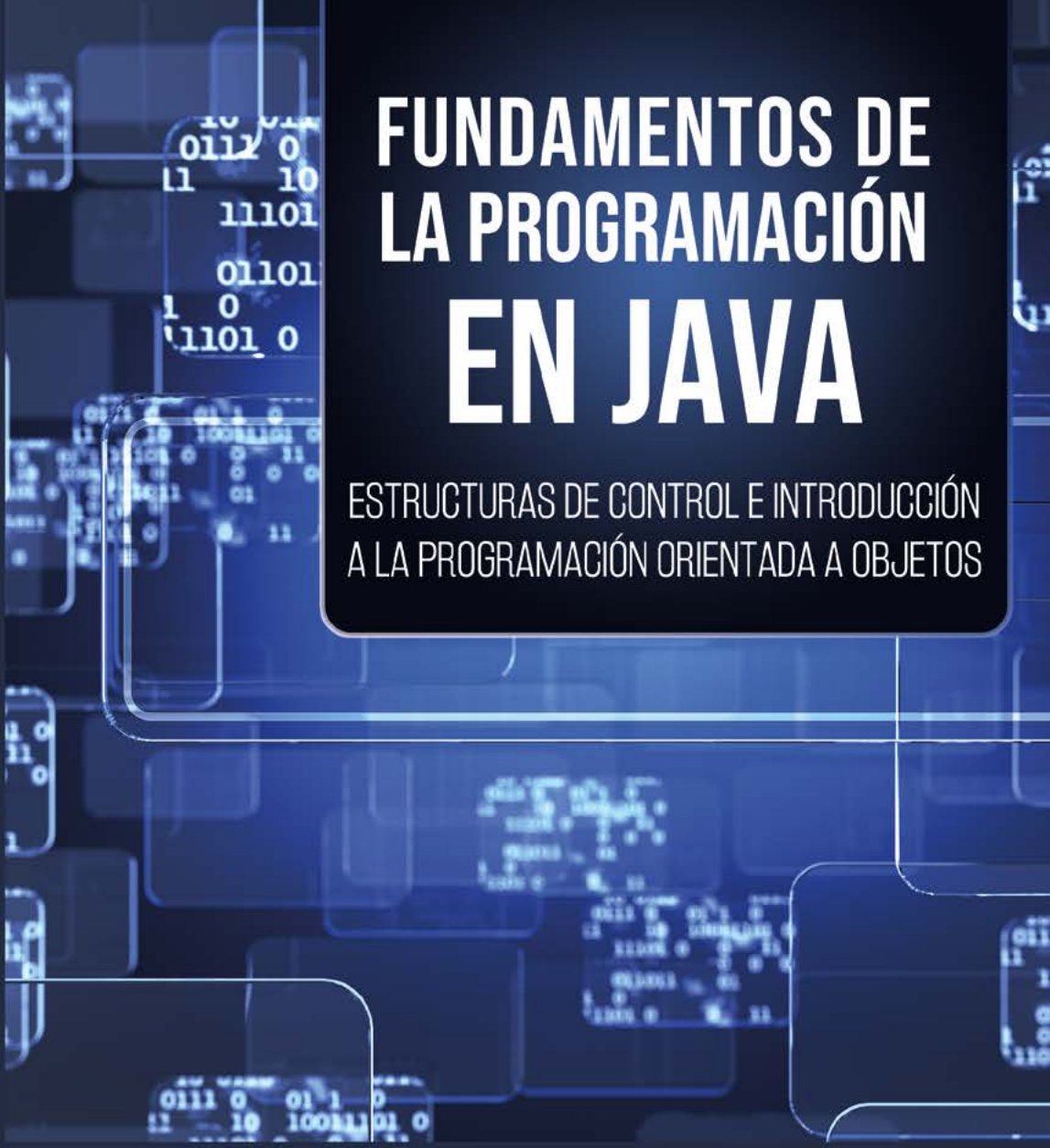

FUNDAMENTOS DE LA PROGRAMACIÓN EN JAVA

ESTRUCTURAS DE CONTROL E INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

Javier Pino, Patricia Martínez, José Antonio Vergara

Círculo Rojo
EDITORIAL

FUNDAMENTOS DE LA PROGRAMACIÓN EN JAVA
Javier Pino, Patricia Martínez, José Antonio Vergara

Fundamentos de la programación en Java. Estructuras de Control e Introducción a la Programación Orientada a Objetos

Fundamentos de la programación en Java. Estructuras de Control e Introducción a la Programación Orientada a Objetos

Javier Pino Herrera, Patricia Martínez Moreno,
José Antonio Vergara Camacho

Círculo Rojo
EDITORIAL

Primera edición: septiembre 2020

Depósito legal: AL 1662-2020

ISBN: 978-84-1374-012-6.

Impresión y encuadernación: Editorial Círculo Rojo

Editorial Círculo Rojo
www.editorialcirculorojo.com
info@editorialcirculorojo.com

Impreso en España — Printed in Spain

Directorio Universidad Veracruzana

Sara Ladrón de Guevara
Rectora

María Magdalena Hernández Alarcón
Secretaria Académica

Salvador F. Tapia Spinoso
Secretario de Administración y Finanzas

Carlos Lamothe Zavaleta
Vicerrector región Coatzacoalcos

Arturo Bocardo Valle
Directora del Área Económico-Administrativa

Liliana I. Betancourt Trevedhan
Directora de Desarrollo Académico e Innovación Educativa

José Antonio Vergara Camacho
Director de la Facultad de Contaduría y Administración Coatzacoalcos

Mercedes Asunción Morán Urcelay
Secretaria Académica de la Facultad
de Contaduría y Administración Coatzacoalcos

Cuerpo Académico
Aplicación y enseñanza de la ingeniería de software

Patricia Martínez Moreno
José Antonio Vergara Camacho
Gerardo Contreras Vega
Javier Pino Herrera
Irwing Alejandro Ibañez Castillo

Índice:

Prólogo

Si ha comenzado a leer este libro, lo más seguro es que ya cuenta con una idea general de lo que es la programación, o cuando menos, es consciente de que los programas en las computadoras han sido creados por personas. Directa o indirectamente, todos hemos usado de los servicios de algún programa informático; al momento de utilizar el cajero automático de un banco, al consultar un correo electrónico, cuando la cajera del supermercado cobra los productos, y la lista podría continuar. Todos estos programas fueron creados por personas que al inicio no contaban con nociones de programación, sin embargo, al igual que usted, en algún momento tomaron la decisión de comenzar a aprender a programar.

Aprender a programar no tiene por qué ser complicado, obviamente, al igual que cualquier otra disciplina, requiere de práctica. Una persona que decide aprender a tocar el piano puede leer, y tal vez, llegar a comprender todo lo que corresponde con los conocimientos teóricos, tales como la posición de las manos, el significado de las notas musicales, etcétera. Las primeras veces que toque el piano, la persona cometerá errores; sin embargo, conforme practique lo aprendido, esta persona irá corrigiendo sus errores, hasta lograr tocar una pieza musical de forma correcta. Entonces, ¿es difícil aprender a programar? La respuesta es no, sin embargo, aprender a programar sin practicar, eso sí es imposible. Por lo que, ya que usted ha decidido adentrarse en este hermoso mundo de la programación, lo invitamos a practicar, practicar mucho,

experimentar, equivocarse; ¡sí!, ¡así es!, lo invitamos a equivocarse, a equivocarse mucho, porque después de cometer y corregir todos esos errores, usted logrará convertirse en un experto, y llegará a ver las líneas de código como un arte.

Introducción

Este libro surge por el contacto del día a día en las aulas con los estudiantes a través de impartir las clases de Programación e Introducción a la Programación, con el fin de apoyar y fortalecer competencias en toda persona que inicie y le interese el aprendizaje en el desarrollo de programas de computadora en el lenguaje JAVA.

Java es un lenguaje de programación que ha tomado fuerza por ser software independiente de la plataforma. Un programa en Java funciona en cualquier computadora existente, situación que ha sido atractiva en los desarrolladores de software, olvidándose de los diversos sistemas operativos: Windows, Linux, MacOS, etc. Esto es porque JAVA cuenta con una máquina virtual para cada plataforma que hace de enlace o puente entre el sistema operativo y el programa de Java.

Ahora aquí se muestra paso a paso cómo introducirse a la programación en Java desde la instalación del mismo software, conocer y manipular las diversas estructuras de control: If, While, Switch, Do While, For, hasta temas de mayor complejidad como es el uso de arreglos y los elementos básicos de la Programación Orientada a Objetos (POO) como la instanciación, constructores, herencia, otros.

De tal manera que el libro se encuentra estructurado en siete capítulos.

Capítulo 1. Instalación y configuración de Java.
Capítulo 2. Introducción a la Programación Java.

1. Instalación y configuración de Java

1.1Introducción

Antes de comenzar a trabajar en Java, se necesita instalar todo lo necesario para crear programas. Este capítulo está enfocado en preparar el equipo de cómputo y realizar los primeros programas en Java. Se comenzará mostrando el proceso de instalación del kit de desarrollo de Java (*JDK, Java Developement Kit*) para Windows, éste es necesario para poder crear nuestros programas en Java. De forma opcional, se puede utilizar algún sitio que permita crear y ejecutar programas de Java en línea. En el apartado 1.4 se presentan algunos sitios en Internet que ofrecen esta posibilidad.

1.2Descargar e instalar JDK

El kit de desarrollo de java (JDK) nos brinda las herramientas para poder crear programas en Java en nuestra computadora. Los pasos de instalación para las diferentes plataformas los encontramos en los siguientes enlaces:

Windows:

```
http://www.oracle.com/technetwork/java/javase/
downloads/index.html
```

MacOS:

```
http://www.oracle.com/technetwork/java/javase/
downloads/index.html
```

Linux:

```
http://www.oracle.com/technetwork/java/javase/
downloads/index.html
```

A partir de la versión 9 de Java, el JDK sólo se brinda para plataformas de 64 bits. Uno de los requisitos es contar con Windows de 64 bits. En caso de contar con una plataforma Windows de 32 bits, realice los mismos pasos, pero con el JDK 8.

Aunque los pasos de instalación son descritos por Oracle en los enlaces oficiales, se describirán los pasos para realizar la instalación del JDK 11 en plataformas Windows.

Lo primero que necesitamos realizar es la descarga del JDK, que se encuentran en:

```
http://www.oracle.com/technetwork/java/javase/
downloads/index.html
```

En su defecto, puede utilizar el buscador de su preferencia con las palabras claves ***Java JDK 11***. Una vez dentro del sitio, haga clic en la versión del JDK a descargar, acepte el acuerdo de licencia y oprima el botón de descarga. El sitio solicita identificarnos para poder realizar la descarga. Si se dispone de una cuenta de Oracle, se podrá iniciar sesión; de lo contrario, se debe hacer clic en el botón de *crear una cuenta*. Una vez que se haya identificado, el proceso de descarga del instalador se iniciará.

Ya descargado el instalador del JDK, se procede a ejecutar y realizar el proceso de instalación. Al finalizar el proceso de instalación, se muestra un mensaje informando que el proceso ha sido realizado de manera exitosa.

1.3Establecer la variable de entorno PATH

Una vez instalado el JDK, se establece la variable de entorno PATH para compilar y ejecutar los programas. El proceso para las diversas plataformas se encuentra en:

```
https://www.java.com/es/download/help/path.xml
```

Como apoyo al proceso, las instrucciones para establecer la variable de entorno PATH en Windows 10 son los siguientes:

1. Seleccionar **Panel de control**, luego a **Sistema y seguridad** y, por último, **Sistema**.

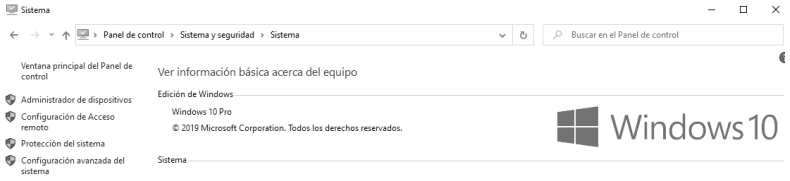

Figura 1.1 | Windows: Panel de control\Sistema y seguridad\Sistema

2. Hacer clic en **Configuración avanzada del sistema** y luego haga clic en **Variables de entorno**.
3. Hacer doble clic sobre la variable PATH.

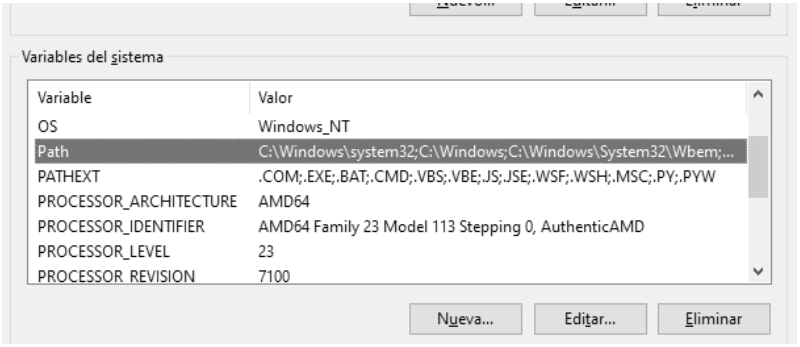

Figura 1.2 | Variables de entorno

4. Agregar la ruta de la carpeta **bin** que se encuentra dentro de la instalación del JDK. Para este ejemplo es la siguiente: C:\Program Files\Java\jdk-11.0.7\bin. Encuentre la ubicación de su carpeta **bin** y agréguela a la variable de entorno **PATH**.
5. Aceptar todas las ventanas.

Una vez finalizados todos los pasos, compruebe estar listo para crear programas en Java al abrir una ventana del **símbolo del sistema** y escribir `java -version`. Al presionar el botón **Intro**, se debe poder visualizar la versión de Java, tal como se muestra en la figura 1.3.

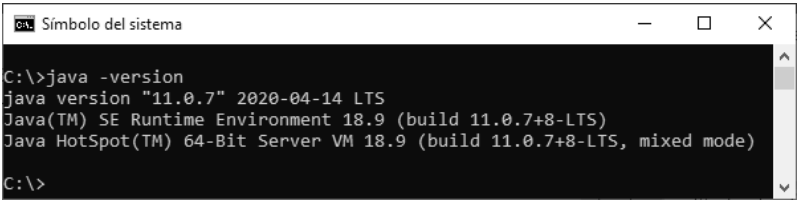

Figura 1.3 | Símbolo del sistema en Windows

1.4Crear y ejecutar programas de Java desde Internet

En caso de no realizar el proceso de instalación, ni la configuración de Java en su computadora, se puede hacer uso de algún sitio de Internet que permita crear programas en Java y ejecutarlos desde ahí mismo. Existen varios sitios que ofrecen esta posibilidad. Con realizar una búsqueda desde el navegador de su preferencia con las palabras ***java editor online***, se mostrarán varias opciones y así crear los programas que se presentan en este libro. Algunos de los posibles sitios que encontrará en los resultados de su búsqueda se muestran a continuación.

CodingGround

```
https://www.tutorialspoint.com/compile_java_online.php
```

Paiza.io

```
https://paiza.io/es/languages/online-java-compiler
```

Repl.it

```
https://repl.it/languages/java10
```

Es probable que los enlaces lleguen a cambiar, sin embargo, el buscador siempre le entregará los enlaces actuales. En cualquier caso, aunque estos sitios son una opción para realizar programas simples y ejecutarlos en línea, no se recomienda el uso para usuarios que apenas se inician en el aprendizaje de este lenguaje de programación, en todo caso, la recomendación es realizar los ejercicios del libro desde un **editor de texto plano**, sin embargo, es bueno contar con la libertad de

utilizar alguno de los sitios antes mencionados, si así lo desea, con la recomendación de investigar y documentarse sobre su funcionamiento.

1.5Editores de código para Java

Hoy en día, es posible encontrar una gran variedad de editores de código para Java, algunos incluyen botones que realizan el proceso de compilación y ejecución. Aunque se asume en este libro que los programas son realizados en un **editor de texto plano**, es posible que utilice un editor de código o algún entorno de desarrollo integrado (IDE, *Integrated Development Environment*).

Algunos de los entornos de desarrollo integrados más utilizados son:

Netbeans

```
https://netbeans.org/
```

Eclipse

```
https://www.eclipse.org/
```

IntelliJ IDEA

```
https://www.jetbrains.com/es-es/idea/
```

jGRASP

```
https://www.jgrasp.org/
```

jBlue

```
https://bluej.org/
```

En todo caso, si utiliza algún editor de código o un IDE, la recomendación es documentarse sobre su uso, ya que el alcance de este libro no abarca el uso de alguno de estos programas. Como ya se mencionó anteriormente, en este libro los programas realizados para Java serán tratados como si hubiesen sido creados desde un **editor de texto plano**.

1.6Conclusión

En este capítulo vimos la forma de preparar el equipo para crear programas en Java. Así mismo, se hizo referencia a algunos sitios que nos permiten realizar programas en Java y ejecutarlos ahí mismo. Por último, se mostraron algunos de los entornos de desarrollo integrados más utilizados para crear programas en Java. En el próximo capítulo se comienza el aprendizaje sobre las bases de la programación en Java con la creación de programas.

2.**Introducción a la programación en Java**

2.1Introducción

El aprendizaje de cualquier lenguaje de programación se inicia con los conceptos básicos y, conforme se avanza en los temas, éstos van incrementando gradualmente la complejidad, no porque los temas sean difíciles, sino porque los temas se van relacionando con otros entre sí. En este capítulo creará programas, se muestra la estructura general de los programas en Java, aprenderá a imprimir texto en pantalla, a identificar los tipos de datos disponibles a utilizar en los programas, además realizará operaciones aritméticas, y solicitará valores de entrada a los usuarios desde los programas.

2.2Imprimir texto en pantalla

Ha llegado el momento de realizar el primer programa. Para crear un programa en Java, se puede hacer uso de un editor de texto plano, el **bloc de notas** en Windows es suficiente para los alcances de este libro. De igual manera, como ya se mencionó antes, si no se desea instalar Java en la computadora, y se desea realizar los ejercicios desde cualquier

equipo de cómputo que tenga acceso a Internet, se puede utilizar alguna de las opciones para crear código en línea a través de alguna plataforma en línea. Será necesario se documente sobre el uso adecuado del mismo. Los ejercicios del libro se muestran asumiendo que están siendo creados desde un editor de texto plano.

En la figura 2.1, se ejemplifica el código de un programa en Java. Tómese un corto tiempo para observar y analizar el código.

MiClase.java

```
1  class MiClase {
2   public static void main(String[] args){
3         System.out.println("Hola mundo");
4      } // Fin del método main
5  } // Fin de la clase MiClase
```

```
Hola mundo.
```

Figura 2.1 | Imprimir texto

Escriba el código de la figura 2.1 en un editor de texto plano. Guarde el archivo con el nombre y extensión

```
MiClase.java
```

es importante respetar las letras en mayúsculas y las letras en minúsculas, es decir, el nombre del archivo debe ser exactamente el que se muestra en la línea 1, así también, la extensión debe ser .java.

Como cualquier nuevo conocimiento, conforme se realicen más ejercicios, poco a poco se irá familiarizando con la estructura general de los programas en Java. Por el momento,

lo más importante es entender que el código mostrado en la figura 2.1, al ser compilado y ejecutado, imprime en pantalla el texto `Hola mundo`.

El programa de la figura 2.1 muestra en términos generales cómo está estructurado el código de un programa en Java. La línea 1

```
1 | class MiClase {
```

inicia la declaración de la clase `MiClase`. En Java, todo programa necesita de por lo menos una clase. Para definir esa clase se utiliza la **palabra reservada** `class`, seguida del nombre a identificar la clase. Las **palabras reservadas** siempre se escriben en minúsculas y sólo son utilizadas por Java. Por último, la llave de apertura, `{`, indica que, a partir de ahí, todo lo que se escriba corresponde a esa clase.

La línea 2,

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

es el **método** que será llamado para iniciar el programa, sin embargo, por el momento no se intenta comprender esta línea por completo, conforme se avance en los temas, se comprenderá poco a poco el significado de cada fragmento en esa línea de código. Lo importante es saber que todo programa en Java requiere de una clase, y esa clase debe definir un **método `main`** para que el programa pueda ser ejecutado. Al final de la línea, se aprecia nuevamente una llave de apertura, `{`, ésta indica que a partir de ahí comienzan las instrucciones que realiza el programa.

La línea 3,

```
3   System.out.println("Hola mundo");
```

realiza la tarea de imprimir el texto `Hola mundo` en pantalla. Nótese que, al ejecutar el programa, el texto que se muestra en pantalla se observa sin las comillas dobles. En Java, el texto es representado entre comillas dobles. Aunque en estos momentos no se logre comprender cada fragmento del código, con certeza se irá avanzando hasta comprender a detalle cada parte del código. Por ahora, si se quiere imprimir texto en pantalla, replique la instrucción mostrada en la línea 3.

La línea 4,

```
4   } // Fin del método main
```

comienza con un cierre de llave, `}`. Esto indica el final de las instrucciones que son parte del **método `main`**. Al ingresar dos barra diagonales, `//`, todo lo que se escriba a partir de ahí, hasta el final de la línea, será ignorado por el compilador de Java, esto se conoce como **comentario de línea**, y sirve para agregar información de apoyo para que el código sea más comprensible.

La línea 5,

```
5   } // Fin de la clase MiClase
```

comienza con un cierre de llave, `}`, esto indica el final del cuerpo de la **clase `MiClase`**. La línea 5 también tiene un comentario de línea, es decir, que todo el texto escrito después de las dos barras diagonales, `//`, será ignorado por el compilador de Java hasta que pase a la siguiente línea.

Todos los programas a realizar en Java tendrán esta estructura.

Es momento de compilar y ejecutar el primer programa en Java. Si aún no se tiene instalado el **Kit de desarrollo de Java (JDK)**, y por el momento no se desea realizar el proceso de instalación y configuración en la computadora, se puede hacer uso de alguna plataforma de codificación en línea, Repl.it (https://repl.it), podría ser una buena opción. Si se desea realizar el proceso de instalación y configuración del JDK, se sugiere revisar el proceso que se explica en el capítulo 1.

Los siguientes pasos corresponden al proceso de compilación y ejecución desde la línea de comandos en un equipo de cómputo con el JDK instalado y configurado.

Lo primero a realizar para compilar un programa en java es abrir una ventana de comandos y ubicarse en el directorio en el que se encuentra el archivo con el código de su programa. En Windows el comando sería algo parecido a

```
cd c:\ejercicios
```

En Linux y MacOS el comando es parecido a

```
cd ~\ejercicios
```

Una vez en la ubicación de la carpeta en la que se encuentra el archivo del código, para compilar el programa, escribir:

```
javac MiClase.java
```

Una vez ejecutado el comando anterior, si no se muestra algún mensaje de error en pantalla, significa que la compilación del programa se realizó con éxito y se tiene un nuevo archivo de nombre `MiClase.class`, este archivo contiene la información necesaria para que el programa sea ejecutado en cualquier computadora que tenga instalado Java.

Ya que se tiene el archivo **.class**, se procede a ejecutar el programa escribiendo:

```
java MiClase
```

Al ejecutar este comando, el programa se inicia y se muestra en pantalla el texto `Hola mundo`. Note que en el comando de ejecución de nuestro programa no se agrega la extensión `.class`.

Ahora, se realiza un pequeño cambio al código del programa, modifique el texto `Hola mundo` de la línea 3 por `Bienvenido a JAVA`. La figura 2.2 muestra el código con esta modificación. Se procede a guardar los cambios, compilar nuevamente y ejecutar el programa.

MiClase.java

```
1  class MiClase {
2    public static void main(String[] args) {
3    System.out.println("Bienvenido a JAVA");
4    } // Fin del método main
5  } // Fin de la clase MiClase
```

```
Bienvenido a JAVA.
```

Figura 2.2 | Imprimir texto

Observe que, al ejecutar el programa, ahora se muestra en pantalla el texto Bienvenido a JAVA. Siempre que se utilice el método `println` para imprimir texto, es necesario colocarlo dentro de los paréntesis, y los textos siempre serán representados en Java con caracteres encerrados entre dobles comillas, tal y como se observa en la línea 3 de nuestro programa.

2.3Secuencias de escape

En estos momentos ya se ha enseñado a imprimir texto en pantalla, sin embargo, existen ciertos caracteres que no se imprimen de forma natural, por ejemplo, por más que se intente, no es posible imprimir dos líneas de texto del siguiente modo:

```
System.out.println("Línea uno. Línea dos.");
```

Cuando se desea escribir más de una línea de texto, lo normal es pensar que, presionando la tecla *Intro* obtengamos dicho resultado, sin embargo, si llegáramos a utilizar la sentencia anterior, obtendremos un error en tiempo de compilación. Entonces, ¿cómo se puede indicar que el texto "`Línea dos.`" se muestre en otra línea? Para eso se utilizar una representación de dicho carácter que se conoce como **secuencia de escape**. La secuencia de escape para poder representar dentro de una cadena de texto (tipo String) el salto de línea, es con la barra diagonal invertida seguida de la letra n, `\n`. Es así que, el fragmento de código queda:

```
System.out.println("Línea uno.\nLínea dos.");
```

Salida del programa:

```
Línea uno. Línea dos.
```

Las secuencias de escape no se imprimen tal cual como se colocan en las cadenas de caracteres, sino que son interpretadas y remplazadas por el carácter que le corresponde. Estas son algunas de las secuencias de escape más comunes en Java:

Secuencia de escape	Descripción
\t	Inserta un espacio de tabulación en el texto.
\n	Inserta un salto de línea en el texto.
\r	Inserta un retorno de carro en el texto.
\”	Inserta un carácter de doble comilla en el texto (“).
\\	Inserta un carácter de barra diagonal invertida (\)

Figura 2.3 | Secuencias de escape más comunes en Java.

Para comprender mejor el uso de las secuencias de escape, se realiza un ejercicio. La figura 2.4 muestra un programa que usa de los especificadores de formato más comunes en Java.

SecuenciasDeEscape.java

```
1  class SecuenciasDeEscape {
2    public static void main(String[] args) {
3      System.out.println("Texto 1\nTexto 2");
4      System.out.println("Texto 3\tTexto 4");
5      System.out.println("Texto 5\rTexto 6");
6      System.out.println(
                        "\"Texto 7\" Texto  8");
7      System.out.println(
                          "Texto 9\\ Texto 10");
8      } // Fin del método main
9  } // Fin de la clase SecuenciasDeEscape
```

```
Texto 1
Texto 2
Texto 3    Texto 4
Texto 6
"Texto 7" Texto 8
Texto 9\ Texto 10
```

Figura 2.4 | Imprimir texto que incluye secuencias de escape.

En la línea 3 del código código,

```
3   System.out.println("Texto 1\nTexto 2");
```

se imprime en pantalla el texto `Texto 1`, después se tiene una barra diagonal invertida seguida de la letra **n**, `\n`, lo que inserta un salto de línea, por último, se imprime el texto `Texto 2`. El método `println` imprime el texto que recibe dentro de los paréntesis y, una vez que finaliza, posiciona el cursor en la siguiente línea.

En la línea 4 del código,

```
4 | System.out.println("Texto 3\tTexto 4");
```

se imprime en pantalla el texto `Texto 3`, la barra diagonal invertida seguida de la letra **t**, `\t`, inserta un espacio de tabulación, por último, se observa que se imprime el texto `Texto 4` y el cursor se posiciona en la siguiente línea.

En la línea 5,

```
5 | System.out.println("Texto 5\rTexto 6");
```

se imprime en pantalla el texto `Texto 5`, la barra diagonal invertida seguida de la letra **r**, `\r`, inserta retorno de carro, lo que implica que el cursor regresa al inicio de la línea. Por último, se imprime el texto `Texto 6`, pero como el cursor se encontraba al inicio de la línea, el texto que ya se tenía es sobrescrito, es por ese motivo que no se observa en pantalla el texto `Texto 5`. Una vez que el método `println` termina de imprimir el valor recibido dentro de los paréntesis, el cursor se posiciona en la siguiente línea.

La línea 6,

```
6 | System.out.println("\"Texto 7\" Texto 8");
```

inicia con una barra diagonal invertida seguida del carácter de doble comilla, `\"`, esta secuencia de escape inserta el carácter de doble comilla, esto debe ser considerado porque de no hacerlo, Java interpretaría las dobles comillas como el final de la cadena de caracteres. Es así como, vemos en pantalla el texto `Texto 7` encerrado entre comillas dobles en el resultado. Posteriormente de que se imprime la segunda secuencia de escape, `\"`, se imprime un espacio en blanco seguido del texto `Texto 8`. Puesto que el método `println` ha terminado de imprimir el valor recibido dentro de los paréntesis, el cursor se posiciona en la siguiente línea.

En la línea 7,

```
7   System.out.println("Texto 9\\ Texto 10");
```

se imprime el texto `Texto 9`, la secuencia de escape `\\` se imprime como una sola barra diagonal invertida, `\`, después se imprime un espacio en blanco seguido del texto `Texto 10`. Puesto que el método `println` ha terminado de imprimir el valor recibido dentro de los paréntesis, el cursor se posiciona en la siguiente línea.

En la línea 8,

```
8   } // Fin del método main
```

se indica el final del cuerpo del método `main`.

Por último, en la línea 9,

```
9   } // Fin de la clase SecuenciasDeEscape
```

se termina el cuerpo de la clase `SecuenciasDeEscape`.

2.4Tipos de datos y variables

Al realizar programas, es necesario manejar distintos datos, éstos pueden ser textos, en otros casos, probablemente se interactúe con números. En Java, en términos generales se tienen dos tipos de datos: los tipos de datos simples y los tipos de datos por referencia. Los tipos de **datos simples**, en Java se conocen como tipos de datos primitivos, por el momento, únicamente nos enfocaremos en los tipos de **datos primitivos**, los tipos de **datos por referencia** se abordarán a profundidad en los temas de Clases y Objetos.

Los tipos de **datos primitivos** permiten trabajar con números, valores de verdad (booleanos), y caracteres. Hasta

el momento se ha trabajado con textos, en Java los textos son de tipo **`String`**, sin embargo, estos datos no son parte de los tipos de datos primitivos. Por el momento, probablemente resulte confuso el concepto de **tipos de datos primitivos**, también conocidos como **tipos de datos simples** en otros lenguajes de programación, pero se irá ampliando este concepto conforme avancemos en nuestro aprendizaje. Lo primero por recordar es que Java cuenta con 8 tipos de datos primitivos. Éstos se pueden ver en la figura 2.5.

Tipo	Tamaño en bits	Valor mínimo	Valor máximo	Valor por defecto (para campos)
byte	8	-128	127	0
short	16	-32,768	32,767	0
int	32	-2^{31}	$2^{31}-1$	0
long	64	-2^{63}	$2^{63}-1$	0L
float	32	Precisión simple IEEE 754 de 32 bits		0.0f
double	64	Doble precisión IEEE 754 de 64 bits		0.0d
boolean	No específica	false o true		false
char	16	'\u0000'	'\uffff'	'\u0000'

Figura 2.5 | Tipos de datos primitivos en Java

Los tipos de datos **byte**, **short**, **int** y **long** permiten trabajar con números enteros. Los tipos **float** y **double,** se usan para trabajar con valores con punto decimal. El tipo de dato **boolean**, se utiliza para trabajar con valores de verdad, los únicos valores de tipo **boolean** son **`false`** y **`true`**. El tipo de dato **char** es un carácter Unicode. Unicode es un estándar que define los caracteres necesarios para poder escribir la mayoría de los idiomas. Todos los tipos de datos primiti-

vos son palabras reservadas, por lo que se escriben siempre en minúsculas, y no se usan como identificadores, es decir, nombres de variables, clases u objetos.

Ahora, que se han conocido los tipos de datos primitivos, se identificaran las **variables**. Las variables permiten guardar y manipular datos en los programas. Imagine estar con un amigo en el supermercado, su amigo observa el precio de un paquete de galletas, lo toma y le pide a usted que recuerde que el precio de las galletas es de 2.25 dólares. Al llegar a la caja, su amigo coloca todos los productos, con excepción del paquete de galletas, la cajera revisa y pasa por la caja cada uno de ellos, al finalizar, ella menciona el costo total. Su amigo observa el dinero con el que cuenta, y le pregunta a usted por el precio del paquete de galletas, usted le menciona el precio memorizado, 2.25 dólares, su amigo observa que sí cuenta con suficiente dinero y agrega el paquete de galletas a la compra. Pero bueno, ¿era realmente necesaria toda esta historia para comprender lo que son las variables?, probablemente no, pero sirve como analogía. Cuando se realizan programas, es común la necesidad de guardar información de manera temporal para realizar operaciones más adelante, para eso utilizaremos las variables. En el ejemplo del supermercado, se realizaron varios pasos, al igual que un programa informático consta de una serie de pasos ordenados; en este ejemplo, usted fungió como **variable** al almacenar información de forma temporal. Cuando su amigo necesito de la información, le preguntó por el precio del paquete de galletas.

Los conceptos de **variables** y **tipo de datos primitivos** son trabajados en el siguiente ejemplo:

Variables.java

```
1  class Variables {
2   public static void main(String[] args) {
3          byte byteDato = 100;
4          short shortDato = 1000;
5          int intDato = 1000000;
6          long longDato = 1000000000000L;
7          float floatDato = 100.25f;
8          double doubleDato = 2000.35;
9          boolean booleanDato = true;
10         char charDato = '\u0040';
11
12         System.out.print("Dato byte: ");
13         System.out.println(byteDato);
14          System.out.print("Dato short: ");
15         System.out.println(shortDato);
16         System.out.print("Dato int: ");
17         System.out.println(intDato);
18         System.out.print("Dato long: ");
19         System.out.println(longDato);
20         System.out.print("Dato float: ");
21         System.out.println(floatDato);
22          System.out.print("Dato double: ");
23         System.out.println(doubleDato);
24          System.out.print("Dato boolean: ");
25         System.out.println(booleanDato);
26         System.out.print("Dato char: ");
27         System.out.println(charDato);
28     } // Fin del método main
29 } // Fin de la clase Variables
```

```
Dato byte: 100
Dato short: 1000
Dato int: 1000000
Dato long: 1000000000000
Dato float: 100.25
Dato double: 2000.35
Dato boolean: true
Dato char: @
```

Figura 2.5 | Uso de variables con tipos de datos primitivos

Revise y analice el programa de la figura 2.5. Con base en el resultado obtenido, se sugiere intentar descifrar las acciones que realiza cada línea del programa. Las partes sombreadas del código de la figura 2.5 han sido puestas de ese modo intencionalmente, con el propósito de resaltar los objetivos principales de aprendizaje en el programa.

El método `main` de nuestro programa inicia definiendo variables de diferentes tipos. En la línea 3,

```
3   byte byteDato = 100;
```

se realiza la definición de una variable de tipo **`byte`** que lleva por identificador el nombre `byteDato`, y tiene asignado el valor numérico de 100. Observe que este patrón ocurre desde la línea 3 hasta la línea 10, en todas estas líneas se están declarando variables. Entonces, ¿qué se necesita para definir variables? Lo primero es **indicar el tipo de dato**, después se debe **dar un nombre** y, por último, se le **asigna un valor**; el valor que se asigna debe corresponder con el tipo de dato que se esté declarando, de lo contrario, el programa no compilará.

Se observa que el valor numérico asignado en la línea 6,

```
6 | long longDato = 1000000000000L;
```

incluye la letra **L**. Generalmente, no se tendrá problema en asignar un número entero sin la letra **L** a una variable de tipo **long**, a excepción de cuando el valor entero exceda de **2,147,483,647**, es decir, $2^{31}-1$; tal como sucede en el valor asignado en la línea 6. Si la línea 6 no incluyera la letra **L** al final del valor numérico, el programa no compilaría.

El valor numérico asignado en la línea 7,

```
7 | float floatDato = 100.25f;
```

incluye la letra **f**. Siempre que se asigne un valor con punto decimal a una variable de tipo **float**, se debe agregar la letra **f** al final de dicho valor, de lo contrario, el programa no compilará; esto se debe a que los valores con punto decimal, en Java por defecto, son de tipo **double**, así que se debe agregar la letra **f** para indicar que el valor numérico es de tipo **float**.

La línea 9,

```
9 | boolean booleanDato = true;
```

define una variable de tipo **boolean** que será identificada como **booleanDato** y le es asignado el valor de **true**. Los únicos posibles valores que pueden ser asignados a una variable de tipo **boolean** es **true** y **false**.

La línea 10,

```
10  char charDato = '\u0040';
```

define una variable de tipo **char**, el valor asignado es **'\u0040'**, el número 0040 es un valor hexadecimal que representa el carácter @ en la tabla de símbolo Unicode. Los tipos de datos **char** también se asignan, encerrando al carácter entre comillas simples, ' '; por ejemplo, la línea 10, también se pudo escribir como:

```
char charDato = '@';
```

La línea 12,

```
12  System.out.print("Dato byte: ");
```

hace uso del método print, éste imprime el texto "Datos byte: ". La diferencia entre usar el método print y el método println, es que el método println agrega un salto de línea luego de imprimir el texto, es decir, posiciona el cursor en la siguiente línea, mientras que el método print imprime el texto que recibe y mantiene el curso en la última posición de la misma línea. Es por lo que el resultado de las líneas 12 y 13 es Dato byte: 100 en la misma línea.

La línea 13,

```
13  System.out.println(byteDato);
```

hace uso del método println para imprimir el valor que posee la variable byteDato. El método println puede recibir cualquier tipo de datos, en esta línea imprime el valor que posee la variable byteDato, el cual es el valor de 100.

Desde la línea 14 hasta la línea 27,

```
14  System.out.print("Dato short: ");
```

```
15    System.out.println(shortDato);
16    System.out.print("Dato int: ");
17    System.out.println(intDato);
18    System.out.print("Dato long: ");
19    System.out.println(longDato);
20    System.out.print("Dato float: ");
21    System.out.println(floatDato);
22    System.out.print("Dato double: ");
23    System.out.println(doubleDato);
24    System.out.print("Dato boolean: ");
25    System.out.println(booleanDato);
26    System.out.print("Dato char: ");
27    System.out.println(charDato);
```

se imprimen textos y valores almacenados en variables, de forma similar a las líneas 12 y 13.

Como se observa, las variables nos brindan la posibilidad de guardar datos que se utilizan durante la ejecución del programa. En este programa, se asignó un valor que posteriormente es utilizado, sin embargo, más adelante se ejemplifica cómo remplazar el valor inicial de las variables por un nuevo valor.

2.5Concatenación de textos

Cuando se desarrollan programas, en ocasiones se necesitan crear nuevos textos con base en otros datos, por ejemplo, si se tiene una variable con un nombre y una variable con el primer apellido, se puede formar un nuevo texto conforma-

do por ambos textos, a esto se llama **concatenar texto**, y se realiza usando el operador **+**. Vease un ejemplo:

Concatenacion.java

```
1  class Concatenacion {
2    public static void main(String[] args) {
3      String nombre = "John";
4      String primerApellido = "Doe";
5      String nombreConApellido =
                  nombre + " " + primerApellido;
6
7      System.out.println(nombreConApellido);
8    } // Fin del método main
9  } // Fin de la clase Concatenacion
```

```
John Doe
```

Figura 2.6 | Concatenación de textos

El programa de la figura 2.6 define 3 objetos de tipo **`String`**. ¿`Objetos`? ¿`String`? ¿Qué es eso? Bueno, no se ha profundizado en esos temas y, de hecho, por ahora se seguirá sin profundizar en esos temas puesto que, es muy probable que, si se intenta comprender lo que son los objetos en estos momentos, puede ser confuso. El tema de Objetos y clases se irá mostrando conforme se avance y adquieran las bases necesarias para entenderlos de forma apropiada, se le asegura abordar el tema de los objetos cuando se consideremos pertinente, por ahora, basta con tomar consciencia de que las líneas 3, 4 y 5 están definiendo objetos de tipo `String`. Al igual que cuando se escribe un número entero, por defecto, este valor es de tipo `int`; cuando se tiene un conjunto de caracteres encerrados entre dobles comillas, por ejemplo, "`John`", éste, por defecto, es un valor de tipo `String`.

La línea 3 define un objeto de tipo `String` que posee el valor de `"John"` y se identifica con el nombre de `nombre`. La línea 4 define un objeto de tipo `String` que posee el valor de `"Doe"` y se identifica con el nombre de `primerApellido`. La línea 5 declara un objeto de tipo `String` con `nombreConApellido` como identificador, y le es asignado el resultado de concatenar los textos del objeto `nombre` con el texto `" "`, el cual es un espacio en blanco, y el texto del objeto `primerApellido`; por lo que, realmente se asigna el resultado de unir los textos de `"John"`, `" "`, y `"Doe"`; es así que, en la línea 7, al imprimir el texto que posee el objeto `nombreConApellido`, vemos en pantalla `"John Doe"`.

2.6 Operadores aritméticos

En Java, al igual que en otros lenguajes de programación, se dispone de operadores que nos permiten realizar cálculos aritméticos. La figura 2.7 muestra los operadores aritméticos disponibles en Java.

Operación	Operador	Ejemplo	Resultado
Suma	+	5 + 3	8
Resta	-	5 - 3	2
Multiplicación	*	5 * 3	15
División	/	5 / 3	1
Residuo (Módulo)	%	5 % 3	2

Figura 2.7 | Operadores aritméticos en Java

Seguramente, se reconocen a la mayoría de los operadores aritméticos que se muestran en la figura 2.7, el único operador que podría llegar a ser nuevo sería el **operador de residuo**, **%**, éste divide un valor numérico con otro, y devuelve el residuo de la división; en el ejemplo, **5 % 3**, al dividir **5** entre

3, el resultado sería **1**, y el residuo sería **2**, por lo que el resultado de **5 % 3** es **2**. Así también, se observa que, en el ejemplo del **operador de división**, **/**, éste devolvió como resultado un valor entero; esto sucede así porque en Java las operaciones con números enteros, devuelven números enteros, pero esto quedará más claro después del siguiente programa:

OperadoresAritmeticos.java

```
1  class OperadoresAritmeticos {
2    public static void main(String[] args) {
3      int numero1 = 7;
4      int numero2 = 5;
5
6      System.out.println("Operaciones con
   \"numero1\" y \"numero2:\"");
7      System.out.println( numero1 + numero2 );
8      System.out.println( numero1 - numero2 );
9      System.out.println( numero1 * numero2 );
10     System.out.println( numero1 / numero2 );
11     System.out.println( numero1 % numero2 );
12   } // Fin del método main
13 } // Fin de la clase OperadoresAritméticos
```

```
Operaciones con "numero1" y "numero2:"
12
2
35
1
2
```

Figura 2.7 | Uso de operadores aritméticos en Java

El programa de la figura 2.7 define dos variables de tipo `int`, identificadas como **`numero1`** y **`numero2`**. En la línea 6 del código del programa se muestra un texto que, para lograr

imprimir el carácter de comillas dobles, ", hace uso de la secuencia de escape, \", es así como la línea 6 imprime el texto `Operaciones con "numero1" y "numero2:"`.

Desde la línea 7 hasta la línea 11, se imprime el resultado de varias operaciones aritméticas realizadas con los valores que poseen las variables **`numero1`** y **`numero2`**. Al ejecutar el programa, se observa que el resultado de sumar **`numero1`** y **`numero2`**, es decir, `7+5`, es `12`. La **resta** de los valores de las variables es `2`, al **multiplicar** resulta `35`, al **dividir** es `1`, y el **residuo** es `2`.

Cuando se realizan operaciones aritméticas con valores enteros, Java da como resultado un valor de tipo entero; es por eso por lo que, cuando realizamos la operación `7/5`, Java devuelve `1` como resultado. Entonces, ¿qué pasaría si en lugar de definir variables de tipo `int`, se hubiésen definido variables de tipo `double`? Veamos.

OperadoresAritmeticos.java

```
1  class OperadoresAritmeticos {
2    public static void main(String[] args) {
3      double numero1 = 7.0;
4      double numero2 = 5.0;
5
6      System.out.println("Operaciones con
   \"numero1\" y \"numero2:\"");
7      System.out.println( numero1 + numero2
   );
8      System.out.println( numero1 - numero2
   );
9      System.out.println( numero1 * numero2
   );
10     System.out.println( numero1 / numero2
   );
```

```
11      System.out.println( numero1 % numero2
    );
12    } // Fin del método main
13  } // Fin de la clase OperadoresAritméticos
```

```
Operaciones con "numero1" y "numero2:"
12.0
2.0
35.0
1.4
2.0
```

Figura 2.8 | Uso de operadores aritméticos en Java con valores de tipo double.

En la figura 2.8, se realizaron cambios ligeros al programa que se tenía en la figura 2.7. Los cambios han sido resaltados en las líneas 3 y 4. ¿Qué cambios son notorios al ejecutar el programa? Lo primero es que ahora los resultados incluyen valores con punto decimal; así también, se observa que el resultado de la división es **1.4**. Entonces, se puede deducir que cuando realizamos operaciones con valores `int` en Java, el resultado es un valor `int`; y cuando se realizan operaciones con valores `double` en Java, el resultado es un valor `double`; eso ha quedado claro, ¿y qué sucede cuando la operación involucra valores de distintos tipos?, por ejemplo, un valor `int` y un valor `double`; si se define la variable `numero1` como `int` y se deja la variable `numero2` como `double`; el único cambio corresponde a la línea 3, la línea 4 permanecerá igual:

```
3   int numero1 = 7;
4   double numero2 = 5.0;
```

Al ejecutar nuevamente el programa y observar el resultado:

```
 Operaciones con "numero1" y "numero2:"
12.0
2.0
35.0
1.4
2.0
```

¿Se nota algún cambio en el resultado?, ¿no?

Efectivamente, al ejecutar el programa, los valores que se imprimen siguen siendo los mismos que se tenían anteriormente. Entonces, ¿cómo resuelve Java las operaciones con valores de distintos tipos? Cuando Java se encuentra con operaciones con valores de tipos diferentes, el resultado lo entrega con respecto al tipo de dato de mayor jerarquía numérica. El tipo de dato de mayor jerarquía es `double`, y el tipo de dato de menor jerarquía es `byte`. Entonces, cuando Java tiene que resolver la división de un valor `int` con un valor `double`, por ejemplo, `7/5.0`, el resultado se entrega como `double`, puesto que, `double` tiene mayor jerarquía que un tipo de dato `int`.

En la figura 2.9, se muestra la jerarquía de los tipos de datos numéricos.

Jerarquía	Tipo
Mayor	double
	float
	long
	int
	short
Menor	byte

Figura 2.9 | Jerarquía de tipos de datos numéricos

2.7Leer valores de entrada

Una parte importante de los programas es la posibilidad de solicitar datos al usuario, si se tiene un programa que realiza sumas, seguramente lo valores numéricos no serán los mismos con cada ejecución. Para continuar con el aprendizaje, se procederá a solicitar al usuario que introduzca información. La figura 2.10 muestra un programa que solicita al usuario 2 valores enteros y muestra la suma de los valores recibidos.

Lectura.java

```
1  import java.util.Scanner;
2  class Lectura {
3    public static void main(String[] args) {
4      Scanner entrada= new Scanner(System.in);
5
6      System.out.print(
   "Ingrese un valor entero: ");
7      int numero1 = entrada.nextInt();
8
9          System.out.print("Ingrese otro valor
                                    entero: ");
10     int numero2 = entrada.nextInt();
11
12     int suma = numero1 + numero2;
13     System.out.println(
                          "La suma es " +  suma);
14   } // Fin del método main
15 } // Fin de la clase Lectura
```

```
Ingrese un valor entero: 10
Ingrese otro valor entero: 20
La suma es 30
```

Figura 2.10 | Lectura de valores del usuario.

En la figura 2.10 se resaltan los fragmentos de código que son nuevos en el proceso de aprendizaje. Las líneas 1 y 4 son necesarias para que el programa tenga la capacidad de solicitar datos al usuario. En estos momentos, aún no cuenta con las bases de conocimiento suficientes para entender por completo la línea 4, cuando se estudie el tema de clases y objetos se comprenderá la línea en su totalidad, si no es que antes. Por ahora, es normal que los conceptos de clases, objetos y métodos resulten un poco confusos.

Volviendo al programa, la línea 4,

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

hace uso de la clase **Scanner** para crear un objeto. Así como se definieron las clases anteriores, Java cuenta con muchas clases que han sido creadas para que se puedan ser usadas; la clase **Scanner** es una de esas clases, esta clase cuenta con varios métodos que permiten leer información del usuario. *¿Cómo se accede a esos métodos?* Para hacer uso de los métodos de la clase `Scanner`, primero se requiere crear un objeto de esta clase, esto es realizado en la línea 4 del programa; una vez que se dispone del objeto de tipo **Scanner**, es posible acceder a los métodos, al escribir el nombre del objeto seguido de un punto y el nombre del método que se desea invocar. En las líneas 7 y 10,

```
7   int numero1 = entrada.nextInt();
```

```
10  int numero2 = entrada.nextInt();
```

se realiza un llamado al método `nextInt` de este modo, es decir, `entrada.nextInt()`. Cuando el método `nextInt` es invocado, éste espera a que el usuario introduzca un valor entero y presione la tecla *Intro*, una vez que esto sucede, el método `nextInt` entrega el valor que el usuario ingresó. En la línea 7, el valor ingresado por el usuario es asignado a

la variable `numero1`, y en la línea 10, el valor que el usuario ingresa es asignado a la variable `numero2`.

Ya se tiene una idea general de cómo obtener valores del usuario a través de un objeto de tipo **Scanner**, sin embargo, para que Java reconozca los objetos de tipo **Scanner,** es necesario realizar una importación de dicha clase, esto se debe realizar antes de comenzar a escribir la clase de nuestro programa. La línea 1 muestra cómo se realiza la importación de la clase **Scanner**. Como se puede observar, al ejecutar el programa, se solicita 2 veces que se introduzca un valor entero, una vez que se ingresan los 2 valores, se imprime en pantalla el resultado de sumar ambos valores.

Por último, observe que en la línea 13 se pide imprimir el resultado de `"La suma es " + suma`, dicho de otra manera, la operación a realizar involucra un valor de tipo `String` y un valor de tipo `int`, ¿cómo resuelve Java este tipo de operación?, cuando Java encuentra este tipo de operaciones, devuelve como resultado un objeto de tipo `String`, es decir, todo se convierte a texto; es así que, con base en los valores ingresados en el ejemplo, el texto resultante es `La suma es 30`.

2.8Conclusión

En este capítulo, estudió la estructura básica de un programa en Java, cómo imprimir textos y las forma de concatenarlos. Se identificaron los tipos de datos primitivos y cómo utilizarlos para definir variables. También, se enseñó a trabajar con variables y a realizar operaciones aritméticas. Por último, se aprendió a leer valores enteros del usuario a través de la clase `Scanner`. En el próximo capítulo, enseña a controlar las sentencias de los programas a través de las **estructuras de control**.

3.**Estructuras de control**

3.1Introducción

La ejecución de las sentencias en los programas se controla de 3 formas: *secuencial*, *de selección*, y *de repetición*; la **estructura secuencial** es la más básica de todas, es la forma de control que se ha realizado hasta el momento. Con la *estructura secuencial*, todas las instrucciones se ejecutan de forma lineal, es decir, una instrucción no puede ser ejecutada hasta que la instrucción anterior haya sido ejecutada. En este capítulo se revisan las sentencias que permiten controlar la forma en que las instrucciones serán ejecutadas, primero las **estructuras de selección**, éstas permiten tomar decisiones con respecto a qué instrucciones deberían ser ejecutadas; posteriormente, las **estructuras de repetición** con las que se ejecuta un conjunto de instrucciones de manera repetitiva. Tanto las *estructuras de selección*, como las *de repetición*, necesitan de un valor de verdad, es decir, un valor `true` o un valor `false`; estos valores suelen ser asignados de forma directa, pero generalmente, el valor de verdad se obtiene al evaluar alguna condición con los **operadores relacionales,** con los **operadores de igualdad** y con los **operadores lógicos**, todos estos operadores serán abordados en este capítulo.

3.2Operadores relacionales y de igualdad

Los **operadores relacionales** permiten comparar valores numéricos entre sí; las comparaciones pueden ser con respecto a si el primer valor es mayor al segundo o si el primer valor es menor al segundo; por otro lado, los **operadores de igualdad** nos permiten saber si dos valores son iguales, o si dos valores son diferentes. La figura 3.1 muestra los *operadores relacionales* y los *de igualdad*.

Operador	Descripción	Ejemplo	Resultado
>	¿Mayor a …?	3 > 2	true
<	¿Menor a …?	3 < 3	false
>=	¿Mayor o igual a …?	3 >= 3	true
<=	¿Menor o igual a …?	3 <= 2	false
==	¿Igual a …?	3 == 3	true
!=	¿No es igual a …?	3 != 3	false

Figura 3.1 | Operadores relacionales y de igualdad en Java

En general, los operadores mostrados en la figura 3.1 permiten comparar valores, se debe prestar atención al modo de escritura en Java, por ejemplo, si se tuviera `numero1 == numero2`, lo que se preguntan es si el valor de la variable `numero1` es igual al valor de la variable `numero2`. Por otra parte, preste atención a los operadores >= y <=, estos operadores devolverán `true` en caso de que se cumpla cualquiera de las dos condiciones que manejan. En los ejemplos de la figura 3.1 se observa que el resultado de `3 >= 3` es `true`, esto es porque aunque `3` no es mayor que `3`, el valor de `3` sí es igual a `3`, y con que una de las dos opciones se cumpla, el resultado será `true`.

Comprobemos el uso de los operadores de la figura 3.1.

Comparaciones.java

```
1  import java.util.Scanner;
2  public class Comparaciones {
3    public static void main(String[] args) {
4      Scanner entrada= new Scanner(System.
   in);
5      int numero1;
6      int numero2;
7
8      System.out.print(
                 "Ingrese un número entero: ");
9      numero1 = entrada.nextInt();
10
11     System.out.print(
              "Ingrese otro número entero: ");
12     numero2 = entrada.nextInt();
13
14     System.out.print(numero1 +
             " es mayor a " + numero2 + ": ");
15     System.out.println(numero1 > numero2 );
16
17     System.out.print(numero1 +
             " es menor a " + numero2 + ": ");
18     System.out.println(numero1 < numero2 );
19
20     System.out.print(numero1 +
       " es mayor o igual a "+numero2 + ": ");
21     System.out.println(numero1 >= numero2
   );
22
23     System.out.print(numero1 +
       " es menor o igual a "+numero2 + ": ");
24     System.out.println(numero1 <= numero2
   );
```

```
25
26      System.out.print(numero1 +
              " es igual a " + numero2 + ": " );
27      System.out.println(numero1 == numero2
    );
28
29      System.out.print(numero1 +
              " no es igual a " + numero2 + ": " );
30      System.out.println(numero1 != numero2 );
31    } // Fin del método main
32  } // Fin de la clase Comparaciones
```

```
Ingrese un número entero: 5
Ingrese otro número entero: 7
5 es mayor a 7: false
5 es menor a 7: true
5 es mayor o igual a 7: false
5 es menor o igual a 7: true
5 es igual a 7: false
5 no es igual a 7: true
```

Figura 3.2 | Comparaciones de valores numéricos

Al analizar el programa de la figura 3.2. La línea 1,

```
1   import java.util.Scanner;
```

indica que dentro del programa se usa de la clase `Scanner`. En la línea 2,

```
2   public class Comparaciones {
```

se define la clase que llevará por nombre `Comparaciones`. Esta línea termina con la apertura de llaves, `{`, lo que indica el inicio del cuerpo de nuestra clase. En la línea 3,

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

se define el método `main`, el cual es indispensable para que el programa pueda ser ejecutado por Java. La línea 3 finaliza con la apertura de llaves, `{`, lo que indica el inicio del cuerpo del método `main`. En la línea 4,

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

se crea un objeto de la clase `Scanner` que servirá para solicitar valores al usuario.

La línea 5 y 6,

```
5 | int numero1;
6 | int numero2;
```

realizan la declaración de dos variables de tipo `int` identificadas como `numero1` y `numero2`. Estas variables no fueron inicializadas con un valor. Cuando en un programa se dice que se va a utilizar una variable de cierto tipo, pero no se asigna un valor, entonces se está **declarando** a esa variable. Por el contrario, cuando se dice que se va a utilizar una variable de cierto tipo y la inicializamos, se está **definiendo** a esa variable. Por lo que en la línea 5 y 6, se están declarando las variables `numero1` y `numero2`, de tipo `int`.

```
8 | System.out.print(
  |             "Ingrese un número entero: ");
9 | numero1 = entrada.nextInt();
```

La línea 8 imprime un texto indicando al usuario que debe ingresar un valor entero. En la línea 9, se realiza la lectura del valor introducido por el usuario a través del método `nextInt`, el valor leído es asignado a la variable `numero1`. Para el ejemplo de la figura 3.2, el valor ingresado es `5`.

```
11  System.out.print(
                "Ingrese otro número entero: ");
12  numero2 = entrada.nextInt();
```

La línea 11 imprime un texto indicando al usuario que debe ingresar otro valor entero. En la línea 12 se realiza la lectura del valor introducido por el usuario a través del método `nextInt`, el valor leído es asignado a la variable `numero2`. Para el ejemplo de la figura 3.2, el valor ingresado es `7`.

```
14  System.out.print(numero1 +
              " es mayor a " + numero2 + ": ");
15  System.out.println(numero1 > numero2 );
```

En la línea 14 se imprime la concatenación de varios valores con base en los valores ingresados en el ejemplo de ejecución. El texto que se imprime es `5 es mayor a 7: `. En vista de que el texto se imprime con el método `print`, el cursor permanece en la misma línea después de imprimir el texto en pantalla. En la línea 15 se comparan los valores de las variables `numero1` y `numero2` a través del operador `>`, es decir, se imprime el resultado de la operación `numero1 < numero2`, lo que da como resultado `false`, con base en los valores ingresados de `5` y `7`.

Las líneas 17, 20, 23, 26 y 29,

```
17  System.out.print(numero1 +
              " es menor a " + numero2 + ": ");
```

```
20  System.out.print(numero1 +
       " es mayor o igual a "+numero2 + ": ");
```

```
23  System.out.print(numero1 +
       " es menor o igual a "+numero2 + ": ");
```

```
26  System.out.print(numero1 +
            " es igual a " + numero2 + ": " );
```

```
29   System.out.print(numero1 +
          " no es igual a " + numero2 + ": " );
```

del mismo modo que ocurrió en la línea 14, imprimen textos conformados por la concatenación de varios valores. Los textos resultantes son mostrados en el ejemplo de la figura 3.2.

Las líneas 18, 21, 24, 27 y 30,

```
18   System.out.println(numero1 < numero2 );
```

```
21   System.out.println(numero1 >= numero2 );
```

```
24   System.out.println(numero1 <= numero2 );
```

```
27   System.out.println(numero1 == numero2 );
```

```
30   System.out.println(numero1 != numero2 );
```

realizan comparaciones con los valores de las variables `numero1` y `numero2`, con los operadores de **<**, **>=**, **<=**, **==** y **!=**, respectivamente. Los resultados para los valores ingresados de 5 y 7 se pueden observar en la figura 3.2.

Ejecute y pruebe el programa de la figura 3.2 con otras entradas de valores enteros hasta que, con seguridad, comprenda el funcionamiento de los *operadores relacionales* y *de igualdad*.

3.3Operadores lógicos

Ahora, ya se identificaron los *operadores relacionales* y *de verdad*, Éstos serán de utilidad para controlar el flujo de ejecución en los programas. Otros tipos de operadores que serán útiles para controlar el flujo en los programas son los **ope-**

radores lógicos. La figura 3.3 resume los operadores lógicos más comunes.

Operador	Descripción	Ejemplo	Resultado
&&	Devuelve **true**, sólo cuando ambos valores son **true**.	true && true	true
		true && false	false
		false && true	false
		false && false	false
||	Devuelve **true**, si por lo menos uno de los valores es **true**.	true || true	true
		true || false	true
		false || true	true
		false || false	false
!	Devuelve el **true**, si el valor es **false**; y devuelve **false**, si el valor es **true**.	!true	false
		!false	true

Figura 3.3 | Operadores lógicos comunes en Java

En la figura 3.3, se muestran los 3 principales *operadores lógicos* utilizados en Java. El operador `&&`, se conoce como operador **AND**, este operador devuelve `true`, solamente cuando ambos valores que evalúa son `true`, con que uno de los valores sea `false`, entonces el resultado será `false`. Por otra parte, al operador `||`, se conoce como operador **OR**, con este operador, el resultado es `true`, siempre y cuando, por lo menos, uno de los valores sea `true`. Dicho de

otro modo, sólo devuelve `false` cuando ambos valores son `false`. Por último, el operador `!`, es el operador **NOT**, este operador invierte el valor de verdad, es decir, cuando evalúa un valor `true`, devuelve un valor `false` y cuando evalúa un valor `false`, devuelve un valor `true`.

OperacionesLogicos.java

```
14      System.out.println( false || false );
1   public class OperadoresLogicos {
2     public static void main(String[] args) {
3       System.out.println(
    "Operador AND (&&):");
4       System.out.println( true && true );
5       System.out.println( true && false );
6       System.out.println( false && true );
7       System.out.println( false && false );
8       System.out.println();
9
10      System.out.println("Operador OR
    (||):");
11      System.out.println( true || true );
12      System.out.println( true || false );
13      System.out.println( false || true );
15      System.out.println();
16
17      System.out.println("Operador NOT
    (!):");
18      System.out.println( !true );
19      System.out.println( !false );
20    } // Fin del método main
21  } // Fin de la clase OperadoresLogicos
```

```
Operador AND (&&):
true
false
false
false
Operador OR (||):
true
true
true
false
Operador NOT (!):
false
true
```

Figura 3.4 | Uso de los principales operadores condicionales en Java

Al analizar el programa de la figura 3.4. La línea 3 imprime en el texto `Operador AND (&&)` **:** en pantalla, y después imprime un salto de línea.

La línea 4 imprime el resultado de la operación `true && true`. Es preciso recordar que el operador `&&` sólo devuelve `true` cuando ambos valores son `true`, y como en esta operación, se evalúan dos valores `true`, el resultado que se imprime es `true`. Para las líneas 5, 6 y 7 también se evalúan valores con el operador `&&`, pero en todas ellas, por lo menos, un valor es `false`, por lo que el resultado para cada una de estas líneas es `false`.

La línea 8 hace uso del método `println` sin argumentos, es decir, sin recibir valores para imprimir un salto de línea.

La línea 10 imprime en pantalla el texto `Operador OR (||)` **:**. En las líneas 11, 12, 13 y 14, se evalúan valores con el operador `||`, este operador devolverá `true` con que uno de los valores evaluados sea `true`, y sólo devuelve `false`, si ambos valores son `false`. Tomando en cuenta la forma en que el operador `||` evalúa los valores, las líneas 11, 12 y

13 imprimen `true`, puesto que al menos uno de los valores evaluados es `true`, y sólo la línea 14, que evalúa la operación `false || false` devuelve `false`.

La línea 15 hace uso del método `println` sin usar argumentos para imprimir un salto de línea.

La línea 17 imprime el texto `Operador NOT (!):`, y posteriormente imprime un salto de línea, puesto que se hace uso del método `println`.

La línea 18 evalúa un valor `true` con el operador `!`. El valor a imprimir es el resultado de la operación `!true`. El resultado es `false`, por lo que ése es el valor que se observa en pantalla.

La línea 19 imprime el resultado de la operación `!false`. El resultado que se imprime es `true`.

3.4Sentencia de selección if

Los operadores estudiados anteriormente permiten condicionar el flujo de la ejecución en los programas, ahora se necesita conocer las instrucciones que permiten agrupar sentencias que se ejecutarán con base en condiciones establecidas.

La sentencia de control de flujo más básica es la instrucción **`if`**. La instrucción **`if`** evalúa una condición, y si ésta es verdadera, es decir, da como resultado `true`, entonces ejecuta un conjunto de instrucciones. En general, la sintaxis de la instrucción **`if`** se muestra en la figura 3.5.

```
if ( CONDICIÓN ) {
   SENTENCIAS A REALIZAR
}
```

Figura 3.5 | Sintaxis de la sentencia **`if`**

La sentencia `if` espera un valor de verdad dentro de los paréntesis, éste es un valor de tipo `boolean`, es decir, se espera un valor `true` o un valor `false`. Evidentemente, no tendría mucho sentido asignar directamente un valor `true` o `false`. Generalmente, se tendrá una operación condicional que sirva para determinar si el conjunto de instrucciones se debe, o no, de ejecutar. Veamos cómo utilizar la instrucción `if` en un programa.

InstruccionIf.java

```
1  import java.util.Scanner;
2  public class InstruccionIf {
3    public static void main(String[] args) {
4      Scanner entrada= new Scanner(System.in);
5
6      System.out.print("Ingrese una edad: ");
7      int edad = entrada.nextInt();
8
9      if( edad >= 18 ){
10       System.out.println("Mayor de edad");
11     }
12     System.out.println("Adiós!");
13   } // Fin del método main
14 } // Fin de la clase InstruccionIf
```

```
Ingrese una edad: 18
Mayor de edad
Adiós!
```

Figura 3.6 | Uso de la instrucción if

El programa de la figura 3.6 solicita una edad al usuario, si la edad es mayor o igual a 18, entonces muestra el texto

"Mayor de edad". A continuación, se analiza con mayor detalle el programa.

Las líneas 1 y 4, se han estado utilizando en los programas en los que se solicitan datos de entrada al usuario. La línea 6 imprime el texto "Ingrese una edad: ", para indicar al usuario la acción que debe realizar. La línea 7 define la variable edad, ésta es de tipo int, y el valor asignado es aquél que devuelva el método nextInt, mismo que se encarga de leer el valor que el usuario introduce por teclado.

En la línea 9 se hace uso de la instrucción if. La operación dentro de los paréntesis indica la condición para que las sentencias dentro del cuerpo del if sean ejecutadas. El cuerpo de la instrucción if, es decir, el inicio y el final, están delimitadas por las llaves de apertura y cierre, { }. En la figura 3.6 se muestra la ejecución cuando el usuario introduce una edad de 18, la condición a evaluar por la instrucción if es edad >= 18, para el ejemplo de esta ejecución, el valor de la variable edad es 18, por lo que la operación a evaluar es si *18 es mayor o igual a 18*, la respuesta es true porque aunque 18 no es mayor a 18, 18 sí es igual a 18, y la condición que se pide es que *18 sea mayor o igual a 18*. En vista de que la condición se cumple, entonces, todas las instrucciones encerradas por las llaves de la instrucción if son ejecutadas, en este programa, se imprime Mayor de edad.

Si se ejecuta nuevamente el programa de la figura 3.6, pero en esta ocasión se ingresa como edad el valor de 17 se observa que el resultado es distinto:

```
Ingrese una edad: 17
Adiós!
```

Al ingresar como edad el valor de 17, la condición de la línea 9 del código, edad >= 18, no se cumple, por lo que la línea 10 del programa no se ejecuta. Sin embargo, la línea 12, siempre se ejecutará, independientemente de que se cumpla

o no la condición de la instrucción `if`, puesto que la línea 12 no se encuentra dentro del cuerpo de la sentencia `if`.

3.5Sentencia de selección if-else

Ya se tuvo oportunidad de conocer la instrucción `if`, en la que se condicionaba si un conjunto de instrucciones debía o no ser ejecutado. Ahora se muestra la instrucción **`if-else`**. Aquí también se tiene una condición que determina si un conjunto de instrucciones debe ser ejecutado en caso de que ésta dé como resultado **`true`**, sin embargo, también se define otro conjunto de instrucciones que será ejecutado en caso de que la condición dé como resultado **`false`**. La sintaxis de la instrucción **`if-else`** se observa en la figura 3.7.

```
if ( CONDICIÓN ) {
   SENTENCIAS A REALIZAR
}else{
   SENTENCIAS A REALIZAR
}
```

Figura 3.7 | Sintaxis de la sentencia **`if-else`**

El programa de la figura 3.8 solicita una edad al usuario y muestra el texto de mayor de edad, si el valor es mayor o igual a 18, y muestra, en caso contrario. En el ejemplo de ejecución se introduce como edad el valor de 17.

InstruccionIfElse.java

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

```
6       System.out.print("Ingrese una edad: ");
7       int edad = entrada.nextInt();
8
9       if( edad >= 18 ){
10        System.out.println("Mayor de edad");
11      }else{
12        System.out.println("Menor de edad");
13      }
14      System.out.println("Adiós!");
15    } // Fin del método main
16  } // Fin de la clase InstruccionIfElse
```

```
Ingrese una edad: 17
Menor de edad
Adiós!
```

Figura 3.8 | Uso de la instrucción if-else

El programa de la figura 3.8 es muy parecido al último programa realizado, es decir, el de la figura 3.6, sin embargo, en este programa en lugar de utilizar una instrucción `if`, se está haciendo uso de la instrucción **`if-else`**. La diferencia es que la condición que tenemos en la línea 9, `edad >= 18`, si devuelve como resultado `true`, entonces ejecuta la línea 10, en caso contrario, se ejecuta el cuerpo de la parte **`else`** que, en este programa, corresponde a la sentencia de la línea 12. Tomando en cuenta lo anterior, como el valor que se recibe durante la ejecución del programa para la edad es **17**, las sentencias que se ejecutan son las de la parte `else`, y las de la parte `if` son ignoradas.

Por otro lado, al ejecutar el programa nuevamente, pero ingresamos como edad un valor mayor o igual a **18**,

```
Ingrese una edad: 18
Mayor de edad
Adiós!
```

se observa que el resultado de la ejecución corresponde con el texto de la parte `if`, es decir, la línea 10 y las sentencias de la parte `else` son ignoradas, en este caso, la línea 12 es ignorada.

3.6Sentencia de selección múltiple switch

La instrucción switch permite tener una serie de casos para ejecutar un conjunto de instrucciones con base en un valor. Para comprender cómo funciona la instrucción `switch`, es preciso revisar la sintaxis de ésta.

```
switch ( VALOR ENTERO ) {
   case 1:  SENTENCIAS;
            break;
   case 2:  SENTENCIAS;
            break;
   case N:  SENTENCIAS;
            break;
   default: SENTENCIAS;
            break;
   }
```

Figura 3.9 | Sintaxis de la sentencia **`switch`**

La figura 3.9 muestra la sintaxis de la instrucción `switch` con base en un valor entero, éstos pueden ser de tipo `int`, `short` o `byte`, sin embargo, también es posible utilizar `switch` con objetos `String`. La figura 3.10 detalla un programa que usa de la instrucción `switch` evaluando un valor `int`.

InstruccionSwitch.java

```
1  import java.util.Scanner;
2  public class InstruccionSwitch {
3    public static void main(String[] args) {
4      Scanner entrada= new Scanner(System.in);
5
6      System.out.print(
   "Ingrese un número de mes: ");
7      int mes = entrada.nextInt();
8      switch( mes ){
9        case 1: System.out.println("Enero");
10               break;
11       case 2: System.out.println("Febrero");
12               break;
13       case 3: System.out.println("Marzo");
14               break;
15       case 4: System.out.println("Abril");
16               break;
17       case 5: System.out.println("Mayo");
18               break;
19       case 6: System.out.println("Junio");
20               break;
21       case 7: System.out.println("Julio");
22               break;
23       case 8: System.out.println("Agosto");
24               break;
25       case 9: System.out.println(
                                  "Septiembre");
26               break;
27       case 10: System.out.println(
                                    "Octubre");
28               break;
```

```
29          case 11: System.out.println(
                                        "Noviembre");
30                   break;
31          case 12: System.out.println(
                                        "Diciembre");
32                   break;
33          default: System.out.println(
                                        "No válido");
34                   break;
35       } // Fin de la instrucción switch
36    } // Fin del método main
37 } // Fin de la clase InstruccionSwitch
```

```
Ingrese un número de mes: 7
Julio
```

Figura 3.10 | Uso de la instrucción switch

Este programa solicita al usuario que ingrese un número de mes, dicho valor es asignado a la variable `mes` que es definida en la línea 7. En la línea 8,

```
8   switch( mes ){
```

se hace uso de la instrucción `switch`. La instrucción evalúa el valor que posee la variable mes para determinar el caso que corresponde con el valor. La apertura de llave, `{`, indica el inicio del cuerpo de la instrucción `switch`. Dentro del cuerpo se deben definir los casos.

La línea 9,

```
9   case 1: System.out.println("Enero");
```

define el caso para cuando el valor es 1. Los casos se definen poniendo la palabra reservada case, seguida del valor que corresponde al caso, posteriormente, se agrega el signo de

dos puntos, :, éste indica el inicio de las sentencias que son ejecutadas para dicho caso. Para este caso, la única instrucción a ejecutar es la tarea de imprimir el texto “Enero”. Para indicar el final del conjunto de instrucciones a ser ejecutadas en dicho caso, es necesario poner la palabra reservada break, de este modo se termina la ejecución de la instrucción switch. La sentencia break es colocada en la línea 10, ésta indica el final de las instrucciones para el caso.

Desde la línea 11, hasta la línea 32, se definen los casos para cada valor del mes. La línea 33,

```
33 | default: System.out.println(“No válido”);
```

define el caso por defecto. Éste se utiliza en caso de que ninguno de los casos se cumpla, es decir, si el usuario llega a ingresar, por ejemplo, el valor de 13, se ejecuta el conjunto de instrucciones del caso por defecto, puesto que el valor que se evalúa no corresponde con ninguno de los casos.

Al ejecutar el programa de la figura 3.10, el resultado es “Julio”, puesto que fue ingresado por el usuario el valor de 6.

```
Ingrese un número de mes: 7
Julio
```

3.7Sentencia if-else-if

Algunos programadores consideran que la instrucción switch no es recomendable si lo que se pretende es generar código limpio, es decir, código que se lea y entienda fácilmente. En este libro no encontrará una opinión con respecto a dicha polémica, por otro lado, se muestra la forma de realizar

el mismo objetivo de la instrucción `switch` a través de la instrucción `if-else`.

En la figura 3.10 se hizo uso de la instrucción `switch` para mostrar el nombre del mes con base en un valor entero recibido por parte del usuario, la figura 3.11 muestra un programa que realiza las mismas acciones utilizando la instrucción `if-else`.

InstruccionIfElseIf.java

```
1  import java.util.Scanner;
2  public class InstruccionIfElseIf {
3    public static void main(String[] args) {
4      Scanner entrada= new Scanner(System.
   in);
5
6      System.out.print(
                 "Ingrese un número de mes: ");
7      int mes = entrada.nextInt();
8
9      if( mes == 1 ) {
10       System.out.println("Enero");
11     }else if( mes == 2){
12       System.out.println("Febrero");
13     }else if( mes == 3){
14       System.out.println("Marzo");
15     }else if( mes == 4){
16       System.out.println("Abril");
17     }else if( mes == 5){
18       System.out.println("Mayo");
19     }else if( mes == 6){
20       System.out.println("Junio");
21     }else if( mes == 7){
22       System.out.println("Julio");
```

```
23      }else if( mes == 8){
24        System.out.println("Agosto");
25      }else if( mes == 9){
26        System.out.println("Septiembre");
27      }else if( mes == 10){
28        System.out.println("Octubre");
29      }else if( mes == 11){
30        System.out.println("Noviembre");
31      }else if( mes == 12){
32        System.out.println("Diciembre");
33      }else{
34        System.out.println("No válido");
35      } // Fin de la instrucción if-else
36    } // Fin del método main
37  } // Fin de la clase InstruccionIfElseIf
```

```
Ingrese un número de mes: 7
Julio
```

Figura 3.11 | Uso de la instrucción if-else en reemplazo de la instrucción switch

En el programa se observa que en la línea 9,

```
9   if( mes == 1 ) {
```

se hace uso de la instrucción if para comprobar si el valor de la variable mes es igual a 1; de ser así, se imprime el texto "Enero". En caso contrario, es decir, la parte `else` que, se observa en la línea 11,

```
11   }else if( mes == 2){
```

en lugar de poner la apertura de llaves, {, se usa una nueva instrucción if para comprobar si el valor de la variable mes es igual a 2. Al final de la línea, ahora sí, se coloca la apertura de llaves, {, para indicar el inicio del conjunto de sentencias que serán ejecutadas si es que se cumple la condición.

Desde la línea 13 hasta la línea 33, se definen condiciones haciendo uso de sentencias `if-else` para imprimir el nombre del mes que corresponda con el valor que posee la variable mes. Sin embargo, en la línea 33,

```
33  }else{
```

ya no se agrega una nueva comprobación, simplemente, se coloca la apertura de llaves, `{`; con esto, se están indicando las sentencias que se realizarán en caso de que ninguna de las comprobaciones anteriores haya sido verdadera. Éste vendría a ser, por así decirlo, como nuestro "case default" de la instrucción `switch`.

2.8Sentencia de repetición while

Se ha tenido la oportunidad de conocer y utilizar las estructuras de selección con las que se determinan conjuntos de sentencias, que deben o no ser ejecutadas. Ahora, se revisarán las **estructuras de repetición**. Éstas permiten que un conjunto de sentencias se ejecute de forma repetitiva. Estas estructuras son muy útiles, ya que permiten reducir el número de líneas del código. La primera estructura de repetición a mostrar es la instrucción `while`. Su sintaxis es muy parecida a la de la instrucción `if`. Se tiene la palabra reservada `while`, luego se establece una condición dentro de los paréntesis de la instrucción y las apertura y cierre de llaves, `{ }`, posteriormente

se define el cuerpo de la instrucción `while`, es decir, donde irán las instrucciones que se ejecutarán de forma repetitiva. La figura 3.12 muestra la sintaxis de la instrucción `while`.

```
while ( CONDICIÓN ) {
   SENTENCIAS A REALIZAR
}
```

Figura 3.12 | Sintaxis de la sentencia **`if`**

La instrucción `while` recibe una condición, y mientras esta condición responda con el valor de `true`, las sentencias dentro del cuerpo del `while` se ejecutan de forma repetitiva, es decir, cada vez que son ejecutadas las sentencias encerradas entre las llaves, `{ }`, se vuelve a preguntar si la condición aún devuelve `true`, de ser así, se ejecutan nuevamente las instrucciones, una vez que finalizan, se vuelve a preguntar si la condición sigue siendo verdadera. Este proceso se repite hasta que la condición devuelva el valor de `false`. Para lograr esto, se incluyen instrucciones dentro del cuerpo de `while`, que permitan modificar la operación de la condición. Véase el ejemplo de la figura 3.13.

ImprimiendoTextos.java

```
1  public class ImprimiendoTextos {
2    public static void main(String[] args) {
3      System.out.println(“Texto 1”);
4      System.out.println(“Texto 2”);
5      System.out.println(“Texto 3”);
6      System.out.println(“Texto 4”);
7      System.out.println(“Texto 5”);
8      System.out.println(“Texto 6”);
9      System.out.println(“Texto 7”);
10     System.out.println(“Texto 8”);
11     System.out.println(“Texto 9”);
```

```
12      System.out.println("Texto 10");
13    } // Fin del método main
14  } // Fin de la clase ImprimiendoTextos
```

```
Texto 1
Texto 2
Texto 3
Texto 4
Texto 5
Texto 6
Texto 7
Texto 8
Texto 9
Texto 10
```

Figura 3.13 | Imprimiendo textos de forma secuencial.

En el programa de la figura 3.13, se imprimen 10 textos. Cada texto es acompañado del número de texto que se imprime. Este programa no hace uso de la instrucción `while`. En este programa, únicamente se usa la *estructura secuencial*, es decir, las instrucciones se ejecutan de forma lineal, una después de otra. Véase cómo obtener el mismo resultado del programa de la figura 3.13 usando la instrucción `while`.

InstruccionWhile.java

```
1  public class InstruccionWhile {
2    public static void main(String[] args) {
3      int numero = 1;
4      while( numero <= 10 ){
5        System.out.println("Texto " + numero);
6        numero = numero + 1;
7      }
8    } // Fin del método main
9  } // Fin de la clase InstruccionesWhile
```

```
Texto 1
Texto 2
Texto 3
Texto 4
Texto 5
Texto 6
Texto 7
Texto 8
Texto 9
Texto 10
```

Figura 3.13A | Imprimiendo textos con la instrucción **while**

Dedique unos minutos para analizar el código de la figura 3.13.

La línea 3 del programa define una variable de nombre `numero` que ha sido inicializada con el valor de 1, esta variable es utilizada para controlar el número de repeticiones en la instrucción `while`. La condición dentro de la instrucción `while` se evalúa, y mientras ésta devuelva `true` como respuesta, las instrucciones dentro del cuerpo de `while` son ejecutadas. Una vez que se termine dicha ejecución, se verifica nuevamente el valor que devuelve la condición. Mientras la condición siga devolviendo `true` como respuesta, las sentencias dentro del cuerpo del `while` se ejecutan continuamente.

Entonces, la primera vez que se evalúa la condición del `while` en la línea 4, el valor que la variable `numero` tiene asignado es `1`, por lo que la condición devuelve `true`, es decir, se evalúa `1 <= 10`. Puesto que la respuesta es `true`, se ejecutan las instrucciones que contiene la instrucción `while`. En la línea 5, se imprime el texto `Texto 1`, como en ese momento, el valor de la variable `numero` es `1`. En la línea 6, se asigna a la variable `número` la suma del valor actual de la variable `número` más `1`, es decir, se asigna el resultado de

1 + 1. En vista de que se llega al cierre del cuerpo de la instrucción while, se vuelve a evaluar la condición de numero <= 10, la cual es true como resultado, puesto que en ese momento, numero posee el valor de 2, por lo que al evaluar 2 <= 10, la respuesta es true. En la segunda ejecución de las instrucciones dentro de while, se imprime el texto Texto 2, y en la línea 6, se asigna a la variable numero la suma del valor actual de la variable número más 1, es decir, se asigna el resultado de 2 + 1. Nuevamente, se llega al final del cuerpo de la instrucción while, por lo que, se evalúa nuevamente la condición del while, en ese momento, la variable numero posee el valor de 3, por lo que la condición sigue siendo verdadera. Todo el proceso se repite hasta que en cierto momento a la variable numero le es asignado el valor de 11, y deja de ser verdadera la condición del while, por lo que las instrucciones dentro del cuerpo de ésta ya no son ejecutadas.

Como se observa, la instrucción while es una poderosa estructura disponible para ejecutar instrucciones repetitivas, sin embargo, se deben realizar las modificaciones necesarias dentro del cuerpo de la instrucción while, para que en algún momento la condición que se haya establecido deje de devolver true como respuesta, de lo contrario, las repeticiones nunca terminarían, es decir, se estaría entrando en un ciclo infinito.

3.9Sentencia de repetición for

Ya se tuvo acercamiento al uso de las estructuras de repetición. La estructura while es la primera instrucción que hemos utilizado. Preste a tención a las partes resaltadas en el fragmento del código de la figura 3.14. Note usted que

hemos resaltado el fragmento en el que se define la variable `numero`, la condición establecida en la instrucción `while` y la asignación realizada a la variable `numero` dentro del cuerpo del `while`.

```
int numero = 1;
while( numero <= 10 ){
    System.out.println("Texto " + numero);
    numero = numero + 1;
}
```

Figura 3.14 | Ejemplo de una instrucción **while**

Las tres partes resaltadas en la figura 3.14, son las que permiten controlar el número de veces que se ejecutan las instrucciones dentro del cuerpo del `while`. Esas partes son comunes para controlar el número de repeticiones en la instrucción `while`. Primero, se necesita una **variable para controlar** el número de ciclos de la estructura de repetición; segundo, se requiere establecer una **condición** que incluya la variable, y tercero, se debe **modificar la variable** definida inicialmente para que en algún momento la condición llegue a ser falsa. En pocas palabras, para una tener una estructura de repetición con un número establecido de ciclos, se necesita de una **variable de control**, una **condición** y **modificar** la variable de control.

```
for ( VARIABLE DE CONTROL ; CONDICIÓN ; MODIFICADOR ){
   SENTENCIAS A REALIZAR
}
```

Figura 3.15 | Sintaxis de la sentencia **`for`**

Las tres partes identificadas, que nos permiten controlar las repeticiones de una instrucción `while` están integradas

por defecto en la instrucción `for`. La figura 3.15 muestra la sintaxis de la estructura `for`.

InstruccionFor.java

```
1  public class InstruccionFor {
2    public static void main(String[] args) {
3      for(int numero=1; numero<=10;
                              numero=numero+1){
4        System.out.println("Texto " + numero);
5      }
6    } // Fin del método main
7  } // Fin de la clase InstruccionFor
```

```
Texto 1
Texto 2
Texto 3
Texto 4
Texto 5
Texto 6
Texto 7
Texto 8
Texto 9
Texto 10
```

Figura 3.16 | Imprimiendo textos con la instrucción **for**

El programa de la figura 3.16 muestra la forma en la que se da solución al mismo programa de la figura 3.13, pero con la instrucción `for`. Préstese atención al código dentro de los paréntesis de la instrucción `for`, ¿resultan familiares dichas instrucciones? Si se observa detenidamente, dentro de los paréntesis, la instrucción `for` cuenta con 3 partes separadas por puntos y comas, `;`, la primer parte está destinada para definir la **variable de control**, la segunda parte está destinada para establecer la **condición** de la estructura, por último, la tercera parte está destinada para establecer la sentencia que **modificará** la

variable de control. Nótese cómo al ejecutar el programa de la figura 3.16, se obtiene el mismo resultado de la figura 3.13.

Entonces, ¿es conveniente utilizar la instrucción **`while`** o utilizar la instrucción **`for`**? Por el momento, considerando que apenas se están conociendo los fundamentos de la programación, se sugiere utilizar la instrucción que domine mejor. Con la experiencia, se descubrirán los beneficios de utilizar una u otra instrucción con base en el problema a resolver.

3.10Sentencia de repetición do-while

La siguiente estructura de repetición a revisar, es la instrucción `do-while`, esta instrucción, al igual que las instrucciones `while` y `for`, permite establecer sentencias que son ejecutadas de forma repetitiva, pero la diferencia es que esta instrucción garantiza que, por lo menos, las sentencias dentro de la estructura son ejecutadas una vez.

Obsérvese el código y la salida del programa de la figura 3.17.

InstruccionDoWhile.java

```
1  import java.util.Scanner;
2  public class InstruccionDoWhile {
3    public static void main(String[] args) {
4      Scanner entrada= new Scanner(System.
   in);
5      int respuesta = 0;
6      do{
7        System.out.println("1) Opción A");
8        System.out.println("2) Opción B");
9        System.out.println("3) Salir");
```

```
10        System.out.print(
    "Elija una opción: ");
11        respuesta = entrada.nextInt();
12
13        System.out.println(
    "Elegiste la opción " + respuesta);
14        System.out.println();
15      }while( respuesta != 3 );
16      System.out.println("Adiós!");
17    } // Fin del método main
18  } // Fin de la clase InstruccionDoWhile
```

```
1) Opción A
2) Opción B
3) Salir
Elija una opción: 1
Elegiste la opción 1
1) Opción A
2) Opción B
3) Salir
Elija una opción: 2
Elegiste la opción 2
1) Opción A
2) Opción B
3) Salir
Elija una opción: 3
Elegiste la opción 3
Adiós!
```

Figura 3.17 | Programa que hace uso de la instrucción **do-while**

El programa hace uso de la instrucción `do-while`, la cual inicia con la palabra reservada **`do`**, seguida de la apertura de llave, **`{`**, esto indica el inicio del cuerpo de la instrucción, y

todas las sentencias dentro de las llaves, **{ }**, son ejecutadas de forma inmediata la primera vez. El cierre de llave, **}**, indica el final del cuerpo del **do**, y se debe agregar inmediatamente la palabra reservada `while`, con la condición a evaluar, finalizando con punto y coma, **;**, de lo contrario, se obtiene un error al intentar compilar el código.

Las sentencias de la línea 7 a la línea 14 corresponden con las acciones a realizar, por lo menos, una vez. De la línea 7 a la 10 se imprime en pantalla un menú de opciones. En la línea 11, se lee el valor numérico ingresado por el usuario. La línea 13 imprime el texto que indica el número de opción que el usuario introduce. La línea 14 imprime un salto de línea.

La línea 15 indica el final del bloque de sentencias de **do**, y se define la condición a evaluar dentro del **while** que determina si las sentencias dentro del bloque **do** deben o no ser ejecutadas nuevamente. En este programa, las sentencias del bloque **do** son ejecutadas nuevamente, mientras el valor que tiene la variable `respuesta` no sea igual a `3`. En la ejecución mostrada en la figura 3.17, la primera vez se ingresa el valor de 1, por lo que la variable `respuesta` toma el valor de **1**. Posteriormente se imprime nuevamente el menú y se introduce el valor de **2**, por lo que en esa ocasión la variable `respuesta` toma el valor de `2`. La condición, para terminar con las repeticiones, sigue devolviendo `true`, por lo que se imprime nuevamente el menú, pero el valor ingresado en esta ocasión es **3**. Así, la condición deja de ser verdadera, es decir, `respuesta != 3` devuelve `false`, por lo que no se ejecutan más repeticiones. Con eso termina la instrucción do-`while`. Por último, la línea 16 imprime en pantalla el texto `Adiós!`.

3.11 Instrucciones break y continue

Existen dos instrucciones disponibles dentro de las estructuras de repetición: **`break`** y **`continue`**. La instrucción **`break`** permite terminar una estructura de repetición por completo, independientemente de que sea una instrucción `while`, `for` o `do-while`. La instrucción **`continue`** permite finalizar una repetición, es decir, sólo va a interrumpe la ejecución de las sentencias de la repetición en la que se encuentra en ese momento, pero la estructura continuará con el resto de las repeticiones.

Analicemos cómo funcionan las instrucciones **`break`** y **`continue`**. Primeramente, se revisa el modo en que trabaja la instrucción **`break`**. La figura 3.18 muestra un programa que usa la instrucción **`break`**.

```
1   public class InstruccionBreak {
2     public static void main(String[] args) {
3       for(int numero=1; numero<=10;
                                numero=numero+1){
4         if( numero == 7 ){
5           break;
6         }
7         System.out.print( numero + " " );
8       }
9     } // Fin del método main
10  } // Fin de la clase InstruccionBreak
```

```
 1 2 3 4 5 6
```

Figura 3.18 | Uso de la instrucción **break** dentro una estructura de repetición

El programa de la figura 3.18, en la línea 3, usa la instrucción `for`, para controlar una estructura de repetición en

la que se define una variable de nombre `numero`, que es inicializada con el valor de `1`. Esta variable se usa dentro de la **condición `for`,** para controlar el ciclo de repetición. En la instrucción `for`, se establece que con cada repetición, el valor de la variable `numero` es incrementado en uno. En general, el programa pretende imprimir los valores del 1 al 10. Para lograrlo, en la línea 7, se imprime el valor que posee la variable `numero` y se agrega un espacio en blanco.

Al observar el resultado de ejecutar el programa, se aprecia que únicamente se imprimen los valores del 1 al 6. Esto sucede porque con cada repetición, en la línea 4 se pregunta si el valor de la variable `numero` es igual a siete, `numero == 7`. Durante las primeras 6 ejecuciones, la condición no es verdadera, sin embargo, en la repetición en el que la variable `numero` tiene asignado el valor de `7`, la condición sí se cumple, por lo que se ejecuta la instrucción **`break`**. La instrucción `break`, al ser utilizada dentro de una estructura de repetición, como en este caso, termina por completo con las repeticiones, por lo que la instrucción `for` es finalizada, teniendo como resultado, que únicamente fueran impresos los valores del 1 al 6.

Ahora, ¿cómo trabaja la instrucción **`continue`**? La figura 3.19 muestra un programa similar al anterior, el de la figura 3.18, con la única diferencia de que en lugar de hacer uso de la instrucción `break`, se usa la instrucción `continue`. Obsérvese el resultado de ejecutar este programa.

InstruccionContinue.java

```
1  public class InstruccionContinue {
2    public static void main(String[] args) {
3      for(int numero=1; numero<=10;
   numero=numero+1){
4        if( numero == 7 ){
5          continue;
```

```
6         }
7         System.out.print( numero + " " );
8       }
9     } // Fin del método main
10
} // Fin de la clase InstruccionContinue
```

```
 1 2 3 4 5 6 8 9 10
```

Figura 3.19 | Uso de la instrucción **continue** dentro una estructura de repetición

Préstese atención al resultado de ejecutar el programa de la figura 3.19, parece que los valores del 1 al 10 se imprimen en pantalla, sin embargo, el valor de **7** no está presente. Anteriormente se comentó que, cuando la instrucción **continue** es ejecutada, hace que se dé por terminada la repetición en la que se encuentra, pero el resto de las repeticiones seguirán siendo ejecutadas. Es por eso que, cuando la variable `numero` posee el valor de `7`, la repetición se da por terminada, por lo que no se logra imprimir el valor en pantalla, sin embargo, el resto de los valores sí son impresos en pantalla.

3.12Conclusión

En el capítulo anterior se hizo uso de la estructura secuencial, la más básica de todas. En este capítulo se conocieron las estructuras de selección y de repetición. Todas las estructuras aprendidas hasta el momento, la secuencial, la de selección y la de repetición, corresponden a las **estructuras de control**, parte importante de la mayoría de los lenguajes de programación. Así también, se usaron los operadores relacionales, de igualdad y los operadores lógicos. Todos estos operadores son parte importante de las estructuras de selección y repetición, pues nos resultan útiles para la creación de condiciones a asignar en dichas estructuras.

4.**Arreglos**

4.1Introducción

Generalmente, cuando se crean programas, se declaran e inicializan variables para realizar operaciones. Éstas son muy útiles para guardar de manera temporal, valores a utilizar durante la ejecución de los programas. En este capítulo se estudian los arreglos, mismos que brindan la posibilidad de tener un contenedor de valores u objetos de cierto tipo. Éstos son útiles cuando existe la necesidad de almacenar un conjunto de valores de un mismo tipo, tal vez un conjunto de calificaciones, un conjunto de nombres de personas, un conjunto de temperaturas; los ejemplos son muy variados. Con seguridad, en algún momento, el uso de los arreglos facilitará las operaciones de los programas que se realicen.

4.2Trabajando con los arreglos

Para trabajar con los arreglos, se requiere entender cómo están estructurados. La figura 4.1 muestra una representación de un arreglo de 10 elementos.

Figura 4.1 | Arreglo de 10 elementos

En la figura 4.1, se muestra la representación de un arreglo de 10 elementos. El arreglo está compuesto por objetos de tipo `String`, es decir, textos. Cuando se desea recuperar alguno de los valores de un arreglo, se debe indicar el índice del elemento que se desea obtener. El índice del primer elemento es el índice 0, siendo un arreglo de siete elementos, el último elemento se encuentra en el índice 6.

Con excepción de los **datos primitivos**, en Java, todos los tipos de datos brindan objetos. Los **arreglos** también son objetos. Una de las formas de definir un arreglo es la siguiente:

```
String[] arregloTextos = new String[7];
```

Para definir un arreglo, se indica el tipo de dato que contendrá el arreglo. Posteriormente, se agregan corchetes de apertura y cierre, `[ ]`. En este ejemplo, se utiliza `String[ ]`, como tipo de arreglo. Posteriormente, se escribe el nombre con el que se identifica al arreglo. Se agrega el operador de asignación, =, la palabra reservada `new`, y nuevamente el tipo de dato que contendrá el arreglo con los corchetes, pero esta vez, indicando dentro de los corchetes la longitud del arreglo. Para este ejemplo, `new String[7]`.

Un punto importante a tomar en cuenta, es que los arreglos pueden ser de cualquier tipo, ya sean de **datos primitivos** u objetos. Podríamos tener un arreglo de tipo `int`, un arreglo de tipo `boolean` o de cualquier otro tipo de dato, siempre y cuando, sea una clase válida. Aunque los arreglos pueden ser de cualquier tipo, no es posible mezclarlos, es decir, por poner un ejemplo, no se puede tener un arreglo que contenga

valores de tipo `String` y al mismo tiempo contenga datos de tipo `double`.

Ya se mostró cómo crear un arreglo, y entonces, ¿cómo se asigna un valor? Para ello, se escribe el nombre de nuestro arreglo, y entre los corchetes de apertura y cierre, se indica el índice en el que se desea asignar un valor. Por ejemplo, la sentencia

```
arregloTextos[0] = "PRIMERO";
```

realiza la asignación del texto "`PRIMERO`" en el índice `0` del arreglo `arregloTextos`. Si posteriormente, dentro del programa, se desea recuperar el valor almacenado en el índice 0 del arreglo, se realiza del siguiente modo:

```
System.out.println( arregloTextos[0] );
```

La sentencia solicita el valor que posee el arreglo `arregloTextos` en el índice 0, que posteriormente se manda a imprimir en pantalla.

Veamos cómo usar los arreglos en un programa.

ArreglosEnteros.java

```
1  public class ArregloEnteros {
2    public static void main(String[] args) {
3      int[] numeros = new int[5];
4
5      for(int indice=0; indice<5;
                          indice=indice+1){
6        System.out.println("Índice: "+ indice
             + ", Valor: " + numeros[indice]);
7      }
8      System.out.println();
9
```

```
10      numeros[0] = 10;
11      numeros[1] = 30;
12      numeros[2] = 50;
13      numeros[3] = 70;
14      numeros[4] = 90;
15
16      for(int indice=0; indice<5; indice++ ){
17        System.out.println("Índice: "+ indice
              + ", Valor: " + numeros[indice]);
18      }
19    } // Fin del método main
20  } // Fin de la clase ArregloEnteros
```

```
Índice: 0, Valor: 0
Índice: 1, Valor: 0
Índice: 2, Valor: 0
Índice: 3, Valor: 0
Índice: 4, Valor: 0
Índice: 0, Valor: 10
Índice: 1, Valor: 30
Índice: 2, Valor: 50
Índice: 3, Valor: 70
Índice: 4, Valor: 90
```

Figura 4.2 | Programa de Arreglo de enteros

El programa de la figura 4.2 define un arreglo de enteros de 5 elementos. Se utiliza la instrucción `for` para leer e imprimir en pantalla cada uno de los valores que posee el arreglo. Posteriormente, se asignan valores en cada uno de los índices del arreglo, y se vuelven a imprimir los valores que posee el arreglo con apoyo de la instrucción `for`. Se muestra el análisis del proceso de ejecución.

En la línea 3, se define un arreglo de tipo `int`, de nombre `numeros`, y se establece que la longitud del arreglo es de

5 elementos, es decir, se pueden guardar valores **desde** el índice **0** **hasta** el índice **4**.

En la línea 5, se implementa una instrucción `for` con una variable de control de nombre `indice` que está inicializada con el valor de `0`. Se establece que la condición es que, se realizarán repeticiones mientras la variable `indice` sea menor al valor de `5`, y al final, se define que con cada repetición a la variable `índice`, le será asignado el valor que posee, más 1.

La línea 6 recupera el valor de la variable `indice` y el valor del arreglo `numeros` en el índice que tenga la variable `indice` para mostrarlos en pantalla al concatenarlos con los textos `"Índice: "` y `", Valor: "`. Para la primera ejecución de la instrucción `for`, `indice` posee el valor de **0**, por lo que `numeros[indice]` devuelve el valor que posee el arreglo `numeros` en el índice 0. En la siguiente repetición, se muestra el valor del arreglo `numeros` en el índice **1**, y así progresivamente hasta el índice **4**. Cuando la variable `indice` posee el valor de **5**, se deja de cumplir la condición del `for` y se da por terminada la instrucción.

El resultado de la instrucción `for`, anteriormente descrita, muestra como salida:

```
Índice: 0, Valor: 0
Índice: 1, Valor: 0
Índice: 2, Valor: 0
Índice: 3, Valor: 0
Índice: 4, Valor: 0
```

Para cada una de las repeticiones el valor mostrado es **0**, esto sucede porque aún no se ha asignado un valor a cada uno de los índices, y **todos los arreglos son inicializados por defecto**. En este caso, por tratarse de valores numéricos, el valor que se asigna por defecto es **0**. Más adelante, en este

capítulo, se abordará el tema de la inicialización de valores por defecto en los arreglos.

En la línea 10, se realiza la asignación del valor **10** al arreglo `numeros` en el índice `0`; en la línea 11 el valor de **30** en el índice `1`; en la línea 12 el valor de **50** en el índice `2`; en la línea 13 el valor de **70** en el índice `3`, y por último, en la línea 14 se asigna el valor de **70** en el índice `4` del arreglo `numeros`.

La línea 16 implementa una instrucción `for`, similar a la que fue definida en la línea 5, sin embargo, se observa que en la última parte de la instrucción, se visualiza `indice++`, en lugar de la operación `indice=indice+1`. Esto ha sido realizado de manera intencional para que se conozca el **operador ++**. Para este ejemplo, el **operador ++** realiza exactamente lo mismo que la operación `indice=indice+1`, es decir, incrementa en 1 el valor que posee la variable y lo asigna a ésta. Se puede utilizar cualquiera de las dos formas en los programas.

Las impresiones de pantalla realizadas por la sentencia de la línea 17 muestran los valores de cada uno de los índices del arreglo `numeros`.

```
Índice: 0, Valor: 10
Índice: 1, Valor: 30
Índice: 2, Valor: 50
Índice: 3, Valor: 70
Índice: 4, Valor: 90
```

4.3Inicialización por defecto de los arreglos

En estos momentos, ya se tiene una idea más clara de cómo hacer uso de los arreglos. Ahora se visualiza la inicialización de los arreglos. En el tema anterior, se mostró que los valores de los arreglos son inicializados por defecto. Estos

valores dependen del tipo de dato que almacena el arreglo. En la figura 4.3, se muestran los valores por defecto que son asignados a los arreglos, según el tipo de dato.

Tipo de dato	Ejemplo de tipo	Valor asignado
Tipos numéricos	`int[]`	`0`
	`double[]`	`0.0`
Tipo char	`char[]`	`'\u0000'`
Tipo boolean	`boolean[]`	`false`
Objetos	`String[]`	`null`

Figura 4.3 | Valores por defecto asignados a los arreglos.

En los arreglos, todos los datos primitivos de tipo numéricos son inicializados con el valor de 0, por supuesto, el valor es acorde al tipo. Por ejemplo, `int` es inicializado con `0`, `float` con `0.0f`, y `double` con `0.0`. Los arreglos de tipo `char` son inicializados con el valor `0` de la tabla de símbolo Unicode. En Java esta asignación se realiza: `'\u0000'`. Los arreglos de tipo `boolean` son inicializados con el valor de `false`. Por último, cualquier otro tipo de arreglo es inicializado como `null`. Éste podría ser de una clase de Java o cualquier otra clase creada por el programador. `null` es la forma que se utiliza en Java para indicar la ausencia de la referencia hacia un objeto. Ese tema se discutirá en el siguiente capítulo. Estas inicializaciones son comprobadas en un programa.

InicializacionArreglos.java

```
1  public class InicializacionArreglos {
2    public static void main(String[] args) {
3      int[] intArreglo = new int[4];
4      double[] doubleArreglo = new double[4];
5      char[] charArreglo = new char[4];
6      boolean[] booleanArreglo=new boolean[4];
7      String[] stringArreglo = new String[4];
```

```
8
9       for(int indice=0; indice<4; indice++){
10                                  System.out.print(
    intArreglo[indice] + "-" );
11      }
12      System.out.println();
13
14      for(int indice=0; indice<4; indice++){
15        System.out.print(
                  doubleArreglo[indice] + "-" );
16      }
17      System.out.println();
18
19      for(int indice=0; indice<4; indice++){
20        System.out.print(
                    charArreglo[indice] + "-" );
21      }
22      System.out.println();
23
24      for(int indice=0; indice<4; indice++){
25        System.out.print(
                 booleanArreglo[indice] + "-" );
26      }
27      System.out.println();
28
29      for(int indice=0; indice<4; indice++){
30        System.out.print(
                  stringArreglo[indice] + "-" );
31      }
32      System.out.println();
33    } // Fin del método main
34  } // Fin de la clase InicializacionArreglos
```

```
0-0-0-0-
0.0-0.0-0.0-0.0-
 - - - -
false-false-false-false-
null-null-null-null-
```

Figura 4.4 | Programa de arreglos de diferentes tipos.

El programa de la figura 4.4 presenta los valores que son inicializados, por defecto, en 5 arreglos de diferentes tipos. Estos son de tipo `int[]`, `double[]`, `char[]`, `boolean[]` y `String[]`.

La línea 9, 10 y 11,

```
9   for(int indice=0; indice<4; indice++){
10    System.out.print(intArreglo[indice]+"-");
11  }
```

corresponden con la definición de la instrucción `for` para mostrar los valores del arreglo `intArreglo`. Con cada repetición se concatena el valor de cada índice del arreglo con el texto "-". Una vez que se imprimen todos los valores del arreglo `intArreglo`, el resultado en pantalla es "0-0-0-0-".

La línea 14, 15 y 16,

```
14  for(int indice=0; indice<4; indice++){
15    System.out.print(
                doubleArreglo[indice] + "-");
16  }
```

hacen uso de la instrucción `for` para imprimir los valores del arreglo `doubleArreglo`, agregando al final de cada impresión el texto "-". El resultado en pantalla es "0.0-0.0-0.0-0.0-".

La línea 19, 20 y 21,

```
19  for(int indice=0; indice<4; indice++){
20    System.out.print(
                  charArreglo[indice] + "-");
21  }
```

utilizan la instrucción `for` de forma similar a las instrucciones `for` anteriores para imprimir los valores del arreglo `charArreglo`. Tomando en cuenta que el valor asignado por defecto a un arreglo de tipo `char[]` es `'\u0000'`, el resultado en pantalla es `" - - - -"`.

La línea 24, 25 y 26,

```
24  for(int indice=0; indice<4; indice++){
25     System.out.print(
             booleanArreglo[indice] + "-" );
26  }
```

recorren los valores del arreglo `booleanArreglo` para imprimir los valores de forma similar a los otros arreglos. El resultado en pantalla es `"false-false-false-false-"`.

La línea 29,30 y 31,

```
29  for(int indice=0; indice<4; indice++){
30      System.out.print(
              stringArreglo[indice] + "-" );
31  }
```

recorren los valores del arreglo `stringArreglo` e imprimen sus valores de forma similar al resto de los arreglos. El resultado en pantalla es `"null-null-null-null-"`. Es necesario recordar que los textos en Java son representados por la clase `String`, la cual no es un tipo de dato primitivo.

Todo arreglo que no sea de algún tipo de dato primitivo, sus valores son inicializados como `null`.

4.4Inicialización de los arreglos

En el tema anterior, se estudió la inicialización por defecto de los arreglos. En este tema, se aborda la creación de arreglos con valores iniciales que se asignan desde su desarrollo.

La figura 4.5 muestra un ejemplo en el que se define un arreglo de tipo `int[]` con valores iniciales.

```
Tipo de arreglo  Identificador  Valores
     int[]         numeros =   {10, 30, 50, 70,90};
```

Figura 4.5 | Inicialización de un arreglo de tipo int[]

Se aprecia que, para crear un arreglo con los valores asignados desde un inicio, se utilizan las llaves de apertura y cierre, `{}`. Dentro de las llaves, se indican los valores separados por comas. Los valores asignados dentro de las llaves deben ser del mismo tipo del arreglo, es decir, si se desea inicializar un arreglo de tipo `int[]`, dentro de las llaves se establecen los valores de tipo `int`. No es necesario indicar la palabra reservada `new`, tampoco es necesario indicar la longitud del arreglo, ya que ésta será calculada por Java al momento de compilar el programa.

DiasSemana.java

```
1  class DiasSemana {
2    public static void main(String[] args) {
3      String[] dias = {"LUN", "MAR", "MIE",
       "JUE","VIE","SAB","DOM"};
4
5      for(int indice=0; indice<dias.length;
       indice++){
```

```
6           System.out.println( dias[indice] );
7         }
8       } // Fin del método main
9     } // Fin de la clase DiasSemana
```

```
LUN
MAR
MIE
JUE
VIE
SAB
DOM
```

Figura 4.6 | Programa de días de la semana

El programa de la figura 4.6, en la línea 3, crea un arreglo de tipo `String[]`. Le asigna como identificador el nombre de `dias`, y lo inicializa con los textos de los días de la semana.

En la línea 5, se implementa la instrucción `for` para recorrer los valores del arreglo. Se estable como variable de control a `indice`, sin embargo, nótese que la condición es `indice<dias.length`. Siempre que se desee conocer la longitud de un arreglo, se agrega `.length` al final del arreglo. Por el momento, no es necesario entrar en detalles sobre el porqué, eso se aborda en el siguiente capítulo. La instrucción `for` hace uso del operador **++** para indicar que la variable `indice` se incremente en `1` con cada ejecución. Al finalizar la ejecución del programa, el resultado en pantalla son los valores con los que fue inicializado el arreglo `dias`.

```
LUN
MAR
MIE
JUE
VIE
SAB
DOM
```

4.5Sentencia for mejorado

En el capítulo anterior, se trabajó con las estructuras de repetición. Entre ellas, la instrucción `for`. En los últimos temas, se ha utilizado la instrucción `for` para recuperar los valores de arreglos. La instrucción `for` tiene un uso especial para cuando se quiere recorrer los elementos de un arreglo. Simplificando esta tarea, esta forma de trabajar con la instrucción `for` se conoce como **for mejorado**.

```
for ( VARIABLE : ARREGLO ) {
    SENTENCIAS A REALIZAR
}
```

Figura 4.7 | Sintaxis de la instrucción **for mejorado**

La figura 4.7 muestra la sintaxis de la instrucción **`for` mejorado**. Para comprender de forma clara el funcionamiento de **`for` mejorado**, se analizará el programa de la figura 4.8.

ForMejorado.java

```
1  class ForMejorado {
2    public static void main(String[] args) {
3      String[] dias = {“LUN”, “MAR”, “MIE”,
   “JUE”,”VIE”,”SAB”,”DOM”};
4
5      for( String dia : dias ){
```

```
6           System.out.println( dia );
7         }
8       } // Fin del método main
9   } // Fin de la clase ForMejorado
```

```
LUN
MAR
MIE
JUE
VIE
SAB
DOM
```

Figura 4.8 | Programa que hace uso del **for mejorado**.

En la línea 3 del programa, se define un arreglo de tipo `String[]` con el nombre de `días`. Éste es inicializado con las iniciales de los días de la semana.

La línea 5 es donde se hace uso de la instrucción **`for` mejorado** para obtener el valor de cada elemento del arreglo `dias`. La forma en la que se implementa **`for` mejorado** es estableciendo dentro de los paréntesis el **arreglo que se desea recorrer** y la **variable a la que le asignará cada valor** del arreglo. Se aprecia que primero se define la variable y después se menciona el arreglo que será recorrido. Ambos están separados por dos puntos, **`:`**.

La **primera vez** que se ejecuta la línea 6, a la variable `dia` le es asignado el valor del **primer elemento** del arreglo `dias`, por lo que el valor en ese momento es `"LUN"`, así que se imprime en pantalla ese valor y un salto de línea.

La **segunda vez** que se ejecuta la línea 6, a la variable `dia` le es asignado el valor del siguiente elemento del arreglo `dias`, es decir, `"MAR"`.

La **tercera vez** que se ejecuta la línea 6, a la variable `dia` le es asignado el siguiente valor del arreglo `dias`, y se imprime en pantalla. Este proceso continúa hasta que todos los ele-

mentos del arreglo `dias` hayan sido leídos. Al finalizar la ejecución del programa, se visualiza en pantalla la impresión de todos los elementos del arreglo `dias`.

```
LUN
MAR
MIE
JUE
VIE
SAB
DOM
```

4.6Conclusión

En este capítulo, se mostraron los arreglos y cómo inicializarlos. Los valores que son inicializados por defecto, se enseñaron a recorrer los arreglos a través de la instrucción **`for`** y con **`for` mejorado**. Los arreglos permiten trabajar con un conjunto de valores del mismo tipo en un único objeto y brindan beneficios al trabajar en la construcción de los programas. Éstos ayudan a reducir código, mejorar la lectura de del código, y permiten trabajar de forma más natural. En el siguiente capítulo se tiene una introducción al paradigma de la Programación Orientada a Objetos (POO); se identificarán los fundamentos y la forma en que se utiliza en Java.

5.**Introducción de la POO**

5.1Introducción

La programación actual es un proceso que ha ido evolucionando. Existen diferentes estilos para programar. A esto se le conoce como **paradigma**. Un paradigma dice cuáles son las características para un determinado estilo de programación. Los lenguajes de programación se apegan a estas características, y por eso se dice que cierto lenguaje de programación *es de tal o cual paradigma*. Java fue creado con base en las características del paradigma de la **Programación Orientada a Objetos** (POO). Es común hacer referencia a este paradigma como **POO**, y a veces como OOP, por sus iniciales en inglés, Object-oriented programming. De aquí en adelante, en este libro, se hará referencia a este paradigma como POO.

En este capítulo, se mencionan características básicas de la POO, y se muestran principalmente los métodos. Se ilustra cómo definir los métodos en las clases, y se comprueba el funcionamiento de éstos.

5.2Paradigma de la POO

Hace algunos años, la programación se realizaba a través de sentencias. Los programas eran ejecutados de forma lineal, es decir, los programas contaban con una lista de instrucciones, pero no existían otras estructuras de control. Con el tiempo, se incorporaron las estructuras de control, y posteriormente se comenzaron a crear estilos de programación en los que se permitía la modularidad, es decir, dividir el código del programa en fragmentos separados.El objetivo, realmente, no es entrar en detalles históricos, por lo que estos serán tratados en términos generales.

Conforme los años pasaron, se crearon diferentes estilos de programación, es decir, diferentes paradigmas. La POO fue uno de ellos. A diferencia de los otros paradigmas, la POO incorporaba la clasificación y creación de entidades. Este paradigma comenzó a popularizarse y actualmente es el paradigma de mayor uso en la programación.

5.3Abstracción

La unidad básica de la POO es la clase. En Java, no es posible crear un programa sin antes tener una clase, pero realmente, ¿qué representan las clases?

Si se deja de pensar por un momento en la programación. Los seres humanos utilizan la clasificación para, prácticamente, todo. Se clasifican a las comunidades por Continentes, Países, Estados, Ciudades. Se clasifican a los animales, según su estructura; con base en su alimentación, según la forma en que se reproducen. Se clasificamos los colores en primarios y secundarios. La clasificación permite organizar y conocer las características de un grupo de entidades. Esta forma de pen-

samiento fue llevado a la programación, donde se pensó en entidades con sus atributos y comportamientos.

En este punto ya se tiene una idea de la forma en que se trabaja la POO. Como ya se mencionó, la clase es la unidad básica de trabajo en la POO. La clase es la representación de una entidad a trabajar en los programas. Está tendrá **características** y **comportamientos,** por ejemplo, una clase Automóvil podría tener las características de número de puertas, tipo de combustible, marca y modelo, y podría tener los comportamientos de acelerar, frenar, encender, apagar. A esta representación dentro de los programas, se le conoce como **abstracción**.

La **abstracción** es una representación más simple de algo complejo. Si se desea utilizar la representación de las personas, seguramente será creada una clase **Persona** con las características que se desean trabajar en los programas. En ella, se establece que la clase `Persona` tendrá un nombre, un apellido paterno, un apellido materno, una edad, un peso, etcétera. Al crear una abstracción de las personas, sólo son consideradas las características de las personas que interesan ser trabajadas en el programa. En la figura 5.1, se trabaja con la **abstracción** para crear una clase **Persona**, sin embargo, será abordado rápidamente el concepto de los **campos** antes de analizar el programa de la figura 5.1.

5.4Campos

Cuando se define una clase, se especifican las características con las que va a contar el futuro objeto. Las clases son definidas por **atributos** y **comportamientos**. Los **atributos** son las variables definidas en el cuerpo de la clase. Éstas son llamadas **campos**. También se conocen como **variables de instancia**, en este libro, se hace referencia a los atributos de la

clase como **campos**. Los **comportamientos** son las acciones que el objeto puede realizar. Éstos, son conocidos dentro de la clase como **métodos**, sin embargo, a los métodos, se abordarán en el siguiente tema. Por ahora el enfoque principal corresponde en entender los campos.

Véase el siguiente ejemplo que define una clase Persona.

Persona.java

```
1   package c05.p01;
2
3   class Persona {
4       String nombre;
5       String apellidoPaterno;
6       String apellidoMaterno;
7       int edad;
8       int peso;
9   } // Fin de la clase Persona
```

Figura 5.1 | Clase Persona

PruebaPersona.java

```
1  package c05.p01;
2
3  class PruebaPersona {
4    public static void main(String[] args) {
5      Persona persona1 = new Persona();
6
7      // Se imprimen los valores iniciales del objeto
   "persona1"
8      System.out.println( "Nombre: " +
                              persona1.nombre);
9      System.out.println("Ap. Paterno: " +
                     persona1.apellidoPaterno);
```

```
10      System.out.println("Ap. Materno: " +
                    persona1.apellidoMaterno);
11      System.out.println("Edad: " +
                                persona1.edad );
12      System.out.println("Peso: " +
                                persona1.peso );
13
14      // Se asignan valores a los campos del objeto "persona1"
15      persona1.nombre = "Alex";
16      persona1.apellidoPaterno = "Torres";
17      persona1.apellidoMaterno = "Flores";
18      persona1.edad = 20;
19      persona1.peso = 56;
20
21      // Se imprimen los valores del objeto "persona1"
22      System.out.println();
23      System.out.println("Nombre: " +
                              persona1.nombre);
24      System.out.println("Ap. Paterno: " +
                    persona1.apellidoPaterno);
25      System.out.println("Ap. Materno: " +
                    persona1.apellidoMaterno);
26      System.out.println("Edad: " +
                               persona1.edad);
27      System.out.println("Peso: " +
                                persona1.peso );
28
29    } // Fin del método main
30  } // Fin de la clase PruebaPersona
```

```
Nombre: null
Ap. Paterno: null
Ap. Materno: null
Edad: 0
Peso: 0
Nombre: Alex
Ap. Paterno: Torres
Ap. Materno: Flores
Edad: 20
Peso: 56
```

Figura 5.2 | Prueba de la clase **Persona**.

La figura 5.1 muestra un ejemplo de una clase Persona. Esta clase sólo es la representación que se utiliza dentro del programa para una persona. La estructura general de una clase, ya se ha mostrado anteriormente. En la línea 3,

```
3   class Persona {
```

se establece que se desea crear una clase al poner la palabra reservada **`class`** y el nombre de la clase será **`Persona`** seguido de la llave de apertura, **`{`**, con lo que define el inicio del cuerpo de la clase. Así también, se indica que la estructura de la clase Persona está formada por 5 atributos, los cuales se conocen como **campos** o **variables de instancia**. Éstos son un campo llamado `nombre` de tipo `String`, un campo llamado `apellidoPaterno` de tipo `String`; un campo llamado `apellidoMaterno` de tipo `String`; un campo llamado `edad` de tipo `int`; y un campo llamado `peso` de tipo `int`. La línea 9 indica el final del cuerpo de la clase a través de la llave de cierre, **`}`**. Pero, ¿qué sucede en la línea 1?

```
1   package c05.p01;
```

La palabra reservada `package`, se utiliza para agrupar un conjunto de clases, y que Java esté enterado de ello. Más adelante, en este capítulo, se retoma el tema de los paquetes con mayor detalle. `c05.p01` significa que la clase se encuentra dentro de una carpeta de nombre **p01**, y esta carpeta se encuentra dentro de una carpeta de nombre **c05**. Es importante respetar la información que se define, por lo que, es necesario guardar el archivo de la clase dentro de las carpetas mencionadas. Un ejemplo de una probable ruta al archivo sería **c:\ejercicios\c05\p01\Persona.java**, independientemente de dónde se encuentre ubicada la carpeta **c05**. Lo importante es que el archivo **Persona.java** esté dentro de una carpeta **p01**, y a su vez, esta carpeta esté dentro de una carpeta de nombre **c05**. Otro dato importante, que no debe de olvidar, es que el nombre del archivo debe ser exactamente igual al nombre de la clase. Esto incluye los caracteres en minúsculas y mayúsculas. Si se nombra a la clase como **Persona**, el archivo se debe guardar como **Persona.java**. Si se nombra al archivo como **persona.java**, la compilación fallará. Para compilar la clase `Persona`, se debe realizar de la siguiente manera:

```
javac -classpath c:\ejercicios; Persona.java
```

Los *paquetes* se irán revisando a lo largo del resto de los ejercicios del libro. Cuando una clase especifica que pertenece a un paquete, al compilar, se debe mencionar la ruta raíz para que Java encuentre las clases. En este caso, se tiene el archivo `Persona.java` en la ruta c:\ejercicios\c05\p01\Persona.java, por lo que el directorio raíz es `c:\ejercicios`. Si se organizan los archivos en un directorio diferente, se tiene que especificar dicho directorio al momento de compilar.

```
  javac  -classpath  DIRECTORIO  RAÍZ;
Persona.java
```

La clase `Persona` (figura 5.1) no cuenta con un método `main`, por lo que, ésta no puede ser ejecutada como un programa. Para probar el funcionamiento de la clase `Persona`, se ha creado una clase llamada `PruebaPersona`, la cual sí incluye un método `main` con las características necesarias para poder ser ejecutada la clase como un programa. La clase **`PruebaPersona`** se muestra en la figura 5.2.

La clase PruebaPersona también indica en la línea 1,

```
1   package c05.p01;
```

que pertenece al paquete `c05.p01`, por lo que el archivo `PruebaPersona.java` debe estar dentro de la misma ruta que la clase `Persona`. Entonces, al momento de compilar, se debe realizar del siguiente modo:

```
javac -classpath DIRECTORIO RAÍZ; Persona.java
```

En este caso, la ruta del archivo PruebaPersona es c:\ejercicios\c05\p01\Persona.java, por lo que el directorio raíz es `c:\ejercicios`. Así que la compilación se realiza del siguiente modo:

```
javac -classpath c:\ejercicios; PruebaPersona.
java
```

Para ejecutar la clase PruebaPersona, se indica el directorio raíz a través de -classpath y el nombre de la clase anteponiendo el paquete. Si el directorio raíz es `c:\ejercicios`; la ejecución de la clase PruebaPersona se realiza de la siguiente forma:

```
java -classpath c:\ejercicios; c06.p01.
PruebaPersona
```

Por el momento, no es necesario la comprensión de cómo funcionan los *paquetes*. En otros temas, más adelante, se conocerá con mayor detalle el funcionamiento de los *paquetes*. Sin embargo, es importante conocer la forma de ejecutar los programas que se realicen de aquí en adelante.

Al analizar el funcionamiento de la clase `PruebaPersona`. En la línea 5,

```
5   Persona persona1 = new Persona();
```

se crea un objeto de tipo `Persona`. Ya antes se mencionaron los objetos, sin embargo, no se había entrado en detalle sobre lo que éstos representan. Todas las clases, ya sean las que proporciona Java de forma predeterminada en su amplia biblioteca de clases o que se creen, pueden ser usadas para crear objetos. Las clases definen cómo estarán compuestos los objetos, y los objetos son creados durante la ejecución del programa. Por poner una analogía, el plano de un edificio es a la construcción terminada, lo que la clase es al objeto creado. La clase define las características que tendrá el objeto, pero éste no existe todavía. El objeto será creado posteriormente con base en la clase. Esto es lo que ocurre en la línea 5. Se define que tendremos un objeto que será identificado con el nombre de **`persona1`**, y éste será del tipo de clase **`Persona`**. La palabra reservada **`new`** es la encargada de preparar el espacio en memoria para el objeto que será creado. **`Persona()`** es lo que se conoce como **constructor**. Más adelante se entrará en detalle con respecto a los constructores. Por el momento, se identifican las partes que generalmente se utilizan para crear un objeto.

La línea 7,

```
7       // Se imprimen los valores iniciales del
                                objeto "persona1"
```

es un comentario de línea que ofrece información sobre las acciones que serán realizadas.

En la línea 8,

```
8   System.out.println("Nombre:
                          "+persona1.nombre);
```

`persona1.nombre` recupera el valor que tiene asignado el objeto en su campo nombre y lo concatena con el texto "Nombre: ". Siempre que se requiera acceder al campo de un objeto; se menciona el nombre del objeto; agregamos un punto, .; y escribiendo el campo al que se desea acceder. En este caso, se accede al campo nombre del objeto persona1 para recuperar el valor e imprimirlo en pantalla. En este caso, se imprime en pantalla el texto "Nombre: null". Esto ocurre porque aún no ha sido asignado un valor al campo nombre, por lo que Java le asigna un valor por defecto. En este caso, por ser de tipo `String`, le es asignado el valor de `null`.

En la línea 9, se recupera el valor del campo `apelli-doPaterno` del objeto `persona1`, el cual es concatenado con el texto `"Ap. Paterno: "`. El texto resultante es `"Ap. Paterno: null"`.

En la línea 10, se recupera el valor del campo `apelli-doMaterno` del objeto `persona1`, el cual es concatenado con el texto `"Ap. Materno: "`. El texto resultante es `"Ap. Materno: null"`.

En la línea 11, se recupera el valor del campo `edad` del objeto `persona1`, el cual es concatenado con el texto `"Edad: "`, puesto que el campo `edad` es de tipo `int` y aún no tiene asignado un valor. Java le asigna, por defecto, el valor de **0**. Es así como el texto que se imprime en pantalla es `"Edad: 0"`.

En la línea 12, se recupera el valor del campo `peso` del objeto `persona1`, el cual es concatenado con el texto

"Peso: ", puesto que el campo peso es de tipo int y aún no tiene asignado un valor. Java le asigna por defecto el valor de **0**, es así como el texto que se imprime en pantalla es "Peso: 0".

```
14  // Se asignan valores a los campos del objeto "persona1"
15  persona1.nombre = "Alex";
16  persona1.apellidoPaterno = "Torres";
17  persona1.apellidoMaterno = "Flores";
18  persona1.edad = 20;
19  persona1.peso = 56;
```

La línea 14, es comentario de línea que brinda información sobre las siguientes acciones que serán realizadas dentro del código. En las siguientes líneas, de la 15 a la 19, se accede a los campos que posee el objeto persona1 y se les asignan valores.

```
21          // Se imprimen los valores del objeto "persona1"
22  System.out.println();
23  System.out.println("Nombre: " + persona1.nombre);
24  System.out.println("Ap. Paterno: " + persona1.apellidoPaterno);
25          System.out.println("Ap. Materno: " + persona1.apellidoMaterno);
26  System.out.println("Edad: "+persona1.edad);
27  System.out.println("Peso: "+persona1.peso);
```

En la línea 21, se tiene un comentario de línea que informa al programador que las siguientes instrucciones tienen el objetivo de imprimir los valores que posee el objeto persona1.

Desde la línea 22 hasta la línea 27, se realiza nuevamente el proceso de imprimir los valores que poseen los campos del

objeto persona1, sin embargo, en vista de que, en las líneas anteriores, a cada uno de los campos les fue asignado un valor, el resultado en este caso es:

```
Nombre: Alex
Ap. Paterno: Torres
Ap. Materno: Flores
Edad: 20
Peso: 56
```

5.5Métodos

En las clases, se definen los **atributos** y **comportamientos** con los que contará el objeto cuando éste sea creado. Por cierto, el proceso de crear un objeto, a partir de una clase, se conoce como **instanciación**, o simplemente, **instanciar una clase**. Los **atributos** de la clase son los **campos** y los **comportamientos** son los **métodos**. Los métodos son un conjunto de instrucciones que son ejecutados cuando éstos son invocados, y ¿cómo son invocamos? De forma similar a como se accede a los campos, es decir, se menciona el objeto; se agrega el símbolo de punto, `.`; y se menciona al método a invocar.

Los métodos son las acciones que el objeto puede realizar. Una clase `Extraterrestre`, en un programa, podría tener definidas las acciones de desplazarse; atacar; y tal vez, defenderse. Cada clase define tantos métodos como sean pertinentes. Pero ¿cómo se crean los métodos? La sintaxis para definir los métodos dentro de las clases se muestra en la figura 5.3.

```
Nivel de acceso   Tipo de dato que devuelve   Identificador (Nombre del método)   Parámetros
  public          void                       imprimirTexto                       ( ){
                    // Sentencias a ejecutar
    }
```

Figura 5.3 | Sintaxis de los métodos

La forma de definir un método es, primeramente, estableciendo el nivel de acceso del método. Éste puede ser **`public`**, **`private`**, **`protected`** o **quedar vacío** (nivel paquete). Por ahora, únicamente se utilizará el nivel de acceso **`public`**. Posteriormente se especifica el tipo de dato que el método va a devolver. Éste puede ser cualquier tipo de dato, sin embargo, cuando el método no devuelve valores, se debe indicar con la palabra reservada **`void`**. Luego, se indica el nombre con el que se va a identificar al método. El identificador puede estar formado por letras, números, guion bajo (_) y el símbolo de moneda ($), pero no puede iniciar el nombre con un número. Un identificador válido podría ser **`imprimirPersona1`**, sin embargo, el identificador **`1imprimirPersona`** no es un identificar válido y generaría un error al compilar la clase. Después del identificador, se agregan paréntesis, `()`, dentro de éstos se definen los parámetros. Los parámetros son variables a las que se asignan valores cuando el método es invocado, si no se definen parámetros, los paréntesis quedan vacíos. Por último, se agrega la llave de apertura, `{`, lo que indica el inicio del cuerpo del método. Dentro del cuerpo del método, se coloca el conjunto de sentencias que con ejecutadas cuando el método es invocado. El final del cuerpo del método es establecido con la llave de cierre, `}`.

5.6Convención de nombres de Clases, Campos y Métodos

Tal vez note que en los nombres de las clases se han creado en programas previos, comienzan con mayúsculas, y cada palabra se ha puesto en mayúscula dejando el resto del nombre en minúsculas. Por ejemplo, en la figura 5.2, fue definida la clase `PruebaPersona`. Por convención, en Java, ***el nombre de las clases se escribe en mayúsculas y minúsculas con la primera letra de cada palabra en mayúsculas***. Esto no significa que, si no se aplica de este modo, el programa falla. No, simplemente es una convención para asignar nombres a las clases. Esto ayuda a mejorar la lectura de los programas, y generalmente los programas especializados para el desarrollo de programas incluyen esta convención.

Con respecto a los campos y los métodos, ***el nombre debe iniciar con la primera letra en minúscula, y posteriormente con la primera letra de cada palabra en mayúsculas***. Por ejemplo, algunos nombres de campos válidos son: `apelliDoPaterno`, `codigoPostal`, `entidadFederativa`. Así también, algunos ejemplos de nombres válidos para métodos son `imprimirPersona`, `agregarRegistro`, `eliminarUsuario`.

Recuerde que la convención de los nombres de clases, campos y métodos es un acuerdo para realizar el código de los programas, y aunque no se obliga a seguir esta convención, es una buena práctica apegarse a ella. Al mismo tiempo que mejora la calidad como programadores en el desarrollo de software.

5.7Métodos sin parámetros

En estos momentos, ya se ha trabajado con métodos. Al imprimir texto en pantalla, se invoca al método `print` o `println`. Al crear programas, se define el método `main`. Al leer un valor entero del usuario. Se invoca al método `nextInt` de la clase `Scanner`.

Ahora se aprenderá a definir métodos propios en una clase. Anteriormente se identificó la sintaxis para definir los métodos. Se mencionó que se establece el nivel de acceso; el valor de retorno; un identificador; y dentro de los paréntesis los parámetros. Primero, se analizará y se procederá a aprender a definir métodos sin parámetros que no retornan valores. Para una clara comprensión, preste atención al programa de la figura 5.4.

Persona.java

```
1  package c05.p02;
2
3  class PruebaPersona {
4   String nombre;
5   String apellidoPaterno;
6   String apellidoMaterno;
7   int edad;
8   int peso;
9
10 public void imprimirCampos(){
11    System.out.println("Nombre: "+nombre);
12    System.out.println("Ap. Paterno: " +
                          apellidoPaterno);
13    System.out.println("Ap. Materno: " +
                          apellidoMaterno);
14    System.out.println("Edad: " + edad);
```

```
15      System.out.println("Peso: " + peso);
16      } // Fin del método imprimirCampos
17
18      } // Fin de la clase Persona
```

Figura 5.4 | Clase Persona

PruebaPersona.java

```
1  package c05.p02;
2
3  class PruebaPersona {
4    public static void main(String[] args) {
5      Persona persona1 = new Persona();
6
7     // Se imprimen los valores iniciales del objeto "persona1"
8      persona1.imprimirCampos();
9      System.out.println();
10
11     // Se asignan valores a los campos del objeto "persona1"
12     persona1.nombre = "Alex";
13     persona1.apellidoPaterno = "Torres";
14     persona1.apellidoMaterno = "Flores";
15     persona1.edad = 20;
16     persona1.peso = 56;
17
18    // Se imprimen los valores del objeto "persona1"
19     persona1.imprimirCampos();
20
21   } // Fin del método main
22 } // Fin de la clase PruebaPersona
```

```
Nombre: null
Ap. Paterno: null
Ap. Materno: null
Edad: 0
Peso: 0
Nombre: Alex
Ap. Paterno: Torres
Ap. Materno: Flores
Edad: 20
Peso: 56
```

Figura 5.5 | Prueba de la clase **Persona**.

La figura 5.4 define una clase Persona, en ésta, se están definiendo los campos de tipo `String` de nombre, `apellidoPaterno` y `apellidoMaterno`. Así también, se definen los campos de tipo `int` `edad` y `peso`.

En la línea 1,

```
1   package c05.p02;
```

se indica que la clase pertenece a un paquete de clases, y que se encuentra dentro de una carpeta de nombre **p02**, que a su vez, está dentro de una carpeta de nombre **c05**.

En la línea 10,

```
10  public void imprimirCampos(){
```

se define un método identificado por el nombre de `imprimirCampos`, el cual se utiliza para ejecutar las instrucciones definidas dentro de cuerpo del método. Al momento de leer el código de una clase, se distinguen los métodos de los campos, porque los métodos indican un valor de retorno, y al

final llevan los paréntesis. En este caso, se tiene un método que no devuelve valores, por lo que es indicado con la palabra reservada `void`. Así también, éste es un método que no requiere información para ejecutar las sentencias, por lo que los paréntesis permanecen vacíos. Más adelante, en este capítulo, se abordan los métodos que definen parámetros.

Desde la línea 11 hasta la línea 15,

```
10  public void imprimirCampos(){
11      System.out.println("Nombre: "+nombre);
12      System.out.println("Ap. Paterno: " +
                                apellidoPaterno);
13      System.out.println("Ap. Materno: " +
                                apellidoMaterno);
14      System.out.println("Edad: " + edad);
15      System.out.println("Peso: " + peso);
16  } // Fin del método imprimirCampos
```

se imprimen los valores de los campos definidos en la clase. Como se está dentro de la misma clase, sólo es necesario indicar el nombre del campo del cual se desea recuperar su valor. Eso se observa en los campos resaltados en las instrucciones del método `imprimirCampos`. La línea 16 indica el final del cuerpo del método a través de la llave de cierre, `}`. Al definir este método, cada vez que se quieren imprimir los valores de los campos de un objeto de tipo `Persona` en pantalla, únicamente se requiere llamar al método `imprimirCampos`.

Ya se tiene claro cuáles son los atributos y comportamiento de la clase `Persona`, es decir, sus campos y métodos. Ahora se aborda cómo crear un objeto de la clase `Persona` y la forma de invocar al método que ésta tiene definido.

En la línea 5 de la figura 5.5,

```
5 | Persona persona1 = new Persona();
```

se está creando un objeto de tipo `Persona`, el cual lo identificamos por el nombre de `persona1`.

En la línea 8

```
8 | persona1.imprimirCampos();
```

se invoca al método `imprimirCampos` del objeto persona1. Así que las acciones que fueron definidas en la clase Persona para cuando el método `imprimirCampos` fuese invocado, son ejecutadas. Es así como después de invocar a este método, se muestran en pantalla los valores asignados por defecto a cada uno de los campos.

```
Nombre: null
Ap. Paterno: null
Ap. Materno: null
Edad: 0
Peso: 0
```

Posteriormente, en la línea 9, se imprime un salto de línea a través del método `println`.

Desde la línea 11 hasta la línea 16,

```
12 | persona1.nombre = "Alex";
13 | persona1.apellidoPaterno = "Torres";
14 | persona1.apellidoMaterno = "Flores";
15 | persona1.edad = 20;
16 | persona1.peso = 56;
```

se asignan valores a los campos del objeto persona1.

En la línea 19,

```
19 | persona1.imprimirCampos();
```

se invoca nuevamente al método `imprimirCampos` del objeto `persona1`, por lo que el resultado en pantalla corresponde con los valores asignados en las recientes líneas anteriores.

```
Nombre: Alex
Ap. Paterno: Torres
Ap. Materno: Flores
Edad: 20
Peso: 56
```

Al crear objetos de nuestras clases, hay que comprender que las características de cada objeto son únicas. Así como a partir de un plano arquitectónico, se crean varios edificios. Cada uno de los edificios construidos es diferente. Cada edificio se encuentra ubicado en un lugar único. Cada edificio puede ser de un color diferente. Cada edificio tendrá su propia dirección postal. De tal manera, las clases y los objetos funcionan igual. A partir de una clase, se crean varios objetos, pero cada objeto tiene su propia ubicación en memoria. Cada objeto posee sus propios atributos, en pocas palabras, los objetos que se crean a partir de una clase, son del mismo tipo, pero cada uno de ellos es independiente de los demás. En la figura 5.7, se crean varios objetos de la clase persona y se asignan diferentes valores para comprender claramente que cada objeto es independiente de los demás

PruebaPersona.java

```
1  package c05.p03;
2
3  class PruebaPersona {
4      String nombre;
5      String apellidoPaterno;
6      String apellidoMaterno;
7      int edad;
8      int peso;
9
10    public void imprimirCampos(){
11     System.out.println("Nombre: "+nombre)
12     System.out.println("Ap. Paterno: " +
                            apellidoPaterno );
13     System.out.println("Ap. Materno: " +
                            apellidoMaterno );
14     System.out.println("Edad: " + edad );
15     System.out.println("Peso: " + peso );
16 } // Fin del método imprimirCampos
17
18     } // Fin de la clase Persona
```

Figura 5.6 | Clase Persona

PruebaPersona.java

```
1  package c05.p03;
2
3  class PruebaPersona {
4    public static void main(String[] args) {
5      // Creación de objetos de tipo Persona
6      Persona persona1 = new Persona();
7      Persona persona2 = new Persona();
8      Persona persona3 = new Persona();
```

```
9
10      // Asignación de valores
11      persona1.nombre = "Alex";
12      persona1.apellidoPaterno = "Torres";
13      persona1.apellidoMaterno = "Flores";
14      persona1.edad = 21;
15      persona1.peso = 56;
16
17      persona2.nombre = "Beto";
18      persona2.apellidoPaterno = "Aguirre";
19      persona2.apellidoMaterno = "Cruz";
20      persona2.edad = 20;
21      persona2.peso = 61;
22
23      persona3.nombre = "Carla";
24      persona3.apellidoPaterno = "Reyes";
25      persona3.apellidoMaterno = "Uscanga";
26      persona3.edad = 19;
27      persona3.peso = 58;
28
29      // Impresión en pantalla
30      persona1.imprimirCampos();
31      System.out.println();
32
33      persona2.imprimirCampos();
34      System.out.println();
35
36      persona3.imprimirCampos();
37
38    } // Fin del método main
39  } // Fin de la clase PruebaPersona
```

```
Nombre: Alex
Ap. Paterno: Torres
Ap. Materno: Flores
Edad: 21
Peso: 56
Nombre: Beto
Ap. Paterno: Aguirre
Ap. Materno: Cruz
Edad: 20
Peso: 61
Nombre: Carla
Ap. Paterno: Reyes
Ap. Materno: Uscanga
Edad: 19
Peso: 58
```

Figura 5.7 | Prueba de la clase **Persona**.

El programa de la figura 5.6 es similar con el programa de la figura 5.4. La única diferencia entre los dos programas es en la línea 1, `package c05.p03`, que indica el paquete al que pertenece la clase.

En las líneas 6,7 y 8 de la clase `PruebaPersona` (figura 5.7) se crean 3 objetos de tipo Persona que son identificados como `persona1`, `persona2` y `persona3`.

De las líneas 11 a la 15, se asignan valores a los campos del objeto `persona1`.

De las líneas 17 a la 21, se asignan valores a los campos del objeto `persona2`.

De las líneas 23 a la 27, se asignan valores a los campos del objeto `persona3`.

La línea 30 invoca al método `imprimirCampos` del objeto `persona1`.

La línea 33 invoca al método `imprimirCampos` del objeto `persona2`.

La línea 36 invoca al método `imprimirCampos` del objeto `persona3`.

Lo importante del programa es comprender que cada uno de los objetos creados a partir de la clase `Persona`, son diferentes en el sentido de que cada uno de ellos cuenta con sus propios valores en los campos, y que al invocar a alguno de sus métodos, en este caso, al método `imprimirCampos`. Las acciones corresponden con los valores que cada objeto posee de forma independiente.

5.8Métodos con un parámetro

Ahora que se conoce la utilidad de los métodos, se mostrará cómo funcionan los métodos con parámetros. Cuando invocamos un método, sabemos que éste realizará un conjunto de instrucciones, pero a veces los métodos van a necesitar información extra para poder realizar sus acciones. Por ejemplo, cuando queremos imprimir un texto en pantalla e invocamos al método `println`, éste sabe cómo realizar sus acciones, pero no sabe cuál es el texto que debe de imprimir. Esa información, nosotros la proporcionamos dentro de los paréntesis. La figura 5.8 muestra la sintaxis de un método que define un parámetro.

```
Nivel de acceso   Tipo de dato que devuelve   Identificador (Nombre del método)   Parámetros
  public            void                      imprimirTexto                       (String texto){
                        // Sentencias a ejecutar
    }
```

Figura 5.8 | Sintaxis de método con un parámetro

Los parámetros que definen los métodos son variables a las que se les asigna el valor enviado al momento de ser invo-

cado. En el ejemplo de la figura 5.8, el parámetro `texto` de tipo `String` es una variable que se utiliza dentro del cuerpo del método para llevar a cabo las acciones que se requieren. La figura 5.9 y 5.10 muestran un ejemplo de cómo se define un método con un parámetro y su invocación.

Operaciones.java

```
1  package c05.p04;
2
3  class Operaciones {
4
5   public void imprimirTexto(String texto){;
6   System.out.println( texto );
7      } // Fin del método imprimirTexto
8
9      } // Fin de la clase Operaciones
```

Figura 5.9 | Clase que define un método con un parámetro

PruebaOperaciones.java

```
1  package c05.p04;
2
3  class PruebaOperaciones {
4    public static void main(String[] args) {
5      Operaciones operaciones =
   new Operaciones();
6
7      operaciones.imprimirTexto(
                              "Primer Texto" );
8      operaciones.imprimirTexto(
                             "Segundo Texto" );
9      operaciones.imprimirTexto(
                              "Tercer Texto" );
10
```

```
11      } // Fin del método main
12  } // Fin de la Clase PruebaOperaciones
```

```
Primer Texto
Segundo Texto
Tercer Texto
```

Figura 5.10 | Prueba de la clase **Operaciones**.

En la figura 5.9 se define una clase de nombre `Operaciones`, dentro de esta clase, se define un método con un parámetro que devuelve valores. El nombre del método es `imprimirTexto`. Tiene un parámetro de tipo `String` con el nombre de texto. El nivel de acceso es `public`, y puesto que no devuelve valores, se hace uso de la palabra reservada `void` para el tipo de dato de retorno. Se observa que en la línea 6 se imprime en pantalla el valor que posee el parámetro `texto`. Ese valor le será asignado al momento de la invocación.

En la figura 5.10 se crea un programa para probar la clase Operaciones. En la línea 5, se crea una instancia de la clase Operaciones, es decir, es creado un objeto de tipo Operaciones. Este objeto es utilizado en las líneas 7, 8 y 9 para invocar al método `imprimirTexto`. En la línea 7, se utiliza el texto de `"Primer Texto"` al momento de la invocación del método `imprimirTexto`, por lo que ese es el valor que le es asignado al parámetro del método `imprimirTexto`, y al ejecutarse las instrucciones de éste, vemos en pantalla el texto `Primer Texto`. En la línea 8 se invoca al método `imprimirTexto` y se le envía el texto `"Segundo Texto"`, por lo que este es el valor que toma el parámetro del método. El resultado en pantalla es `Segundo Texto`. Por último, en la línea 9, se invoca al método `imprimirTexto` enviando el texto `"Tercer Texto"`, con lo que el resultado en pantalla es `Primer Texto`.

5.9Métodos con varios parámetros

Los métodos no sólo definen un parámetro, en realidad, se definen tantos parámetros como lo consideremos pertinente. Para definir más de un parámetro, lo realizamos separando por comas, **,** , cada uno de los parámetros.

```
Nivel de acceso   Tipo de dato que devuelve   Identificador (Nombre del método)   Parámetros
public            void                       concatenar-Textos                   (String text1, String text2){

                        // Sentencias a ejecutar

    }
```

Figura 5.11 | Sintaxis de método con varios parámetros

Se procede a analizar un programa en el que se tengan métodos con más de un parámetro.

Operaciones.java

```
1   package c05.p05;
2
3   class Operaciones {
4
5     public void concatenarTexto(String texto1,
                                  String texto2){
6       System.out.println( texto1 + " " +
                                      texto2 );
7     } // Fin del método concatenarTexto
8
9     public void imprimirSuma(int entero1,
                                int entero2){
10      int suma = entero1 + entero2;
11      System.out.println( "Suma: " + suma );
12    } // Fin del método imprimirSuma
```

```
13
14  } // Fin de la clase Operaciones
```

Figura 5.12 | Clase que define un método con un parámetro

PruebaOperaciones.java

```
1   package c05.p05;
2
3   class PruebaOperaciones {
4     public static void main(String[] args) {
5       Operaciones operaciones =
                                new Operaciones();
6
7       operaciones.concatenarTexto("Primero",
                                      "Segundo");
8       operaciones.concatenarTexto("Tercero",
                                       "Cuarto");
9
10      operaciones.imprimirSuma(10,20);
11      operaciones.imprimirSuma(30,40);
12
13    } // Fin del método main
14  } // Fin de la Clase PruebaOperaciones
```

```
Primero Segundo
Tercero Cuarto
Suma: 30
Suma: 70
```

Figura 5.13 | Prueba de la clase **Operaciones**.

En la clase Operaciones de la figura 5.12, se definen dos métodos con 2 parámetros, el primero es definido en la línea 5.

```
5  public void concatenarTexto( String texto1,
                                String texto2){
```

El método es identificado como `concatenarTexto` y se establecen 2 parámetros de tipo `String`, el primero de nombre `texto1`, y el segundo de nombre `texto2`. Estos parámetros son utilizados en la línea 6,

```
6  System.out.println(texto1 + " " + texto2);
```

para imprimir la concatenación de los textos dejando un espacio de separación entre ambos textos.

El segundo método es definido en la línea 9.

```
9  public void imprimirSuma(int entero1,
                            int entero2){
```

El identificador del método es `imprimirSuma`, en éste, se definen 2 parámetros de tipo `int`. El nombre del primer parámetro es `entero1`, y el nombre del segundo parámetro es `entero2`.

En las líneas 10 y 11,

```
10  int suma = entero1 + entero2;
11  System.out.println( "Suma:  " + suma );
```

se define una variable de tipo `int` con el identificador de `suma`, y le es asignado el resultado de la suma de los valores que posee `entero1` y `entero2`. Posteriormente, la suma se muestra en pantalla agregando al inicio el texto de "`Suma:  `".

Con respecto al programa de la figura 5.13, en la línea 5,

```
5  Operaciones operaciones=new Operaciones();
```

se crea un objeto a partir de la clase Operaciones.

En la línea 7 y 8 ,

```
7  operaciones.concatenarTexto("Primero",
                                "Segundo");
8  operaciones.concatenarTexto("Tercero",
                                "Cuarto");
```

se invoca al método `concatenarTexto` del objeto operaciones. En la línea 7, se envían los textos `"Primero"` y `"Segundo"`, éstos serán asignados a los parámetros del método `concatenarTexto`, que son utilizados para imprimirlos en pantalla. El resultado en pantalla es `Primero Segundo`. En la línea 8, se invoca nuevamente al método `concatenarTexto` del objeto operaciones, se envían los valores de `"Tercero"` y `"Cuarto"` como argumentos, el resultado en pantalla es `Tercero Cuarto`. A los valores que son enviados durante la invocación de un método, se conocen como **argumentos**; en la línea 7 y 8, los valores `"Primero"`, `"Segundo"`, `"Tercero"` y `"Cuarto"` son argumentos.

Las líneas 10 y 11,

```
10  operaciones.imprimirSuma(10,20);
11  operaciones.imprimirSuma(30,40);
```

invoca al método `imprimirSuma` del objeto operaciones. En la línea 10, se envían como argumentos los valores de tipo `int 10` y `20`. El método `imprimirSuma` manipula estos valores para realizar la suma e imprimirla en pantalla, el resultado en pantalla es `Suma: 30`. En la línea 11, se envían como argumentos los valores de tipo `int 30` y `40` al ser invocado el método `imprimirSuma`, el resultado es `Suma: 70`.

5.10Métodos que devuelven datos

Los métodos se utilizan para ejecutar un conjunto de instrucciones. Éstos pueden definir o no parámetros para llevar a cabo las operaciones. Hasta el momento, únicamente se han utilizado métodos que no retornan datos, sin embargo, los métodos pueden entregarnos información. Cuando un método retorna un dato como respuesta, primero se ejecutan las instrucciones definidas dentro del cuerpo del método, y una vez que se tiene el valor que se desea devolver, éste es devuelto a través de la palabra reservada `return`.

```
Nivel de acceso   Tipo de dato que devuelve   Identificador (Nombre del método)   Parámetros

public      int        sumar     ( int numero1, int numero2 ){

         int suma = numero1 + numero2;        return suma;

}
```

Figura 5.14 | Sintaxis de método que retorna un dato

En el método de la figura 5.14, se establece que el tipo de dato que devolverá el método será un dato de tipo `int`. El identificador del método es `sumar`. El método define 2 parámetros de tipo `int`. El primero de nombre `numero1`, y el segundo con el nombre `numero2`. En el cuerpo del método se define una variable de nombre `suma`, a la que le es asignado el resultado de sumar los valores que poseen los parámetros `numero1` y `numero2`. Finalmente, el método retorna el valor de tipo `int` que posee la variable `suma`. Para que el método devuelva un dato, debe indicarlo con la palabra reservada `return` y el valor a devolver.

Si un método indica que va a devolver un tipo de dato, y dentro del cuerpo del método no devuelve un dato, se

obtiene un error al intentar compilar el código, por ejemplo, si el método de la figura 5.14 no tuviera la línea con la sentencia `return suma;`, al intentar compilar, se obtendría un error.

Operaciones.java

```
1   package c05.p06;
2
3   class Operaciones {
4
5     public int sumar(int numero1,int numero2)
    {
6       int suma = numero1 + numero2;
7       return suma;
8     } // Fin del método sumar
9
10    public int restar( int numero1,
                                  int numero2){
11      return numero1 - numero2;
12    } // Fin del método restar
13
14    public int cuadrado( int numero ){
15      return numero * numero;
16    } // Fin del método cuadrado
17
18  } // Fin de la clase Operaciones
```

Figura 5.15 | Clase que define un método con un parámetro

PruebaOperaciones.java

```
1  package c05.p06;
2
3  class PruebaOperaciones {
4    public static void main(String[] args) {
5      Operaciones operaciones =
                              new Operaciones();
6
7      int resultado=operaciones.sumar(10, 20);
8      System.out.println( resultado );
9
10     System.out.println(
                   operaciones.restar(30,9) );
11
12     System.out.println(
                   operaciones.cuadrado(11) );
13
14   } // Fin del método main
15 } // Fin de la Clase PruebaOperaciones
```

```
30
21
121
```

Figura 5.16 | Prueba de la clase **Operaciones**.

Para entender mejor la forma en que definimos los métodos que retornan valores, se analizará el código de las clases de las figuras 5.15 y 5.16. La primera define una clase de nombre `Operaciones`, en la que se tienen 3 métodos que retornan valores de tipo `int`. La figura 5.16 muestra una clase en la que se crea una instancia de la clase `Operaciones` y se hace uso de los métodos que posee.

En la clase Operaciones (figura 5.15), el primer método es definido desde la línea 5 hasta la línea 8.

```
5  public int sumar(int numero1, int numero2){
6      int suma = numero1 + numero2;
7      return suma;
8  } // Fin del método sumar
```

En la línea 5, la parte resaltada indica que el método devolverá un valor de tipo `int`. En la misma línea, se indica que el nombre con el que es identificado el método es `sumar`, y éste define dos parámetros de tipo `int`, el primero con el nombre `numero1`, y el segundo parámetro con el nombre `numero2`. Dentro del cuerpo del método, se declara una variable de nombre suma de tipo `int` a la que le es asignado el resultado de la operación `numero1 + numero2`. En la línea 7, se devuelve el valor de la variable `suma`, con lo que se cumple el retorno de un valor de tipo `int`, tal y como fue indicado en la línea 5.

El método de la línea 10,

```
10  public int restar(int numero1,int numero2){
11      return numero1 - numero2;
12  } // Fin del método restar
```

indica que devolverá un dato de tipo `int`. Este método es nombrado como `restar`, y define dos parámetros de tipo `int`. En la línea 11, se indica que se debe devolver el resultado de restar el valor del parámetro `numero1` menos el valor del parámetro `numero2`. A diferencia del método `sumar`, en este método, se retorna directamente el resultado de una operación aritmé-

tica, en lugar de almacenar el resultado en una variable y devolverlo, tal como ocurre en el método `sumar`.

El último método definido en la clase Operaciones es el método cuadrado en la línea 14.

```
14  public int cuadrado( int numero ){
15      return numero * numero;
16  } // Fin del método cuadrado
```

Este método establece que será devuelto un dato de tipo `int`. El método define un parámetro de tipo `int` con el nombre de `numero`. En la línea 15, se indica que se devuelva el resultado de `numero * numero`.

La clase `PruebaOperaciones`, de la figura 5.16, es la encargada de comprobar el funcionamiento de los métodos de la clase `Operaciones` (figura 5.15).

En la línea 5,

```
5  Operaciones operaciones=new Operaciones();
```

se crea una instancia de la clase `Operaciones`. Este objeto es utilizado en las siguientes líneas para comprobar el funcionamiento de los métodos definidos en la clase `Operaciones`.

Las líneas 7 y 8,

```
7  int resultado = operaciones.sumar(10, 20);
8  System.out.println( resultado );
```

son establecidas para comprobar el resultado devuelto por el método `sumar`. En la línea 7 se invoca al método `sumar` del objeto `operaciones`. Al método le son pasados como argumento los valores de `10` y `20`. Recuerde que un **argumento** *es un valor que es enviado durante la invocación de un método*

para que sea asignado a un parámetro. El valor devuelto por el método `sumar` es asignado a la variable resultado. En la línea 8, se imprime en pantalla el valor que posee la variable `resultado`. Se puede observar que el resultado en pantalla es 30.

La línea 10,

```
10 | System.out.println(
   |                  operaciones.restar(30,9) );
```

invoca al método `restar` del objeto `operaciones`, y se envían como argumentos los valores de `30` y `9`. El resultado de invocar al método `restar` se imprime en pantalla obteniendo `21`.

En la línea 12,

```
12 | System.out.println(
   |                  operaciones.cuadrado(11) );
```

se imprime en pantalla el valor devuelto por el método cuadrado del objeto operaciones. Al invocar al método cuadrado, se envía como argumento el valor de 11. El resultado en pantalla es `121`.

En este programa, se realizaron llamados a métodos que devolvían datos. Así también, todos los métodos invocados recibían valores. Si bien se realizaron invocaciones a métodos definidos en la clase Operaciones, hay un método más que recibe un argumento, pero no fue definido por nosotros, y lo estuvimos invocando varias veces. ¿Lo identifica? Probablemente se ha percatado que `println` es un método que recibe un argumento, y ese argumento es pasado al parámetro que tiene definido el método `println` para realizar las operaciones pertinentes e imprimir en pantalla el valor enviado. Claro que, aún queda por entender qué significa cada parte de `System.out.println()`, pero eso lo iremos deduciendo se avance en los temas.

5.11Sobrecarga de métodos

La sobrecarga de métodos es una potente característica que permite definir varios métodos con un mismo nombre como identificador. Esto resulta muy útil para definir acciones a realizar con base en los argumentos que son enviados durante la invocación.

Si los métodos tienen el mismo nombre, ¿cómo Java determina qué método ejecutar? Para determinar el método a ser ejecutado, Java revisa los parámetros de los métodos sobrecargados. Java no toma en cuenta el tipo de datos de retorno o el nivel de acceso de éste. Java realizará la selección con base en los parámetros. Esta selección está basada en el número de parámetros y el tipo de dato de los parámetros.

Veamos un ejemplo de la sobrecarga de métodos.

Operaciones.java

```
1  package c05.p07;
2
3  class Operaciones {
4
5    public void mostrarDatos( int numero ){
6       System.out.println( "Parámetros: int" );
7    } // Fin del método mostrarDatos
8
9    public void mostrarDatos( String texto ){
10     System.out.println(
                          "Parámetros: String");
11   } // Fin del método mostrarDatos
12
13   public void mostrarDatos( int numero,
                                  String texto){
14     System.out.println(
                     "Parámetros: int, String");
```

```
15    } // Fin del método mostrarDatos
16
17        public void mostrarDatos( String texto,
                                    int numero ){
18      System.out.println(
                    "Parámetros: String, int" );
19    } // Fin del método mostrarDatos
20
21  } // Fin de la clase Operaciones
```

Figura 5.17 | Clase con métodos sobrecargados

PruebaOperaciones.java

```
1   package c05.p07;
2
3   class PruebaOperaciones {
4     public static void main(String[] args) {
5       Operaciones operaciones =
                              new Operaciones();
6
7       operaciones.mostrarDatos( 99 );
8       operaciones.mostrarDatos( "Hola" );
9       operaciones.mostrarDatos( 99, "Hola" );
10      operaciones.mostrarDatos( "Hola", 99 );
11
12    } // Fin del método main
13  } // Fin de la Clase PruebaOperaciones
```

```
Parámetros: int
Parámetros: String
Parámetros: int, String
Parámetros: String, int
```

Figura 5.18 | Prueba de la clase **Operaciones**.

En la clase Operaciones de la figura 5.17, se definen 4 métodos sobrecargados, es decir, tienen el mismo nombre. Para que clase Operaciones sea ser compilada, debe existir una diferencia con respecto a los parámetros. Java analiza 3 características de los parámetros; el **número de parámetros**, el **tipo de dato** de los parámetros y el **orden de los parámetros**. Si dos o más de los métodos sobrecargados tienen las mismas características que Java analiza, la compilación fallará.

El método de la línea 5 tiene un parámetro de tipo `int`. El método de la línea 9 tiene un parámetro, pero éste es de tipo `String`, por lo que los métodos no tendrán problemas al compilar. El método de la línea 13 tiene dos parámetros, uno de tipo `int` y otro de tipo `String`. Este método no tiene conflicto con los anteriores porque el número de parámetros es diferente, así que Java no tendrá problema en determinar cuando se quiera invocar a este método. Por último, el método de la línea 17 tiene dos parámetros, uno de tipo `String` y otro de tipo `int`. Este método tiene el mismo número de parámetros y tipos de datos que el método de la línea 13,

```
5    public void mostrarDatos( int numero ){…}

     public void mostrarDatos( String texto )
9    {…}

     public void mostrarDatos( int numero,
13   String texto ){…}

     public void mostrarDatos( String texto,
17   int numero ){…}
```

sin embargo, el método de la línea 17 tiene un distinto **orden de parámetros** al del método de la línea 13. En el método de la línea 17, primero define un parámetro de tipo `String`, y

después un parámetro de tipo `int`, mientras que el método de la línea 13 define primero un parámetro de tipo `int`, y después un parámetro de tipo `String`, así que ninguno de los métodos sobrecargados entra en conflicto, porque todos pueden ser diferenciados por Java al momento de la invocación.

Recuerde, los métodos sobrecargados son diferenciados por Java por:

- El número de parámetros,
- el tipo de dato de los parámetros,
- y el orden de los parámetros.

Las acciones definidas dentro de cada método son la impresión en pantalla de los tipos de datos definidos como parámetros.

El programa de la figura 5.18 crea una instancia de la clase `Operaciones` para comprobar el funcionamiento de los métodos definidos en dicha clase.

En la línea 7,

```
7 | operaciones.mostrarDatos( 99 );
```

se invoca al método `mostrarDatos` del objeto `operaciones`, ese método corresponde con un método de nombre `mostrarDatos` que espera recibir un dato de tipo `int`. Java verifica que exista un método definido con dichas características y procede a ejecutar las acciones de ese método. Es así como se imprime en pantalla el texto `Parámetros: int`.

En la línea 8,

```
8 | operaciones.mostrarDatos( "Hola" );
```

se invoca al método `mostrarDatos` del objeto `operaciones`. Para está invocación Java verifica que se tenga definido un método de nombre `mostrarDatos y` un parámetro de tipo `String`. El resultado en pantalla con respecto a la invocación del método mostrar de la línea 8 es `Parámetros: String`.

En la línea 9,

```
9 | operaciones.mostrarDatos( 99, "Hola" );
```

la invocación del método `mostarDatos` corresponde con el método sobrecargado que define como primer parámetro un dato de tipo `int`, y como segundo parámetro un dato de tipo `String`. El método que será ejecutado debe cumplir con las 3 características analizadas por Java: el número de parámetros, los tipos de datos, y el orden de los parámetros. La ejecución de la sentencia de la línea 9 da como resultado el texto `Parámetros: int, String`.

En la línea 10,

```
10 | operaciones.mostrarDatos( "Hola", 99 );
```

se tiene la invocación del método `mostrarDatos`. En está invocación, se envía como primer argumento un dato de tipo `String`, y como segundo argumento, un dato de tipo `int`. El método sobrecargado que cumple con las características de

los datos utilizados como argumentos es ejecutado, y el resultado en pantalla es `Parámetros: String, int`.

5.12Conclusión

En este capítulo se aprendieron conceptos básicos sobre la Programación Orientada a Objetos. Aprendimos a definir clases, e instanciar objetos. Así también, definimos y accedimos a campos de objetos, pero principalmente, trabajamos con los métodos. Éstos son muy importantes porque nos brindan muchos beneficios al crear nuestros programas como la reducción de código; mejora en la lectura de nuestro código; reduce la complejidad de nuestros programas; mejora el rendimiento; entre otras razones.

En el siguiente capítulo, continuaremos profundizando sobre algunos temas y características de la POO, y el modo en que lo llevamos a cabo en Java.

6.**POO: Constructores e Instancias**

6.1Introducción

En el capítulo anterior, se dieron a conocer conceptos básicos de la POO, y se trabajó con los métodos. En este capítulo, se muestran algunas otras características de la POO. Principalmente los niveles de acceso. Se trabajará con los constructores, los campos y métodos *static*, y se utilizarán objetos como argumentos. Todos estos conceptos que trabajaremos en Java, brindarán las bases aplicables para cualquier lenguaje de programación que implemente la POO.

6.2Niveles de acceso

Primeramente, serán estudiados los **niveles de acceso**. En temas anteriores, sólo se mencionaron los **niveles de acceso**, para que se fuera familiarizando con el término, pero no llegamos a utilizarlos más allá de colocar niveles de acceso `public` o simplemente, no colocar nada. De hecho, también es un nivel de acceso el no poner nada, pero eso se aborda en un momento más.

En Java, tenemos 4 niveles de acceso: **`public`**, **`protected`**, ***`default`*** (sin modificador) y **`private`**. Cada uno de ellos brinda acceso o no desde la misma clase; desde las clases del paquete; desde una subclase; o desde cualquier clase. La figura 6.1 muestra los niveles de acceso.

Modificador	Clase	Paquete	Subclase	Cualquiera
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
Sin modificador (default)	Sí	Sí	No	No
private	Sí	No	No	No

Figura 6.1 | Niveles de acceso

Los niveles de acceso se asignan a través de los modificadores de acceso, que son las palabras reservadas que hemos mencionado anteriormente y que se pueden observar en la figura 6.1.

Los **niveles de acceso** definen las clases que pueden acceder a nuestros campos o invocar a nuestros métodos.

El modificador `public` establece que el elemento permite el acceso desde cualquier clase; ya sea desde la misma clase; desde una clase dentro del mismo paquete; o desde una subclase. Éste es el nivel de acceso que brinda mayor libertad. Una subclase es una clase que hereda las características de otra clase. A las subclases, las veremos en el tema de herencia.

El modificador `protected` permite que el elemento sea accedido desde la misma clase; desde una clase dentro del mismo paquete o desde una subclase.

El modificador *`default`*, es decir, cuando **no se aplica un modificador** de acceso, permite que el elemento esté disponible desde la misma clase o por las clases dentro del mismo

paquete. Este nivel de acceso también es conocido como **nivel paquete**.

Por último, el modificado `private` es el nivel de acceso de mayor restricción. El elemento sólo estará disponible dentro de la misma clase.

Los niveles de acceso permiten tener control sobre la visibilidad de los elementos desde otras clases. Por el momento, sólo se analizan y comprueban los niveles de acceso `public` y `private`. El resto de los niveles de acceso, serán retomados después de comprender y trabajar con la herencia.

Persona.java

```
1  package c06.p01;
2
3  class Persona {
4    public String nombre;
5    public int edad;
6
7    public void imprimirCampos(){
8      System.out.println("Nombre: " +
   nombre);
9      System.out.println("Edad: " + edad);
10     System.out.println();
11   } // Fin del método imprimirCampos
12
13 } // Fin de la clase Persona
```

Figura 6.2 | Clase con campos y métodos de nivel de acceso `public`.

PruebaPersona.java

```
1  package c06.p01;
2
3  class PruebaPersona {
```

```
4     public static void main(String[] args) {
5       Persona persona = new Persona();
6
7       persona.imprimirCampos();
8       persona.nombre = "Alex";
9       persona.edad = 20;
10      persona.imprimirCampos();
11
12    } // Fin del método main
13  } // Fin de la clase PruebaPersona
```

```
Nombre: null
Edad: 0
Nombre: Alex
Edad: 20
```

Figura 6.3 | Prueba de la clase **Persona**.

El archivo de la clase `Persona`, de la figura 6.2, lleva por nombre **Persona.java**, y éste debe estar dentro de una carpeta de nombre **p01**, que a su vez, está dentro de una carpeta de nombre **c06**. Esto debe ser así, puesto que se indica en la línea 1,

```
1   package c06.p01;
```

que la clase está dentro del paquete `c06.p01`.

No hay mucho por decir de la clase `Persona`. Prácticamente ya se han realizado clases de características similares, sin embargo, en las líneas 4 y 5,

```
4   public String nombre;
5   public int edad;
```

se tienen 2 campos a los que se les ha asignado el **modificado de acceso** `public`. En ejercicios anteriores, se había dejado

vacío, lo que significaba que los campos tenían nivel de acceso *default* o **nivel paquete**. Para efectos de los ejercicios anteriores, e incluso de éste, no se tiene un cambio apreciable. Lo que sí se sabe, es que los campos `nombre` y `edad` pueden ser accedidos desde cualquier clase, incluso desde clases que no estén dentro del mismo paquete.

Desde la línea 7 hasta la 11,

```
7    public void imprimirCampos(){
8      System.out.println("Nombre: " + nombre);
9      System.out.println("Edad: " + edad);
10     System.out.println();
11   } // Fin del método imprimirCampos
```

se define un método de nombre `imprimirCampos`, al cual le es asignado el nivel de acceso `public`. Lo que implica que el método puede ser invocado desde una instancia que haya sido creada en cualquier clase.

La clase `PruebaPersona` de la figura 6.03 es guardada en un archivo con el nombre de **PruebaPersona.java**, y ésta debe estar dentro de la misma carpeta en la que se encuentra el archivo `Persona.java` de la figura 6.02.

La instrucción `Persona persona = new Persona();` de la línea 5 crea una instancia con base en la clase Persona.

En la línea 7,

```
persona.imprimirCampos();
```

se invoca al método `imprimirCampos`, lo que da como resultado en pantalla:

```
Nombre: null
Edad: 0
```

En las líneas 8 y 9,

```
8 | persona.nombre = "Alex";
9 | persona.edad = 20;
```

se accede a los campos del objeto `persona`. En la línea 8, se asigna el valor de `"Alex"` al campo `nombre` del objeto `persona`. En la línea 9, se asigna el valor de `20` al campo `edad` del objeto `persona`.

Por último, en la línea 10,

```
persona.imprimirCampos();
```

se invoca nuevamente al método `imprimirCampos` del objeto `persona`, lo que esta vez da como resultado en pantalla:

```
Nombre: Alex
Edad: 20
```

Hasta el momento, el programa de las figuras 6.2 y 6.3 no parecieran que brinde nuevos conocimientos, y tal vez así sea. Se realizarán unos pequeños cambios a la clase `Persona`. Será cambiado el **modificador de acceso** de las líneas 4 y 5 por el **modificador de acceso** `private`. El cambio en las líneas anterior debe de verse similar a:

```
4 | private String nombre;
5 | private int edad;
```

Se procede a guardar los cambios, y compilar nuevamente la clase `Persona`. Al compilar nuevamente la clase `PruebaPersona`. ¿Qué sucedió? ¿Se obtuvo el siguiente error?:

```
c:\c06\p01>javac -classpath c:\ PruebaPersona.
java
PruebaPersona.java:8: error: nombre has private
access in Persona
        persona.nombre = "Alex";
                ^
PruebaPersona.java:9: error: edad has private
access in Persona
        persona.edad = 20;
                ^
2 errors
```

Es correcto, ese error debía de ocurrir. Lo que informa el compilador de Java al intentar compilar el archivo **PruebaPersona.java,** es que no es posible acceder al campo `nombre` del objeto `persona`, porque éste tiene nivel de acceso `private`. Lo mismo ocurre con la línea 9, no es posible acceder al campo `edad` del objeto `persona` porque tiene nivel de acceso `private`. Es pertinente recordar que el nivel de acceso `private` significa que esos elementos sólo son utilizados dentro de la misma clase, para este caso, sólo pueden ser usados dentro de la clase `Persona`. Entonces, si no se accede a esos elementos, ¿cómo será posible asignar o recuperar los valores de esos campos? La respuesta es a través de métodos de nivel de acceso `public`. Nuevamente, es necesario recordar que los elementos con el nivel de acceso `private` sólo son utilizados dentro de la misma clase, y los elementos con nivel de acceso `public`, son utilizados desde cualquier clase. Entonces no se puede acceder al campo `nombre` o `edad` desde fuera de la clase `Persona`, pero sí crear un método dentro de la clase `Persona` con nivel

de acceso `public`, y que éste permita recuperar o asignar valores a los campos `nombre` y `edad` de la clase `Persona`. Piénsese por un momento. Imagínese un método que retorne el valor del campo `nombre`. Podría ser llamado `getNombre`. ¿y qué tal otro método que nos permita asignar un valor al campo `nombre`? Podría ser llamado `setNombre`, y éste sería definido con un parámetro que reciba el valor y lo asigne al campo. ¿Qué tal suena? ¿Parece confuso? Bueno, quizás porque aún, no se construye en código, pero una vez realizado, la idea será más clara.

Analice por un momento el código de la clase **`Persona`** de la figura 6.4

Persona.java

```
1   package c06.p02;
2
3   class Persona {
4     private String nombre;
5     private int edad;
6
7     public void imprimirCampos(){
8       System.out.println("Nombre: " +
    nombre);
9       System.out.println("Edad: " + edad);
10      System.out.println();
11    } // Fin del método imprimirCampos
12
13    public String getNombre(){
14      return nombre;
15    } // Fin del método getNombre
16
17    public void setNombre(String valor){
18      nombre = valor;
```

```
19    } // Fin del método setNombre
20
21    public int getEdad(){
22      return edad;
23    } // Fin del método getEdad
24
25    public void setEdad(int valor){
26      edad = valor;
27    } // Fin del método setEdad
28
29  } // Fin de la clase Persona
```

Figura 6.4 | Clase con métodos sobrecargados

En el código de la clase `persona`, se definien los campos `nombre` y `edad` con el nivel de acceso `private`, lo que significa, que estos campos únicamente pueden ser utilizados dentro de la clase `Persona`. Sin embargo, desde la línea 13 hasta la línea 27, se definen 4 métodos con nivel de acceso `public`, lo que significa, que estos métodos son invocados desde cualquier otra clase, sin excepción. Estos métodos interactúan con los campos definidos en la clase. El método de la línea 13,

```
13   public String getNombre(){
14     return nombre;
15   } // Fin del método getNombre
```

devuelve un valor de tipo `String`, y dicho valor corresponde a lo que tenga asignado el campo `nombre`. Así que, aunque el campo `nombre` no es utilizado directamente desde fuera de la clase, sí se conoce el valor asignado a través de este método con nivel de acceso `public`.

El método setNombre,

```
17   public void setNombre(String valor){
18     nombre = valor;
19   } // Fin del método setNombre
```

define un parámetro de tipo `String` con el nombre de valor. Este parámetro es utilizado para asignar el valor que éste posee al campo `nombre`. De este modo, aunque no se accede desde afuera de la clase al campo `nombre`, sí realiza una asignación a través de este método con nivel de acceso `public`.

Los métodos `getEdad` y `setEdad`,

```
21   public int getEdad(){
22     return edad;
23   } // Fin del método getEdad
24
25   public void setEdad(int valor){
26     edad = valor;
27   } // Fin del método setEdad
```

realizan lo mismo que los métodos `setNombre` y `getNombre`, sólo que `getEdad` y `setEdad` realizan sus operaciones con respecto al campo `edad`.

Si puedo manipular campos públicos directamente, ¿por qué crear métodos públicos para manipular campos privados? Es una pregunta razonable. La respuesta es, porque es una buena práctica realizarlo de esta manera. Se tiene mayor control de los datos que serán asignados; se pueden realizar validaciones de los datos y; entre otros motivos, se disminuye la probabilidad de errores. Antes de continuar la discusión sobre los beneficios de realizar la asignación y lectura de esta manera, se procederá a comprobar el funcionamiento de la clase `Persona`.

PruebaPersona.java

```
1  package c06.p02;
2
3  class PruebaPersona {
4    public static void main(String[] args) {
5      Persona persona = new Persona();
6
7      System.out.println(persona.getNombre());
8      System.out.println(persona.getEdad());
9
10     persona.setNombre( "Alex" );
11     persona.setEdad( 20 );
12
13     persona.imprimirCampos();
14
15   } // Fin del método main
16 } // Fin de la clase PruebaPersona
```

```
null
0
Nombre: Alex
Edad: 20
```

Figura 6.5 | Prueba de la clase **Persona**.

El programa de la figura 6.5 usa los métodos con nivel de acceso `public` para la asignación y recuperación de los atributos de la clase `Persona`.

La línea 5,

```
5  Persona persona = new Persona();
```

crea una instancia de la clase `Persona`.

En la línea 7,

```
7  System.out.println( persona.getNombre() );
```

se invoca al método `getNombre` del objeto `persona`.

Este método devuelve el valor que tiene asignado el campo `nombre`, el cual tiene el valor que Java le asignó por defecto, por lo que se imprime `null` en pantalla. En esta línea, se observa que, aunque no es posible acceder al campo `nombre`, que tiene nivel de acceso `private`, sí es posible hacer uso del método `getNombre`, que es public, el cual puede leer ese campo y devolvernos el valor al invocarlo.

En la línea 8,

```
8 | System.out.println( persona.getEdad() );
```

se invoca al método `getEdad` del objeto `persona` con base en las instrucciones definidas en la clase `Persona` (figura 6.4). Éste método devuelve el valor que posee el campo `edad`, que en ese instante es `0`, por lo que el resultado en pantalla es `0`.

La línea 10,

```
10 | persona.setNombre( "Alex" );
```

se invoca al método `setNombre` y se le pasa como argumento el valor de `"Alex"`. Con base en las instrucciones definidas en la clase Persona (figura 6.4), el método utiliza el valor recibido para asignarlo al campo edad.

En la línea 11,

```
11 | persona.setEdad( 20 );
```

se invoca al método `setEdad` y se envía como argumento el valor `int` de `20`. Dentro de las instrucciones del método `setEdad`, se asigna el valor de `20` al campo `edad`.

En la línea 13,

```
13 | persona.imprimirCampos();
```

se invoca al método imprimirCampos, el cual hace uso de los campos `nombre` y `edad` para imprimirlos en pantalla.

```
Nombre: Alex
Edad: 20
```

Al ver en pantalla el resultado de la invocación del método `imprimirCampos`, se aprecia que, efectivamente los

valores de los campos `nombre` y `edad` del objeto `persona` fueron modificados en las invocaciones a los métodos `setNombre` y `setEdad` de las líneas 11 y 12.

Bien, pero aún no sé ¿por qué debería utilizar métodos para asignar o recuperar los datos de los campos? Ésa sería una pregunta válida. Se verá un ejemplo de por qué es recomendable que los campos siempre se manejen con nivel de acceso *private* en lugar de *public*. Si se tuviiese un campo en una clase `Persona` como este:

```
public int edad;
```

Se da oportunidad a que cualquier clase acceda al campo para recuperar o asignar un valor. Si en alguna otra clase, se crea una instancia de la clase `Persona` y le dan por nombre el identificador de `persona`, una asignación sería realizada de la siguiente manera:

```
persona.edad = 20;
```

Eso está bien, incluso, pensar que es una forma más práctica de realizarlo, *si se viviera en un mudo perfecto y sin errores*. Piénsese qué sucedería si alguien por error o intencionalmente asignara un valor negativo.

```
persona.edad = -4;
```

La lógica del programa no tendrá sentido, y muy seguramente funcionará de forma incorrecta, puesto que se espera que el valor del campo `edad` siempre sea un valor igual o mayor a 0.

Ahora, ¿cómo validar y controlar los valores que serán asignado al campo `edad`? Si en lugar del campo `edad` como `public`, se asigna como `private`, ninguna otra clase podrá manipular el campo. Ahora, en el método `setEdad`, se agregar una validación antes de asignar el valor al campo `edad`. Se muestra cómo queda el fragmento de código:

```
 private int edad; public void setEdad( int valor ){
if( valor < 0 ){
        edad = 0;
    }else{
        edad = valor;
    }
 } // Fin del método setEdad
```

En este fragmento de código, el campo `edad` ha sido establecido con un nivel de acceso `private`, por otra parte, el método `setEdad` asigna `0` en caso de que el valor recibido sea menor a `0`, y si no se asigna el valor recibido, es decir, se garantiza que nunca se asignen valores negativos al campo `edad`, de este modo, si se realizará una invocación como la siguiente:

```
 persona.setEdad( -4 );
```

El valor que se asigna al campo `edad` es `0`, porque el método setEdad realiza una validación antes de asignar el valor.

Recordando, está forma de trabajar con los campos es sólo una recomendación, no significa quesea obligatorio trabajar los campos de este modo, sin embargo, es una buena práctica realizarlo así, y es parte de la convención de Java.

6.3Constructores

Los **constructores** son los encargados de preparar al objeto que será creado. Toda clase cuenta con un constructor, así es, toda clase cuenta con un constructor. Cuando en una clase no se define un constructor, Java define un constructor por defecto. En los programas elaborados, se han realizado llamadas a los constructores de las clases cada vez que creaba una

instancia. Por ejemplo, el último programa (figura 6.5) incluía la siguiente línea:

```
Persona persona = new Persona();
```

En esta línea se crea una instancia de la clase `Persona`. La parte resaltada, `Persona()` es la llamada al constructor. La palabra reservada `new` es el encargado de asignar memoria al objeto que es creado, y devuelve la referencia de dicha ubicación en memoria. El **operador** `new` siempre es seguido por la llamada a un constructor.

Anteriormente se mencionó la analogía de un plano arquitectónico y un edificio. El plano es como la clase de la POO, éste define las características con las que contará el futuro objeto. Cuando se lleva a cabo la construcción del edificio, se realiza con base en el plano arquitectónico. Cuando un objeto en la POO es creado, se realiza con base en la clase. Ahora piénsese antes de crear el edificio. Se necesita saber en qué ubicación será construido. Algún espacio de suelo es designado para realizar la construcción del edificio. Pues bueno, lo mismo pasa con los objetos en la POO. Antes de construir un objeto, se necesita saber dónde será construido el objeto, es decir, en qué espacio en la memoria de la computadora será construido el objeto. De eso se encarga el **operador** `new`. El **constructor** vendría a ser la acción de construir y levantar el edificio.

La sintaxis de un **constructor** tiene parecido a la sintaxis de los métodos.

```
Nivel de acceso   Identificador (mismo nombre de la clase)   Parámetros
  public                  Persona                            ( ){
                           // Sentencias
     }
```

Figura 6.6 | Sintaxis del constructor

Los **constructores** cuentan con **nivel de acceso**, al igual que los métodos, pero los constructores nunca devuelven datos,

por lo que no existe un tipo de datos de retorno. El **nombre del constructor** siempre es el mismo que el de la clase, es decir, si estamos dentro de una clase de nombre `Persona`, el nombre del constructor es `Persona`. Por último, en los constructores también se definen parámetros como en los métodos, éstos van dentro de los paréntesis. En caso de no requerir parámetros, los paréntesis quedan vacío.

Los constructores realizan el proceso de creación de los objetos. Dentro de las instrucciones del constructor, se establecen acciones para inicializar los campos de la instancia. Será retomada la clase `Persona`, pero está vez haciendo uso de un constructor.

Persona.java

```
1   package c06.p03;
2
3   class Persona {
4     private String nombre;
5     private int edad;
6
7     public Persona(){
8       nombre = "(Sin nombre)";
9       edad = 0;
10    } // Fin del constructor
11
12    public void imprimirCampos(){
13      System.out.println("Nombre: " +
    nombre);
14      System.out.println("Edad: " + edad);
15      System.out.println();
16    } // Fin del método imprimirCampos
17
18    public String getNombre(){
```

```
19      return nombre;
20    } // Fin del método getNombre
21
22    public void setNombre(String valor){
23      nombre = valor;
24    } // Fin del método setNombre
25
26    public int getEdad(){
27      return edad;
28    } // Fin del método getEdad
29
30    public void setEdad(int valor){
31      edad = valor;
32    } // Fin del método setEdad
33
34  } // Fin de la clase Persona
```

Figura 6.7 | Clase con constructor

La clase `Persona` de la figura 6.7 es muy parecida a la clase `Persona`, que se trabajó anteriormente en la figura 6.4. Lo único que cambia es que esta clase se encuentra en el paquete `c06.p03`, y se agrega un constructor que se muestra resaltado en las líneas 7 hasta la 10. En estas líneas,

```
7   public Persona(){
8       nombre = "(Sin nombre)";
9       edad = 0;
10  } // Fin del constructor
```

se observa que el constructor tiene nivel de acceso `public`. El nombre del constructor siempre es el mismo nombre que el de la clase, por lo que el constructor lleva por nombre `Persona`. El constructor no define parámetros, y en el cuerpo del constructor se inicializan los valores de los campos

nombre y edad. En el caso del campo nombre, le es asignando el String "(Sin nombre)". El campo edad es inicializado con el valor de 0.

Se comprueba la clase Persona a través de un programa que crea una instancia de esta clase.

PruebaPersona.java

```
1  package c06.p03;
2
3  class PruebaPersona {
4    public static void main(String[] args) {
5      Persona persona = new Persona();
6
7      persona.imprimirCampos();
8
9      persona.setNombre( "Alex" );
10     persona.setEdad( 20 );
11
12     persona.imprimirCampos();
13
14   } // Fin del método main
15 } // Fin de la clase PruebaPersona
```

```
Nombre: (Sin nombre)
Edad: 0
Nombre: Alex
Edad: 20
```

Figura 6.8 | Prueba de la clase **Persona**.

En la clase PruebaPersona, en la línea 5, se crea una instancia de un objeto de la clase Persona. La creación del objeto es realizada por el operador new, seguido de la llamada al constructor Persona(). En la llamada al construc-

tor `persona`, todas las instrucciones definidas en éste, son ejecutadas. Para esta situación, el constructor `Persona()` inicializa los campos `nombre` y `edad` con los valores especificados en la clase `Persona` (figura 6.7).

En la línea 7, se invoca al método `imprimirCampos` del objeto `persona`. Este método imprime en pantalla los valores de los campos `nombre` y `edad`. El resultado en pantalla es

```
Nombre: (Sin nombre)
Edad: 0
```

En programas anteriores, después de crear un objeto, los valores iniciales de los campos eran asignados por Java, sin embargo, en este programa los valores iniciales han sido definidos en el constructor de la clase `Persona`. Es así como el valor inicial para el campo `nombre` es "`(Sin nombre)`". En el caso del campo `edad`, el valor inicial asignado desde el constructor es similar al que Java asigna por defecto, por lo que se observa diferencia aparente.

En las líneas 9 y 10,

```
9   persona.setNombre( "Alex" );
10  persona.setEdad( 20 );
```

se utilizan los métodos `setNombre` y `setEdad` para asignar los valores de "`Alex`" y `20` a los campos `nombre` y `edad`. Es así como la invocación del método `imprimirCampos` de la línea 12, muestra como resultado:

```
Nombre: Alex
Edad: 20
```

6.4Constructores con parámetros

En el programa anterior, se vio que los constructores son el primer conjunto de instrucciones en ser ejecutadas durante la creación de un objeto. Así también, toda clase cuenta con por lo menos un constructor. No puede existir una clase sin un constructor, y cuando no se define un constructor en una clase, Java agregará un constructor sin parámetros por defecto.

Cuando se definen métodos en las clases, éstos pueden o no tener parámetros. En los constructores ocurre lo mismo, los constructores pueden o no definir parámetros. Los parámetros en los constructores son muy útiles para enviar información al constructor durante la creación de un objeto. Por ejemplo, si se tiene una **clase** `Cuadrado`, se puede enviar información al constructor para inicializar el valor de un campo `lado`. Así también, una **clase** `Rectangulo`, informa al constructor el valor de los campos `base` y `altura` para que sean inicializados durante la construcción del objeto.

Se procede a hacer que la clase `Persona` tenga un constructor con parámetros que permita inicializar los campos de la instancia.

Persona.java

```
1  package c06.p04;
2
3  class Persona {
4    private String nombre;
5    private int edad;
6
7    public Persona(String pNombre, int pEdad){
8      nombre = pNombre;
```

```
9       edad = pEdad;
10    } // Fin del constructor
11
12    public void imprimirCampos(){
13      System.out.println("Nombre: " + nombre);
14      System.out.println("Edad: " + edad);
15      System.out.println();
16    } // Fin del método imprimirCampos
17
18    public String getNombre(){
19      return nombre;
20    } // Fin del método getNombre
21
22    public void setNombre(String valor){
23      nombre = valor;
24    } // Fin del método setNombre
25
26    public int getEdad(){
27      return edad;
28    } // Fin del método getEdad
29
30    public void setEdad(int valor){
31      edad = valor;
32    } // Fin del método setEdad
33
34  } // Fin de la clase Persona
```

Figura 6.9 | Clase con un constructor con parámetros

La clase Persona que se crea en la figura 6.9 establece que pertenece al paquete `c06.p04`, que debe estar dentro de una carpeta de nombre **p04**, y ésta a su vez está dentro de una carpeta de nombre **c06**.

Con excepción de la línea 7 hasta la línea 10,

```
7    public Persona(String pNombre, int pEdad){
8        nombre = pNombre;
9        edad = pEdad;
10   } // Fin del constructor
```

ya se ha trabajado en programas anteriores con del resto del código. La línea 7 define un constructor con nivel de acceso `public` y con dos parámetros: `pNombre` de tipo `String` y `pEdad` de tipo `int`. El parámetro `pNombre` es utilizado en la línea 8 para ser asignado e inicializar el campo `edad`. El parámetro `pEdad` es usado en la línea 9 para ser asignado e inicializar el campo `edad`.

Ya creada la clase `Persona`, se verá cómo utilizar el constructor que fue definido para crear un objeto que tenga valores inicializados.

PruebaPersona.java

```
1    package c06.p04;
2
3    class PruebaPersona {
4      public static void main(String[] args) {
5        Persona persona = new Persona(
                                        "Alex", 20);
6
7        persona.imprimirCampos();
8
9        persona.setNombre( "Beto" );
10       persona.setEdad( 19 );
11
12       persona.imprimirCampos();
13
14     } // Fin del método main
15   } // Fin de la clase PruebaPersona
```

```
Nombre: Alex
Edad: 20
Nombre: Beto
Edad: 19
```

Figura 6.10 | Prueba de la clase **Persona**.

En este programa, se crea un objeto de tipo `Persona` haciendo uso del constructor al que le son enviados los argumentos de `"Alex"` y `20`. Una vez creado el objeto, sus campos ya cuentan con los valores que fueron pasados como argumento. Es por lo que el método `imprimirCampos` del objeto `persona` en la línea 7 muestra como resultado en pantalla

```
Nombre: Alex
Edad: 20
```

En la línea 9, se invoca al método `setNombre` para asignar el texto `"Beto"` al campo nombre. En la línea 10, se invoca al método `setEdad` para asignar el valor de `19` al campo `edad`. Es así como el resultado en pantalla al invocar al método `imprimirCampos` del objeto `persona` en la línea 12 da como resultado:

```
Nombre: Beto
Edad: 19
```

6.5Sobrecarga de constructores

En el capítulo anterior, se trabajaron con los métodos sobrecargados y las características de los parámetros que utiliza Java para poder diferenciarlos, puesto que todos los métodos sobrecargados comparten el mismo nombre. Ahora serán estudiado los constructores sobrecargados. En sí, un **constructor sobrecargado** es la definición de más de un constructor en

un a clase, no tendría sentido decir que son constructores con el mismo nombre, porque al ser un constructor, está implícito el hecho de tener el mismo nombre que la clase, así que al mencionar que existe más de un constructor, se da por hecho que el nombre será el mismo de la clase para cada constructor.

Los métodos sobrecargados son diferenciados por las características de los parámetros. En los constructores sobrecargados es igual. Los constructores sobrecargados serán diferenciados por Java en base a 3 características:

- El número de parámetros.
- El tipo de dato de los parámetros.
- El orden de los parámetros.

Comprobemos el uso de los constructores sobrecargados con una clase.

Rectangulo.java

```
1  public class Rectangulo {
2    private int base;
3    private int altura;
4
5    public Rectangulo(){
6      base = 1;
7      altura = 1;
8    } // Fin del constructor
9
10   public Rectangulo(int pBase, int pAltura){
11     base = pBase;
12     altura = pAltura;
13   } // Fin del constructor
14
15   public void imprimirCampos(){
16     System.out.println( "Base: " + base );
17     System.out.println( "Altura: "+altura);
18     System.out.println();
19   } // Fin del método imprimirCampos
20
21 } // Fin de la clase Rectángulo
```

Figura 6.11 | Clase con constructores sobrecargados

La clase `Rectangulo` define dos campos de tipo `int` y nivel de acceso `private`. El primero con el nombre de `base` y el segundo con el nombre de `altura`. La clase `Rectangulo` define dos constructores sobrecargados, uno sin parámetros y otro con dos parámetros. Éstos, serán analizados con más detalle en un momento. Por último, la clase define un método de nombre `imprimirCampos` que se encarga de imprimir en pantalla los valores de los campos `base` y `altura`.

Veamos las acciones que realiza cada uno de los constructores.

En primer constructor definido en la clase,

```
5  public Rectangulo(){
6      base = 1;
7      altura = 1;
8  } // Fin del constructor
```

es un constructor sin parámetros. Este constructor inicializa los campos `base` y `altura`. En la línea 6, se asigna el valor de `1` al campo `base`. En la línea 7, se asigna el valor de `1` al campo `altura`.

Por otro lado, se tiene otro constructor de la línea 10 a la 14.

```
10  public Rectangulo(int pBase, int pAltura){
11      base = pBase;
12      altura = pAltura;
13  } // Fin del constructor
```

Este constructor define dos parámetros de tipo `int`, uno de nombre `pBase` y otro de nombre `pAltura`. Los parámetros definidos por el constructor son utilizados para inicializar los campos `base` y `altura`. En la línea 11, se asigna el valor del parámetro `pBase` al campo `base`. En la línea 12 se asigna el valor del parámetro `pAltura` al campo `altura`.

En esta clase `Rectángulo`, se ha decidido no permitir que los valores de los campos sean modificados fuera de la clase, es decir, los campos poseen nivel de acceso `private`, y no se han incluido métodos con nivel de acceso `public` que permitan recuperar o asignar valores a los campos. Hemos decidido realizarlo de esta manera para no extender demasiado la clase `Rectangulo`, y así podamos enfocarnos

en comprender y atender, principalmente el tema de los **constructores sobrecargados**.

Realicemos un programa que permita poner a prueba nuestra clase `Rectangulo`.

PruebaRectangulo.java

```
1  package c06.p05;
2
3  public class PruebaRectangulo {
4    public static void main(String[] args) {
5                    Rectangulo rectangulo1=new
                                      Rectangulo();
6      Rectangulo rectangulo2 =
                             new Rectangulo(7,11);
7      Rectangulo rectangulo3 =
                            new Rectangulo(13,17);
8
9      rectangulo1.imprimirCampos();
10     rectangulo2.imprimirCampos();
11     rectangulo3.imprimirCampos();
12
13   } // Fin del método main
14 } // Fin de la clase PruebaRectangulo
```

```
Base: 1
Altura: 1
Base: 7
Altura: 11
Base: 13
Altura: 17
```

Figura 6.12 | Prueba de la clase **Rectangulo**.

En el programa de la figura 6.12, se crean 3 objetos de la clase `Rectangulo` y se imprimen los valores iniciales de los campos `base` y `altura`. Las instancias de la clase `Rectan-`

gulo son creadas haciendo uso de los dos constructores sobrecargados definidos en la clase Rectangulo. En la línea 5,

```
5   Rectangulo rectangulo1 = new
                                    Rectangulo();
```

se hace uso del constructor sin argumentos de la clase Rectangulo para construir un objeto que es nombrado rectangulo1. Con base en las instrucciones establecidas en el cuerpo de este constructor, los campos base y altura son inicializados con los valores de 1.

En la línea 6,

```
6   Rectangulo rectangulo2 =
                            new Rectangulo(7,11);
```

se crea un objeto con el nombre de rectangulo2, éste es de tipo Rectángulo. Al constructor se le envían como argumentos los valores int de 7 y 11. El constructor con 2 parámetros de tipo int, dentro de sus acciones, utiliza los parámetros para inicializar los valores de los campos base y altura; es así como el campo base es inicializado con el valor de 7, y el campo altura es inicializado con el valor de 11.

El último objeto de tipo Rectangulo es creado en la línea 7,

```
7   Rectangulo rectangulo3 =
                           new Rectangulo(13,17);
```

Éste es nombrado como rectangulo3. Este objeto es creado a partir de la llamada al constructor que recibe dos argumentos de tipo int. Como el constructor es invocado con los valores int de 13 y 17, el campo base es inicializado con el valor de 13 y el campo altura es inicializado con el valor de 17.

En las líneas 9, 10 y 11;

```
9   rectangulo1.imprimirCampos();
10  rectangulo2.imprimirCampos();
11  rectangulo3.imprimirCampos();
```

se invoca al método `imprimirCampos` de cada uno de los objetos de tipo `Rectangulo`. En el resultado en pantalla,

```
Base: 1
Altura: 1
Base: 7
Altura: 11
Base: 13
Altura: 17
```

se observa que efectivamente los campos del objeto `rectangulo1` fueron inicializados con el valor de `1`, los campos del objeto `rectangulo2` fueron inicializados con los valores de `7` y `11` y los campos del objeto `rectangulo3` fueron inicializados con los valores `13` y `17`.

Los constructores son parte importante de la POO, éstos permiten realizar acciones durante la creación de los objetos. Es posible crear tantos constructores sobrecargados como sean necesarios, sólo con el cuidado de permitir que Java pueda diferenciar los constructores a través de los parámetros, de lo contrario, se obtendrá un error al intentar compilar la clase.

6.6Campos static

En la POO, se definen clases que permiten especificar las características que tendrá el objeto cuando sea creado. Los atributos, se trabajan como **campos de la instancia** y los com-

portamientos, se trabajan como **métodos**. Los campos de la instancia son independientes para cada objeto, es decir, cada objeto cuenta con su propio espacio en memoria para ese recurso. En todos los programas realizados hasta el momento, cada vez que se ha creado un nuevo objeto de alguna clase definida por nosotros, se ha observado que los valores de los campos de cada objeto no se comparten, es decir, cuando se trabajó con objetos de la clase `Persona`, el valor asignado al campo `edad` de un objeto no realizaba cambios en el campo `edad` de cualquier otro objeto de tipo `Persona`. Es importante comprender el uso de los campos de los objetos para entender lo que son los **campos `static`**, también conocidos como **campos de clase**.

Los **campos `static`** o **campos de clase** son campos cuyo valor es compartido por todos los objetos que sean creado de la clase. Véase un ejemplo.

ClaseA.java

```
1  package c06.p06;
2
3  public class ClaseA {
4      public int campoA;
5      public int campoB;
6      public static int campoCompartido;
7  } // Fin de la clase ClaseA
```

Figura 6.13 | Clase con campos de instancia y un campo de clase.

La clase `ClaseA` de la figura 6.13 es una clase sencilla que define dos campos de tipo `int` con nivel de acceso `public` y un campo `static` de tipo `int`. Los **campos de instancia** han sido nombrados como `campoA` y `campoB`. El **campo de clase** o **campo `static`** ha sido llamado `campoCompartido`. Para indicar que un campo es `static`, se asigna la

palabra reservada `static` después del nivel de acceso como se puede observar en la línea 6.

Probemos la clase `ClaseA` y sus campos en un programa.

PruebaClaseA.java

```
1  package c06.p06;
2
3  public class PruebaClaseA {
4    public static void main(String[] args) {
5      ClaseA obj1 = new ClaseA();
6      ClaseA obj2 = new ClaseA();
7
8      obj1.campoA = 10;
9      obj2.campoA = 20;
10     System.out.println( obj1.campoA + ", " +
       obj2.campoA);
11
12     obj1.campoB = 100;
13     obj2.campoB = 200;
14     System.out.println( obj1.campoB + ", " +
       obj2.campoB);
15
16     obj1.campoCompartido = 1000;
17     obj2.campoCompartido = 2000;
18     System.out.println( obj1.campoCompartido + ",
       " +
       obj2.campoCompartido);
19
20   } // Fin del método main
21 } // Fin de la clase PruebaClaseA
```

```
10, 20
100, 200
2000, 2000
```

Figura 6.14 | Prueba de la clase **ClaseA**.

El programa de la figura 6.14 comienza con la instanciación de 2 objetos en las líneas 5 y 6.

```
5  ClaseA obj1 = new ClaseA();
6  ClaseA obj2 = new ClaseA();
```

Estos objetos son creados a partir de la clase `ClaseA`. El objeto de la línea 5 es nombrado como `obj1` y el objeto de la línea 6 es nombrado como `obj2`. Ambos objetos han sido creados a partir de la llamada al constructor `ClaseA()` que Java define por defecto, puesto que en la clase `ClaseA` no fue creado ningún constructor.

En las líneas 8 y 9,

```
8  obj1.campoA = 10;
9  obj2.campoA = 20;
```

se accede al `campoA` de cada uno de los objetos y se le asignan valores. En la línea 8, se asigna el valor de `10` al campo `campoA` del objeto `obj1`. En la línea 9, se asigna el valor de `20` al campo `campoA` del objeto `obj2`. En la línea 10,

```
10  System.out.println( obj1.campoA + ", " +
                                    obj2.campoA);
```

se imprime el valor del `campoA` de los objetos `obj1` y `obj2`. El resultado en pantalla es `10, 20`. Cada objeto tiene de forma independiente un espacio en memoria para guardar el valor del campo `campoA`.

En las líneas 12 y 13,

```
12  obj1.campoB = 100;
13  obj2.campoB = 200;
```

se asignan valores al campoB de los objetos obj1 y obj2. En la línea 12, se asigna el valor de `100` al campo campoB del

objeto obj1. En la línea 13, se asigna el valor de `200` al campo campoB del objeto obj2. Al imprimir los valores de ambos objetos, tal y como se realiza en la línea 14,

```
14   System.out.println( obj1.campoB + ", " +
     obj2.campoB);
```

el resultado en pantalla es `100, 200`.

En las líneas 16 y 17,

```
16   obj1.campoCompartido = 1000;
17   obj2.campoCompartido = 2000;
```

los objetos `obj1` y `obj2` acceden al **campo `static`** y asignan valores. En la línea 16, el objeto obj1 accede al **campo `static`** `campoCompartido` y asigna el valor de `1000`. En la línea 17, el objeto `obj2` accede **al campo `static`** `campoCompartido` y asigna el valor de `2000`. En la línea 18,

```
18   System.out.println( obj1.campoCompartido + ", "
                       +  obj2.campoCompartido);
```

se imprime el **campo `static`** `campoCompartido` a través del objeto `obj1` y `obj2`. El resultado en pantalla es `2000, 2000`. El resultado en esta situación es diferente que cuando utilizamos campos de instancia. Cuando un valor es asignado a un **campo `static`**, este valor se ve reflejado para todos los objetos, es por eso por lo que, aunque el objeto `obj1` había asignado el valor de `1000` al **campo `static`** `campoComparto`, éste no se imprime, ya que inmediatamente después el objeto `obj2` asigna el valor de `2000` al **campo `static`** `campoCompartido`, y el resultado en pantalla es `2000, 2000`.

Es importante observar que los **campos `static`** son compartidos entre todas las instancias de la clase. Cualquier instancia de la clase que realice cambios en el valor de un **campo**

`static`, se verá reflejado cuando cualquier otra instancia recupere el valor de dicho **campo `static`**.

Algo más a saber sobre los *campos static*, es que los campos pueden ser utilizados sin necesidad de instanciar un objeto, por eso también se les conoce como *campos de clase*. Para acceder a los *campos static* sin necesidad de una instancia, se escribe el nombre de la clase seguido un punto, `.`, y el nombre del campo. Para la clase `ClaseA` de la figura 6.13., es posible acceder al *campo static* `campoCompartido` a través de la sentencia `ClaseA.campoCompartido`.

Ahora, se realiza un programa que accede a un **campo `static`** a través de la mención de la clase y a través de una instancia para comprobar que el campo se puede acceder sin necesidad de un objeto.

ClaseB.java

```
1  package c06.p07;
2
3  public class ClaseB {
4      public static int campoStatic;
5  } // Fin de la clase ClaseA
```

Figura 6.15 | Clase con un **campo de clase**.

La clase `ClaseB` únicamente define un **campo `static`** que ha sido nombrado como `campoStatic` y es de tipo `int`.

PruebaClaseB.java

```
1  package c06.p07;
2
3  public class PruebaClaseB {
4   public static void main(String[] args) {
5     ClaseB obj1 = new ClaseB();
6     ClaseB obj2 = new ClaseB();
7
8     System.out.println("obj1: " +
   obj1.campoStatic);
9     System.out.println("obj2: " +
   obj2.campoStatic);
10    System.out.println("ClaseB: " +
   ClaseB.campoStatic +"\n");
11
12    obj1.campoStatic = 10;
13    System.out.println("obj1: " +
   obj1.campoStatic);
14    System.out.println("obj2: " +
   obj2.campoStatic);
15    System.out.println("ClaseB: " +
   ClaseB.campoStatic +"\n");
16
17    obj2.campoStatic = 200;
18    System.out.println("obj1: " +
   obj1.campoStatic);
19    System.out.println("obj2: " +
   obj2.campoStatic);
20    System.out.println("ClaseB: " +
   ClaseB.campoStatic +"\n");
21
22    ClaseB.campoStatic = 3000;
23    System.out.println("obj1: " +
   obj1.campoStatic);
```

```
24      System.out.println("obj2: " +
    obj2.campoStatic);
25      System.out.println("ClaseB: " +
    ClaseB.campoStatic +"\n");
26
27    } // Fin del método main
28  } // Fin de la clase PruebaClaseB
```

```
obj1: 0
obj2: 0
ClaseB: 0
obj1: 10
obj2: 10
ClaseB: 10
obj1: 200
obj2: 200
ClaseB: 200
obj1: 3000
obj2: 3000
ClaseB: 3000
```

Figura 6.16 | Prueba de la clase **ClaseB**

El programa de la figura 6.16 comprueba las diferentes formas para acceder y asignar valores al **campo `static`** `campoStatic` de la clase `ClaseB`.

En la línea 5 y 6,

```
5   ClaseB obj1 = new ClaseB();
6   ClaseB obj2 = new ClaseB();
```

se crean 2 instancias de la clase `ClaseB`, los objetos son nombrados como `obj1` y `obj2`.

Desde la línea 8 hasta la línea 10,

```
8   System.out.println( "obj1:" +
                              obj1.campoStatic);
9   System.out.println( "obj2:" +
                              obj2.campoStatic);
```

```
10  System.out.println( "ClaseB:" +
                            ClaseB.campoStatic);
```

se imprime el valor del **campo `static`** `campoStatic` de la clase `ClaseB`. En la línea, 8 se imprime `campoStatic` haciendo uso del objeto `obj1`. En la línea, 9 se imprime `campoStatic` haciendo uso del objeto `obj2`. Por último, en la línea 10, se imprime `campoStatic` haciendo uso de la clase `ClaseB`. Se observa que se accede al campo sin necesidad de un objeto, es decir, únicamente mencionando la Clase. Al visualizar el resultado en pantalla,

```
obj1:0
obj2:0
ClaseB:0
```

se observa que el valor inicial para el **campo `static`** `campoStatic` es de `0`.

En la línea 12,

```
12  obj1.campoStatic = 10;
13  System.out.println( "obj1:" +
                            obj1.campoStatic);
14  System.out.println( "obj2:" +
                            obj2.campoStatic);
15  System.out.println( "ClaseB:" +
                          ClaseB.campoStatic);
```

se asigna el valor de `10` a `campoStatic` haciendo uso del objeto `obj1`. Puesto que `campoStatic` es un campo de clase, cuando la clase u otra instancia de la clase `ClaseB` llegue a recuperar el valor de `campoStatic`, éste será de `10`. Es así como las impresiones en pantalla de las líneas 13, 14 y 15 dan como resultado:

```
obj1: 10
obj2: 10
ClaseB: 10
```

En la línea 17,

```
17 | obj2.campoStatic = 200;
```

se asigna el valor de `200` al campo `campoStatic` a través del objeto `obj2`. Cualquier otro objeto de tipo `ClaseB` que recupere el valor de `campoStatic` obtendrá el valor de `200`. De igual forma, si se recupera el valor de `campoStatic` a través de la mención de la clase `ClaseB`, se obtendrá el valor de `200`. Tal y como ocurre en las líneas 18, 19 y 20,

```
18 | System.out.println( "obj1: " +
   |                              obj1.campoStatic);
19 | System.out.println( "obj2: " +
   |                              obj2.campoStatic);
20 | System.out.println( "ClaseB: " +
   |                    ClaseB.campoStatic + "\n");
```

Al recuperar `campoStatic` a través de `obj1`, `obj2` o `ClaseB`, el valor recuperado es el mismo al que fue asignado por `obj2` en la línea 17. El resultado en pantalla es:

```
obj1: 200
obj2: 200
ClaseB: 200
```

En la línea 22,

```
22 | ClaseB.campoStatic = 3000;
```

se asigna nuevamente un valor a `campoStatic`, pero esta vez a través de la clase `ClaseB`. Ya se ha mencionado antes que los **campos `static`** pueden ser trabajados sin necesi-

dad de una instancia. En esta línea, se trabaja con el **campo `static`** de nombre `campoStatic` mencionando únicamente la clase, que en este caso, es `ClaseB`. El resultado de imprimir los valores en pantalla es:

```
obj1: 3000
obj2: 3000
ClaseB: 3000
```

6.7Métodos static

En los últimos 2 programas, se trabajó con los *campos static*. Aprendimos a trabajar con los *campos static*, también conocidos como *campos de clase*, sin necesidad de una instancia. Ahora vamos a trabajar con los ***métodos static***. Si se tiene claro cómo funcionan los *campos static*, no se tendrá ningún problema en comprender los ***métodos static***. Cuando a un método le es asignada la palabra `static`, este método puede ser utilizado sin necesidad de instanciar un objeto, tal y como ocurre con los *campos static*. Véase este programa para comprenderlo mejor.

Operaciones.java

```
1  package c06.p08;
2
3  public class Operaciones {
4    public static int sumar(int numero1,
                                 int numero2){
5      return numero1 + numero2;
6    } // Fin del método sumar
7
8    public static int restar(int numero1,
                                 int numero2){
9      return numero1 - numero2;
```

```
10    } // Fin del método restar
11
12  } // Fin de la clase Operaciones
```

Figura 6.17 | Clase con métodos `static`.

La clase `Operaciones`, define 2 **métodos `static`**, el primer método es definido en la línea 4 con el nombre de `sumar`, el segundo es definido en la línea 8 con el nombre de `restar`. Ambos métodos retornan un valor de tipo `int` y ambos métodos definen 2 parámetros de tipo `int` con los nombres de `numero1` y `numero2`. El método `sumar` utiliza los parámetros para devolver el resultado de `numero1 + numero2`. El método `restar` utiliza los parámetros para devolver el resultado de `numero1 - numero2`.

PruebaOperaciones.java

```
1   package c06.p08;
2
3   public class PruebaOperaciones {
4     public static void main(String[] args) {
5       Operaciones obj1 = new Operaciones();
6
7       System.out.println( obj1.sumar(10,20)
    );
8       System.out.println(
    Operaciones.sumar(10,20) );
9
10      System.out.println(obj1.restar(30,20));
11      System.out.println(
    Operaciones.restar(30,20) );
12
13    } // Fin del método main
14  } // Fin de la clase PruebaOperaciones
```

```
30
30
10
10
```

Figura 6.18 | Prueba de la clase **PruebaOperaciones**

En la clase `PruebaOperaciones` se crea una instancia de la clase `Operaciones`, el objeto es nombrado `obj1`. En la línea 7, es utilizado el objeto `obj1` para invocar al ***método static*** `sumar`, los valores de `10` y `20` son enviados como argumentos. El valor devuelto por el método `sumar` es mostrado en pantalla. En la línea 8, se realiza la invocación al método `sumar` del mismo modo que en la línea anterior, pero aquí se realiza la invocación a través de la clase `Operaciones`, es decir, sin depender de una instancia. El resultado en pantalla de la ejecución de las líneas 7 y 8 es:

```
30
30
```

En la línea 10, se invoca al *método static* `restar` haciendo uso del objeto `obj1`. En la línea 11 se invoca, de igual manera, al *método static* `restar`, pero mencionando la clase `Operaciones`. En las dos invocaciones, se envían como argumentos los valores de `30` y `20`. El resultado devuelto por los métodos se imprimen en pantalla dando como resultado:

```
10
10
```

6.8Objetos como argumentos

Un aspecto importante de entender al trabajar con parámetros y argumentos es comprender las implicaciones de utilizar **tipos de datos primitivos** y **tipos de datos por referencia**. Primeramente, recordar que los *parámetros* son definidos al momento de escribir la clase, mientras que los *argumentos* son los datos que se envían al momento de realizar la invocación de un método.

No se tiene la intención profundizar con respecto a los tipos de **datos por referencia**, sin embargo, es importante comprender que cualquier dato que no sea de tipo *primitivo* es un tipo de *dato por referencia*, es decir, un objeto. En Java, únicamente existen 8 tipos de *datos primitivos*: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` y `char`.

Para entender la diferencia entre trabajar con *datos primitivos* y *datos por referencia* como argumentos, se necesita verlo en un programa y analizar los resultados.

Acciones.java

```
1  package c06.p09;
2
3  public class Acciones {
4
5    public static void imprimeIncremento(
                                    int valor ){
6      valor = valor + 10;
7      System.out.println( "Valor: " + valor );
8    } // Fin del método imprimeIncremento
9
10 } // Fin de la clase Acciones
```

Figura 6.19 | Clase con un **campo de clase**.

Principal.java

```
1  package c06.p09;
2
3  public class Principal {
4    public static void main(String[] args) {
5      int variable1 = 50;
6      System.out.println(
             "Variable (antes):" + variable1 );
7      Acciones.imprimeIncremento( variable1
   );
8      System.out.println(
          "Variable (después): " + variable1 );
9
10   } // Fin del método main
11 } // Fin de la clase Principal
```

```
Variable (antes):50
Valor: 60
Variable (después): 50
```

Figura 6.20 | Programa para comprobar una variable primitiva como argumento

La clase de la figura 6.19 únicamente define un *método static*, por lo que no es necesario crear una instancia para hacer uso de ese método. En la línea 5,

```
5        public static void imprimeIncremento(
                                    int valor ){
```

se define el *método static* con el nombre de `imprimeIncremento`. Este método define un parámetro de tipo `int` y es nombrado `valor`. En la línea 6,

```
6  valor = valor + 10;
```

se asigna al parámetro `valor` el valor numérico que posee más `10`, es decir, el resultado de `valor + 10`. Posteriormente, en la línea 7,

```
7 | System.out.println( "valor: " + valor );
```

se imprime el valor del parámetro `valor`.

En la clase `Principal` (figura 6.20), se declara una variable de tipo `int` que es utilizada para mostrar su valor **antes** y **después** de invocar al método `imprimeIncremento` de la clase `Acciones`. Analicemos cada línea.

En la línea 5,

```
5 | int variable1 = 50;
```

es definida la variable `variable1` de tipo `int` con el valor asignado de `50`. Posteriormente, en la línea 6,

```
6 | System.out.println(
  |           "variable (antes):" + variable1 );
```

se imprime en pantalla el valor de variable `variable1`. El resultado en pantalla es `variable (antes):50`. El resultado en pantalla intenta describir que se está mostrando el valor de `variable1` antes de realizar la invocación del método `imprimeIncremento` de la línea 7.

```
  | Acciones.imprimeIncremento(
7 |                              variable1 );
```

Para la invocación al método `imprimeIncremento`, se envía como argumento la variable `variable1`. El método `imprimeIncremento` recibe el valor y lo utiliza para sumarle `10` e imprimirlo en pantalla, el resultado es `valor: 60`. El resultado en pantalla corresponde con las acciones

dentro del cuerpo del método `imprimeIncremento` de la clase `Acciones` (figura 6.19).

Finalmente, en la línea 8,

```
8    System.out.println(
     "variable (después): " + variable1 );
```

se imprime el valor de la variable `variable1` después de la ejecución del método `imprimeIncremento` de la línea anterior. El resultado en pantalla es `variable (después): 50`. Se observa que, el valor de la variable variable1 no se ve afectado por el hecho de utilizarlo como argumento durante la invocación del método `imprimeIncremento`, esto ocurre porque el método únicamente recibe una copia del valor que posee variable1.

Ahora que se comprende el uso de los *datos primitivos* como argumentos, se estudiará el uso de *objetos* como argumentos, es decir, tipos de **datos por referencia**.

Persona.java

```
1  package c06.p10;
2
3  public class Persona {
4      public int edad;
5  } // Fin de la clase Persona
```

Figura 6.21 | Clase Persona.

Acciones.java

```
1  package c06.p10;
2
3  public class Acciones {
4
5    public static void imprimeCambioDeEdad(
                              Persona persona ){
```

```
6       persona.edad = persona.edad + 10;
7       System.out.println(
                        "Edad: " + persona.edad );
8     } // Fin del método imprimeCambioDeEdad
9
10  } // Fin de la clase Acciones
```

Figura 6.22 | Clase con un **campo de clase**.

Principal.java

```
1   package c06.p10;
2
3   public class Principal {
4     public static void main(String[] args) {
5       Persona objeto1 = new Persona();
6       objeto1.edad = 15;
7       System.out.println(
              "Edad (antes):" + objeto1.edad );
8       Acciones.imprimeCambioDeEdad( objeto1
    );
9       System.out.println(
            "Edad (después): " + objeto1.edad );
10
11    } // Fin del método main
12  } // Fin de la clase Principal
```

```
Edad (antes):15
Edad: 25
Edad (después): 25
```

Figura 6.23 | Programa para comprobar un objeto como argumento

Este programa está compuesto por 3 clases: la clase **`Persona`**, la clase **`Acciones`** y la clase **`Principal`**. La

clase `principal` es utilizada para iniciar el programa puesto que es la que tiene definido el método `main`.

La clase `Persona` (figura 6.21) es una clase que define un campo `public` de nombre `edad` y de tipo `int`.

La clase `Acciones` (figura 6.22) define un *método static* con nivel de acceso `public` de nombre `imprimeCambioDeEdad`. El método no retorna ningún valor, pero sí define un parámetro de tipo `Persona` el cual es nombrado `persona`. Dentro del cuerpo del método, al campo `edad` del objeto `persona` le es asignado el valor que posee más `10`, es decir, `persona.edad = persona.edad + 10`. Antes de terminar, el método `imprimeCambioDeEdad` muestra en pantalla el texto `"Edad: "` seguido del valor que posee el campo `edad` del objeto `persona`, es decir, `"Edad: " + persona.edad`. En términos generales, este método recibe un objeto de tipo `Persona`, incrementa el valor del campo `edad` en `10` e imprime el nuevo valor del campo `edad`.

La clase Principal (figura 6.23) es la encargada de comprobar lo que sucede con un objeto cuando uno de sus campos es modificado dentro de un método en el que es utilizado como argumento. En la línea 5,

```
5 | Persona objeto1 = new Persona();
```

se crea un objeto de tipo Persona con el nombre de objeto1. En la línea 6,

```
6 | objeto1.edad = 15;
```

es asignado el valor de 15 al campo edad del objeto objet1. En la línea 7,

```
7 | System.out.println(
  | "Edad (antes):" + objeto1.edad );
```

se imprime en pantalla el texto "Edad (antes):" junto con el valor actual del campo edad. El resultado en pantalla es Edad (antes):15.

En la línea 8,

```
8 | Acciones.imprimeCambioDeEdad( objeto1 );
```

se invoca al método imprimeCambioDeEdad y se pasa como argumento el objeto1. Dentro de las acciones del método imprimeCambioDeEdad, se incrementa en 10 el valor del campo edad y se imprime en pantalla. El resultado es Edad: 25.

Finalmente, en la línea 9,

```
9 | System.out.println(
  | "Edad (después): " + objeto1.edad );
```

se imprime el texto "Edad (después): " seguido del valor actual del campo edad del objeto objeto1. El resultado es Edad (después): 25.

Observemos con más detalle el resultado en pantalla de la ejecución del programa.

```
Edad (antes):15
Edad: 25
Edad (después): 25
```

El campo edad del objeto objeto1 tenía el valor de **15** antes de la invocación del método imprimeCambioDeEdad. Durante la ejecución del método imprimeCambioDeEdad, el método recibió el objeto, incrementó en 10 el valor del campo edad y lo mostró en pantalla donde se observa que el valor es de 25. Una vez que el método imprimeCambioDeEdad termina de realizar sus acciones, se imprime nuevamente el valor del campo edad del objeto

`objeto1` y se observa que el cambio que se realizó dentro del método se ve reflejado en el objeto `objeto1`. Entonces es necesario tomar en cuenta que, al utilizar objetos como argumentos, si el método que recibe nuestro objeto realiza cambios en los campos, éstos persistirán cuando el método haya terminado de ejecutar sus acciones tal y como ocurrió en este programa.

6.9Conclusión

En este capítulo se aprendió a trabajar con los niveles de acceso `public` y `private`. En el siguiente capítulo se hará uso del resto de ellos. Así también, en este capítulo se trabajó con los constructores, los parámetros de los constructores y la sobrecarga de éstos. Se estudiaron los campos y métodos `static`. Por último, se revisaron los objetos como argumentos en los métodos y la diferencia al trabajar con tipos de datos primitivos como argumentos. En el siguiente capítulo se aprenderán los fundamentos de la Herencia, la cual conforma parte importante de las características de la POO.

7POO: Herencia

7.1Introducción

La herencia es una característica importante en la POO. Con ésta se crean clases que generalicen las características para que otra clase pueda tomar estas características y agregar las propias.

7.2Herencia

La herencia brinda la posibilidad de tomar una clase y extender las características de ésta, es decir, hacer uso de aquello que haya sido definido en la primera clase, pero agregando nuevos campos o métodos o reescribiendo las acciones de alguno de los métodos de la primera clase. Se realizará un programa que muestre cómo funciona la herencia en la POO.

Persona.java

```
1  package c07.p01;
2
3  public class Persona {
4    private String nombre;
5    private String apellidoPaterno;
6    private String apellidoMaterno;
```

```
7
8     public Persona(){
9       nombre ="";
10      apellidoPaterno = "";
11      apellidoMaterno = "";
12    } // Fin del constructor
13
14    public String getNombre() {
15      return nombre;
16    } // Fin del método getNombre
17
18    public void setNombre(String pNombre) {
19      nombre = pNombre;
20    } // Fin del método setNombre
21
22    public String getApellidoPaterno() {
23      return apellidoPaterno;
24    } // Fin del método getApellidoPaterno
25
26    public void setApellidoPaterno(
                    String pApellidoPaterno) {
27      apellidoPaterno = pApellidoPaterno;
28    } // Fin del método getApellidoPaterno
29
30    public String getApellidoMaterno() {
31      return apellidoMaterno;
32    } // Fin del método getApellidoMaterno
33
34    public void setApellidoMaterno(
                    String pApellidoMaterno) {
35      apellidoMaterno = pApellidoMaterno;
36    } // Fin del método setApellidoMaterno
```

```
37
38    public void imprimirCampos() {
39      System.out.println( "Persona:" +
40              "\n\tnombre = " + nombre +
41              "\n\tapellidoPaterno = " +
                                  apellidoPaterno +
42              "\n\tapellidoMaterno = " +
                                 apellidoMaterno );
43    } // Fin del método imprimirCampos
44
45  } // Fin de la clase Persona
```

Figura 7.1 | Clase Persona

La clase `Persona` de la figura 7.1 no es muy diferente a otras clases utilizadas antes. Esta clase está definida con 3 campos `String` con nivel de acceso `private`, un constructor. Algunos métodos para asignar y recuperar los campos y un método para imprimir los campos de la clase. Realmente no hay mucho que decir de esta clase. Ya se ha trabajado con clases similares. Se crearán instancias de la clase `Persona` en un programa.

PruebaPersona.java

```
1  package c07.p01;
2
3  public class Principal {
4    public static void main(String[] args) {
5      Persona persona1 = new Persona();
6
7      persona1.setNombre("Alberto");
8      persona1.setApellidoPaterno("Arenas");
9      persona1.setApellidoMaterno("Aguirre");
10
11     persona1.imprimirCampos();
```

```
12
13    } // Fin del método main
14  } // Fin de la clase PruebaPersona
```

```
Persona:
  nombre = Alberto
  apellidoPaterno = Arenas
  apellidoMaterno = Aguirre
```

Figura 7.2 | Programa para comprobar la clase Persona

La clase `PruebaPersona` (figura 7.2) crea una instancia de la clase `Persona`. Asigna valores a los campos de la clase a través de algunos métodos, por último, se imprimen en pantalla los valores del objeto `persona1`.

Hasta el momento, las clases `Persona` y `Principal` muestran código y características que ya se han trabajado anteriormente, hasta aquí, nada nuevo. Analícese la situación en la que se solicita trabajar en el programa con una clase **`Cliente`** y esa clase tuviera campos para trabajar el *nombre del cliente*, su *apellido paterno*, su *apellido materno*, y un *número de cliente*.

Cliente.java

```
1   package c07.p01;
2
3   public class Cliente {
4     private String nombre;
5     private String apellidoPaterno;
6     private String apellidoMaterno;
7     private int numeroDeCliente;
8
9     public Cliente(){
10      nombre ="";
11      apellidoPaterno = "";
```

```
12      apellidoMaterno = "";
13      numeroDeCliente = 0;
14    } // Fin del constructor
15
16    public String getNombre() {
17      return nombre;
18    } // Fin del método getNombre
19
20    public void setNombre(String pNombre) {
21      nombre = pNombre;
22    } // Fin del método setNombre
23
24    public String getApellidoPaterno() {
25      return apellidoPaterno;
26    } // Fin del método getApellidoPaterno
27
28    public void setApellidoPaterno(
                  String pApellidoPaterno) {
29      apellidoPaterno = pApellidoPaterno;
30    } // Fin del método getApellidoPaterno
31
32    public String getApellidoMaterno() {
33      return apellidoMaterno;
34    } // Fin del método getApellidoMaterno
35
36    public void setApellidoMaterno(
                  String pApellidoMaterno) {
37      apellidoMaterno = pApellidoMaterno;
38    } // Fin del método setApellidoMaterno
39
40    public int getNumeroDeCliente(){
41      return numeroDeCliente;
```

```
42    } // Fin del método getNumeroDeCliente
43
44    public void setNumeroDeCliente(
      int pNumeroDeCliente){
45      numeroDeCliente = pNumeroDeCliente;
46    } // Fin del método setNumeroDeCliente
47
48    public void imprimirCampos() {
49      System.out.println( "Cliente:" +
50                  "\n\tnombre = " + nombre +
51                  "\n\tapellidoPaterno = " +
                              apellidoPaterno +
52                  "\n\tapellidoMaterno = " +
                              apellidoMaterno +
53                  "\n\tnumeroDeCliente = " +
                            numeroDeCliente );
54    } // Fin del método imprimirCampos
55
56  } // Fin de la clase PruebaCliente
```

Figura 7.3 | Clase Cliente

La clase `Cliente` de la figura 7.3 es muy parecida a la clase `Persona`, sin embargo, hay fragmento de código diferentes que han sido resaltados. En la clase `Cliente` se tienen 4 campos: `nombre`, `apellidoPaterno`, `apellidoMaterno` y `numeroDeCliente`. Existe un constructor que inicializa los campos; se cuenta con métodos para manipular los campos y por último un método que permite imprimir en pantalla los campos de la instancia. Las partes resaltadas muestran las diferencias de la clase `Cliente` con la clase `Persona`. Comprobaremos el funcionamiento de la clase `Cliente` en un programa.

PruebaCliente.java

```
1  package c07.p01;
2
3  public class PruebaCliente {
4    public static void main(String[] args) {
5      Cliente cliente1 = new Cliente();
6
7      cliente1.setNombre("Alberto");
8      cliente1.setApellidoPaterno("Arenas");
9      cliente1.setApellidoMaterno("Aguirre");
10     cliente1.setNumeroDeCliente(1001);
11
12     cliente1.imprimirCampos();
13
14   } // Fin del método main
15 } // Fin de la clase PruebaCliente
```

```
 Cliente:
nombre = Alberto
apellidoPaterno = Arenas
apellidoMaterno = Aguirre
numeroDeCliente = 1001
```

Figura 7.4 | Programa para comprobar la clase Cliente

En la clase `PruebaCliente` se crea una instancia de la clase `Cliente`. Se asignan valores a sus campos a través de la invocación de algunos métodos. Por último, se imprimen los valores de los campos del objeto `cliente1` a través de la invocación del método `imprimirCampos`.

El objetivo de que se crearan las clases `Persona` y `Cliente`, es identificar que existen muchos campos y métodos similares entre ambas clases. Entonces, ¿por qué no aprovechar que la clase `Persona` ya cuenta con muchas de las características de la clase `Cliente`?, es decir, tomar lo

que ya tiene la clase `Persona` y sólo trabajar en las nuevas características de la clase `Cliente`. Ahí es donde entra la herencia.

Persona.java

```
1   package c07.p02;
2
3   public class Persona {
4       private String nombre;
5       private String apellidoPaterno;
6       private String apellidoMaterno;
…       ...
45  } // Fin de la calse Persona
```

Figura 7.5 | Clase Persona

La clase `Persona` de la figura 7.5, con excepción de la línea 1, es similar a la clase `Persona` de la figura 7.1. Para no repetir todo el código, se han puesto puntos suspensivos después de la línea 6, sin embargo, se reitera que el resto del código es similar a la clase `Persona` de la figura 7.1. Lo único que cambia es que esta clase se encuentra en el paquete `c07.p02`, tal como lo indica la línea 1.

Véase ahora cómo aprovecha la clase `Persona` para crear la clase `Cliente`.

Cliente.java

```
1   package c07.p02;
2
3   public class Cliente extends Persona{
4
5     private int numeroDeCliente;
6
7     public Cliente(){
```

```
8       setNombre("");
9       setApellidoPaterno("");
10      setApellidoMaterno("");
11      numeroDeCliente = 0;
12    } // Fin del constructor
13
14    public int getNumeroDeCliente(){
15      return numeroDeCliente;
16    } // Fin del método getNumeroDeCliente
17
18    public void setNumeroDeCliente(
                          int pNumeroDeCliente){
19      numeroDeCliente = pNumeroDeCliente;
20    } // Fin del método setNumeroDeCliente
21
22    public void imprimirCampos() {
23      System.out.println( "Cliente:" +
24              "\n\tnombre = " + getNombre() +
25              "\n\tapellidoPaterno = " +
                          getApellidoPaterno() +
26              "\n\tapellidoMaterno = " +
                          getApellidoMaterno() +
27              "\n\tnumeroDeCliente = " +
                              numeroDeCliente );
28    } // Fin del método imprimirCampos
29
30  } // Fin de la clase Cliente
```

Figura 7.6 | Clase Cliente que extiende a la clase Persona

La clase `Cliente` de la figura 7.6 funciona de la misma forma que la clase `Cliente` de la figura 7.3, pero esta clase tiene 30 líneas, en lugar de las 56 líneas de la otra clase. Antes de analizar las partes de esta clase, se comprobará que funcione correctamente.

PruebaCliente.java

```
1  package c07.p01;
2
3  public class PruebaCliente {
4    public static void main(String[] args) {
5      Cliente cliente1 = new Cliente();
6
7      cliente1.setNombre("Alberto");
8      cliente1.setApellidoPaterno("Arenas");
9      cliente1.setApellidoMaterno("Aguirre");
10     cliente1.setNumeroDeCliente(1001);
11
12     cliente1.imprimirCampos();
13
14   } // Fin del método main
15 } // Fin de la clase PruebaCliente
```

```
Cliente:
  nombre = Carlos
  apellidoPaterno = Cadena
  apellidoMaterno = Castillo
  numeroDeCliente = 2001
```

Figura 7.7 | Programa para comprobar la clase Cliente

Al ver el funcionamiento del programa de la figura 7.7, se observa que se crea un objeto de la clase `Cliente`. Se inicializan los campos a través de algunos métodos de la instancia y se imprimen los valores de los campos en pantalla a través de la invocación del método `imprimirCampos`. Pero, si los métodos `setNombre`, `setApellidoPaterno` y `setApellidoMaterno` no fueron definidos en la clase `Cliente`, ¿de dónde surgen estos métodos? La respuesta es simple, de la clase `Persona`. Como la clase `Cliente` está

extendiendo a la clase `Persona`, la clase `Cliente` hereda todos los métodos de acceso `public`.

En la clase Persona (figura 7.3), en la línea 3,

```
3 | public class Cliente extends Persona{
```

se observa la palabra reservada `extends` seguida del nombre de clase `Persona`, esto indica que la clase `Cliente` va a extender a la clase `Persona` por lo que heredará las características de la clase `Persona`. A la clase que hereda, se le como **subclase** y a la clase de la cual hereda como **superclase**.

Algo a tomar en cuenta al momento de trabajar con la herencia es que, aunque una clase herede las características de otra, esto no significa hacer uso de todos sus campos o métodos. Eso dependerá del nivel de acceso que la **superclase** haya definido.

Como la clase `Cliente` está heredando de la clase `Persona`, en la clase `Cliente` únicamente se definen aquellos campos y métodos extra que necesita. Primeramente, en la línea 5,

```
5 | private int numeroDeCliente;
```

se define un campo de nombre `numeroDeCliente` con el tipo de dato `int`. El resto de los campos necesarios no son definidos porque la clase `Persona` ya los incluye e incorpora los métodos para manipularlos.

De la línea 7 a la 12,

```
7  | public Cliente(){
8  |     setNombre("");
9  |     setApellidoPaterno("");
10 |     setApellidoMaterno("");
11 |     numeroDeCliente = 0;
12 | } // Fin del constructor
```

se define un constructor para inicializar los campos. Observe que el valor para el campo `nombre` es asignado a través del método `setNombre`. Debido a que el campo `nombre` en la clase `Persona` fue definido con nivel de acceso `private`, en la clase `Cliente`, no se accede directamente al campo, pero eso no significa que no se pueda trabajar con él, sí se puede, pero a través de los métodos con nivel de acceso `public` que haya definido la clase `Persona`, en este caso, el método `setNombre`. Lo mismo sucede con los campos `apellidoPaterno` y `apellidoMaterno`, a los que se les asigna valor a través de los métodos `setApellidoPaterno` y `setApellidoMaterno`, los cuales fueron definidos por la clase `Persona` y heredados por la clase `Cliente`. En el caso del campo `numeroDeCliente`, éste sí puede ser accedido directamente, puesto que es definido en la clase `Cliente`, es así como en la línea 11 se asigna el valor de `0` a este campo.

De la línea 14 a la 20,

```
14  public int getNumeroDeCliente(){
15      return numeroDeCliente;
16  } // Fin del método getNumeroDeCliente
17
18  public void setNumeroDeCliente(
    int pNumeroDeCliente){
19      numeroDeCliente = pNumeroDeCliente;
20  } // Fin del método setNumeroDeCliente
```

se definen los métodos `getNumeroDeCliente` y `setNumeroDeCliente` para recuperar y asignar valores al campo `numeroDeCliente`.

Por último, de la línea 22 a la 28,

```
22  public void imprimirCampos() {
23      System.out.println( "Cliente:" +
24              "\n\tnombre = " + getNombre() +
25              "\n\tapellidoPaterno = " +
                        getApellidoPaterno() +
26              "\n\tapellidoMaterno = " +
                        getApellidoMaterno() +
27              "\n\tnumeroDeCliente = " +
                             numeroDeCliente );
28  } // Fin del método imprimirCampos
```

se redefine el método `imprimirCampos`. El método `imprimirCampos` había sido definido por la clase `Persona` y utilizardo desde la clase `Cliente`. El problema está en que el método `imprimirCampos` que se hereda de la clase `Persona` imprime en pantalla un texto diferente al que se desea para la clase `Cliente`. Pero no hay problema porque la herencia no obliga a utilizar los métodos tal y como son heredados. Si las acciones no sirven para los propósitos de la clase, los métodos pueden ser reescritos tal y como se realiza entre las líneas 22 y 28. Otro detalle a observar es que para hacer uso de los campos `nombre`, `apellidoPaterno` y `apellidoMaterno` en las líneas 24, 25 y 26, se usan los métodos con acceso `public`, puesto que los campos fueron definidos con acceso `private` en la clase `Persona`. En el caso del campo `numeroDeCliente`, se accede directamente en la línea 27, puesto que fue definida en la clase `Cliente`.

La **herencia** es una característica muy poderosa, permite reutilizar código y hace más sencillo trabajar con el código. Siempre que se tengan clases con campos o métodos comunes, es bueno analizar qué tan factible es utilizar la herencia con estas clases.

7.3Nivel de acceso protected y default

En el capítulo anterior se trabajó con los niveles de acceso `public` y `private`. Ahora se muestran los niveles de acceso `protected` y *`default`* ***(sin modificador)***.

Modificador	Clase	Paquete	Subclase	Cualquiera
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
Sin modificador (default)	Sí	Sí	No	No
private	Sí	No	No	No

Figura 7.8 | Niveles de acceso

Un elemento al que se le asigna el nivel de acceso `protected` sólo puede ser accedido desde la misma clase, desde una clase del mismo paquete, o desde una subclase. Por otra parte, un elemento al que no se le asigna un modificar de acceso, le es asignado el nivel de acceso `default`, también llamado ***nivel paquete*** y éste sólo puede ser accedido desde la misma clase o desde una clase dentro del mismo paquete.

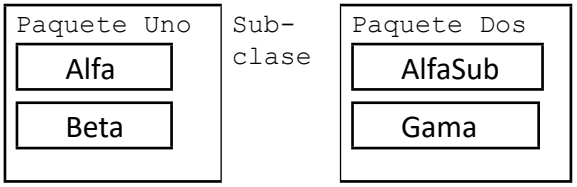

Figura 7.9 | Clases en paquetes diferentes

La figura 7.9 muestra dos paquetes cada uno con dos clases. La clase `Alfa` y `Beta` están dentro del paquete **Uno**, mientras que la clase `AlfaSub` y `Gama` dentro del paquete **Dos**, así también, la clase `AlfaSub` es una subclase de `Alfa`. Si un elemento de la clase `Alfa` tuviera un elemento con el nivel de acceso `protected`, éste podría ser accedido por la clase `Alfa`, la clase `AlfaSub` y por la clase `Beta`. La clase

`Beta` podría acceder al elemento, puesto que está dentro del mismo paquete y la clase `AlfaSub`, aunque no está dentro del mismo paquete, sí está heredando de la clase `Alfa`, por lo que podría acceder al elemento con nivel de acceso `protected`. La única clase que no podría acceder al elemento sería la clase `Gama`. Ahora, si un elemento de la clase `Alfa` tuviera un elemento con el nivel de acceso *`default`*, este elemento podría ser accedido por la clase `Alfa` y por la clase `Beta`. La clase `AlfaSub` y `Gama` no podrían acceder puesto que están en otro paquete.

Elemento en clase Alfa	Clase			
Modificador	Alfa	Beta	AlfaSub	Gama
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
Sin modificador (default)	Sí	Sí	No	No
private	Sí	No	No	No

Figura 7.10 | Visibilidad de los elementos de la clase **Alfa**.

La figura 7.10 resume las clases que tendrían acceso a los miembros de la clase `Alfa` en base al modificador de acceso que se asignara. Por ejemplo, la primera fila muestra que, si la clase `Alfa` tiene un elemento `public`, éste puede ser accedido por todas las clases. Por el contrario, la última fila muestra que, si un elemento de la clase `Alfa` tiene modificador de acceso `private`, sólo puede ser accedido por la clase `Alfa`. En el caso del nivel de acceso `protected`, la clase `AlfaSub` no puede acceder al campo desde una instancia de `Alfa`, pero sí hereda el campo. Esa parte quedará explicada al crear las clases y comprobar los niveles de acceso.

Alfa.java

```
1  package c07.p03.uno;
2
3  public class Alfa {
4
5      public int campoPublic;
6      int campoDefault;
7      protected int campoProtected;
8      private int campoPrivate;
9
10     public Alfa(){
11         campoPublic = 0;
12         campoDefault = 0;
13         campoProtected = 0;
14         campoPrivate = 0;
15     } // Fin del campo Alfa
16
17 } // Fin de la clase Alfa
```

Figura 7.11 | Clase **Alfa** con campo con diferentes niveles de acceso.

La clase `Alfa` de la figura 7.11 se encuentra definida dentro del paquete `c07.p03.uno`. La clase define 4 campos: en la línea 5 un campo con nivel de acceso `public`, en la línea 6 un campo con nivel de acceso `default`, en la línea 7 uno con nivel de acceso `protected` y en la línea 8 uno con nivel de acceso `private`. Dentro del constructor se inicializan los campos, puesto que estamos dentro de la misma clase. Todos los campos pueden ser accedidos de manera directa tal y como ocurre en las líneas 11, 12, 13 y 14. Sin importar el nivel de acceso, todos los campos que defina una clase siempre podrán ser accedidas desde la misma clase.

Beta.java

```
1  package c07.p03.uno;
2
3  public class Beta {
4
5    public Beta(){
6      Alfa alfa = new Alfa();
7
8      alfa.campoPublic = 0;
9      alfa.campoDefault = 0;
10     alfa.campoProtected = 0;
11     // alfa.campoPrivate = 0; /* No se tiene
   acceso */
12   } // Fin del constructor
13
14 } // Fin de la clase Beta
```

Figura 7.12 | Clase **Beta** que crea una instancia de la clase **Alfa** y comprueba el acceso a los campos.

La clase `Beta` de la figura 7.12 es definida dentro del paquete `c07.p03.uno`, el mismo paquete de la clase `Alfa`. La clase `Beta` no define campos, solamente crea una instancia de la clase `Alfa` y accede a los elementes que tiene permitido. En la línea 8 accede al campo `campoPublic`, el cual no tiene problema en acceder puesto que este campo tiene nivel de acceso `public`, por lo que puede ser accedido desde cualquier clase. En la línea 9, se accede al campo `campoDefault` para asignarle el valor de `0`, en este caso, tampoco existe inconveniente. Puesto que el campo tiene nivel `default`, por lo que puede ser accedido desde cualquier clase que pertenezca al mismo paquete y en esta situación, tanto la clase `Alfa` como la clase `Beta`, ambas son definidas dentro del paquete `c07.p03.uno`. Por último, en la línea 11, se muestra una instrucción comentada que muestra que el campo `campoPrivate`

no puede ser accedido, si se llegase a quitar el comentario de línea, la clase lanzaría un error al intentar compilarla, puesto que los elementos con nivel de acceso `private` sólo pueden ser accedidos desde la misma clase en la que son definidos.

AlfaSub.java

```
1  package c07.p03.dos;
2
3  import c07.p03.uno.Alfa;
4
5  public class AlfaSub extends Alfa {
6
7    public AlfaSub(){
8      Alfa alfa = new Alfa();
9
10     alfa.campoPublic = 0;
11     // alfa.campoDefault = 0;   /* No se
   tiene acceso */
12     // alfa.campoProtected = 0; /* No se
   tiene acceso */
13     // alfa.campoPrivate = 0;   /* No se
   tiene acceso */
14
15     campoPublic = 0;
16     // campoDefault = 0; /* No se tiene
   acceso */
17     campoProtected = 0;
18     // campoPrivate = 0; /* No se tiene
   acceso */
19   } // Fin del constructor
20
21 } // Fin de la clase
```

Figura 7.13 | Clase **AlfaSub** que extiende a la clase **Alfa** y comprueba el acceso a los campos.

De las 4 clases que trabajandas (`Alfa`, `Beta`, `AlfaSub` y `Gama`) probablemente la clase `AlfaSub` (figura 7.13) sea la que nos aporte más información para entender los niveles de acceso. Esta clase se encuentra en un paquete diferente a la clase `Alfa` y `Beta`. Ésta se encuentra en el paquete `c07.p03.dos`. La clase `AlfaSub` extiende a la clase `Alfa`, por lo que hereda sus campos, pero no significa que pueda accederlos de forma directa. Así también, la clase `AlfaSub` comprueba dos cosas: a qué campos puede acceder de una instancia `Alfa` y a qué campos heredados, de la clase `Alfa`, puede acceder de forma directa. En las líneas 10, 11, 12 y 13, se muestra que la clase sólo puede acceder al campo `campoPublic`. El resto de los campos no pueden ser accedidos. En las 15, 16, 17 y 18, se muestra el acceso a los campos heredados de la clase `Alfa`. El campo `campoPublic`, en la línea 15, es accedido sin problema, puesto que es `public`. El campo `campoDefault`, en la línea 16, no es accedido, puesto que la clase no se encuentra dentro del mismo paquete que la clase `Alfa`. El campo `campoProtected`, en la línea 17, sí es accedido por el hecho de estar heredando de la clase `Alfa` aún y cuando no estamos dentro del mismo paquete. Por último, el campo `campoPrivate` no es accedido, lo cual en estos momentos, entendemos que sólo podría realizarse dentro de la misma clase en la que es definido.

Gama.java

```
1  package c07.p03.dos;
2
3  import c07.p03.uno.Alfa;
4
5  public class Gama {
6
7    public Gama(){
8      Alfa alfa = new Alfa();
9
10     alfa.campoPublic = 0;
11     // alfa.campoDefault = 0;   /* No se
       tiene acceso */
12     // alfa.campoProtected = 0; /* No se
       tiene acceso */
13     // alfa.campoPrivate = 0;   /* No se
       tiene acceso */
14   } // Fin del constructor
15
16 } // Fin de la clase Gama
```

Figura 7.14 | Clase **Gama** que comprueba el acceso a los campos de la clase **Alfa**.

La clase `Gama` comprueba los campos de una instancia de la clase `Alfa` a los que puede acceder. En la línea 10, se observa que el único campo al que puede acceder la clase `Gama` del objeto `alfa` es al campo `campoPublic`. Cualquier intento por acceder a los campos `campoDefault`, `campoProtected` o `campoPrivate` lanzaría un error al intentar compilar la clase.

Como se observa en las clases Alfa, Beta, AlfaSub y Gama, los modificadores de acceso no permiten controlar la visibilidad de los campos en las distintas clases. Una de las recomen-

daciones al trabajar con clases, es brindar el mínimo privilegio de acceso necesario a los miembros de la clase.

7.4Conclusión

En este capítulo se aprendió sobre la herencia, muy importante en la creación de nuestros programas, puesto que permite crear clases que generalicen las características comunes de otras, de este modo, es posible tomar una clase con características generales y especializarla con características específicas en una nueva clase, sin necesidad de reescribir el código. Así también se profundizó con mayor detalle en la comprensión de los modificadores de acceso con los que se controla la visibilidad de los miembros de una clase.

Referencias

Asignación, aritmética y operadores unarios (Tutoriales de Java ™> Aprendizaje del lenguaje Java> Conceptos básicos del lenguaje) . Docs.oracle.com. (2006) Consultado el 14 de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html.

Caracteres (Los Tutoriales Java ™> Aprendiendo el Lenguaje Java> Números y Cadenas). Docs.oracle.com. (2006) Consultado el 14 de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/data/characters.html.

Ceballos Sierra, F. J. (2015). *Java 2: lenguaje y aplicaciones.* RA-MA Editorial.

Codificaciones de caracteres: conceptos esenciales . W3.org. (2018) Consultado el 14 de julio de 2020, en https://www.w3.org/International/articles/definitions-characters/index.en.

Deitel, P. y Deitel, H. (2008). *Cómo Programar en Java.* Pearson Educación.

Comprensión de los miembros de la clase (Los Tutoriales Java ™> Aprendiendo el lenguaje Java> Clases y objetos) . Docs.oracle.com. (2006) Consultado el 14 de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html.

Control del acceso a los miembros de una clase (Tutoriales de Java ™> Aprendizaje del lenguaje Java> Clases y objetos) . Docs.oracle.com. (2010) Consultado el 14

de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Convenciones de código para el lenguaje de programación Java: 9. Convenciones de nomenclatura. Oracle.com. (2010) Consultado el 14 de julio de 2020, en https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html.

Creando Objetos (Los Tutoriales Java ™> Aprendiendo el Lenguaje Java> Clases y Objetos). Docs.oracle.com. (2006) Consultado el 14 de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html.

Guía de instalación. Centro de ayuda de Oracle. (2018) Consultado el 14 de julio de 2020, en https://docs.oracle.com/en/java/javase/11/install/installation-jdk-microsoft-windows-platforms.html.

Las declaraciones if-then y if-then-else (The Java ™ Tutorials> Aprendiendo el lenguaje Java> Conceptos básicos del lenguaje) . Docs.oracle.com. (2006) Consultado el 14 de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html.

Matrices (Tutoriales de Java ™> Aprendizaje del lenguaje Java> Conceptos básicos del lenguaje) . Docs.oracle.com. (2006) Consultado el 14 de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html.

Operadores de igualdad, relacionales y condicionales (Los Tutoriales de Java ™> Aprendizaje del lenguaje Java> Conceptos básicos del lenguaje) . Docs.oracle.com. (2006) Consultado el 14 de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html.

Sánchez Allende, J., Fernández Manjón, B., Moreno Díaz, P., Sánchez, C., y Hueca Fernández Toribio, G. (2009). *Programación en JAVA (3a. ed.).* McGraw-Hill España.

Schildt, H. y Rojas, E. (2010). *Fundamentos de Java*. McGraw-Hill Interamericana.

Santini, S. (2011). *A discipline of java programming.* Editorial Universidad Autónoma de Madrid.

Tipos de datos primitivos (Tutoriales de Java ™> Aprendizaje del lenguaje Java> Conceptos básicos del lenguaje) . Docs.oracle.com. (2006) Consultado el 14 de julio de 2020, en https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html.

Zohonero Martínez, I. y Joyanes Aguilar, L. (2008). *Estructuras de datos en Java.* McGraw-Hill España.

Zukowski, J. (2007). *Usando For-Loops mejorados con tus clases* . blogs.oracle.com. Consultado el 14 de julio de 2020, en https://blogs.oracle.com/corejavatechtips/using-enhanced-for-loops-with-your-classes.

Zohonero Martínez, I. y Joyanes Aguilar, L. (2008). *Estructuras de datos en Java.* McGraw-Hill España

Este libro surge por el contacto del día a día en las aulas con los estudiantes a través de impartir las clases de Programación e Introducción a la Programación, con el fin de apoyar y fortalecer competencias en toda persona que inicie y le interese el aprendizaje en el desarrollo de programas de computadora en el lenguaje JAVA.

En este libro se muestra paso a paso cómo introducirse a la programación en Java desde la instalación del mismo *software*, conocer y manipular las diversas estructuras de control: If, While, Switch, Do While, For, hasta temas de mayor complejidad como es el uso de arreglos y los elementos básicos de la Programación Orientada a Objetos (POO), como la instanciación, constructores, herencia, otros.

9 788413 740126

Círculo Rojo
EDITORIAL

FUNDAMENTOS DE LA PROGRAMACIÓN EN JAVA

Javier Pino, Patricia Martínez, José Antonio Vergara