior que a un programador debería hacerle ruido: el índice de la última vuelta fue el número 5 porque se tiene un arreglo de 6 posiciones.

¿Qué sucede si esas mismas instrucciones las reutilizo para mostrar otro arreglo de 10 posiciones?

Resulta que las primeras dos instrucciones de salida funcionan bien, pero la tercera, que imprime `vueltas[5]` no está imprimiendo el valor de la última posición, sino la sexta. El índice de la última vuelta en un arreglo de 10 posiciones es 9.

Habrás notado que, para obtener el valor de la última posición de cualquier arreglo, se debe conocer su longitud.

### Obtener longitud del arreglo

Todo arreglo incorpora un atributo entero llamado `length`, que representa su longitud. Para obtener el valor de la longitud de cualquier arreglo se escribe el nombre de este seguido de un punto y a continuación la palabra `length`.

El siguiente programa muestra por consola cuánto vale la longitud del arreglo `vueltas`:

```
1  package prueba;
2
3  public class Prueba {
4      public static void main(String[] args) {
5          int[] vueltas = {64,62,65,68,63,65};
6          System.out.println("Longitud: " + vueltas.length );
```

```
7     }  
8 }
```

#### Código 49: Obtención de la longitud de un arreglo.

Gracias a conocer la longitud, podés ahora obtener la última posición de cualquier arreglo: **el índice del último elemento es la longitud del arreglo menos 1** (porque el índice comienza en **0**).

El siguiente programa muestra el valor de la última vuelta del arreglo **vuel**tas, pero usando una salida que funcionaría para cualquier longitud:

```
1 package prueba;  
2  
3 public class Prueba {  
4     public static void main(String[] args) {  
5         int[] vueltas = {64,62,65,68,63,65};  
6         int ult = vueltas.length - 1;  
7         System.out.println("Última vuelta: " + vueltas[ult] );  
8     }  
9 }
```

#### Código 50: Obtención del último elemento de cualquier arreglo.

### Obtener todos los valores

Para obtener todos los valores de un arreglo, simplemente hay que pasar por todos sus índices. Como ya sabés, el índice de cualquier arreglo comienza en **0** y termina en la longitud menos **1**.

El siguiente programa muestra todos los valores de cualquier arreglo (uno por línea) utilizando un ciclo **for**:

```
1 package prueba;  
2  
3 public class Prueba {  
4     public static void main(String[] args) {  
5         int[] vueltas = {64,62,65,68,63,65};  
6         for (int i = 0; i < vueltas.length; i++) {  
7             System.out.println( i + "º valor: " + vueltas[i] );  
8         }  
9     }  
10 }
```

#### Código 51: Muestra de todos los elementos de cualquier arreglo.

Pero si el que visualiza esa salida es un usuario, no debería ver los elementos listados desde **0**, por lo tanto, manipulo la variable **i** en la salida para que se vea incrementada una unidad (en vez de

mostrar de 0 a 5, mostrará de 1 a 6), pero seguiré accediendo a los valores del arreglo utilizando el valor original de i:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         int[] vueltas = {64,62,65,68,63,65};
6         for (int i = 0; i < vueltas.length; i++) {
7             System.out.println( (i+1) + "º valor: " + vueltas[i] );
8         }
9     }
10 }
```

Código 52: Muestra de todos los elementos de cualquier arreglo numerados desde el valor 1.

## Establecer un valor

Para establecer un valor en un arreglo, se debe escribir el nombre de este seguido de un par de corchetes. Dentro de los corchetes, se escribe el número de índice del dato que se desea reemplazar (recordando que la cuenta empieza desde 0). A través del operador de asignación, se puede establecer un nuevo valor, por supuesto, del mismo tipo que la definición del arreglo.

El siguiente programa reemplaza los tiempos de la primera, la cuarta y la última vuelta del arreglo **vueltas**. A través del procedimiento **mostrarArreglo()**, se imprime dos veces el mismo arreglo, antes y después de las asignaciones:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         int[] vueltas = {64,62,65,68,63,65};
6         int tam = vueltas.length - 1;
7         System.out.println("Arreglo original");
8         mostrarArreglo(vueltas);
9         vueltas[0] = 50;
10        vueltas[3] = 70;
11        vueltas[tam] = 60;
12        System.out.println("Arreglo modificado");
13        mostrarArreglo(vueltas);
14    }
15 }
```

```
16 static void mostrarArreglo (int[] a) {
17     for (int i = 0; i < a.length; i++) {
18         System.out.print( a[i] + " " );
19     }
20     System.out.println(""); // Línea en blanco
21 }
22 }
```

Código 53: Reemplazo de valores individuales en un arreglo.

Una representación abstracta de lo que acabo de hacer sería la siguiente

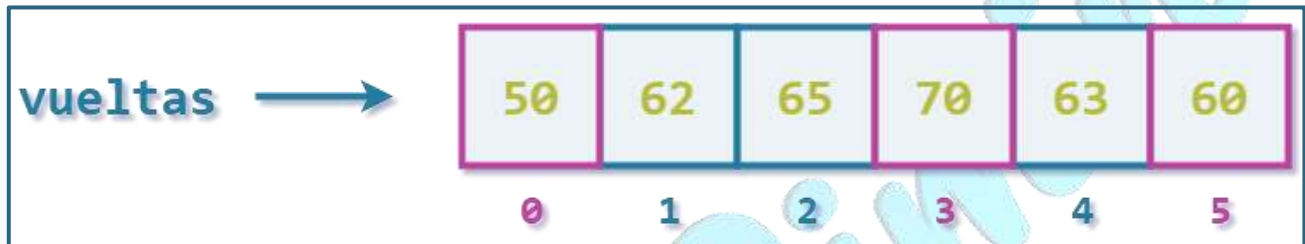


Ilustración 22: Reemplazo de valores individuales en un arreglo.

## Establecer todos los valores

Para establecer todos los valores de un arreglo, por ejemplo, cuando se lo crea con valores por defecto y luego se espera que el usuario los ingrese, la metodología es similar a la de mostrar todos los valores: pasar por todos los índices.

Lo ideal, como de costumbre, es modularizar, y por ello, haré una función llamada **crearArregloDeEnteros()**, que recibe como parámetro un número que representa la longitud deseada para el nuevo arreglo. El procedimiento crea un nuevo arreglo y va guardando números enteros a medida que se van leyendo desde la consola (usando otra función llamada **leerEntero()**). Por último, se devuelve el arreglo a quien haya invocado a la función.

```
1 package prueba;
2
3 import java.util.Scanner;
4
5 public class Prueba {
6     public static void main(String[] args) {
7         int[] numeros = crearArregloDeEnteros(5);
8         System.out.println("El arreglo queda:");
9         mostrarArreglo(numeros);
10    }
11
12    static void mostrarArreglo (int[] a) {
```



```
13     for (int i = 0; i < a.length; i++) {
14         System.out.print(a[i] + " ");
15     }
16     System.out.println(""); // Línea en blanco
17 }
18
19 static int[] crearArregloDeEnteros (int tam) {
20     int[] a = new int[tam];
21     for (int i = 0; i < a.length; i++) {
22         a[i] = leerEntero("Valor " + (i+1) + " de " + a.length + ": ");
23     }
24     return a;
25 }
26
27 static int leerEntero (String cartel) {
28     Scanner entrada = new Scanner (System.in);
29     System.out.print(cartel);
30     double x = entrada.nextDouble();
31     while ( (int) x != x ) {
32         System.out.print("ERROR. " + cartel);
33         x = entrada.nextDouble();
34     }
35     return (int) x;
36 }
37 }
```

Código 54: Creación y carga de un arreglo por el usuario a través de funciones.

## Arreglos como argumentos

En el apartado de modularización, observaste que Java trabaja con pasaje de argumentos por valor. Observá un caso similar cuando el argumento se trata de un arreglo:

```
1 package prueba;
2
3 public class Prueba {
4     public static void main(String[] args) {
5         int[] x = {1,2,3,4};
```

```
6      System.out.println("Muestro arreglo x antes de invocar:");
7      mostrarArreglo(x);
8      duplicarValores(x);
9      System.out.println("Muestro arreglo x luego de invocar:");
10     mostrarArreglo(x);
11 }
12
13 static void duplicarValores(int[] vec) {
14     for (int i = 0; i < vec.length; i++) {
15         vec[i] *= 2;
16     }
17 }
18
19 static void mostrarArreglo (int[] a) {
20     for (int i = 0; i < a.length; i++) {
21         System.out.print(a[i] + " ");
22     }
23     System.out.println(""); // Línea en blanco
24 }
25 }
```

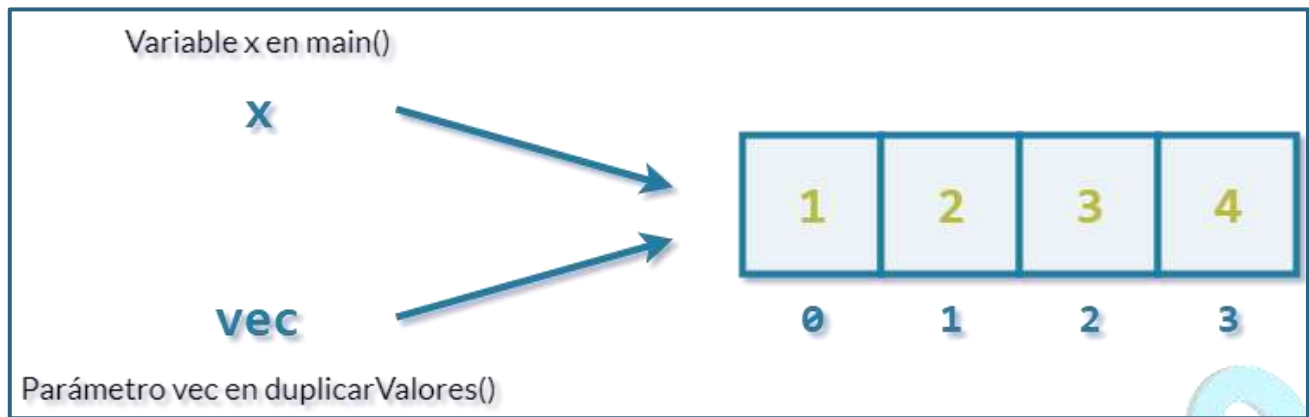
#### Código 55: Pasaje de argumentos de tipo arreglo.

En este caso, las salidas no son similares. En la línea 7 se invoca al procedimiento que permite mostrar el arreglo, imprimiendo los valores 1, 2, 3 y 4. Sin embargo, tras la invocación del procedimiento `duplicarValores()`, el `mostrarArreglo()` de la línea 10 imprime los valores 2, 4, 6 y 8.

La explicación para tal efecto es que el pasaje sigue siendo por valor, pero los arreglos son objetos, y, como he dicho anteriormente, **una variable de tipo objeto no guarda más que una referencia hacia un objeto**. Esto hace que cuando se invoca al procedimiento `duplicarValores()` enviando como argumento la variable `x` (de tipo arreglo), **lo que se esté enviando como copia es la dirección hacia donde está el arreglo `x`**.

Es como si tanto vos como yo tuviéramos cada uno un papel escrito con la dirección de mi casa. Ambos tendríamos manera de llegar a mi casa y hacer modificaciones. Cualquier cambio que hagás en mi casa, yo lo vería. Y viceversa.

Por consiguiente, la dirección del arreglo `x` llega como parámetro a la definición del procedimiento `duplicarValores()` con el nombre `vec`. A partir de allí, todo cambio que se haga sobre `vec`, tiene efecto para todas las variables que apunten hacia el mismo arreglo. Por ello, cuando se retorna a `main()` y se intenta mostrar el arreglo (usando siempre la misma referencia), se observa que han cambiado sus valores.



**Ilustración 23: Demostración de cómo llegar al mismo arreglo a través de su referencia.**

Lo explicado anteriormente funciona exactamente igual con cualquier pasaje como argumentos de variables de tipo objeto, como los arreglos o los **String**.

Estos conceptos se entenderán mejor cuando veas el paradigma orientado a objetos.

# Cadenas de caracteres

## Breve introducción al concepto de clase y objeto

Para comprender el tema del todo, primero es preciso definir brevemente, por fin, qué es una **clase** y qué es un **objeto**.

Una clase es como un plano o un boceto: no es un "algo" concreto sino la definición de cómo debe ser ese "algo".

Lo primero que se hace en una obra en construcción es el plano del edificio a construir. A partir del plano, se concretan uno o más edificios físicos. Un amigo no vive en un plano de edificio. No puedo tocar el timbre al plano del edificio. La correspondencia no llega al plano del edificio.

Por ello, en Java **una clase modela un nuevo tipo de dato**. Dentro de la clase se define cómo deben ser los objetos creados a partir de ese "molde".

Para obtener un objeto concreto a partir de una clase, se genera una instancia de la misma, a través del operador **new**.

La primera vez que viste esto fue cuando creabas un objeto llamado **entrada** de tipo **Scanner**, para leer datos desde la consola:

```
Scanner entrada = new Scanner(System.in);
```

**Scanner** es la clase que define cómo serán todos los objetos de tipo **Scanner**, cuya definición se encuentra en el archivo **Scanner.java**, que viene incluido entre las librerías de Java y que era necesario que importes cada vez que requerías trabajar con la clase **Scanner**. La variable **entrada** guarda una referencia hacia un objeto concreto de tipo **Scanner** creada a través de la palabra **new** y asignada a **entrada** a través del operador **=**. Las operaciones no se las pedías a **Scanner**, sino a **entrada**.

**Los objetos entonces son variables que guardan referencias a instancias concretas de una clase**. Y la característica principal de los objetos es que no solo modelan los datos que definen su **estado**, sino que además están provistos de **comportamiento** a través de sus **métodos**.

Por ello, cuando a **entrada** le invocabas el método **nextInt()** a través de un punto, este respondía devolviendo el valor que el usuario introducía en la consola:

```
int valor = entrada.nextInt();
```

El método **nextInt()** está definido por única vez en la clase **Scanner**. Por ello, todas las instancias concretas de **Scanner** (**entrada** es una de ellas) incorporan el método **nextInt()**.

Lo que acabo de darte es una introducción muy básica acerca de los conceptos de **clase** y **objeto**. Por supuesto que el tema merece un mayor análisis. En realidad, un libro completo y mucha práctica.

## La clase String

En Java, todo conjunto de caracteres se denomina "**cadena**" (en inglés, "**string**") y como abstracción puedo decir que **una cadena es un arreglo unidimensional de variables de tipo char**.

Hay operaciones básicas para cualquier cadena de caracteres cuyos algoritmos son tediosos y llenos de combinaciones. Recordá que un caracter simple puede ser una letra minúscula, una letra

mayúscula, un número o un símbolo. Por eso, Java provee (y has estado utilizando) la clase **String** que provee muchos métodos para manejar cadenas.

Cuando escribís la siguiente sentencia:

```
String mensaje = "Hola";
```

Lo que está pasando es que estás creando un objeto de la clase **String** cuya referencia es guardada por la variable **mensaje**. Implícitamente, Java está haciendo lo siguiente (si lo probás, funciona):

```
String mensaje = new String("Hola");
```

Java intenta "emular" a las cadenas como si se tratasen de un dato primitivo, por lo que permite asignar directamente una cadena entre comillas y ocultar la creación de la instancia, pero, en realidad, debés saber que **las cadenas son objetos**.

## Concatenar cadenas

Como habrás visto, es posible unir varias cadenas en una sola más grande, utilizando el operador de concatenación.

Cualquier otro tipo de dato primitivo que se concatene con una cadena, se convertirá primero en su representación en formato **String**. El resultado es una cadena.

La siguiente tabla lo ejemplifica:

| Concatenación               | Devuelve            |
|-----------------------------|---------------------|
| "Hola" + " mundo"           | "Hola mundo"        |
| "Tengo " + 2 + " cuadernos" | "Tengo 2 cuadernos" |
| "Pi vale " + 3.14           | "Pi vale 3.14"      |
| "Empieza con " + 'A'        | "Empieza con A"     |
| "¿Difícil? " + false        | "¿Difícil? false"   |

## Comparar cadenas

Hay que tener cuidado cuando se realizan comparaciones entre objetos, pues lo que Java compara no son precisamente sus valores (en realidad, un objeto no tiene valores sino **estado**, determinado por sus atributos), sino que compara las referencias de cada objeto. A no ser un caso particular, que sería el de comparar un objeto con sí mismo, todas las comparaciones entre objetos usando el operador **==** resultarán **false**, pues se supone que cada objeto, por más que tenga el mismo estado, es único. Es como preguntarse si dos personas gemelas son la misma persona: falso.

Sin embargo, como Java tiene un tratamiento especial para los **String**, la siguiente comparación resulta **true**:

```
String cadena = "Hola";
String otraCadena = "Hola";
System.out.println( cadena == otraCadena ); // Devuelve true
```

Sin embargo, exactamente el mismo proceso utilizando el operador **new** de forma explícita resulta false:

```
String cadena = new String("Hola");  
String otraCadena = new String("Hola");  
System.out.println( cadena == otraCadena ); // Devuelve false
```

La explicación de este fenómeno es bastante técnica y tiene que ver con cómo Java trata de manera especial a los **String**. Lo que queda claro es que en el primer caso compara realmente los valores **"Hola"**, los cuales al ser idénticos provocan que el resultado de compararlos resulte **true**. En el segundo caso se están comparando las referencias de cada una de las cadenas. Las variables **cadena** y **otraCadena** tienen cada una una referencia unívoca, que, al compararse, por supuesto, provocan que el resultado sea **false**.

Para estar siempre seguro de que lo que se está comparando no son las referencias, sino los verdaderos valores de los **String**, se debe utilizar el método **equals()** que incorpora todo objeto. El método **equals()** espera recibir como argumento un **String** y devolverá un valor booleano que indica si el **String** recibido como parámetro es idéntico al **String** por el cual se invocó al método.

Así entonces, se pueden comparar las dos cadenas anteriores de dos maneras:

```
String cadena = new String("Hola");  
String otraCadena = new String("Hola");  
System.out.println( cadena.equals(otraCadena) ); // Devuelve true  
System.out.println( otraCadena.equals(cadena) ); // Devuelve true
```

La conclusión es que todas las cadenas deben compararse utilizando el método **equals()** y no el operador de comparación.

## Procesar cadenas

Como las cadenas son objetos de la clase **String**, traen incorporados métodos muy útiles para procesarlas.

Lo que es importante destacar es que los **String** son inmutables, es decir, que una vez definidos, no pueden cambiar su estado. Cualquier operación sobre un **String**, por ejemplo, convertirlo a mayúsculas, no modifica al actual, sino que devuelve un nuevo **String** con el resultado.

A continuación, las operaciones más típicas.

### Conocer la longitud de una cadena

Como toda cadena en realidad es un arreglo, todos los **String** cuentan con el método **length()**, que devuelve un entero que indica la longitud de la misma:

```
String cadena = "Cadena de prueba";  
System.out.println( cadena.length() ); // Devuelve 16
```

A diferencia del atributo público **length** que incorporaba todo arreglo (notar que accedías sin usar los paréntesis), la longitud de una cadena es privada, es decir, que no es posible acceder a ella directamente, sino a través de un método que devuelve su valor. En la programación orientada a objetos, esto se denomina **encapsulamiento**.



## Obtener un caracter de una cadena

Para obtener un único caracter de cierta cadena, se utiliza el método `charAt()`, que recibe un número entero como argumento que representa la posición del caracter requerido (comenzando a contar desde 0) y devuelve un `char`, que representa el caracter en la posición correspondiente.

```
String cadena = "Cadena de prueba";  
System.out.println( cadena.charAt(0) ); // Devuelve 'C'  
System.out.println( cadena.charAt(2) ); // Devuelve 'd'  
System.out.println( cadena.charAt( cadena.length() - 1 ) ); // Devuelve la última 'a'
```

## Extraer una porción de la cadena

Para extraer una subcadena de otra, se utiliza el método `substring()` que presenta dos variantes. Se puede invocar a `substring()` con un solo argumento de tipo entero. Devuelve la subcadena correspondiente entre la posición recibida como argumento (incluyendo a ese caracter) y el final de la cadena:

```
String cadena = "Sacacorchos";  
System.out.println( cadena.substring(4) ); // Devuelve "corchos"
```

La otra variante es utilizando dos argumentos enteros. En tal caso, devuelve la subcadena correspondiente entre la posición recibida como primer parámetro y la posición recibida como segundo parámetro, sin incluir a este último:

```
String cadena = "Sacacorchos";  
System.out.println( cadena.substring(0,4) ); // Devuelve "Saca"
```

## Otras operaciones

La siguiente tabla resume las operaciones que incorpora todo `String`:

| Método                          | Parámetro(s)        | Devuelve             | Descripción  |
|---------------------------------|---------------------|----------------------|--|
| <code>charAt()</code>           | <code>int</code>    | <code>char</code>    | Devuelve el carácter que se encuentre en la posición recibida como parámetro.  |
| <code>equals()</code>           | <code>String</code> | <code>boolean</code> | Devuelve si la cadena es igual a la recibida por parámetro.  |
| <code>equalsIgnoreCase()</code> | <code>String</code> | <code>boolean</code> | Devuelve si la cadena es igual a la recibida por parámetro, ignorando mayúsculas y minúsculas.                                     |
| <code>indexOf()</code>          | <code>String</code> | <code>int</code>     | Devuelve la posición donde comienza la cadena recibida por parámetro, desde el principio. Si no existe, devuelve <code>-1</code> . |
| <code>indexOf()</code>          | <code>char</code>   | <code>int</code>     | Devuelve la posición del carácter recibido por parámetro, desde el principio. Si no existe, devuelve <code>-1</code> .             |

|                             |                              |                      |  |
|-----------------------------|------------------------------|----------------------|--|
| <code>indexOf()</code>      | <code>String , int</code>    | <code>int</code>     | Devuelve la posición donde comienza la cadena recibida como primer parámetro, desde el valor del segundo parámetro. Si no existe, devuelve <b>-1</b> .             |
| <code>indexOf()</code>      | <code>char , int</code>      | <code>int</code>     | Devuelve la posición del carácter recibido como primer parámetro, desde el valor del segundo parámetro. Si no existe, devuelve <b>-1</b> .                         |
| <code>isEmpty()</code>      | <code>-</code>               | <code>boolean</code> | Devuelve si la cadena está o no vacía.   |
| <code>lastIndexOf()</code>  | <code>String</code>          | <code>int</code>     | Devuelve la posición donde comienza la cadena recibida por parámetro, desde el final. Si no existe, devuelve <b>-1</b> .   |
| <code>lastIndexOf()</code>  | <code>char</code>            | <code>int</code>     | Devuelve la posición del carácter recibido por parámetro, desde el final. Si no existe, devuelve <b>-1</b> .   |
| <code>lastIndexOf()</code>  | <code>String , int</code>    | <code>int</code>     | Devuelve la posición donde comienza la cadena recibida como primer parámetro, desde el valor del segundo parámetro hacia atrás. Si no existe, devuelve <b>-1</b> . |
| <code>lastIndexOf()</code>  | <code>char , int</code>      | <code>int</code>     | Devuelve la posición del carácter recibido como primer parámetro, desde el valor del segundo parámetro hacia atrás. Si no existe, devuelve <b>-1</b> .             |
| <code>length()</code>       | <code>-</code>               | <code>int</code>     | Devuelve la longitud de una cadena.  |
| <code>replace()</code>      | <code>char , char</code>     | <code>String</code>  | Devuelve una cadena, donde todas las ocurrencias del carácter recibido como primer parámetro son reemplazadas por el carácter recibido como segundo parámetro.     |
| <code>replace()</code>      | <code>String , String</code> | <code>String</code>  | Devuelve una cadena, donde todas las ocurrencias de la cadena recibida como primer parámetro son reemplazadas por la cadena recibida como segundo parámetro.       |
| <code>replaceFirst()</code> | <code>String , String</code> | <code>String</code>  | Devuelve una cadena, donde la primera ocurrencia de la cadena recibida como primer parámetro es reemplazada por la cadena recibida como segundo parámetro.         |

|                      |                     |                 |   |
|----------------------|---------------------|-----------------|---|
| <b>split()</b>       | <b>String</b>       | <b>String[]</b> | Separa la cadena por la cadena recibida como parámetro y devuelve cada segmento en un arreglo de cadenas.   |
| <b>startsWith()</b>  | <b>String</b>       | <b>boolean</b>  | Devuelve si una cadena comienza con la cadena recibida como parámetro.  |
| <b>startsWith()</b>  | <b>String , int</b> | <b>boolean</b>  | Devuelve si una cadena comienza con la cadena recibida como parámetro, desde la posición recibida como segundo parámetro.                               |
| <b>substring()</b>   | <b>int</b>          | <b>String</b>   | Devuelve la cadena resultante entre la posición recibida como primer parámetro y el final de la cadena.   |
| <b>substring()</b>   | <b>int , int</b>    | <b>String</b>   | Devuelve la cadena resultante entre la posición recibida como primer parámetro y la posición recibida como segundo parámetro (ésta última sin incluir). |
| <b>toCharArray()</b> | <b>-</b>            | <b>char[]</b>   | Devuelve la cadena como un arreglo de caracteres.   |
| <b>toLowerCase()</b> | <b>-</b>            | <b>String</b>   | Devuelve la cadena en minúsculas.   |
| <b>toUpperCase()</b> | <b>-</b>            | <b>String</b>   | Devuelve la cadena en mayúsculas.   |
| <b>trim()</b>        | <b>-</b>            | <b>String</b>   | Devuelve la cadena sin espacios en blanco al principio y al final.  |

## Conversiones

Hay ocasiones donde es necesario convertir un valor representado por una cadena en el verdadero valor primitivo y viceversa.

### Clases envoltorio (wrapper)

Todo tipo primitivo tiene en Java una clase asociada, que integra operaciones y valores especiales para el correspondiente tipo de dato. La siguiente tabla asocia cada dato primitivo (de los que utilizo) con su correspondiente clase envoltorio:

| Dato primitivo | Clase envoltorio |
|----------------|------------------|
| <b>char</b>    | <b>Character</b> |
| <b>int</b>     | <b>Integer</b>   |
| <b>double</b>  | <b>Double</b>    |
| <b>boolean</b> | <b>Boolean</b>   |

## Convertir un primitivo en una cadena

Para convertir cualquier dato primitivo a una cadena, se utiliza el método

`toString()` de la correspondiente clase envoltorio, cuyo argumento es el tipo de dato primitivo. El método devuelve el argumento representado en una cadena.

```
String enteroACadena = Integer.toString(50); // Convierte el 50 en "50"
String doubleACadena = Double.toString(1.44); // Convierte el 1.44 en "1.44"
String charACadena = Character.toString('x'); // Convierte la 'x' en "x"
String booleanACadena = Boolean.toString(true); // Convierte true en "true"
```

Aunque hay un "truco" más sencillo, que implica saber que todo primitivo concatenado con una cadena, resulta en una cadena. Por lo tanto, podrías a cada primitivo concatenarlo con una cadena nula, obteniendo el mismo efecto:

```
String enteroACadena = 50 + ""; // Convierte el 50 en "50"
String doubleACadena = 1.44 + ""; // Convierte el 1.44 en "1.44"
String charACadena = 'x' + ""; // Convierte la 'x' en "x"
String booleanACadena = true + ""; // Convierte true en "true"
```

## Convertir una cadena en un primitivo

Suponiendo que tengas valores numéricos expresados como cadenas y necesites realizar operaciones matemáticas con ellos, será necesario convertirlos a su verdadero tipo de dato primitivo. Esto se denomina "parsear".

```
int aEntero = Integer.parseInt("5"); // Convierte el "5" en 5
double aDouble = Double.parseDouble("1.44"); // Convierte el "1.44" en 1.44
char aChar = "x".charAt(0); // Convierte la "x" en 'x'
boolean aBoolean = Boolean.parseBoolean("true"); // Convierte "true" en true
```

Cualquier intento de "parseo" de una cadena que no represente un valor numérico a un tipo de dato primitivo como `int` o `double` provocará una excepción.

## Tabla de ilustraciones

|   |    |
|---|----|
| Ilustración 1: Diferencia entre JVM, JRE y JDK.....   | 8  |
| Ilustración 2: Conversión de un código a un programa .....  | 8  |
| Ilustración 3: Guardando el código Java. ....   | 10 |
| Ilustración 16: Modificando la variable de entorno Path. ....   | 11 |
| Ilustración 17: Compilando un código Java. ....   | 12 |
| Ilustración 18: Ejecutando un código Java.....  | 12 |
| Ilustración 19: Creando un proyecto en NetBeans.....  | 13 |
| Ilustración 20: Configurando el nuevo proyecto en NetBeans. ....  | 13 |
| Ilustración 21: Escribiendo código Java en NetBeans. ....   | 14 |
| Ilustración 22: Resultado del programa en NetBeans. ....  | 14 |
| Ilustración 23: Accediendo a las propiedades del proyecto. ....   | 15 |
| Ilustración 24: Ventana de propiedades del proyecto.....  | 16 |
| Ilustración 25: Ventana de apertura de proyecto. ....   | 17 |
| Ilustración 26: Cuadro completo de promoción de tipos.....  | 31 |
| Ilustración 27: Flujo del programa al invocar un procedimiento. ....                                    | 65 |
| Ilustración 28: Los argumentos de una invocación se convierten en los parámetros de la definición. .... | 67 |
| Ilustración 29: Las funciones se invocan y devuelven un valor. ....                                     | 71 |
| Ilustración 30: Representación abstracta de un arreglo.....   | 78 |
| Ilustración 31: Arreglo con valores por defecto.....  | 79 |
| Ilustración 32: Arreglo inicializado con valores. ....  | 80 |
| Ilustración 33: Obtención de valores individuales de un arreglo. ....                                   | 81 |
| Ilustración 34: Reemplazo de valores individuales en un arreglo. ....                                   | 84 |
| Ilustración 35: Demostración de cómo llegar al mismo arreglo a través de su referencia..                | 87 |



## Tabla de códigos

|  |    |
|--|----|
| Código 1: Primer programa en Java.....   | 9  |
| Código 2: Código mínimo para cualquier programa en Java. ....  | 18 |
| Código 3: Ejemplo de código comentado.....   | 19 |
| Código 4: Salidas con println(). ....  | 20 |
| Código 5: Salidas con print(). ....  | 21 |
| Código 6: Caracteres de escape. ....   | 22 |
| Código 7: Prueba de división entre enteros. ....   | 24 |
| Código 8: Prueba de división con valores de tipo double. ....  | 24 |
| Código 9: Declaración, inicialización, operación y muestra de variables. ....                                  | 27 |
| Código 10: Declaración, inicialización, operación y muestra de variables con el operador de concatenación..... | 28 |
| Código 11: Actualización de una variable.....  | 29 |
| Código 12: Demostración de operadores incrementales. ....  | 30 |
| Código 13: Demostración de casting implícito por ensanchamiento.....   | 31 |
| Código 14: Demostración de casting explícito por estrechamiento. ....  | 31 |
| Código 15: Importación de la clase Scanner.....  | 32 |
| Código 16: Declaración e inicialización del objeto entrada.....  | 33 |
| Código 17: Suma de dos números ingresados por el usuario. ....   | 34 |
| Código 18: Ingreso de un caracter simple. ....   | 34 |
| Código 19: Ingreso de una cadena. ....   | 35 |
| Código 20: Programa que explicita el problema de usar nextLine(). ....   | 36 |
| Código 21: Solución al problema de nextLine(). ....  | 36 |
| Código 22: Demostración de algunos métodos de la clase Math. ....  | 39 |
| Código 23: Verificación de expresiones booleanas con operadores relacionales entre números. ....               | 41 |
| Código 24: Algoritmo que valida un pase según la edad. ....  | 42 |
| Código 25: Algoritmo que valida o no un pase según la edad.....  | 44 |
| Código 26: Algoritmo que determina la hora del día utilizando bloques de selección dentro de otros.....        | 45 |
| Código 27: Algoritmo que determina la hora del día utilizando bloques de selección anidados. ....              | 46 |
| Código 28: Demostración del operador NOT.....  | 47 |
| Código 29: Algoritmo que determina la hora del día con operadores lógicos. ....                                | 49 |



|   |    |
|---|----|
| Código 30: Menú de opciones con switch. ....  | 53 |
| Código 31: Salida por consola repetida 10 veces. ....   | 55 |
| Código 32: Salida por consola repetida 10 veces, comenzando a contar desde 0.....                 | 55 |
| Código 33: Sumatoria de números ingresados por el usuario con while. ....                         | 57 |
| Código 34: Sumatoria de números ingresados por el usuario con do...while.....                     | 58 |
| Código 35: Intento de acceso a una variable declarada dentro de un bloque.....                    | 59 |
| Código 36: Declaración de la variable fuera del bloque.....                                       | 60 |
| Código 37: Programa que imprime la letra de una canción. ....                                     | 62 |
| Código 38: Dónde se escribe la definición de un procedimiento.....                                | 63 |
| Código 39: Definición de un procedimiento.....  | 64 |
| Código 40: Programa que imprime la letra de una canción utilizando un procedimiento...            | 65 |
| Código 41: Procedimiento que presenta a un usuario según sus parámetros.....                      | 66 |
| Código 42: Procedimiento que presenta al usuario según datos ingresados por consola. .            | 69 |
| Código 43: Definición e invocación de una función que calcula el factorial. ....                  | 71 |
| Código 44: Programa que comprueba que toda ejecución comienza y termina en la función main()..... | 74 |
| Código 45: Demostración del pasaje de parámetros por valor. ....                                  | 75 |
| Código 46: Programa demasiado acoplado y con poca cohesión. ....                                  | 77 |
| Código 47: Programa con funciones altamente cohesivas y poco acopladas. ....                      | 77 |
| Código 48: Obtención de valores individuales de un arreglo.....                                   | 81 |
| Código 49: Obtención de la longitud de un arreglo. ....   | 82 |
| Código 50: Obtención del último elemento de cualquier arreglo. ....                               | 82 |
| Código 51: Muestra de todos los elementos de cualquier arreglo.....                               | 82 |
| Código 52: Muestra de todos los elementos de cualquier arreglo numerados desde el valor 1 .....   | 83 |
| Código 53: Reemplazo de valores individuales en un arreglo.....                                   | 84 |
| Código 54: Creación y carga de un arreglo por el usuario a través de funciones. ....              | 85 |
| Código 55: Pasaje de argumentos de tipo arreglo.....  | 86 |