Curso Java

Formación Sopra Steria

Formación JAVA

Bloque Java Básico

Versión 1.2.3 del miércoles, 18 de junio de 2018

Destinatario(s)

Historial

Versión	Fecha	Origen de la actualización	Redactado por	Validado por
1.0	26/04/2018		Sergio Comes Marí Silvia Carpena Tobarra María Pechero Arroyo	
1.1	30/04/2018		Silvia Carpena Tobarra Sergio Comes Marí	
1.1.1	21/05/2018		Silvia Carpena Tobarra	
1.1.2	25/05/2018		Silvia Carpena Tobarra	
1.2	28/05/2018	Expresiones lambda	Silvia Carpena Tobarra Claudio Jesús Fernández Antón	
1.2.1	30/05/2018		Silvia Carpena Tobarra	
1.2.2	07/06/2018		Silvia Carpena Tobarra	
1.2.3	18/06/2018		Silvia Carpena Tobarra	

Índice

1. Java

1.1. Historia de Java

Para hablar de la historia de java, hay que remontarse a los años 80, donde C podía considerarse el lenguaje por antonomasia. Era un lenguaje versátil, que podía actuar a bajo nivel y resolvían problemas muy complejos. Era la cima de la programación estructurada, para resolver estos complejos algoritmos, se generaban grandes procedimientos con un código muy complicado de mantener a largo plazo. Por ello empezó a surgir como alternativa la programación orientada a objetos, y con ella nació C++.

El objetivo de java era crear un lenguaje de programación parecido a C++ en estructura y sintaxis, fuertemente orientado a objetos, pero con una máquina virtual propia. Esto se hizo bajo el principio, de poder ser usado bajo cualquier arquitectura "Write Once, Run Anywhere (escríbelo una vez, ejecútalo en cualquier sitio)".

En 1992 se presenta el proyecto, con los prototipos a bajo nivel. Entre 1993 y 1994 se trabaja para poder presentar un prototipo funcional (hotJava) donde se ve todo el potencial que JAVA puede ofrecer

En 1995 finalmente, es presentada la versión alpha de java, y un año después en 1996 es lanzado el primer JDK (JDK 1.0): software que provee herramientas de desarrollo para la creación de programas en java. El desarrollo de java a partir de entonces es imparable, se van presentando nuevos paquetes y librerías hasta la actualidad.

A día de hoy, se puede decir, que Java es uno de los lenguajes más importantes del mundo. Con una comunidad extendida en todos los componentes y más de 4 millones de desarrolladores, existen millones de dispositivos que lo usan. Además, tras el surgimiento de android, java es establecido como el lenguaje de programación para móviles más extendido del planeta.

En las primeras versiones de Java 1.1, 1.2 y 1.3 es donde el lenguaje va tomando forma, con la inclusión de tecnologías como JavaBeans, JDBC para el acceso a base de datos, RMI para las invocaciones en remoto, Collections para la gestión de múltiples estructuras de datos o AWT para el desarrollo gráfico, entre otros.

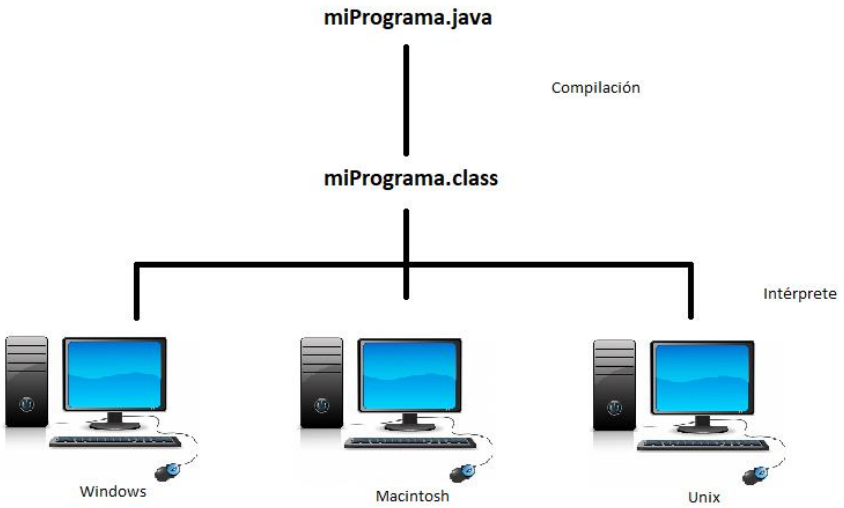
Una de las cosas que sucede en noviembre 2006 es que Sun Microsystems lo convierte en Open Source mediante una licencia GNU General Public License (GPL). Dando lugar en mayo de 2008 a lo que se conoce como OpenJDK, con OpenJDK 6.

Llegado julio de 2011 ve la luz Java 7, la cual trae como novedades el soporte de lenguajes dinámicos, dotando a la JVM de un soporte de mútiples lenguajes y una nueva librería I/O para el manejo de ficheros. También aparecen cosas menores, pero muy útiles como el manejo de String, dentro de la validación en una estructura switch, o la capacidad de poner guiones de subrayado como separadores de miles en los números para facilitar su lectura (p. ej. 1_000_000).

La última versión de Java distribuida es Java 8, aparecida en marzo de 2014. Entre las características de Java 8 se tiene el soporte para expresiones Lambda y uso de Streams, que permiten un estilo más similar a la programación funcional para los programas Java. Dentro de este enfoque más cercano a la programación funcional, también aparecen las transformaciones MapReduce. Ve la luz el Proyecto Nashorn para disponer de un engine Javascript y así poder incluir este lenguaje dentro de las aplicaciones Java. Otras cosas son un nuevo API Date y Time y la inclusión de JavaFX 8 dentro de la JDK de Java.

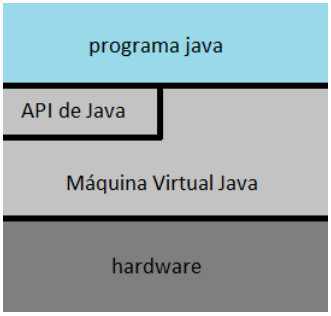
1.2. La Máquina Virtual Java

El lenguaje Java es a la vez compilado e interpretado. Con el compilador, se convierte el código fuente que reside en archivos cuya extensión es **.java,** a un conjunto de instrucciones que recibe el nombre de *bytecodes*, que se guardan en un archivo cuya extensión es **.class**. Estas instrucciones son independientes del tipo de ordenador. El intérprete ejecuta cada una de estas instrucciones en un ordenador específico (Windows, Macintosh, etc). Solamente es necesario, por tanto, compilar una vez el programa, pero se interpreta cada vez que se ejecuta en un ordenador.



Cada intérprete Java es una implementación de la Máquina Virtual Java (JVM). Los *bytecodes* posibilitan el objetivo de "write once, run anywhere", de escribir el programa una vez y que se pueda ejecutar en cualquier plataforma que disponga de una implementación de la JVM. Por ejemplo, el mismo programa Java puede ejecutarse en Windows 98, Solaris, Macintosh, etc.

Java es, por tanto, algo más que un lenguaje, ya que la palabra Java se refiere a dos cosas inseparables: el lenguaje que nos sirve para crear programas y la Máquina Virtual Java que sirve para ejecutarlos. Como se puede ver en la figura, el API de Java y la Máquina Virtual Java forman una capa intermedia (Java platform) que aísla el programa Java de las especificidades del hardware (hardware-based platform).



La Máquina Virtual Java (JVM) es el entorno en el que se ejecutan los programas Java. Su misión principal es la de garantizar la portabilidad de las aplicaciones Java. Define esencialmente un ordenador abstracto y especifica las instrucciones (bytecodes) que este ordenador puede ejecutar. El intérprete Java específico ejecuta las instrucciones que se guardan en los archivos cuya extensión es .class. Las tareas principales de la JVM son las siguientes:

- Reservar espacio en memoria para los objetos creados
- Liberar la memoria no usada (garbage collection)
- Asignar variables a registros y pilas
- Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos
- Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java

Esta última tarea, es una de las más importantes que realiza la JVM. Además, las propias especificaciones del lenguaje Java contribuyen extraordinariamente a este objetivo:

- Las referencias a arrays son verificadas en el momento de la ejecución del programa
- No hay manera de manipular de forma directa los punteros
- La JVM gestiona automáticamente el uso de la memoria, de modo que no queden huecos.
- No se permiten realizar ciertas conversiones (casting) entre distintos tipos de datos.

1.3. El lenguaje Java

Java es un lenguaje de propósito general, de alto nivel, y orientado a objetos.

Java es un lenguaje de programación orientado a objetos puro, en el sentido de que no hay ninguna variable, función o constante que no esté dentro de una clase. Se accede a los miembros dato y las funciones miembro a través de los objetos y de las clases. Por razones de eficiencia, se han conservado los tipos básicos de datos, int, float, double, char, etc, similares a los del lenguaje C/C++.

La API de Java es muy rica, está formada por un conjunto de paquetes de clases que le proporcionan una gran funcionalidad. El núcleo de la API viene con cada una de las implementaciones de la JVM:

- Lo esencial: tipos de datos, clases y objetos, arrays, cadenas de caracteres (strings), subprocesos (threads), entrada/salida (I/O), propiedades del sistema, etc.
- Applets
- Manejo de la red (networking)
- Internacionalización
- Seguridad
- Componentes (JavaBeans)
- Persistencia (Object serialization)
- Conexión a bases de datos (JDBC)

Java proporciona también extensiones, por ejemplo, define un API para 3D, para los servidores, telefonía, reconocimiento de voz, etc.

2. Entorno de desarrollo

2.1. Instalación del JDK (Java Development Kit)

En primer lugar se deberá comprobar si el JDK ya está instalado en el equipo. Para ello, se deberá escribir lo siguiente en la línea de comandos:

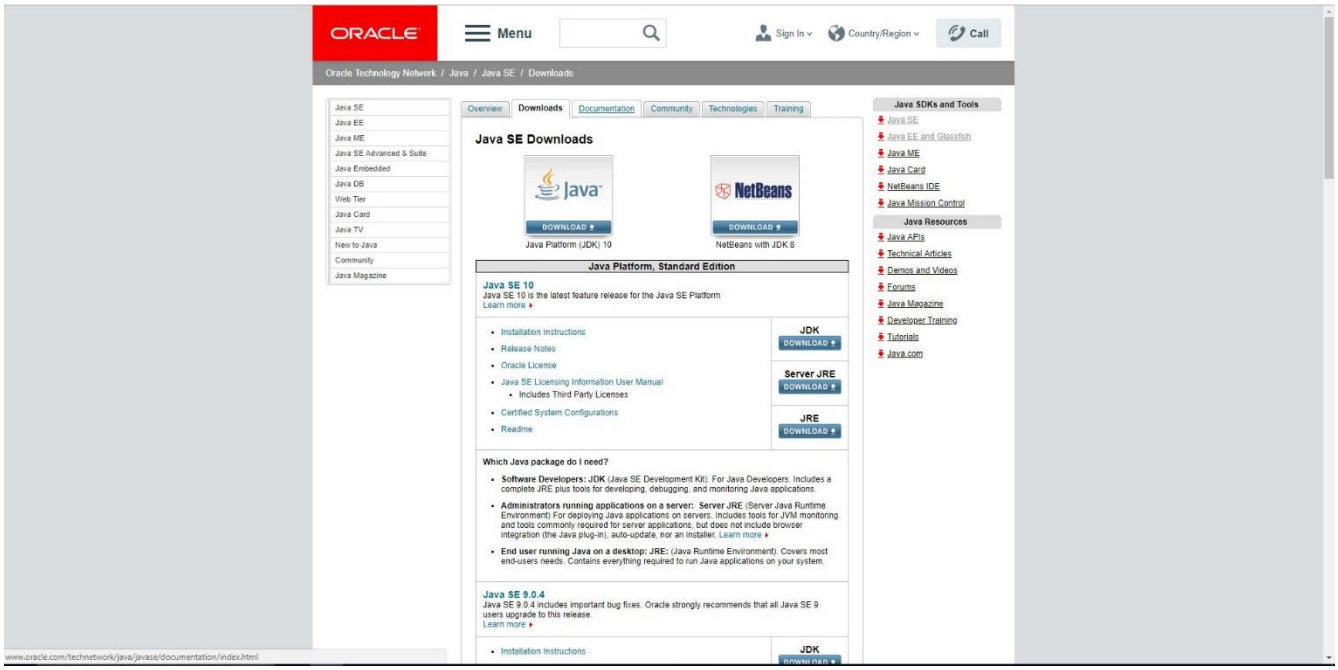
```
java -version
```

En caso de tener el JDK instalado, se podrá ver como resultado del comando algo similar a lo que se muestra en la siguiente imagen y se podrá continuar con la instalación del entorno de desarrollo Eclipse.

```
C:\>java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
```

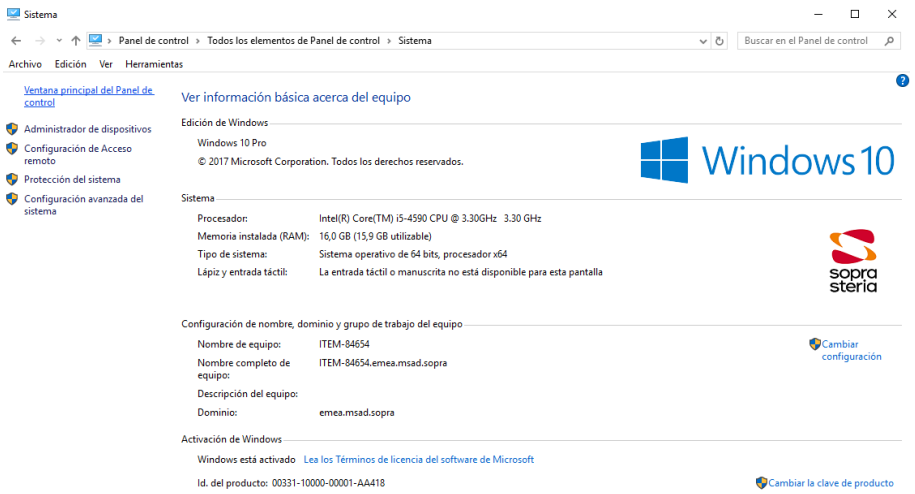
En caso de no tener instalado el JDK, se deberán seguir los siguientes pasos:

1. Se deberá descargar el JDK desde la siguiente URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html

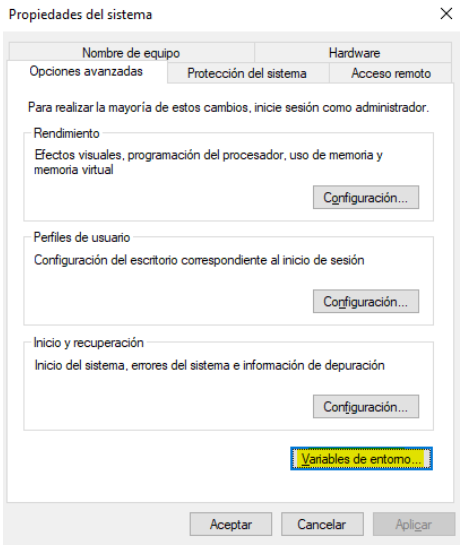


2. Se añadirá el ejecutable de Java que se acaba de descargar al PATH del sistema, para ello:

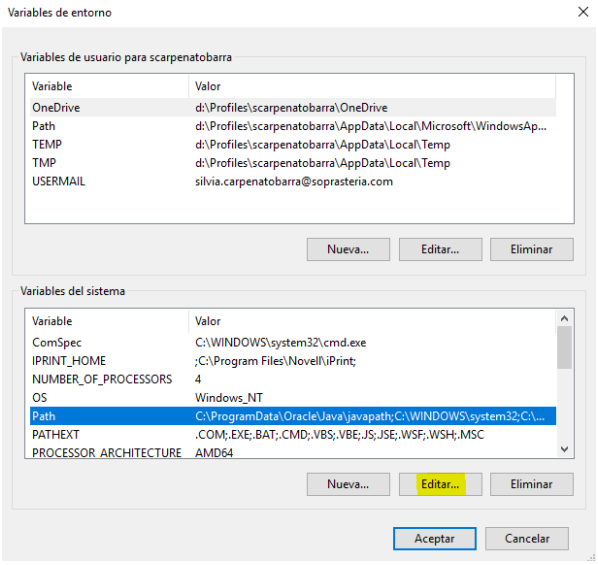
- Se accede a la siguiente ubicación del pc "Panel de control\Todos los elementos de Panel de control\Sistema".



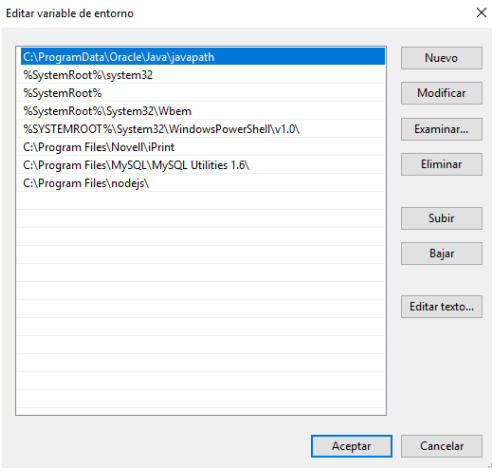
- A continuación, se accede a “Configuración avanzada del sistema”, “Variables de entorno”.



- Es posible que se haya añadido el JDK de manera automática al equipo, por lo que se deberá comprobar haciendo clic sobre "Path" (que está seleccionado en la imagen), y después sobre "Editar..."

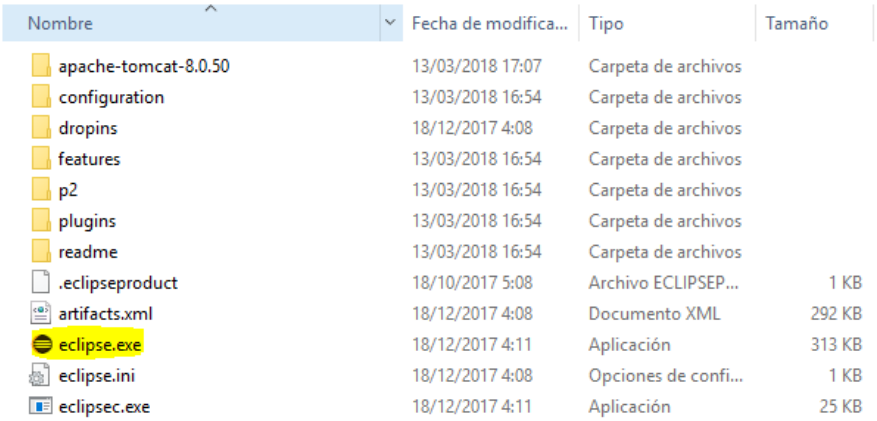


- En la imagen se puede ver que Java está instalado en el sistema, concretamente en "C:\ProgramData\Oracle\Java\javapath". De no ser así, habrá que añadirlo haciendo clic en "Nuevo".



2.2. Instalación del entorno de desarrollo Eclipse

1. Se accede a la siguiente URL y se descarga la versión de Eclipse con la que se vaya a trabajar: https://eclipse.org/downloads/packages/technologyeppdownloadsreleaseoxygenm2eclipse-jee-oxygen-m2-win32-x8664zip
2. Se descomprime el .zip en el disco D: .
3. Para ejecutar Eclipse hay que hacer doble click en el archivo eclipse.exe que se encuentra dentro de la carpeta. Se puede crear un acceso directo al .exe de la aplicación y guardarlo en el escritorio para mayor comodidad.



3. Bloques básicos en Java

3.1. Comprender la estructura de clases de Java

Las clases son los componentes básicos en un programa Java. Cuando se define ***una clase***, se describen todas las partes y características que la componen. Para utilizar la mayoría de las clases, se tienen que crear objetos. Un ***objeto*** es una instancia de la clase en tiempo de ejecución. Los objetos de las distintas clases representan el estado del programa. En las siguientes secciones se verán los componentes de una clase Java, así como la relación entre clases y archivos.

3.1.1. Propiedades y métodos

Las clases Java tienen dos elementos principales, ***métodos***, a menudo llamados funciones o procedimientos, y ***propiedades***, también conocidas como atributos. Las propiedades mantienen el estado del programa, y los métodos operan en ese estado. Si el cambio es relevante en el estado del programa, se almacenará en una propiedad. Es el programador quien crea y organiza estos elementos de tal manera que el código sea útil y fácil de entender para otros programadores.

La clase Java más simple que puede escribirse es como la que sigue:

(Los números de línea solo están ahí para hacer la lectura del código más fácil, no intervienen en el programa).

```
1: public class Fruit {
2: }
```

En Java existen palabras reservadas, también llamadas palabras clave o ***keywords***. Estas palabras poseen un significado especial para Java, como, por ejemplo, la palabra reservada *public* (línea 1), que permite a la clase Fruit ser utilizada por otras clases. La palabra reservada *class* indica que se está definiendo una clase. Fruit es el nombre de la clase. A continuación, se añadirá a la clase Fruit su primera propiedad:

```
1: public class Fruit {
2:   String name;
3: }
```

En la línea 2, se define la propiedad llamada *name*. También se define el tipo de dato de la propiedad como una cadena de texto (*String*). Un *String* es un tipo de dato que contiene un texto, como por ejemplo "Esto es un *String*". *String* también es una clase Java, predefinida en el propio lenguaje. Como se mencionó anteriormente, una clase también puede contener métodos, los cuales se declaran como se ve en el siguiente código:

```
1: public class Fruit {
2:   String name;
```

```
3:   public String getName() {
4:     return name;
5:   }
6:   public void setName(String newName) {
7:     name = newName;
8:   }
9: }
```

En las líneas 3-5 se ha declarado el primer método. Un **método** es una operación que puede ser llamada. En la declaración del método se hace también uso de la palabra reservada *public* para indicar que el método puede ser llamado desde cualquier otra clase. La siguiente palabra, *String* en este caso, hace referencia al tipo de dato que devuelve el método. En este caso, el tipo de dato devuelto es un *String*. En las líneas 6-8 se ha introducido un segundo método, el cual no devuelve nada (palabra reservada *void*). Este método requiere que se le proporcione información cada vez que se le llama; esta información se llama **parámetro**. El método *setName* tiene un parámetro llamado *newName*, y es de tipo *String*. La persona que llama al método le debe pasar, por lo tanto, un parámetro de tipo *String*, y, tras realizar la operación de asignar el valor del parámetro *newName* a la propiedad *name*, puesto que es *void*, el método *setName* no devolverá nada.

3.1.2. Comentarios

Los **comentarios** son muy habituales en el código de un programa Java. Son líneas de texto que no son ejecutadas, y que pueden ser colocadas en cualquier lugar. Los comentarios tienen como finalidad hacer la lectura del código más sencilla. Hay tres tipos de comentarios en Java.

1- El comentario de una sola línea, comienza con dos barras diagonales.

```
// comment until end of line
```

2- El comentario de varias líneas o comentario de líneas múltiples, comienza con el símbolo / * hasta el símbolo * /. Se suele escribir * al comienzo de cada línea de un comentario de líneas múltiples para que sea más fácil de leer.

```
/* Multiple
 * line comment
 */
```

3- El comentario de Javadoc, muy similar a un comentario de líneas múltiples. Los comentarios de Javadoc tienen una estructura específica que la herramienta Javadoc sabe cómo interpretar para poder crear la documentación del código. La diferencia con los comentarios de líneas múltiples es que el comienzo debe ser /**

```
/**
* Javadoc multiple-line comment
*/
```

3.1.3. Javadoc

Javadoc es una utilidad de Oracle que sirve para generar documentación de APIs en formato HTML a partir del código. Al documentar una clase, se debe incluir lo siguiente:

- Descripción general de la clase, número de versión, nombre de autores.
- Documentación de métodos: incluye descripción general, nombre y tipos de parámetros, descripción de parámetros, tipo de retorno, descripción del valor que devuelve.

Para que javadoc pueda generar la documentación automáticamente, la información debe incluirse entre símbolos de comentario de forma que empiece con una barra simple y doble asterisco (/**), y termine con un asterisco y barra simple (*/). Dependiendo de la ubicación, el comentario puede representar una cosa distinta, si está incluido delante de una clase, es un comentario de clase pero, si está incluido delante de un método, es un comentario de ese método.

Para crear la documentación de javadoc, se hace uso de palabras reservadas o "tags" que van precedidas por el carácter "@". En la siguiente tabla se puede encontrar las palabras reservadas que suelen utilizarse más a menudo.

TAG	DESCRIPCIÓN	COMPRENDE
@author	Nombre del desarrollador.	Nombre autor o autores
@deprecated	Indica que el método o clase es obsoleto (propio de versiones anteriores) y que no se recomienda su uso.	Descripción
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.	Nombre de parámetro y descripción
@return	Informa de lo que devuelve el método, no se aplica en constructores o métodos "void".	Descripción del valor de retorno
@see	Asocia con otro método o clase.	Referencia cruzada

		referencia (#método(); clase#método(); paquete.clase; paquete.clase#método()).
@version	Versión del método o clase.	Versión

A continuación, se verá un ejemplo de documentación javadoc para una clase y un método.

```
/**
 * Clase que implementa un ejemplo de una suma a partir de dos números dados
 * y devuelve el resultado por pantalla.
 *
 * @author Sergio
 * @version 2.0
 * @since 2018
 */
class EjemploSuma {
   public static void main(String args[]) {
      int num1 = 10;
      int num2 = 5;
      int result = getSuma(num1,num2);
      System.out.println("La suma es igual a "+result);
   }

   /**
        * Devuelve la suma de dos números enteros (int)
        *
        * @param a primer sumando
        * @param b segundo sumando
        * @return resultado de la suma
        */
   public static int getSuma(int a, int b) {
           return a+b;
   }
}
```

3.1.4. Clases vs. Archivos

Normalmente, cada clase de Java se define en su propio archivo * .java. Por lo general las clases son *public*, lo que significa que cualquier código puede utilizarlas. Las clases; no obstante, no tienen por qué ser públicas.

Por ejemplo, definir una clase como sigue (sin la palabra reservada *public*), también es posible.

```
1: class Fruit {
2:   String name;
3: }
```

También es posible que dos clases Java estén en el mismo archivo. En este caso, como mucho una de las clases del archivo puede ser pública.

Por ejemplo:

```
1: public class Fruit {
2:   private String name;
3: }
4: class FruitSqueezer {
5: }
```

En caso de tener una clase pública en el archivo, el nombre de la clase debe coincidir con el nombre del archivo; es decir, *public class FruitSqueezer* provocaría un error de compilación en un archivo llamado *Fruit.java*.

3.2. El método main()

Un programa Java comienza su ejecución en su método *main()*. El método *main()* es lo primero que se ejecuta cuando se inicia el programa, y es llamado automáticamente por la Máquina Virtual Java (JVM). La JVM es la encargada de comunicarse con el sistema subyacente para asignar recursos como la memoria, la CPU, el acceso a archivos, etc.

El método *main()* nos sirve como enlace para conectar las distintas partes del código a partir del inicio de la ejecución del programa.

Un ejemplo del tipo de clase más simple que puede contener el método *main()* sería como la siguiente:

```
1: public class Fruit {
2:   public static void main(String[] args) {
3:
4:   }
5:}
```

Este código no hace nada útil. No tiene ninguna instrucción más allá que declarar el punto de inicio de la ejecución del programa. Lo que se pretende con este código de ejemplo es mostrar que el método main() se puede declarar en cualquier parte del código Java. De hecho, la única razón de que este método esté dentro de una clase se debe a que el lenguaje así lo requiere.

Para compilar (**javac**) y ejecutar (**java**) el siguiente código, este debería escribirse dentro del archivo Fruit.java, y posteriormente se deberían escribir las siguientes instrucciones en la consola:

```
$ javac Fruit.java
$ java Fruit
```

En caso de no obtener ningún mensaje de error, el programa se ejecutó con éxito. En caso de obtener un mensaje de error, se debería verificar que el *Java Development Kit* (JDK) está correctamente instalado y que no existe ningún error sintáctico en la escritura del código.

Para compilar código Java, el archivo debe tener la extensión .java. El nombre del archivo debe coincidir con el nombre de la clase definida en el archivo. El resultado es otro archivo de *bytecode* con el mismo nombre, pero con una extensión .class. El *bytecode* no es código legible por las personas, pero sí por la JVM.

Las reglas sobre cómo debe ser el contenido de un archivo de código Java son más extensas de lo que se ha visto hasta ahora; no obstante, por el momento y con la finalidad de hacerlo más simple, solo se tendrán en cuenta las siguientes:

1- Cada archivo puede contener solo una clase (ya se ha visto que no tiene por qué ser así, pero de esta forma el código será más sencillo).

2- El nombre del archivo debe coincidir con el nombre de la clase que contiene, y la extensión de este siempre será .java.

Se supone que se reemplaza la línea 3 en Fruit.java con System.out.println ("Hola"). Cuando se compila y se ejecuta nuevamente el código, el programa imprimirá en la consola la palabra "Hola".

A continuación se va a revisar en profundidad el método *main()*, ya que aparecen en él una serie de palabras clave del lenguaje Java, algunas ya vistas anteriormente.

La palabra reservada *public* es lo que se llama un **modificador de acceso**. Declara el nivel de exposición de este método a la hora de ser llamado en otras partes del programa. En este caso, *public* hace referencia a que el método puede ser llamado en cualquier parte del programa.

La palabra reservada *static* enlaza un método a su clase para que pueda ser llamado solo por el nombre de la clase, como, por ejemplo, *Fruit.main()*. No es necesario crear un objeto para llamar al método *main()*. Si el método *main()* no está presente en la clase .java que se ejecuta en la consola, el proceso generará un error y terminará. Incluso si el método *main()* está presente y no es estático (palabra reservada *static*), Java lanzará una excepción.

La palabra reservada *void* representa el tipo de dato que devuelve el método. Un método que no devuelve datos (es decir, que es *void*) devuelve el control de la ejecución del programa al punto donde quedó antes de ser ejecutado de manera "silenciosa". En general, es una buena práctica usar *void* para métodos que cambian el estado de un objeto. En ese caso, el método *main()* cambia el estado del programa de iniciado a finalizado.

Finalmente se llega a la lista de parámetros del método *main()*, representada como un *array* de objetos de tipo *java.lang.String*. El compilador acepta cualquiera de las siguientes formas a la hora de declarar lo parámetros: *String[] args*, *String args[]* o *String...args*. La variable *args* contiene los argumentos que se leyeron de la consola cuando se inició la JVM. Los caracteres [] son corchetes, utilizados para representar que *args* es un *array*. Un *array*es una lista de tamaño fijo que contiene elementos de un mismo tipo de dato. Los caracteres ... se denominan *varargs* (lista de argumentos variable).

En el siguiente ejemplo se ve cómo usar el parámetro *args*. Primero se modifica el programa anterior para imprimir los primeros dos argumentos que sean pasados al programa:

```
public class Fruit {
    public static void main(String[] args) {
        System.out.println(args[0]);
        System.out.println(args[1]);
} }
```

args[0] accede al primer elemento del *array*. Los índices de los *array* en Java siempre comienzan por 0. Para ejecutar el código anterior, se podría escribir en la consola:

```
$ javac Fruit.java
$ java Fruit Banana Apple
```

Y la consola imprimiría:

```
Banana
Apple
```

El programa identifica correctamente las dos primeras palabras como argumentos separados por espacios. En caso de querer espacios dentro de un argumento, se usan comillas tal que así:

```
$ javac Fruit.java
$ java Fruit "Big Banana" Apple
```

Y la consola imprimiría:

```
Big Banana
Apple
```

Todos los argumentos de la línea de comandos se tratan como objetos de tipo *String*, incluso si representan otro tipo de datos:

```
$ javac Fruit.java
$ java Fruit Fruit 1
```

A pesar de que Fruit es una clase y 1 es un número, el programa los interpretará a ambos como la cadena de texto “Fruit” y la cadena de texto “1”, y por lo tanto, la consola imprimirá lo siguiente:

```
Fruit
1
```

Finalmente, ¿Qué sucede si no se escriben suficientes argumentos?

```
$ javac Fruit.java
$ java Fruit Fruit
```

La lectura de args[0] es correcta y "Fruit" se imprime por pantalla, pero al no tener otro argumento, Java imprime una excepción mostrando que no existe ningún argumento en args[1].

La consola imprimirá por lo tanto:

```
      FruitException in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1
      at mainmethod.Fruit.main (Fruit.java:7)
```

Es necesario tener un JDK (Java Development Kit) para compilar código Java; no obstante, el JDK no es necesario para ejecutar el código, basta con un JRE (Java Runtime Environment). Los archivos de clase Java que se ejecutan en la JVM (Java Virtual Machine) se ejecutan en cualquier máquina con Java independientemente de la máquina original donde fueron compilados o el sistema operativo de esta.

3.3. Paquetes e importaciones

Java tiene miles de clases por defecto y muchas otras creadas por otros desarrolladores, por lo que es necesario una forma de organizarlas.

En Java existen paquetes (*packages*), los cuales sirven para organizar las clases de manera lógica. Es necesario especificar en qué paquete se encuentra cada clase para que Java pueda utilizar la clase correcta.

Se supone que el siguiente código está por compilar:

```
public class ImportExample {
    public static void main(String[] args) {
        Random r = new Random(); // ¡No compila!
        System.out.println(r.nextInt(10));
    }
}
```

El compilador de Java mostrará un error como el siguiente:

```
Random cannot be resolved to a type
```

Este error podría significar un error tipográfico en el nombre de la clase, o puede que se haya omitido un *import* necesario (la importación de la clase Random). Las sentencias de importación (*import*) indican a Java en qué paquetes buscar las clases.

Intentarlo de nuevo con la importación de la clase Random hace que el programa compile:

```
import java.util.Random; // Indica a la clase dónde encontrar la clase Random

                         //(Paquete java.util)
public class ImportExample {
    public static void main(String[] args) {
       Random r = new Random();
       System.out.println(r.nextInt(10)); //Imprime por pantalla un número

                                          // del 0 al 9
       }
}
```

Ahora el código se ejecuta e imprime un número aleatorio entre 0 y 9. Al igual que ocurre con los *arrays*, la cuenta empieza en 0. Las clases de Java se agrupan en paquetes (*packages*). La sentencia *import* le indica al compilador en qué paquete buscar para encontrar una clase.

A modo de analogía, si se imagina una carta cuyo destinatario es la dirección C/San Andrés nº20 1ºC. El cartero irá en primer lugar al edificio número 20 de la calle San Andrés, y luego buscará el piso 1ºC. La dirección (calle y número de edificio) serían los paquetes Java, y 1ºC sería el nombre de la clase Java, la cual se ha podido encontrar porque se sabía dentro de qué paquetes estaba. Hay muchos pisos 1ºC, pero solo hay un 1ºC en el edificio nº20 de la calle San Andrés.

Los nombres de los paquetes son jerárquicos, al igual que ocurre con el correo. El servicio de correo mirará en primer lugar el país, luego la región, la ciudad, la calle, el edificio y por último el piso para poder entregar una carta. Con los paquetes Java ocurre algo muy similar. Si el paquete comienza por *java* o *javax* significa que es un paquete por defecto del JDK; si empieza por cualquier otra cosa, hará referencia al dominio del propietario del paquete. Por ejemplo, el paquete com.amazon.laClaseQueSea indica que el código viene de amazon.com. Tras el nombre del dominio los nombres del resto de la jerarquía de paquetes y clases pueden ser los que el programador guste, aunque es una buena práctica que estos sean lo más descriptivos posible.

En los ejemplos que se muestran a continuación es posible encontrar paquetes que no provengan de ningún dominio concreto, como, por ejemplo a.b.c (la clase c está dentro del paquete a.b), esto no es un problema, ya que se trata únicamente de ejemplos.

3.3.1. Wildcards

Las clases que se encuentran en el mismo paquete se pueden importar todas juntas, y para ello, se utiliza el siguiente carácter comodín o ***wildcard.*** Actualmente se considera una mala práctica, ya que se importan clases innecesarias que pueden no necesitarse.

```
import java.util.*; //El asterisco indica que se importarán todas las clases dentro
```

```
                    //del paquete java.util
```

Ejemplo:

```
import java.util.*;
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10));
  }
}
```

En este caso, únicamente se utiliza la clase java.util.Random en el programa; no obstante, se han importado todas las clases del paquete java.util (* es una wildcard que indica todas las clases). Únicamente se importarán las clases, no los paquetes hijos que contenga.

El número de *import* que tenga un programa no ralentiza el programa. El compilador lee cada clase en el momento en que la necesita.

3.3.2. *Imports* redundantes

Como se ha podido ver en el código de ejemplos anteriores, se han utilizado clases, como la clase *System* o la clase *String* en los programas sin importar nada, ¿Cómo es que el código funcionaba si no se han importado estas clases? Esto se debe a que existe un paquete en Java llamado java.lang que siempre se importa de manera automática. Aun así, Java permite también importarlo manualmente. Esto, no obstante, no se debe hacer, ya que importar algo que se importa por defecto o que ya está importado crea código redundante.

Ejemplo:

```
1: import java.lang.System;
2: import java.lang.*;
3: import java.util.Random;
4: import java.util.*;
5: public class ImportExample {
6:   public static void main(String[] args) {
7:     Random r = new Random();
8:     System.out.println(r.nextInt(10));
9:   }
10:}
```

¿Cuántos de los *import* son redundantes?

La respuesta es tres. Las líneas 1 y 2 son redundantes porque, como se ha comentado anteriormente, el paquete java.lang es importado de forma automática y por lo tanto no necesita de un *import*. La línea

4 es también redundante. La clase *Random* es importada al código mediante la sentencia de la línea 3 (*import java.util.Random;*), por lo que no es necesario importar, además, el resto del paquete java.util de la línea 4.

Otro caso de redundancia sería importar una clase que se encuentra en el mismo paquete que la clase que se está ejecutando, ya que, de forma automática, Java tiene en cuenta a todas las clases del mismo paquete sin necesidad de importarlas unas en otras.

Otro ejemplo:

```
public class InputImports {
  public void read(Files files) {
    Paths.get("name");
  }
}
```

Se supone que las clases *Files* y *Paths* están en el mismo paquete java.nio.file, y que la clase del ejemplo, InputInmports, está en un paquete diferente. ¿Qué se debería importar para que el programa compile?

Hay dos posibles formas de hacerlo:

```
import java.nio.file.*; //Importa ambos paquetes debido al uso de la wildcard *
```

o

```
import.java.nio.file.Files; //Importa la clase File
import.java.nio.file.Paths; //Importa la clase Paths
```

A continuación, se muestran unos ejemplos que NO funcionarían:

```
import java.nio.*; // NO FUNCIONA. Recordar que la wildcard *

                   //solo importa clases, no otros paquetes
import java.nio.*.*; // NO FUNCIONA. Solo puede haber una wildcard

                     //por import y esta debe estar al final
import java.nio.file.Paths.*; // NO FUNCIONA. No se pueden importar

                                //métodos sin la palabra clave static
```

3.3.3. Conflictos de nombrado

Una de las razones para utilizar paquetes en Java es que el nombre de las clases no tiene por qué ser único. Esto quiere decir que algunas veces se querrá importar una clase que tenga el mismo nombre que otras que se encuentran en otras partes del programa. Un ejemplo de esto es la clase Date. Java posee implementadas las clases java.util.Date y java.sql.Date. Son dos clases diferentes con el mismo nombre; no obstante, cada una se encuentra en un paquete diferente.

Ejemplo:

```
import java.util.*;
import java.sql.*; // No compila
public class Conflicts {
  Date date;
  //Más código
}
```

El código no compila ya que la clase Date existe en ambos paquetes, por lo que Java dará el siguiente error de compilación: "*The type Date is ambiguous*".

Tampoco compilará esto:

```
import java.util.Date;
import java.sql.Date;
```

Java lanzará el error de compilación "The import java.sql.Date collides with another import statement".

En caso de que se quiera usar las clases de ambos paquetes: clases de java.util.* y clases de java.sql.*, y la clase Date que será la que está en java.util.*, se deberá importar lo siguiente:

```
import java.util.Date;
import java.sql.*;
```

Ahora, al importar de manera explícita la clase Date de java.util.*, esta tiene precedencia por encima de todas las clases importadas con la *wildcard* *.

¿Qué ocurre si realmente se quiere utilizar en el código dos clases con el mismo nombre?

En ocasiones podría ocurrir que se quisiera utilizar la clase Date de ambos paquetes. Cuando esto pasa, una opción sería importar únicamente una de las clases y, cuando se quiera utilizar la otra, llamarla haciendo uso de su nombre plenamente cualificado (es decir, nombre de paquete, punto, nombre de la clase). Por ejemplo:

```
import java.util.Date;

public class Conflicts {
  Date date; //Usa la clase java.util.Date
  java.sql.Date sqlDate; //Usa la clase java.sql.Date
}
```

3.3.4. Crear un nuevo paquete

Hasta ahora, parte del código que se ha escrito en el capítulo se ha encontrado en el paquete por defecto *default package.* Este es un paquete especial que carece de nombre y que es mejor no utilizar.

En la vida real, será necesario nombrar los paquetes donde se encuentran las clases para evitar conflictos de nombrado y para permitir la reutilización de código.

Ahora, se quiere crear un nuevo paquete. La estructura de directorios en un ordenador normal está relacionado con la estructura de paquetes en Java. Se debe tener en cuenta, por ejemplo, la existencia de las siguientes clases:

```
C:\temp\packagea\ClassA.java
package packagea;

public class ClassA {
}
```

```
C:\temp\packageb\ClassB.java
package packageb;
import packagea.ClassA;

public class ClassB {
  public static void main(String[] args) {
    ClassA a;
    System.out.println("Got it");
  }
}
```

Cuando se ejecuta un programa Java, Java sabe "dónde buscar" los paquetes necesarios. En este caso, ejecutando el programa en el directorio C:\temp todo funcionaría correctamente ya que ambos paquetes: *packagea* y *packageb* están en en C:\temp.

3.3.5. Compilar código con *packages*:

Es posible, y de hecho mucho más cómodo, utilizar un IDE (*Integrated Development Environment*) para programar código Java, como Eclipse o Netbeans; no obstante, es posible ejecutar un programa Java desde la línea de comandos y es importante saber cómo hacerlo.

Se comienza por crear ambas clases y situarse en el directorio C:\temp:

Windows

- Crear los archivos:

```
C:\temp\packagea\ClassA.java
C:\temp\packageb\ClassB.java
```

- Ir a C:\temp

```
cd C:\temp
```

Mac OS/Linux

- Crear los archivos:

```
/tmp/packagea/ClassA.java
/tmp/packageb/ClassB.java
```

- Escribir el comando

```
cd /tmp
```

A continuación es necesario **compilar** ambos archivos, por lo que hay que escribir lo siguiente en la línea de comandos:

```
javac packagea/ClassA.java packageb/ClassB.java
```

En caso de que el comando no funcione, se mostrará por pantalla un mensaje de error. En estos casos es recomendable revisar los archivos con cuidado para detectar algún posible error de sintaxis. En caso de que el comando funcione correctamente, se crearán dos nuevos archivos: *packagea/ClassA.class* y *packageb/ClassB.class*.

Por último, quedaría **ejecutar** el programa. Para ello habría que poner lo siguiente en la línea de comandos:

```
java packageb.ClassB
```

Si todo funciona correctamente, la frase "*Got it*" se mostrará en la consola. Es importante percatarse de que a la hora de ejecutar un programa se escribe únicamente *ClassB*, no *ClassB.class*. En Java no se escribe la extensión cuando se ejecuta un programa.

3.3.6. Ruta de las clases y JARs

También es posible especificar la ubicación de otros archivos, explícitamente, utilizando rutas. Esta técnica es muy útil cuando los archivos de las clases están situados en otra parte que no sea el directorio actual o en archivos JAR. Un archivo JAR es como un archivo ZIP que únicamente contiene clases Java. A continuación se verán unos ejemplos:

En Windows

```
java -cp ".;C:\temp\someOtherLocation;c:\temp\myJar.jar" myPackage.MyClass
```

En MAC OS/Linux

```
java -cp ".:/tmp/someOtherLocation:/tmp/myJar.jar" myPackage.MyClass
```

El punto inicial indica que se quiere incluir el directorio actual en la ruta de la clase. El resto del comando incluirá también las clases (o paquetes) en *someOtherLocation* y en *myJar.jar*. Windows utiliza el carácter punto y coma para separar las partes de la ruta, y MAC OS y Linux utiliza el carácter dos puntos.

También es posible utilizar la *wildcard* (*) para hacer referencia a todos los JARs de un directorio. Como por ejemplo:

```
java -cp "C:\temp\directoryWithJars\*" myPackage.MyClass
```

Este comando añadirá todos los JARs que se encuentren en el directorio *directoryWithJars* a la ruta de la clase. No obstante, no incluirá los JARs que se encuentren en los subdirectorios de *directoryWithJars*.

3.4. Crear objetos

Los objetos son una parte fundamental de un programa Java. Un objeto es una instancia de una clase. En las siguientes secciones, se profundizará acerca de los constructores, las propiedades y cómo éstas han de ser inicializadas correctamente.

3.4.1. Constructores

Para crear una instancia de una clase, siempre se utilizará la palabra reservada **new**. Por ejemplo:

```
Random r = new Random();
```

En primer lugar se declara la variable declarando el tipo de dato que tendrá (en este caso *Random*) y su nombre (r). Esto permitirá a Java reservar en memoria una referencia al objeto. Es al escribir *new Random()* cuando realmente el objeto es creado.

Random() parece un método, ya que está seguido de paréntesis. Este tipo de métodos son llamados constructores, y son los encargados de crear nuevos objetos. A continuación, se verá un ejemplo de cómo se define un método constructor:

```
public class Chick {
  public Chick() {
    System.out.println("in constructor");
  }
}
```

Cabe destacar un par de detalles del siguiente código: en primer lugar, el nombre del constructor es el mismo que el nombre de la clase, y en segundo lugar, no hay **return**.

El método public void Chick() { } NO sería un constructor.

Es necesario prestar especial atención a estos detalles. En caso de ver un método que comience por mayúscula pero tenga tipo de **return** (aunque este sea **void**), nunca será un constructor. Se tratará de un método normal que no será ejecutado cuando se escriba *new Chick()*.

El propósito de un constructor es inicializar las propiedades del objeto (aunque se pueda escribir dentro de él lo que se desee). Otra forma de inicializar propiedades es hacerlo directamente en la línea de código donde estas son declaradas. En el siguiente ejemplo se pueden observar ambos casos:

```
public class Chicken {
  int numEggs = 0;// Inicializada directamente
  String name;
```

```
  public Chicken() {
    name = "Duke";// Inicializada en el constructor
  }
}
```

Muchas clases no precisan de un constructor explícito, por lo que el compilador las provee de un constructor por defecto que no hace nada (es decir, su única finalidad es permitir crear objetos, pero no tiene código alguno en su interior).

3.4.2. Leer y modificar las propiedades de los objetos

Es posible leer y modificar las propiedades de un objeto a partir de sí mismo, por ejemplo:

```
public class Swan {
  int numberEggs;// variable de instancia o propiedad de la clase Swan
  public static void main(String[] args) {
    Swan mother = new Swan();
    mother.numberEggs = 1; // modifica la propiedad
    System.out.println(mother.numberEggs); // lee la propiedad
  }
}
```

En este caso, la propiedad *numberEggs* es obtenida directamente cuando es imprimida por consola. En el método *main*, la variable *numberEggs* es modificada escribiendo en ella el valor 1.

Más adelante se verá la forma de prevenir que cualquiera pueda modificar las propiedades de un objeto directamente, para que, por ejemplo, no se pueda poner un número de huevos negativo.

También es posible leer y escribir propiedades directamente al declararlas:

```
1: public class Name {
2:   String first = "Theodore";
3:   String last = "Moose";
4:   String full = first + last;
5: }
```

Las líneas 2 y 3 modifican las propiedades *first* y *last*, respectivamente, mientras que en la línea 4 las propiedades *first* y *last* son leídas y, después, se modifica la propiedad *full* con el valor de la concatenación de ambas.

3.4.3. Bloques inicializadores de instancias

A lo largo de los capítulos anteriores se han visto diferentes métodos, todos ellos conteniendo los caracteres {}. El código entre llaves "{}" es llamado bloque de código.

La mayoría de las veces hay bloques de código dentro de métodos, que se ejecutan cuando el método es llamado. Otras veces, los bloques de código pueden aparecer fuera de un método. Estos bloques son llamados inicializadores de instancia.

¿Cuántos bloques hay en el siguiente código? ¿Cuántos de estos bloques son bloques inicializadores de instancias?

```
3: public static void main(String[] args) {
4:   { System.out.println("Feathers"); }
5: }
6: { System.out.println("Snowy"); }
```

Hay tres bloques y un inicializador de instancia. Contar el número de bloques es sencillo: únicamente hay que contar los pares de llaves que hay en el código. Si no hay el mismo número de "{" que de "}" el código no compilará.

Para poder identificar un bloque inicializador de instancia es necesario tener en cuenta que estos nunca podrán aparecer dentro de un método, por lo que el inicializador de instancia del código anterior es el bloque que aparece en la línea 6.

3.4.4. Orden de inicialización

Cuando se escribe código que inicializa propiedades en diferentes partes del programa, es necesario tener en cuenta el orden de inicialización de las mismas. Por el momento, se tendrá en cuenta las siguientes reglas a la hora de inicializar propiedades:

- Tanto las propiedades como los bloques de iniciación de instancias son ejecutados en el orden en el que aparezcan en el código.
- El constructor se ejecuta después de que las propiedades y los bloques de inicialización de instancias se hayan ejecutado.

A continuación, se verá el siguiente ejemplo:

```
1: public class Chick {
2:   private String name = "Fluffy";
3:   { System.out.println("setting field"); }
4:   public Chick() {
5:     name = "Tiny";
6:     System.out.println("setting constructor");
7:   }
8:   public static void main(String[] args) {
9:     Chick chick = new Chick();
10:    System.out.println(chick.name); } }
```

Que imprimirá por pantalla lo siguiente:

```
setting field
setting constructor
Tiny
```

En primer lugar, el programa comienza su ejecución en el método *main()* (como ocurre en cualquier programa Java). En la línea 9, se llama al constructor de la clase Chick, y Java crea un nuevo objeto de la misma. Primero la propiedad *name* es inicializada con el valor "Fluffy" en la línea 2. Después se ejecuta *System.out.println("setting field");* en la línea 3. Una vez las propiedades y los bloques inicializadores de instancias se han ejecutado, Java vuelve al constructor. La línea 5 cambia el valor de *name* a "Tiny" y la línea 6 vuelve a imprimir por pantalla mediante la sentencia *System.out.println("setting constructor");* En este punto, el constructor ha terminado de ejecutarse, y se retorna al método *main*, concretamente, a la línea 10.

El orden del código es importante. No se puede hacer referencia a una variable antes de que esta haya sido declarada.

```
{ System.out.println(name); } // NO COMPILA
private String name = "Fluffy";
```

Otro ejemplo, ¿qué imprimirá por pantalla el siguiente código?

```
public class Egg {
  public Egg() {
    number = 5;
  }
  public static void main(String[] args) {
    Egg egg = new Egg();
    System.out.println(egg.number);
  }
  private int number = 3;
  { number = 4;}
}
```

La respuesta correcta será "5". Las propiedades y los bloques de código inicializadores de instancia se ejecutarán en primer lugar, cambiando *number* a 3 y luego a 4. Posteriormente, el constructor modifica *number* a 5, y esto es lo que se imprime por pantalla.

3.5. Diferencia entre referencias a objetos y primitivas de datos

Las aplicaciones Java contienen dos grupos de tipos de dato: variables de referencia y primitivas de datos. En este apartado se expondrán las diferencias entre ambos grupos.

3.5.1. Primitivas de datos

Java posee ocho tipos de dato incorporados en el propio lenguaje por defecto que se conocen como primitivas de datos. Estos ocho tipos de dato representan los cimientos de cualquier objeto Java, ya que todos los objetos son composiciones más o menos complejas de las primitivas de datos.

La siguiente tabla muestra todas las primitivas de datos junto con su tamaño en bytes y el rango de valores que pueden adoptar.

Palabra reservada (keyword)	**Tipo**	**Ejemplo**
boolean	true o false	True
byte	Valor entero de 8 bits	123
short	Valor entero de 16 bits	123
int	Valor entero de 32 bits	123
long	Valor entero de 64 bits	123
float	Valor en coma flotante de 32 bits	123.45f
double	Valor en coma flotante de 64 bits	123.456
char	Valor Unicode de 16 bits	'a'

A continuación se verá, detalladamente, la información de la tabla anterior:

- float y double son utilizados para representar valores decimales.
- Los datos float necesitan poner la letra f seguido del número para que Java los identifique como float.
- byte, short, int y long se utilizan para representar números enteros.
- Cada tipo numérico utiliza dos veces más bits que el tipo numérico más pequeño anterior a él; por ejemplo, short utiliza el doble de bits que byte.

Es necesario tener presente que un byte puede contener valores de -128 a 127, ¿Por qué?

Un byte son 8 bits. Un bit puede tener dos posibles valores. 2^8 es 256. Ya que el 0 también ha de ser incluido en el rango de números a abarcar, Java lo considera en la parte positiva, por lo que 256/2=128 (de -128 a 127).

El número de bits utilizado por Java cuando necesita reservar memoria para una primitiva de datos será el número de bits que ocupe cada tipo de dato. Por ejemplo, Java reservará 32 bits de memoria cuando se escriba lo siguiente:

```
int num;
```

Continuando con las primitivas de datos numéricas, cabe destacar que cuando un número está presente en el código, es llamado "literal". Por defecto, Java asume que el valor definido es del tipo int. En el ejemplo siguiente, el valor que se muestra es demasiado grande para ser de tipo int.

```
int max = 3123456789; //NO COMPILA
```

Java mostrará que ese número está fuera de rango, lo cual es cierto teniendo en cuenta que pretende interpretarlo como un número de tipo int. No obstante, no se pretende que sea el número de tipo int. La solución es añadir la letra L al número.

```
long max = 3123456789L; //Ahora es cuando Java comprende que el número es un long
```

Otra manera de escribir números es cambar la base. Cuando se aprende a contar, se estudian los dígitos del 0 al 9 (números en base 10 o sistema decimal, ya que hay 10 números). Java permite escribir números de otras maneras:

- Octal (dígitos del 0-7). Para expresar un número en octal, este ha de ir precedido por un 0. Por ejemplo: 017

- Hexadecimal (dígitos del 0-9, letras de la A-F). Para expresar un número en hexadecimal, este ha de ir precedido por el prefijo 0x o 0X. Por ejemplo: 0xFF.

- Binario (dígitos 0-1). Para expresar un número en binario, este ha de ir precedido por el prefijo 0b o 0B. Por ejemplo: 0b10.

Lo último a tener en cuenta sobre literales numéricos es una característica añadida en Java 7, y es que es posible añadir barras bajas a los números para facilitar su lectura:

```
int million1 = 1000000;
int million2 = 1_000_000;
```

Las barras bajas se podrán añadir en cualquier parte del literal numérico excepto:

- Al principio del literal.
- Al final del literal.
- Antes de un punto decimal.
- Después de un punto decimal.

Por ejemplo:

```
double notAtStart = _1000.00; //NO COMPILA
double notAtEnd = 1000.00_; //NO COMPILA
double notByDecimal = 1000_.00; //NO COMPILA
double goodOne = 1_00_0.0_0; //COMPILA
```

3.5.2. Tipos referencia

Un tipo referencia es un objeto (un objeto es una instancia de una clase). Al contrario que ocurre con las primitivas de datos, que guardan sus valores directamente en la memoria reservada por la variable, los tipos referencia no guardan los valores de los objetos en la dirección de memoria reservada cuando las variables son declaradas. En lugar de eso, la referencia "apunta" al objeto asignado a la variable guardando la dirección de memoria donde este se encuentra, concepto también conocido como puntero. Al contrario que en otros lenguajes, Java no permite conocer cuál es la dirección física en memoria donde están guardados sus objetos.

La forma de declarar e inicializar tipos referencia es la siguiente, se consideran declaradas dos referencias, una de tipo *java.util.Date*, y otra de tipo *String*:

```
java.util.Date hoy;
String saludo;
```

La variable *hoy* es una referencia de tipo Date, y únicamente puede "apuntar" a un objeto de tipo Date. La variable *saludo* es una variable que únicamente puede "apuntar" a un objeto de tipo String. Es posible asignar un valor a una referencia de las siguientes formas:

- A una referencia se le puede asignar un objeto de su mismo tipo de dato.

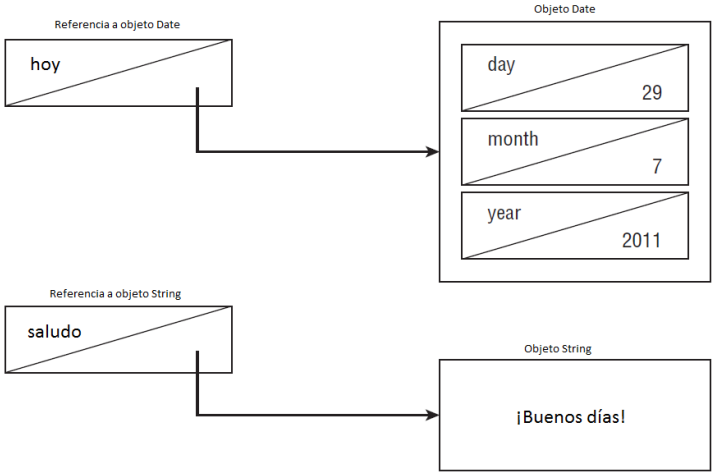
- A una referencia se le puede asignar un nuevo objeto utilizando para ello la palabra reservada *new*.

Por ejemplo, el siguiente código muestra dos asignaciones a nuevos objetos:

```
hoy   = new java.util.Date();
saludo = "¡Buenos días!";
```

La referencia *hoy* apunta a un Nuevo objeto de tipo Date en memoria, por lo que a partir de ahora *hoy* podrá ser utilizada para acceder a los métodos y atributos de la clase Date. De igual manera, la referencia *saludo* apunta a un nuevo objeto de tipo String, el cual tiene el valor "¡Buenos días!". Los objetos String y Date únicamente pueden ser accedidos a través de sus referencias.

La siguiente figura muestra un ejemplo de cómo las variables referencia se representan en memoria:



3.5.3. Diferencias clave

Existen algunas diferencias muy importantes a tener en cuenta entre primitivas de datos y variables referencia. En primer lugar, las variables referencia pueden tener el valor *null*, lo que quiere decir que no tienen ningún objeto asignado. Las primitivas de datos mostrarán un error de compilación si se les intenta asignar el valor *null.* En el siguiente ejemplo, *valor* no puede ser *null*, ya que es de tipo *int*.

```
int valor = null; //NO COMPILA
String s = null;
```

En segundo lugar, los tipos referencia pueden ser utilizados para llamar métodos siempre y cuando no sean *null*. Las primitivas de datos no tienen métodos. En el siguiente ejemplo, es posible llamar al método *length()*desde la variable *texto* porque *texto* es una variable de referencia. Se puede saber que *length* es un método ya que utiliza (). En la siguiente línea, cuando se intenta llamar al método *length()* sobre *longitud*, que es de tipo *int*(primitiva de datos), aparecerá un error de compilación, ya que las primitivas de tipo int no tienen una longitud definida.

```
String texto = “hola”;
int longitud = texto.length();
int mal = longitud.length(); //NO COMPILA
```

Por último cabe destacar que las primitivas de datos están escritas en el lenguaje comenzando con letra minúsucla, mientras que las clases, siempre comienzan con letra mayúscula.

3.6. Declarando e inicializando variables

Hasta ahora se han visto ya numerosas variables. Una variable es, en realidad, un espacio de memoria donde se guardan datos. Cuando se declara una variable, es necesario especificar tanto su nombre como su tipo de dato. Por ejemplo, el siguiente código declara dos variables. Una es llamada *zooName* y es de tipo *String*. La otra es llamada *numberAnimals* y es de tipo *int*.

```
String zooName;
int numberAnimals;
```

Ahora que se han declarados las variables, es posible asignarles valores; es decir, es posible inicializar las variables. Para inicializar una variable, es necesario escribir el nombre de la variable seguido del carácter =, seguido del valor a inicializar:

```
zooName = "El mejor Zoo";
numberAnimals = 100;
```

También es posible declarar e inicializar las variables en la misma línea:

```
String zooName = " El mejor Zoo ";
int numberAnimals = 100;
```

3.6.1. Declarar múltiples variables

También es posible declarar e inicializar varias variables en la misma línea. Por ejemplo, ¿Cuántas variables se han declarado e inicializado en las siguientes dos líneas?

```
String s1, s2;
String s3 = "yes", s4 = "no";
```

La respuesta es cuatro variables de tipo *String*: *s1, s2, s3* y *s4*. Es posible declarar tantas variables como se desee seguidas siempre y cuando estas pertenezcan al mismo tipo de dato. También es posible inicializar estas variables en la misma línea (todas, varias, o lo que se desee). En el ejemplo anterior, las variables *s3* y *s4* están inicializadas, pero las variables *s1* y *s2* están únicamente declaradas.

Un ejemplo algo más complejo:

```
int i1, i2, i3 = 0;
```

En este caso se han declarado tres variables: *i1*, *i2* e *i3*, pero únicamente *i3* ha sido inicializada con el valor 0. Es necesario tener en cuenta que, por cada " , ", hay una variable diferente cuya inicialización es independiente a las demás.

El código siguiente, por ejemplo, no compilaría, ya que no es posible declarar en la misma sentencia variables de tipos de dato diferente:

```
int num, String value; // NO COMPILA
```

Por último, se verá el siguiente ejemplo, ¿Cuáles de las siguientes líneas serían válidas y cuáles no compilarían?

```
1: boolean b1, b2;
2: String s1 = "1", s2;
3: double d1, double d2;
4: int i1; int i2;
5: int i3; i4;
```

La línea 1 es correcta. Declara dos variables de tipo *boolean* sin inicializar. La línea 2 también es correcta, ya que declara dos variables de tipo *String* e inicializa la primera de ellas con el valor "1". La tercera línea no es correcta. Java no permite declarar dos variables de diferente tipo en la misma sentencia, y, a pesar de que en este caso ambas variables serían de tipo *double*, si se quieren declarar dos variables de mismo tipo en la misma línea, estas deben compartir la misma declaración del tipo de dato, y no repetirlo.

La línea 4 también es correcta, y es que, aunque pueda parecer un caso igual al anterior, no es así, ya que la línea cuatro muestra dos sentencias en la misma línea (separadas por ";").

La línea 5 no es correcta, ya que, de nuevo, presenta dos sentencias, y la segunda de ella no tiene el tipo de dato necesario para declarar la variable *i4*. Esto sería como escribir:

```
int i3;
i4; //NO COMPILA
```

3.6.2. Identificadores

Probablemente no sorprenda que Java tenga reglas precisas sobre nombres de identificadores. Afortunadamente, las mismas reglas para los identificadores se aplican a cualquier cosa que puedas nombrar, incluyendo variables, métodos, clases y campos. Sólo hay tres reglas a recordar para los identificadores legales:

- El nombre debe comenzar con una letra o el símbolo $ o _.

- Los caracteres posteriores también pueden ser números.

- No se puede usar el mismo nombre que una palabra reservada de Java. Una palabra reservada es una palabra clave que Java ha reservado para que no se le permita utilizarla. Recordar que Java distingue entre mayúsculas y minúsculas, por lo que se puede utilizar versiones de las palabras clave que sólo difieren en mayúsculas y minúsculas, aunque es aconsejable no hacerlo.

No hay que preocuparse, no se necesitará memorizar la lista completa de palabras reservadas. La siguiente lista es una lista de todas las palabras reservadas en Java. const y goto no se utilizan realmente en Java. Están reservados para que las personas que vienen de otros idiomas no los usen por accidente - y en teoría, en caso de que Java quiera usarlos algún día.

abstract	assert	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto*	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

Los siguientes ejemplos son legales:

```
okidentifier
$OK2Identifier
_alsoOK1d3ntifi3r
__SStillOkbutKnotsonice$
```

Estos ejemplos no son legales:

```
3DPointClass // Los identificadores no pueden comenzar por un número
hollywood@vine // @ no es una letra, dígito, $ o _
*$coffee // * no es una letra, dígito, $ o _
public // public es una palabra reservada
```

Aunque se puede hacer locuras con nombres de identificación, no se debería. Java tiene convenciones para que el código sea legible y consistente. Esta consistencia incluye CamelCase. En CamelCase, cada palabra comienza con una letra mayúscula. Esto facilita la lectura de nombres de variables de más de una palabra. ¿Qué sería más legible: thisismyclass name o ThisIsMyClass name? Cuando aparezca un identificador no estándar, hay que asegurarse de verificar si es legal.

Identificadores en el mundo real

La mayoría de los desarrolladores siguen estas convenciones para identificar nombres:

- Los nombres de métodos y variables comienzan con una letra minúscula seguida de CamelCase.

- Los nombres de las clases comienzan con una letra mayúscula seguida de CamelCase. No se deben usar identificadores con $. El compilador usa este símbolo para algunos archivos.

Además, las letras válidas en Java no son sólo caracteres del alfabeto inglés. Java soporta el conjunto de caracteres Unicode, por lo que hay más de 45.000 caracteres que pueden iniciar un identificador Java legal. Unos pocos centenares más son números no árabes que pueden aparecer después del primer carácter en un identificador válido.

3.7. Comprensión de la inicialización por defecto de las variables

Antes de poder utilizar una variable, necesita un valor. Algunos tipos de variables obtienen este valor automáticamente y otras requieren que el programador lo especifique. En las siguientes secciones, se verán las diferencias entre los valores predeterminados para las variables locales, de instancia y de clase.

3.7.1. Variables locales

Una variable local es una variable definida dentro de un método. Las variables locales deben ser inicializadas antes de su uso. No tienen un valor por defecto y contienen datos basura hasta que se inicializan. El compilador no le permitirá leer un valor sin inicializar. Por ejemplo, el siguiente código genera un error de compilación:

```
4: public int notValid() {
5:   int y = 10;
6:   int x;
7:   int reply = x + y; // DOES NOT COMPILE
8:   return reply;
9: }
```

y se inicializa a 10. Sin embargo, debido a que x no se inicializa antes de ser utilizado en la expresión de la línea 7, el compilador genera el error de seguimiento:

```
Test.java:5:

variable x might not have been initialized
int reply = x + y;
^
```

Hasta que a x no se le asigne un valor, no puede aparecer dentro de una expresión, y el compilador le recordará esta regla. El compilador sabe que su código tiene el control de lo que sucede dentro del método y puede esperar que inicialice los valores.

El compilador es lo suficientemente inteligente como para reconocer variables que han sido inicializadas después de su declaración, pero antes de ser usadas. Aquí hay un ejemplo:

```
public int valid() {
  int y = 10;
  int x; // x is declared here
  x = 3; // and initialized here
  int reply = x + y;
  return reply;
}
```

El compilador también es lo suficientemente inteligente como para reconocer inicializaciones más complejas. En este ejemplo, hay dos ramas de código y answer se inicializa en ambas. onlyOneBranch sólo se inicializa si check es verdadero. El compilador sabe que existe la posibilidad de que check sea falso y de que por lo tanto onlyOneBranch no se inicialice nunca, por lo que en el código hay un error de compilación. Se aprenderá más sobre la declaración if en el próximo capítulo.

```
public void findAnswer(boolean check) {
  int answer;
  int onlyOneBranch;
  if(check) {
    onlyOneBranch = 1;
    answer = 1;
  } else {
    answer = 2;
  }
  System.out.println(answer);
```

```
  System.out.println(onlyOneBranch); // DOES NOT COMPILE
}
```

3.7.2. Variables de clase e instancia

Las variables que no son variables locales se conocen como variables de instancia o variables de clase. Las variables de instancia también se denominan fields. Las variables de clase se comparten entre varios objetos. Se puede decir que una variable es una variable de clase porque tiene la palabra clave static delante de ella. Se encontrará más información al respecto en el capítulo 4. Por ahora, sólo se debe saber que una variable es una variable de clase si tiene la palabra clave static en su declaración.

Las variables de instancia y clase no requieren ser inicializadas.

Tipo de variable	**Valor por defecto de inicialización**
boolean	false
byte, short, int, long	0 (en la longitud de bits del tipo)
float, double	0.0 (en la longitud de bits del tipo)
char	'\u0000' (NUL)
Todas las referencias a objetos	Null

3.8. Entendiendo el ámbito de las variables

Se ha aprendido que las variables locales se declaran dentro de un método. ¿Cuántas variables locales se ven en este ejemplo?

```
public void eat(int piecesOfCheese) {
  int bitesOfCheese = 1;
}
```

Hay dos variables locales en este método. bitesOfCheese se declara dentro del método. piecesOfCheese se llama parámetro de método. También es local al método. Se dice que ambas variables tienen un alcance local al método. Esto significa que no se pueden utilizar fuera del método.

Las variables locales nunca pueden tener un alcance mayor que el método definido. Sin embargo, pueden tener un alcance menor. Ejemplo:

```
3: public void eatIfHungry(boolean hungry) {
4:   if (hungry) {
5:     int bitesOfCheese = 1;
6:   } // bitesOfCheese goes out of scope here
```

```
7:   System.out.println(bitesOfCheese);// NO COMPILA
8: }
```

hungry tiene un alcance de todo el método. bitesOfCheese tiene un alcance más pequeño. Sólo está disponible para su uso en la sentencia if porque se declara dentro de ella. Cuando se vea un juego de llaves ({ }) en el código, significa que se ha introducido un nuevo bloque de código. Cada bloque de código tiene su propio alcance. Cuando hay bloques múltiples, se emparejan de dentro hacia fuera. En este caso, el bloque de sentencia if comienza en la línea 4 y termina en la línea 6. El bloque del método comienza en la línea 3 y termina en la línea 8.

Dado que bitesOfCheese se declara en tal bloque, el ámbito de aplicación se limita a dicho bloque. Cuando el compilador llega a la línea 7, muestra que no sabe nada sobre bitesOfCheese y da un error:

```
bitesOfCheese cannot be resolved to a variable
```

Recordar que los bloques pueden contener otros bloques. Estos bloques más pequeños contenidos pueden referenciar variables definidas en los bloques de mayor tamaño, pero no viceversa. Por ejemplo:

```
16: public void eatIfHungry(boolean hungry) {
17:   if (hungry) {
18:     int bitesOfCheese = 1;
19:     {
20:     boolean teenyBit = true;
21:     System.out.println(bitesOfCheese);
22:     }
23:   }
24:   System.out.println(teenyBit); // NO COMPILA
25: }
```

La variable definida en la línea 18 está en alcance hasta que el bloque termina en la línea 23. Usarlo en el bloque más pequeño de las líneas 19 a 22 es seguro. La variable definida en la línea 20 queda fuera del alcance en la línea 22. No está permitido su uso en la línea 24. No hay que preocuparse si aún no se está familiarizado con las declaraciones o los bucles. No importa lo que haga el código, ya que se está hablando del alcance. A continuación, se intentará determinar, en qué línea, cada una de las variables locales entra y sale del alcance:

```
11: public void eatMore(boolean hungry, int amountOfFood) {
12:   int roomInBelly = 5;
13:   if (hungry) {
14:     boolean timeToEat = true;
15:     while (amountOfFood > 0) {
16:       int amountEaten = 2;
17:       roomInBelly = roomInBelly - amountEaten;
18:       amountOfFood = amountOfFood - amountEaten;
19:     }
```

```
20:    }
21:    System.out.println(amountOfFood);
22: }
```

El primer paso para determinar el alcance es identificar los bloques de código. En este caso, hay tres bloques. Se puede decir esto porque hay tres juegos de llaves. Partiendo del conjunto más interno, se puede ver dónde comienza y termina el bloque del bucle while. Se repetirá esto mientras se sale, para el bloque de sentencia if y el bloque de método. La Tabla 1.3 muestra los números de línea que cada bloque comienza y termina.

Bloque	Primera línea del bloque	Ultima línea del bloque
while	15	19
if	13	20
method	11	22

Se necesitará practicar mucho esto. La identificación de bloques debe ser algo natural. La buena noticia es que hay muchos ejemplos de códigos para practicar.

Ahora que se sabe dónde están los bloques, se puede ver el alcance de cada variable. hungry y amountOfFood son parámetros de método, por lo que están disponibles para todo el método. Esto significa que su ámbito de aplicación son las líneas 11 a 22. roomInBelly entra en el ámbito de aplicación de la línea 12 porque ahí es donde se declara. Se mantiene en el alcance para el resto del método y por lo tanto se sale del alcance en la línea 22. timeToEat entra en el alcance en la línea 14 donde se declara. Se sale del alcance en la línea 20 donde termina el bloque if. amountEaten entra en el ámbito de aplicación de la línea 16 donde se declara. Se sale del alcance en la línea 19 donde termina el bloque de tiempo.

Todo eso era para variables locales. Afortunadamente, las reglas para las variables de instancia son más fáciles: están disponibles tan pronto como se definen y duran toda la vida útil del objeto en sí. La regla para las variables de clase (static) es aún más fácil: entran en alcance cuando se declaran como los otros tipos de variables. Sin embargo, permanecen en el alcance durante toda la vida del programa.

A continuación, se hará un ejemplo más para asegurarse de que se tiene control sobre esto. Una vez más, se debe averiguar el tipo de las cuatro variables y cuándo entran y salen del alcance.

```
1: public class Mouse {
2:    static int MAX_LENGTH = 5;
3:    int length;
4:    public void grow(int inches) {
5:      if (length < MAX_LENGTH) {
6:        int newSize = length + inches;
7:        length = newSize;
8:      }
```

```
9:    }
10:}
```

En esta clase, se tiene una variable de clase (MAX_LENGTH), una variable de instancia (length) y dos variables locales (inches y newSize.) MAX_LENGTH es una variable de clase porque tiene la palabra clave static en su declaración. MAX_LENGTH entra en el ámbito de aplicación de la línea 2 donde se declara. Permanece en el alcance hasta que finaliza el programa. Length entra en el ámbito de la aplicación en la línea 3, donde se declara. Permanece en el alcance mientras el objeto Mouse exista. inches entra en alcance donde se declara en la línea 4. Se sale del alcance al final del método en la línea 9. newSize entra en el alcance donde se declara en la línea 6. Dado que está definido dentro del bloque de sentencia if, se sale del alcance cuando ese bloque termina en la línea 8. ¿Se ha resuelto todo? Ahora, se revisarán las reglas sobre el alcance:

- Variables locales: en el ámbito de aplicación desde la declaración hasta el final del bloque

- Variables de instancia: en el ámbito de aplicación de la declaración hasta que se recoge la basura del objeto.

- Variables de clase: en el ámbito de aplicación de la declaración hasta que finalice el programa.

3.9. Ordenando elementos en una clase

Ahora que se ha visto las partes más comunes de una clase, se echará un vistazo al orden correcto para escribirlas en un archivo. Los comentarios pueden ir en cualquier parte del código. Más allá de eso, es necesario memorizar las reglas en la tabla siguiente:

Elemento	*Ejemplo*	*¿Necesario?*	*¿Dónde va?*
Declaración de paquete	*package abc;*	*No*	*Primera línea del archivo*
Sentencias de importación	*import java.util*;*	*No*	*Inmediatamente después del paquete*
Declaración de clase	*public class C*	*Si*	*Inmediatamente después de la importación*
Declaración de campos	*int value;*	*No*	*Cualquier lugar dentro de la clase*
Declaración de métodos	*void method()*	*No*	*Cualquier lugar dentro de la clase*

Se verán algunos ejemplos para ayudar a recordar esto. El primer ejemplo contiene uno de cada elemento:

```
package structure; // package must be first non-comment
import java.util.*; // import must come after package
```

```
public class Meerkat { // then comes the class
  double weight; // fields and methods can go in either order
  public double getWeight() { return weight; }
  double height; // another field – they don't need to be together
}
```

Hasta ahora todo bien. Este es un patrón común que se debe conocer. Véase otro ejemplo:

```
/* header */
package structure;
// class Meerkat
public class Meerkat { }
```

Sigue siendo bueno. Se puede poner comentarios en cualquier lugar, y las importaciones son opcionales. En el siguiente ejemplo, hay un problema:

```
import java.util.*;
package structure; // DOES NOT COMPILE
String name; // DOES NOT COMPILE
public class Meerkat { }
```

Hay dos problemas aquí. Una es que el paquete y las declaraciones de importación se anulan. Aunque ambos son opcionales, el paquete debe llegar antes de la importación si está presente. La otra cuestión es que un campo intenta la declaración fuera de una clase. Esto no está permitido. Los campos y métodos deben estar dentro de una clase. ¿Se tiene todo?

Un consejo, se debe recordar el acrónimo PIC (picture): package, import y class. Los campos y métodos son más fáciles de recordar porque simplemente tienen que estar dentro de una clase. Se pueden definir varias clases en el mismo archivo, pero sólo una de ellas puede ser pública. La clase pública coincide con el nombre del archivo. Por ejemplo, estas dos clases deben estar en un archivo llamado Meerkat. java:

```
1: public class Meerkat { }
2: class Paw { }
```

En un archivo también se permite que ninguna de las clases sea pública. Mientras no haya más de una clase pública en un archivo, está bien.

3.10. Destruyendo Objetos

Ahora que se ha trabajado con objetos, es hora de deshacerse de ellos. Por suerte, Java se encarga automáticamente de esto, proporcionando una herramienta llamada Garbage Collector ("recolector de basura"), el cual busca automáticamente objetos que ya no están en uso.

Todos los objetos Java se almacenan en la memoria del programa, siendo el *heap* un gran grupo de memoria no utilizada asignada a una aplicación Java. El *heap* puede ser bastante grande dependiendo

del entorno, pero siempre tiene un límite de tamaño. Si el programa sigue instanciando objetos y dejándolos en el *heap*, llegará un momento en el que se quedará sin memoria.

En las siguientes secciones se verá la Garbage Collection o "recolección de basura" y el método finalize().

3.10.1. Garbage Collection

El concepto Garbage Collection en Java hace referencia al proceso de liberar automáticamente la memoria en el *heap* borrando objetos que ya no son accesibles en un programa; es decir, objetos que han quedado desreferenciados. Existen muchos algoritmos diferentes para la Garbage Collection, pero no es necesario conocer ninguno de ellos. Sí que es necesario; no obstante, familiarizarse con el método System.gc(), la cual ejecuta el Garbage Collector, recogiendo aquellos objetos ya no referenciados en el código.

El método System.gc() no siempre puede ejecutar el Garbage Collector, sino que realmente, a pesar de que sí se le ordena a Java que lleve a cabo esta acción, esta podría ser ignorada.

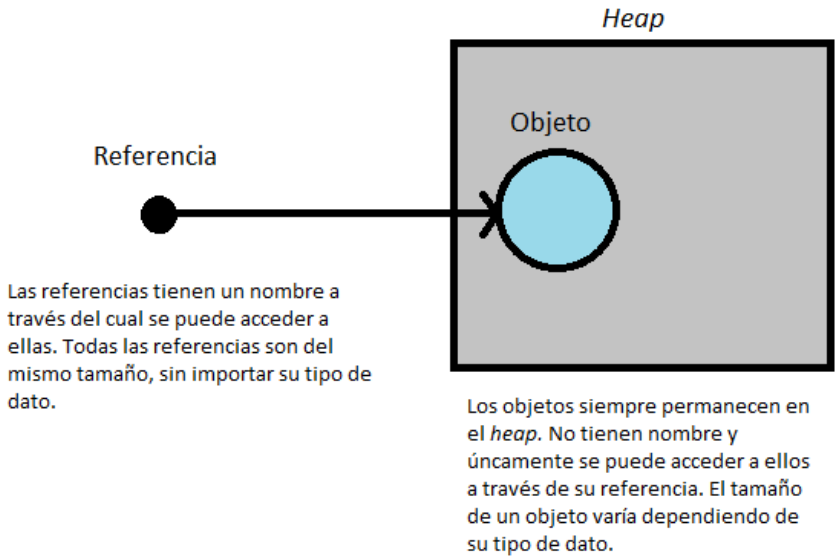
La parte más interesante de la Garbage Collection es cuando la memoria perteneciente a un objeto ya desreferenciado es recuperada. Java es capaz de detectar cuándo un objeto ya no es utilizado en ninguna parte del código. Esto puede darse en dos situaciones:

- El objeto ya no tiene referencias que lo apunten.
- Todas las referencias al objeto están fuera de ámbito.

Objetos vs Referencias

No debe confundirse una referencia con el objeto al que se refiere; son dos entidades diferentes. La referencia es una variable que tiene un nombre y se puede utilizar para acceder al contenido de un objeto. Se puede asignar una referencia a otra referencia, pasarla a un método o devolverla desde un método. Todas las referencias son del mismo tamaño, independientemente de su tipo.

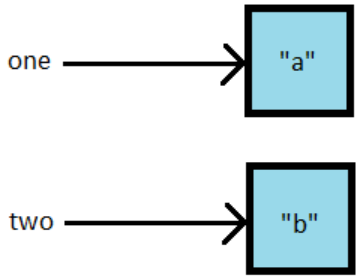
Un objeto se deposita en el heap y no tiene nombre. Por lo tanto, no hay manera de acceder a un objeto excepto a través de una referencia. Los objetos vienen en todas las formas y tamaños diferentes y consumen cantidades variables de memoria. Un objeto no se puede asignar a otro objeto, ni tampoco se puede pasar un objeto a un método o devolverlo desde un método. Es el objeto quien recibe el Garbage Collection, no su referencia.



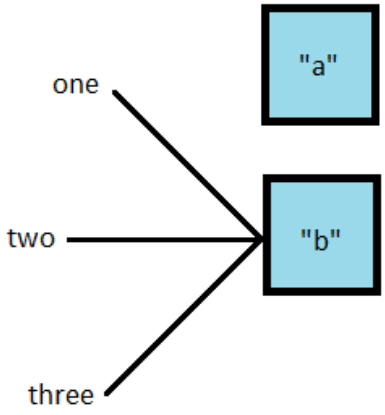
Comprender la diferencia entre una referencia y un objeto ayuda mucho a entender la Garbage Collection, el operador ***new*** y muchas otras facetas del lenguaje Java. Observando el siguiente código se puede ver cuándo un objeto es elegible para ser recolectado por el Garbage Collector:

```
1: public class Scope {
2:   public static void main(String[] args) {
3:     String one, two;
4:     one = new String("a");
5:     two = new String("b");
6:     one = two;
7:     String three = one;
8:     one = null;
9:  } }
```

A la hora de llevar a cabo una traza del código es posible cometer errores, por lo que habrá que prestar especial atención. En la línea 3 se crean dos variables: one y two. Sólo las variables, que son referencias que no apuntan todavía a nada. No hay necesidad de dibujar cajas o flechas que ayuden a comprender el código, ya que no hay objetos en el *heap* todavía. En la línea 4 se crea el primer objeto. Se dibujará una caja con la cadena "a" dentro y luego una flecha de la palabra one a esa caja. La línea 5 es similar. Se dibujará otra caja con la cadena "b" en ella esta vez y una flecha desde la palabra two. En este punto, el esquema debería verse como la figura siguiente.



En la línea 6, la variable one cambia a "b". Se borrará la flecha de one y se dibujará una nueva flecha de one a "b". En la línea 7, se tiene una nueva variable, así que se escribirá la palabra three y se dibujará una flecha de three a "b". Existen tres variables que señalan a "b" en este momento, y ninguna que señale a "a".



Finalmente habría que eliminar la flecha entre one y "b", ya que la línea 8 establece esta variable como nula. Este seguimiento pretendía encontrar cuándo los objetos eran elegibles para ser recolectados por el Garbage Collector. En la línea 6 se eliminó la única referencia que apuntaba a "a", haciendo que ese objeto fuera apto para el Garbage Collector. "B" tiene referencias apuntando hacia él hasta que sale del ámbito del código. Esto significa que "b" no es escogido por el GC hasta el final del método en la línea 9.

3.10.2. finalize()

Java permite a los objetos implementar un método llamado finalize(). Este método se llama si el Garbage Collector intenta recoger el objeto. Si el Grabage Collector no se ejecuta, el método no es llamado nunca. Si el Garbage Collector no puede recoger el objeto e intenta ejecutarse de nuevo más tarde, el método no se vuelve a llamar por segunda vez.

En la práctica, esto significa que es muy poco probable que se utilice en proyectos reales. Sólo hay que tener en cuenta que puede que no se llame y que definitivamente no será llamado dos veces.

Dicho esto, esta llamada no produce ninguna salida cuando la ejecutamos:

```
public class Finalizer {
  protected void finalize() {
    System.out.println("Calling finalize");
  }
  public static void main(String[] args) {
    Finalizer

    f = new Finalizer();
  }

}
```

La razón es que el programa termina antes de que exista la necesidad de ejecutar el Garbage Collector. A pesar de que f es elegible para la Garbage Collection, Java no ejecuta el Garbage Collector constantemente. Ahora un ejemplo más interesante:

```
public class Finalizer {
  private static List objects = new ArrayList();
  protected void finalize() {
    objects.add(this); // Don't do this
  }

}
```

Es preciso recordar que finalize() sólo se ejecuta cuando el objeto es elegible para el Garbage Collector. El problema aquí es que al final del método, el objeto ya no es elegible para el Garbage Collector porque una variable estática hace referencia a él, y las variables estáticas pertenecen al ámbito de la aplicación hasta que el programa termina. Java es lo suficientemente inteligente como para darse cuenta de esto y aborta el intento de eliminar el objeto. Ahora supóngase que más tarde, en el programa, los objetos

toman el valor de cero. Finalmente, se puede eliminar el objeto de la memoria. Java recuerda que ya se ha ejecutado finalize (en este objeto), y no lo hará de nuevo.

3.11. Beneficios de Java

Java tiene algunos beneficios clave que se necesitará conocer:

Orientado a Objetos:
Java es un lenguaje orientado a objetos, lo que significa que todo el código se define en clases y la mayoría de esas clases pueden ser instanciadas en objetos. Muchos idiomas antes de Java eran procedurales, lo que significaba que había rutinas o métodos, pero no había clases. Otro enfoque común es la programación funcional. Java permite la programación funcional dentro de una clase, pero el objeto orientado sigue siendo la organización principal del código.

Encapsulamiento:
Java soporta modificadores de acceso para proteger los datos de accesos y modificaciones no deseados. Se considera que la encapsulación es un aspecto de los lenguajes orientados a objetos.

Plataforma Independiente:
Java es un lenguaje interpretado porque se compila en bytecode. Una ventaja clave es que el código Java se compila una vez en lugar de tener que ser recompilado para diferentes sistemas operativos. Esto se conoce como "write once, run everywhere”.

Robusto:
Una de las principales ventajas de Java sobre C++ es que previene las fugas de memoria. Java maneja la memoria por su cuenta y ejecuta el Garbage Collector automáticamente. La mala gestión de la memoria en C++ es una gran fuente de errores en los programas.

Simple:
Java pretendía ser más simple que C++. Además de eliminar los punteros, eliminó la sobrecarga del operador. En C++, se podría escribir a + b y hacer que signifique casi cualquier cosa.

Seguro:
El código Java se ejecuta dentro del JVM. Esto crea una *sandbox* que dificulta que el código Java lleve a cabo acciones maliciosas en el ordenador en el que se está ejecutando.

3.12. Resumen

En este capítulo, se ha visto que:

Estructura de las clases Java

- Las clases Java consisten en miembros llamados campos y métodos.

-Un objeto es una instancia de una clase Java.

-Hay tres estilos de comentario: un comentario de una sola línea (//), un comentario de varias líneas (/* */), y un comentario Javadoc (/** */).

Método main

-Java inicia la ejecución del programa con un método main(). La cabecera más común para este método se ejecuta desde la línea de comandos: public static void main (String[] args).

-Los argumentos se pasan después del nombre de la clase, como en java NameOfClass firstArgument.

-Los argumentos son indexados comenzando con 0.

Paquetes e importaciones

- El código Java está organizado en carpetas llamadas paquetes.

- Para hacer referencia a clases en otros paquetes, se utilizará una declaración de importación.

- El carácter * al final de una sentencia de importación significa que desea importar todas las clases en ese paquete. No incluye paquetes que estén dentro de ese paquete. - - -java. lang es un paquete especial que no necesita ser importado.

Crear objetos Java

- Los constructores crean objetos Java.

- Un constructor es un método que corresponde al nombre de la clase y omite el tipo de devolución.

- Cuando un objeto es instanciado, los bloques de código se inicializan primero. Una vez inicializados, se puede hacer uso de los constructores.

Referencias a objetos y primitivas de datos

- Los tipos primitivos son los bloques básicos de construcción de los tipos Java. Se montan en tipos de referencia.

- Las clases de referencia pueden tener métodos y asignarse a nulo. Además de los números "normales", se permite que los literales numéricos comiencen con 0 (octal), 0x (hex), 0X (hex), 0b (binario), o 0B (binario).

- Los literales numéricos también pueden contener guiones bajos siempre y cuando estén directamente entre otros dos números.

Declarar e inicializar variables

- Declarar una variable implica indicar el tipo de datos y darle un nombre a la variable.

- Las variables que representan campos en una clase se inicializan automáticamente a su correspondiente valor cero o nulo durante la instanciación del objeto.

- Las variables locales deben ser inicializadas específicamente.

- Los identificadores pueden contener letras, números, $ o _. Los identificadores no pueden comenzar con números.

- El ámbito de aplicación se refiere a la parte del código en la que se puede acceder a una variable.

- Hay tres tipos de variables en Java, dependiendo de su alcance: variables de instancia, variables de clase y variables locales.

- Las variables de instancia son los campos no estáticos de su clase.

- Las variables de clase son los campos estáticos dentro de una clase.
- Las variables locales se declaran dentro de un método.

Ordenando elementos de una clase

- Para algunos elementos de la clase, el orden importa dentro del código.
- La declaración del paquete viene primero si está presente. Luego vienen las declaraciones de importación si están presentes. Luego viene la declaración de clase. Los campos y métodos pueden estar en cualquier orden dentro de la clase.

Destruyendo objetos

- El Garbage Collector es el responsable de retirar objetos de la memoria cuando no se pueden volver a utilizar.
- Un objeto se convierte en elegible para el Garbage Collector cuando no hay más referencias a él o sus referencias han salido del alcance.
- El método finalize() se ejecutará una vez para cada objeto cuando es ejecutado el Garbage Collector.

4. Operadores y sentencias

4.1. Entender los operadores de Java

Los operadores de java son símbolos especiales que se aplican un grupo de variables, valores o "literales" y que devuelven un valor. Existen tres tipos de operadores en Java: unario, binario y ternario. Estos tipos de operadores pueden ser aplicados a uno, dos o tres operandos.

Los operadores de java no siempre se evalúan de izquierda a derecha. En el siguiente ejemplo se puede observar como la expresión se evalúa de derecha a izquierda:

```
int y = 4;
double x = 3 + 2 * --y;
```

En el ejemplo primero se reduce en uno la variable y, después se multiplica por 2 y por último se le suman 3. El resultado final será convertido de 9 a 9.0 y asignado a la variable x. Los valores de x e y serán 9.0 y 3. A menos que estén entre paréntesis, los operadores de Java siguen un orden de operación que se encuentran listados en la tabla. Si dos operadores están en el mismo nivel se evaluarán de izquierda a derecha.

Operador	Símbolos y ejemplos
Operadores post-unarios	Expresión++, Expresión--
Operadores pre-unarios	++Expresión, --Expresión
Operadores unarios	+, -, !
Multiplicación, División, Módulo	*, /, %
Suma, Resta	+, -
Operadores de cambio	<<, >>, >>>
Operadores relacionales	<, >, <=, >=, instanceof
Igual, Distinto	==, !=
Operadores lógicos	&, ^, |
Operadores lógicos de cortocircuito	&&, ||
Operadores ternarios	Expresión booleana ? expresión1 : expresión 2
Operadores de asignación	=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

4.2. Trabajando con operadores binarios aritméticos

4.2.1. Operadores aritméticos

Los operadores aritméticos son los utilizados en las matemáticas y son los siguiente: la suma (+), la resta (-), la multiplicación (*), la división (/) y el módulo (%). También se incluyen los operadores unarios ++ y --. Como puedes comprobar en la tabla anterior los operadores *, / y % tienen mayor orden de preferencia que los operadores + y -, por lo que la siguiente expresión:

```
int x = 2 * 5 + 3 * 4 – 8;
```

Se evaluará primero 2*5 y 3*4 simplificando la expresión a la siguiente:

```
int x = 10 + 12 – 8;
```

Después se evaluará la expresión de izquierda a derecha dando como resultado 14.

Se puede cambiar el orden de la operación añadiendo paréntesis a las secciones que se quiere evaluar primero. Se compara el ejemplo de antes con el siguiente que incorpora paréntesis:

```
int x = 2 * ((5 + 3) * 4 – 8);
```

Esta vez se deberá evaluar la suma 5 + 3 primero, que reducirá la expresión a:

```
int x = 2 * (8 * 4 - 8);
```

Se podrá reducir la expresión multiplicando los dos primeros valores dentro del paréntesis:

```
int x = 2 * (32 - 8);
```

Después se restará los valores de dentro del paréntesis antes de multiplicarlos por el valor de fuera:

```
int x = 2 * 24;
```

Finalmente se multiplicarán los valores restantes, dando un resultado de 48.

Todos los operadores aritméticos pueden ser aplicados a cualquier tipo primitivo de Java, salvo los tipos boolean y string. Además, solo los operadores + y += pueden ser aplicados a variables string, lo que resulta ser una concatenación de string.

El módulo es el resto de la división de dos números, por ejemplo 9 divido entre 3 no tiene resto por lo que 9 % 3 será 0. Por otra parte 11 divido entre 3 si tiene resto por lo tanto 11 % 3 será igual a 2.

Hay que tener clara la diferencia entre la división y el módulo. La división devuelve el cociente, mientras que el módulo devuelve el resto de la división. Los siguientes ejemplos ilustran la diferencia entre ambas operaciones:

```
System.out.print(9 / 3);  // Outputs 3
System.out.print(9 % 3);  // Outputs 0
System.out.print(10 / 3); // Outputs 3
System.out.print(10 % 3); // Outputs 1
System.out.print(11 / 3); // Outputs 3
System.out.print(11 % 3); // Outputs 2
System.out.print(12 / 3); // Outputs 4
System.out.print(12 % 3); // Outputs 0
```

Hay que tener en cuenta que el resultado de la división solo se incrementa cuando el valor de la izquierda se incrementa de 9 a 12, mientras que el módulo incrementa en 1, cada vez que el valor de la izquierda se incrementa, hasta que se convierte en 0. Para un divisor dado y, que es 3 en estos ejemplos, la operación de módulo da como resultado un valor entre 0 y (y - 1) para dividendos positivos. Esto significa que el resultado de una operación de módulo es siempre 0,1 o 2.

El funcionamiento del módulo no se limita a los valores enteros positivos en Java y también puede aplicarse a números enteros negativos y números enteros de coma flotante. Para un divisor "y" y dividendo negativo dado, el valor del módulo resultante está entre (-y + 1) y 0.

4.2.2. Promoción numérica

Reglas de la promoción numérica:

1. Si dos variables tienes diferente tipo, Java automáticamente convertirá una de las variables al tipo más grande de las dos.

2. Si una de las variables es Integer y la otra de coma flotante(float), Java automáticamente convertirá la variable Integer en una de coma flotante.

3. Los tipos de datos pequeños como byte, short y char son convertidos a Int siempre que son usados por un operador aritmético binario, incluso si ninguno de los operandos es un Int.

4. Después de que se hayan convertido las variables y los operandos tengan el mismo tipo, el resultado se guardará con el mismo tipo al que se hayan convertido los operandos.

Las dos últimas reglas son aquellas con las que la mayoría de la gente tiene problemas. Por lo que respecta a la tercera regla, debe tenerse en cuenta que los operadores unitarios están excluidos de esta. Por ejemplo, aplicar ++ a un valor short dará como resultado un valor short.

4.3. Trabajando con operadores unarios

Por definición un operador unario es uno que requiere exactamente un operando o variable para funcionar. Como se muestra en la tabla suelen realizar tareas simples como incrementar en uno una variable o negar el valor de un boolean.

Operador unario	Descripción
+	Indica que un número es positivo, aunque se supone que los números son positivos en Java a menos que vayan acompañados de un operador negativo unario.
-	Indica que un número es negativo o niega una expresión
++	Incremento una variable en 1
--	Decremento de una variable en 1
!	Invierte el valor lógico de un boolean

4.3.1. Complemento lógico y operadores de negación

El operador de complemento lógico(!) invierte el valor de una expresión booleana. Por ejemplo, si el valor es true, se invertiría a false, y viceversa. Para demostrar esto, se comparará la salida de las siguientes instrucciones:

```
boolean x = false;
System.out.printl(x);     //false
x = !x;
System.out.println(x);   //true
```

Por otra parte, el operador de negación (-) cambia el signo de una expresión numérica como se muestra a continuación:

```
double x =

1.21;
System.out.println(x);   //1.21
x = -x;
System.out.println(x);   //-1.21
x = -x;
System.out.println(x);   //1.21
```

Si uno se basa en la descripción, podría ser obvio que algunos operadores requieran la variable de un tipo específico. No se puede utilizar un operador de negación, -, con una expresión booleana, ni se puede

aplicar un complemento lógico a una expresión numérica. Por ejemplo, ninguna de las siguientes líneas de código compilará:

```
int x = !5; // DOES NOT COMPILE
boolean y = -true; // DOES NOT COMPILE
boolean z = !0; // DOES NOT COMPILE
```

La primera declaración no se compilará debido al hecho de que en Java no se puede realizar una inversión lógica de un valor numérico. La segunda declaración no compilará porque no se puede negar numéricamente un valor booleano, es necesario utilizar el operador lógico inverso. Finalmente, la última sentencia no se compila porque no se puede tomar el complemento lógico de un valor numérico, ni se puede asignar un entero a una variable booleana.

4.3.2. Operadores de incremento y decremento.

Los operadores de incremento y decremento (++, --) pueden aplicarse a operandos numéricos y tienen mayor preferencia que los operadores binarios.

Los operadores de incremento y decremento requieren un cuidado especial ya que el orden en el que son utilizados con los operandos pueden provocar un procesamiento distinto de la expresión. Si el operador está situado delante del operando entonces el operador se aplica primero y luego se devuelve el valor. Por otro lado, si el operador se encuentra detrás del operando se devuelve primero el valor original y luego se aplica el operador. El siguiente código ilustra las diferencias:

```
int counter = 0;
System.out.println(counter); // Outputs 0
System.out.println(++counter); // Outputs 1
System.out.println(counter); // Outputs 1
System.out.println(counter--); // Outputs 1
System.out.println(counter); // Outputs 0
```

El primer operador de pre-incremento actualiza el valor del contador y devuelve el nuevo valor de 1. El siguiente operador de post-decremento también actualiza el valor del contador pero devuelve el valor antes de que se produzca la disminución. El siguiente ejemplo es todavía más complicado ya que se modifica el valor 3 veces en la misma línea:

```
int x = 3;
int y = ++x * 5 / x-- + --x;
System.out.println("x is " + x);
System.out.println("y is " + y);
```

Cada vez que se modifica, la expresión pasa de izquierda a derecha, el valor de x cambia, con diferentes valores asignados a la variable. Como recordarás de nuestra discusión sobre la precedencia del operador, el orden de la operación juega un papel importante en evaluando este ejemplo.

Primero, la x se incrementa y se devuelve a la expresión, que se multiplica por 5:

```
int y = 4 * 5 / x-- + --x; // x con valor 4
```

A continuación, se decrementa x, pero el valor original de 4 se utiliza en la expresión, lo que lleva a esto:

```
int y = 4 * 5 / 4 + --x; // x con valor 3
```

La asignación final de x reduce el valor a 2, y como se trata de un operador pre-incremental, se devuelve ese valor:

```
int y = 4 * 5 / 4 + 2; // x con valor 2
```

Finalmente, se realiza la multiplicación y a continuación la división, y por último la suma dando como resultado:

```
x is 2
y is 7
```

4.4. Usando los operadores binarios adicionales

4.4.1. Operadores de asignación

Los operadores de asignación son operadores binarios que modifican la variable con el valor del lado derecho de la ecuación. El operador de asignación más simple es el operador "=":

```
int x = 1;
```

Esta declaración asigna a x el valor 1.

Java transformará automáticamente de tipos de datos pequeños a grandes, como vimos en la sección anterior, pero lanzará una excepción de compilador si detecta que está intentando convertir tipos de datos grandes a pequeños.

Vamos a ver unos ejemplos para demostrar como el "casting" puede resolver estos problemas:

```
int x = 1.0; // DOES NOT COMPILE
short y = 1921222; // DOES NOT COMPILE
int z = 9f; // DOES NOT COMPILE
long t = 192301398193810323; // DOES NOT COMPILE
```

La primera sentencia no compila porque está tratando de asignar un double a un valor int. A pesar de que el valor es un entero, al sumar ". 0", está indicando al compilador que lo trate como un double. La segunda declaración no compila porque el valor 1921222 está fuera del rango de short. La tercera declaración no compila debido a la "f" añadida al final del número que indica al compilador que ha de tratar al número como un float. Finalmente, el último enunciado no compila porque Java interpreta el valor como un int y nota que es mayor de lo que permite int. El último caso necesitaría un postfix L para ser considerado un long.

4.4.2. Casting de valores primitivos

Se pueden arreglar los ejemplos de la sección anterior, haciendo "casting" a los resultados. El "casting" es necesario siempre que se pase de un dato numérico más grande a un tipo de datos numéricos más pequeños, o la conversión de un número float a un valor int.

```
int x = (int)1.0;
short y = (short)1921222; // Stored as 20678
int z = (int)9l;
long t = 192301398193810323L;
```

Algunos ejemplos:

```
short x = 10;
short y = 3;
short z = x * y; // DOES NOT COMPILE
```

Basando todo en lo que se ha aprendido hasta ahora, ¿por qué las últimas líneas de esta declaración no compilarán? Recordar, los valores short se transforman automáticamente a int con cualquier operador aritmético, con el valor resultante de tipo int. Tratar de establecer una variable corta en una int resulta en un error del compilador, como piensa Java, está intentando convertir implícitamente de un tipo de datos más grande a uno más pequeño.

Hay ocasiones en las que se puede querer anular el comportamiento predeterminado del compilador. Por ejemplo, en el ejemplo anterior, se sabe que el resultado de 10 * 3 es 30, que puede ser guardado en un short. Sin embargo, si se necesita que el resultado sea un short puede sustituirse este comportamiento haciendo "casting" del resultado de la multiplicación:

```
short x = 10;
short y = 3;
short z = (short)(x * y);
```

Al realizar este "casting" de un tipo de datos mayor a un tipo de datos más pequeño, se está ordenando al compilador que ignore su comportamiento predeterminado. En otras palabras, se le está diciendo al compilador que se está tomando medidas adicionales para evitar el overflow o el underflow.

Overflow y Underflow

Las expresiones en el ejemplo anterior ahora compilan, aunque hay un coste. El segundo valor, 1.921.222, es demasiado grande para ser almacenado como un valor short, por lo que se produce un overflow y se convierte en 20.678. El overflow es cuando un número es tan grande que ya no se puede guardar dentro de un tipo de datos, por lo que el sistema se "envuelve" en el siguiente valor mínimo y cuenta desde ahí. También hay un underflow analogo, cuando el número es demasiado bajo para guardarlo en el tipo de datos.

Por ejemplo, la siguiente sentencia genera un número negativo:

```
System.out.println(2147483647+1); // -214748364848
```

Dado que 2147483647 es el valor máximo de int, sumando cualquier valor estrictamente positivo a él se obtendrá lo siguiente al siguiente número negativo.

4.4.3. Operadores de asignación compuestos

Además del operador simple de asignación, =, existen también numerosos operadores de asignación compuestos. Sólo se requieren dos de los operadores compuestos enumerados en la Tabla 2.1, += y -=. Los operadores complejos son en realidad una mejora de los operadores de asignación simple con una operación aritmética o lógica incorporada que se aplican de izquierda a derecha de la expresión y almacena el valor resultante en la variable de la parte izquierda de la pantalla. Por ejemplo, las dos expresiones siguientes después de la declaración de x y z son equivalentes:

```
int x = 2, z = 3;
x = x * z; // Simple assignment operator
x *= z; // Compound assignment operator
```

El lado izquierdo del operador compuesto sólo se puede aplicar a una variable que ya esté definida y no se puede utilizar para declarar una nueva variable. En el ejemplo anterior, si x no estaba ya definida, entonces la expresión x *=z no compilaría.

Los operadores compuestos son útiles para algo más que la mera abreviatura, también se puede ahorrar el tener que hacer "casting" a un valor. Por ejemplo, si se considera el siguiente ejemplo en que la última línea no compilará debido a que el resultado es transformado a long y asignado a una variable int:

```
long x = 10;
int y = 5;
y = y * x; // DOES NOT COMPILE
```

Si uno se basa en las dos últimas secciones, se debería poder detectar el problema en la última línea. Esta última línea podría ser fijada con un "casting" a int, pero hay una manera mejor, usando el operador de asignación compuesto:

```
long x = 10;
int y = 5;
y *= x;
```

El operador compuesto hará un "casting" primero de x a long, aplicará la multiplicación de dos valores long, y luego devolverá el resultado a un int. A diferencia del ejemplo anterior, en el que el compilador lanzará una excepción, en este ejemplo vemos que el compilador hará un "casting" automáticamente del valor resultante al tipo de datos de la variable del lado izquierdo del operador compuesto.

Una cosa importante que se debe saber acerca del operador de asignación es que el resultado de la asignación es una expresión en sí misma, igual al valor de la asignación. Por ejemplo, el siguiente fragmento de código es perfectamente válido, aunque un poco extraño:

```
long x = 5;
long y = (x=3);
System.out.println(x); // Outputs 3
System.out.println(y); // Also, outputs 3
```

La clave aquí es que (x=3) hace dos cosas. En primer lugar, fija el valor de la variable x para que sea 3. En segundo lugar, devuelve el valor de la asignación, que también es 3.

4.4.4. Operadores relacionales

Ahora se pasará a los operadores relacionales, que comparan dos expresiones y devuelven un valor booleano. Los primeros cuatro operadores relacionales (ver la tabla se aplican solo a tipos de datos primitivos numéricos. Si los dos operando numéricos no son del mismo tipo de datos, el parámetro más pequeño se transforma de la manera anteriormente discutida.

Operador relacional	Descripción
<	Estrictamente menos que
<=	Menos que o igual a
>	Estrictamente mayor que
>=	Mayor que o igual a

A continuación, se verán unos ejemplos de estos operadores:

```
int x = 10, y = 20, z = 10;
System.out.println(x < y); // Outputs true
System.out.println(x <= y); // Outputs true
System.out.println(x >= z); // Outputs true
System.out.println(x > z); // Outputs false
```

Observar que el último ejemplo produce una salida false, porque, aunque x y z son el mismo valor, x no es estrictamente mayor que z.

El quinto operador relacional (ver la tabla de abajo) se aplica a objetos y clases, o bien a interfaces.

Operador relacional	Descripción
a instanceof b	Verdadero si la referencia a la que apunta “a” es una instancia de una clase, subclase o clase que implementa una interfaz particular, como se nombra en b

Cabe destacar lo siguiente respecto al operador instanceof:

```
String str = null;
System.out.println(str instanceof String );  // false
```

Es decir: no basta con declarar la variable como de un tipo sino que debe ser realmente un objeto de ese tipo (como consecuencia, no es necesario comparar contra null).

```
if (x  != null && x instanceof X) ...
```

Se puede simplificar a:

```
if (x instanceof X) ...
```

4.4.5. Operadores lógicos

Los operadores lógicos, (&), (|) y (^), se pueden aplicar a datos de tipo numéricos y booleanos. Cuando se aplican a tipos de datos booleanos, se los denomina operadores lógicos. Alternativamente, cuando se aplican a tipos de datos numéricos, se les llama operadores a nivel de bit, ya que realizan comparaciones bit a bit de los bits que componen el número.

Se debe familiarizar con las tablas de las figuras, donde se supone que x e y son tipos de datos booleanos.

	Y=true	Y=false
X = true	True	False
X = false	False	False

X & Y (AND)

	Y=true	Y=false
X = true	True	True
X = false	True	False

X | Y (Inclusive OR)

	Y=true	Y=false
X = true	False	True
X = false	True	False

X ^ Y (Exclusive OR)

Aquí hay algunos consejos para ayudar a recordar esta tabla:

- AND solo es verdadero si ambos operandos son verdaderos.
- OR inclusivo solo es falso si ambos operandos son falsos.
- Exclusive OR solo es verdadero si los operandos son diferentes.

Finalmente, se presentan los operadores condicionales, && y ||, que a menudo se conocen como operadores de cortocircuito. Los operadores de cortocircuito son casi idénticos a los operadores lógicos, & y |, respectivamente, excepto que el lado derecho de la expresión nunca puede ser evaluado si el

resultado final puede ser determinado por el lado izquierdo de la expresión. Por ejemplo, si se considera la siguiente declaración:

```
boolean x = true || (y < 4);
```

En referencia a las tablas de verdad, el valor x solo puede ser falso si ambos lados de la expresión son falsas. Como sabemos que el lado izquierdo es verdadero, no hay necesidad de evaluar el lado derecho, ya que nada hará que el valor de x sea diferente de verdadero. Para ilustrar este concepto prueba a ejecutar la línea de código anterior para varios valores de y.

El ejemplo más común de dónde se usan los operadores de cortocircuito es la comprobación de objetos nulos antes de realizar una operación, como esta:

```
if(x != null && x.getValue() < 5) {
  // Do something
}
```

En este ejemplo, si x fuera nulo, entonces el cortocircuito evita lanzar una excepción NullPointerException, ya que la evaluación de x.getValue () <5 nunca se alcanza. Alternativamente, si usamos un & lógico, entonces ambos lados siempre se evaluarán y cuando x fuese nulo esto arrojaría una excepción:

```
if(x != null & x.getValue() < 5) { // Throws an exception if x is null
  // Do something
}
```

¿Cuál es el resultado del siguiente código?

```
int x = 6;
boolean y = (x >= 6) || (++x <= 7);
System.out.println(x);
```

Como x >= 6 es verdadero, el operador de incremento en el lado derecho de la expresión nunca se evalúa, por lo que la salida es 6.

4.4.6. Operadores de igualdad

La determinación de la igualdad en Java puede no ser trivial, ya que hay una diferencia semántica entre "dos objetos son lo mismo" y "dos objetos son equivalentes". Es aún más complicado por el hecho de que para tipos primitivos numéricos (entre los que se incluye también el tipo de dato char) y booleanos, no existe tal distinción.

Comenzando por lo más sencillo: el operador de igualdad (==) y el operador de desigualdad (!=). Como los operadores relacionales, comparan dos operandos y devuelven un valor booleano si las expresiones o valores son iguales o no iguales, respectivamente.

Los operadores de igualdad se utilizan en uno de tres casos:

1. Comparando dos tipos primitivos numéricos. Si los valores numéricos son de diferente tipo de datos, los valores se transforman automáticamente como se describió anteriormente. Por ejemplo, 5 == 5.00 devuelve verdadero ya que el lado izquierdo se transforma a un double.
2. Comparando dos valores booleanos.
3. Comparando dos objetos, incluidos los valores nulos y String.

Las comparaciones para la igualdad se limitan a estos tres casos, por lo que no se pueden mezclar tipos. Por ejemplo, cada uno de los siguientes devolvería en un error del compilador:

```
boolean x = true == 3; // DOES NOT COMPILE
boolean y = false != "Giraffe"; // DOES NOT COMPILE
boolean z = 3 == "Kangaroo"; // DOES NOT COMPILE
```

Si se observa el siguiente fragmento:

```
boolean y = false;
boolean x = (y = true);
System.out.println(x); // Outputs true
```

A primera vista, se podría pensar que la salida debería ser falsa, y si la expresión fuese (y == true), se tendría razón. En este ejemplo, sin embargo, la expresión está asignando el valor true a y, y como se vio en la sección de operadores de asignación, la asignación misma tiene el valor de la asignación. Por lo tanto, la salida sería verdadera.

Para la comparación de objetos, el operador de igualdad se aplica a las referencias a estos objetos, no a los objetos a los que apuntan. Dos referencias son iguales si y solo si apuntan a lo mismo objeto, o ambos apuntan a nulo. Si se ven algunos ejemplos se entenderá mejor:

```
File x = new File("myFile.txt");
File y = new File("myFile.txt");
File z = x;
System.out.println(x == y); // Outputs false
System.out.println(x == z); // Outputs true
```

Aunque todas las variables apuntan a la misma información de archivo, solo dos, x y z, son iguales en términos de ==.

En el Capítulo 3, " Core Java APIs", se continuará la discusión sobre la igualdad de objetos introduciendo lo que significa que dos objetos diferentes sean equivalentes. También se tratará la igualdad de los String y se mostrará cómo este puede ser un tema no trivial.

4.5. Comprender las sentencias de Java 1

Los operadores Java permiten crear muchas expresiones complejas, pero están limitadas en la manera en que pueden controlar el flujo del programa. Por ejemplo, imaginar que se quiere una sección de código que solo sea ejecutado bajo ciertas condiciones que no pueden ser evaluadas hasta que no se ejecute. O si se desea que un segmento particular de código se repita una vez por cada elemento en alguna lista.

Como se dijo en el Capítulo 1, una declaración de Java es una unidad completa de ejecución en Java, terminando con un punto y una coma (;). En el capítulo, se mostrarán varias declaraciones de control de flujo en Java. Las declaraciones de control de flujo rompen el flujo de ejecución mediante la toma de decisiones, el bucle y la ramificación, permitiendo que la aplicación seleccione segmentos particulares del código para ejecutar.

Las declaraciones se pueden aplicar a expresiones simples, o a un bloque de código. Un bloque de código en Java es un grupo de cero o más declaraciones entre llaves, ({}), y se puede usar en cualquier lugar donde se permita usar una declaración simple.

4.5.1. if-then

De vez en cuando, solo se quiere ejecutar un bloque de código según ciertas condiciones. El if-then, como se muestra en el código siguiente, permite que la aplicación ejecute un bloque particular de código si y solo si una expresión booleana se evalúa como verdadera en tiempo de ejecución.

```
if(booleanExpression){
  //Branch if true
}
```

Por ejemplo, si se tiene una función que utiliza la hora del día para mostrar un mensaje al usuario:

```
if(hourOfDay < 11)
  System.out.println(“Good Morning”);
```

Si la hora es menor que 11 entonces la función mostrará el mensaje. Si se quisiera además incrementar algún valor, morningGreetingCount, cada vez que el mensaje se muestra se podría repetir la declaración if-then, pero Java permite escribirlo a continuación del output convirtiendo el código en un bloque:

```
if(hourOfDay < 11){
  System.out.println(“Good Morning”);
  morningGreetingCount++;
}
```

El bloque permite que se ejecuten múltiples instrucciones basadas en la evaluación del if-then. Hay que tener en cuenta que la primera instrucción no contiene un bloque dentro de la sección de impresión, pero podría tenerlo. Para mejorar la legibilidad del código, se considera una buena práctica colocar los bloques dentro de las sentencias if-then, así como muchas otras sentencias de control de flujo, aunque no es obligatorio.

a. Sangría y llaves

Echar un vistazo a esta forma ligeramente modificada del ejemplo:

```
if(hourOfDay < 11)
  System.out.println("Good Morning");
  morningGreetingCount++;
```

Si se observan las sangrías, se puede pensar que la variable morningGreetingCount solo se va a incrementar cuando la hora sea menor que 11, pero no es lo que hace el código. Mostrará el mensaje solo si la hora es menor que 11 pero incrementará el valor de la variable siempre. Recuerda que en Java los espacios en blanco no son considerados parte de la ejecución.

4.5.2. if-then-else

Se va a complicar el ejemplo anterior, ¿qué pasaría si se quisiera mostrar otro mensaje cuando la hora sea mayor o igual a 11?

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
}
if(hourOfDay >= 11) {
  System.out.println("Good Afternoon");
}
```

Esto es un poco redundante, ya que estamos evaluando hourOfDay dos veces, y esto puede ser computacionalmente costoso. Java permite solucionar esto gracias a la sentencia if-then-else que se muestra en el siguiente código:

```
if(booleanExpression){
  // Branch if true
} else {
  // Branch if false
}
```

Veamos este ejemplo:

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else {
  System.out.println("Good Afternoon");
}
```

Ahora el código se está ramificando entre una de las dos opciones posibles, con la evaluación booleana solo una vez. El operador else utiliza una declaración o un bloque, de la misma manera que la sentencia if. De esta manera, se puede agregar declaraciones if-then adicionales a un bloque else para llegar a un ejemplo más refinado:

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else {
  System.out.println("Good Evening");
}
```

En este ejemplo, el proceso de Java continuará la ejecución hasta que encuentre un if-then que se evalúe como verdadero. Si ninguna de las dos primeras expresiones es verdadera, se ejecutará el código del bloque else del final. Una cosa a tener en cuenta al crear complejas declaraciones if-then-else es que el orden es importante. Por ejemplo, si se reordena el fragmento de código anterior de la siguiente manera:

```
if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else if(hourOfDay < 11) {
  System.out.println("Good Morning"); // UNREACHABLE CODE
} else {
  System.out.println("Good Evening");
}
```

Para horas menores que 11, este código se comporta de forma diferente al anterior. Si un valor es menor que 11, será también menor que 15. Por lo tanto, si se puede alcanzar la segunda rama en el ejemplo, también se puede alcanzar la primera rama. Ya que la ejecución de cada rama es excluyente, solo una rama se puede ejecutar, si se ejecuta la primera rama no se podrá ejecutar la segunda.

Por lo tanto, no hay forma de que la segunda rama se ejecute alguna vez, y el código se considera inalcanzable.

a. Verificando si la sentencia if se evalúa en una expresión booleana

Echar un vistazo a las siguientes líneas de código:

```
int x = 1;
if(x) { // DOES NOT COMPILE
...
}
```

Esta declaración puede ser válida en algunos otros lenguajes de programación y scripting, pero no en Java, donde 0 y 1 no se consideran valores booleanos. Además, hay que tener cuidado con los operadores de asignación que se utilizan como si fueran operadores iguales (==) en declaraciones if-then:

```
int x = 1;
if(x = 5) { // DOES NOT COMPILE
...
}
```

4.5.3. Operador Ternario

Ahora que se ha visto las declaraciones if-then-else, se puede volver brevemente a la discusión de los operadores y presentar al último operador. El operador condicional, ? :, también conocido como operador ternario, es el único operador que utiliza tres operandos y tiene la forma:

```
booleanExpression ? expression1 : expression2
```

El primer operando debe ser una expresión booleana, y el segundo y el tercero pueden ser cualquier expresión que devuelva un valor. La operación ternaria es realmente una forma condensada de un if-then-else que devuelve un valor. Por ejemplo, los siguientes dos fragmentos de código son equivalentes:

```
int y = 10;
final int x;
if(y > 5) {
  x = 2 * y;
} else {
  x = 3 * y;
}
```

Comparar el código anterior con el siguiente código equivalente con el operador ternario:

```
int y = 10;
int x = (y > 5) ? (2 * y) : (3 * y);
```

Tener en cuenta que a menudo es útil para la legibilidad agregar paréntesis alrededor de las expresiones en operaciones ternarias, aunque no es necesario.

No es necesario que la segunda y tercera expresiones tengan los mismos tipos de datos, aunque puede entrar en juego cuando se combina con la asignación operador. Comparar las siguientes dos afirmaciones:

```
System.out.println((y > 5) ? 21 : "Zebra");
int animal = (y < 91) ? 9 : "Horse"; // DOES NOT COMPILE
```

Ambas expresiones evalúan valores booleanos similares y devuelven un int y un String, aunque solo se compilará la primera línea. El System.out.println () no se preocupa de que las sentencias sean de tipos completamente diferentes, ya que pueden convertir ambos a String. Por otro lado, el compilador sabe que "Horse" es de un tipo de datos incorrecto y no puede ser asignado a un int; por lo tanto, no permitirá que se compile el código.

a. Evaluación de las expresiones ternarias

A partir de Java 7, solo una de las expresiones de la derecha del operador ternario será evaluado en tiempo de ejecución. De manera similar a los operadores de cortocircuito, si una de las dos expresiones de la derecha del operador ternario realiza un efecto secundario, entonces no se puede aplicar en tiempo de ejecución. Vamos a ilustrar este principio con el siguiente ejemplo:

```
int y = 1;
int z = 1;
final int x = y<10 ? y++ : z++;
System.out.println(y+","+z); // Outputs 2,1
```

Tener en cuenta que como la parte de la izquierda de la expresión es verdadera solo se incrementa y. Contrástar con el siguiente ejemplo:

```
int y = 1;
int z = 1;
final int x = y>=10 ? y++ : z++;
System.out.println(y+","+z); // Outputs 1,2
```

Ahora que la expresión booleana de la izquierda se evalúa como falsa, solo z se incrementa. De esta manera, se ve cómo las expresiones en un operador ternario pueden no aplicarse si la expresión particular no se usa.

4.5.4. switch

Una sentencia switch, como se muestra en la Figura 2.4, es una estructura compleja de toma de decisiones en el que se evalúa un solo valor y el flujo se redirige a la primera rama correspondiente, conocida como "case". Si no se encuentra dicha declaración de caso que coincida con el valor, una opción predeterminada será ejecutada. Si no hay tal opción predeterminada disponible, la totalidad de la sentencia switch será omitida.

a. Tipos de datos soportados

Como se muestra en el siguiente código, una instrucción switch tiene una variable que se evalúa en tiempo de ejecución. Antes de Java 5.0, esta variable solo podía ser valores int o aquellos valores que podían ser transformados a int, de forma específica byte, short, char o int.

Cuando se agregó "enum" en Java 5.0, se agregó el soporte para que los switch pudiesen admitir valores "enum". En Java 7, las sentencias switch se actualizaron aún más para permitir los valores string. Finalmente, la sentencia switch es compatible con cualquiera de las clases primitivas de carácter numérico, como Byte, Short, Character o Integer.

```
switch(variableToTest) {
  case constantExpression1:
    // Branch for case1;
    break;
  case constantExpression2:
    // Branch for case2;
    break;
  …
```

```
  default:
}
```

Tipos de datos soportados por la sentencia switch:

- int y Integer
- byte y Byte
- short y Short
- char y Character
- String
- Valores enum

b. Valores constantes en tiempo de compilación

Los valores de cada case deben ser del mismo tipo de dato que el valor introducido en el switch. Esto significa que solo se pueden utilizar literales, constantes enum o variables constantes finales del mismo tipo de datos.

Ejemplo:

```
int dayOfWeek = 5;
switch(dayOfWeek) {
  default:
    System.out.println("Weekday");
    break;
  case 0:
    System.out.println("Sunday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
}
```

Con el valor de dayOfWeek de 5 el código devolverá “Weekday”.

Lo primero que se puede notar es que hay una sentencia break al final de cada case y la sección predeterminada(default). Se discutirá las sentencias break en detalle cuando se vean los bucles, pero por ahora, todo lo que se necesita saber es que finaliza la sentencia switch y el control de flujo vuelve a la sentencia adjunta. Como se verá pronto, si se omite la declaración del break, el flujo continuará hasta el siguiente caso en curso o bloque predeterminado.

Otra cosa que se puede notar es que el bloque predeterminado no está al final del switch. No es un requisito que el caso o las declaraciones predeterminadas estén en un orden en particular, a menos que tenga vías que lleguen a múltiples secciones del bloque switch en una sola ejecución.

Para ilustrar lo aprendido se considerará la siguiente variación:

```
int dayOfWeek = 5;
switch(dayOfWeek) {
  case 0:
```

```
    System.out.println("Sunday");
  default:
    System.out.println("Weekday");
  case 6:
    System.out.println("Saturday");
    break;
}
```

Este código se parece mucho al ejemplo anterior, excepto que dos de las declaraciones break han sido eliminadas y el orden ha cambiado. Esto significa que por el valor dado de dayOfWeek, 5, el código saltará al bloque predeterminado y luego ejecutará todos los case en orden hasta que encuentre una sentencia break o finalice la estructura.

El orden de los case y del bloque predeterminado es importante ahora ya que dejando el bloque predeterminado al final del switch, este, solo devolverá una palabra. Si el valor de dayOfWeek fuese 6 el switch devolvería "Saturday".

Aunque el bloque predeterminado estaba antes del bloque case, solo se ejecutó el bloque case. Si se recuerda la definición del bloque predeterminado, solo se ramificará si no ha coincidido el valor del caso con el valor del switch, independientemente de su posición dentro del switch.

Por último, si el valor de dayOfWeek fuese 0, la salida mostraría:

```
Sunday
Weekday
Saturday
```

Tener en cuenta que, en este último ejemplo, se ejecuta el bloque predeterminado porque no hay una instrucción break al final de los bloques de case anteriores. Mientras el código no se ramifique hacia la sentencia predeterminada si hay un valor de case que coincida dentro de la declaración del switch, se ejecutará la instrucción predeterminada si se encuentra después de una declaración de case para la que no hay declaración de break.

Conclusión, se acepta que el tipo de datos para las declaraciones de case, debe coincidir con el tipo de datos de la variable del switch. Como ya se discutió, el valor de la declaración de case también debe ser una constante literal, enum o una variable final. Por ejemplo, dada la siguiente instrucción switch, observar qué afirmaciones de case compilarán y cuales no:

```
private int getSortOrder(String firstName, final String lastName) {
  String middleName = "Patricia";
  final String suffix = "JR";
  int id = 0;
  switch(firstName){
    case "Test":
      return 52;
    case middleName: // DOES NOT COMPILE
      id = 5;
```

```
      break;
    case suffix:
      id = 0;
      break;
    case lastName: // DOES NOT COMPILE
      id = 8;
      break;
    case 5: // DOES NOT COMPILE
      id = 7;
      break;
    case 'J': // DOES NOT COMPILE
      id = 10;
      break;
    case java.time.DayOfWeek.SUNDAY: // DOES NOT COMPILE
      id = 15;
      break;
  }
  return id;
}
```

La primera declaración de case compila sin problemas usando un String, y es un buen ejemplo de cómo una declaración de return, se puede usar para salir del switch. La segunda declaración de case no se compila porque middleName no es una variable final, a pesar de tener un valor conocido en esta línea de ejecución en particular. La tercera declaración compila sin problema porque suffix es una variable constante final.

La cuarta declaración, aunque LastName sea final, no es una constante ya que se ha pasado a la función, por lo tanto, esta línea tampoco compila. Finalmente, los tres últimos case no compilan porque ninguno de ellos coincide con el tipo String de la variable del switch, el último es de tipo enum value.

4.5.5. while

Una estructura de control de repetición, también conocida como bucle, ejecuta el mismo código varias veces seguidas. Mediante el uso de variables no constantes, cada repetición de la sentencia puede ser diferente. Por ejemplo, una declaración que se itera sobre una lista de nombres únicos y las salidas devolverán un nuevo nombre en cada ejecución del bucle.

La estructura de control de repetición más simple en Java es la instrucción while, descrita en el siguiente código. Como todas las estructuras de control de repetición, tiene una condición de terminación, implementada como una expresión booleana, que continuará mientras la expresión se evalúe a true.

```
while(booleanExpression){
  //Body
}
```

Como se muestra en el código, un bucle while es similar a las sentencias if-then ya que está compuesta por una expresión booleana e instrucciones o bloque de instrucciones. Durante la ejecución, la expresión booleana es evaluada antes de cada iteración del bucle y termina si la evaluación devuelve un false. Es importante darse cuenta que el bucle while puede terminar después de su primera evaluación de la expresión booleana. De esta forma el bloque de instrucciones puede no ejecutarse nunca.

Si se vuelve al ejemplo de ratón del capítulo 3 y se muestra un bucle que puede ser utilizado para modelar un ratón comiendo una comida:

```
int roomInBelly = 5;
public void eatCheese(int bitesOfCheese) {
  while(bitesOfCheese > 0 && roomInBelly > 0) {
    bitesOfCheese--;
    roomInBelly--;
  }
  System.out.println(bitesOfCheese+" pieces of cheese left");
}
```

Este método coge una cantidad de comida, en este caso de queso, y continua hasta que el ratón no tenga más espacio en su estómago o no quede comida para comer. Con cada iteración del bucle, el ratón come un trozo de comida y pierde una parte del espacio en su estómago. Utilizando una sentencia booleana compuesta, nos aseguramos de que el bucle while pueda terminar por cualquiera de las condiciones.

a. Bucles infinitos

Considerar el siguiente código:

```
int x = 2;
int y = 5;
while(x < 10)
  y++;
```

Se puede observar un evidente problema con esta declaración: nunca acabará. La expresión booleana que se evalúa antes de cada iteración del bucle nunca se modifica por lo que la expresión (x < 10) siempre será evaluada a true. El resultado es que el bucle nunca acabará creando lo que se conoce como bucle infinito.

Los bucles infinitos son algo que se debe tener en cuenta al crear bucles. Se tiene que estar seguro de que el bucle termina bajo alguna condición. Primero comprobar que la variable del bucle se modifica. Luego, asegurarse que la condición de terminación se alcanza bajo cualquier circunstancia. Como se verá en "Entendiendo los flujos de control avanzados" de un bucle se puede salir también con otras condiciones como con la sentencia break.

4.5.6. do-while

Java también permite crear bucles do-while, que como al bucle while, es una estructura de control de repetición con una condición de terminación e instrucciones o bloques de instrucciones, como se muestra en el siguiente código. Al contrario que el bucle while, el bucle do-while garantiza que la instrucción o el bloque de instrucciones se ejecutará mínimo una vez.

```
do{
  //Body
}while(booleanExpression);
```

La principal diferencia entre la estructura del do-while y del while es que ordena intencionadamente las instrucciones o los bloques de instrucciones antes de la expresión condicional, para recalcar que la instrucción será ejecutada antes de evaluar la expresión. Por ejemplo, mirar la salida del siguiente código:

```
int x = 0;
do {
  x++;
}while(false);
System.out.println(x); // Outputs 1
```

Java ejecutará primero el bloque de instrucciones, y luego comprobará la condición del bucle. A pesar de que el bucle termina inmediatamente, el bloque de instrucciones se ejecuta una vez y el programa devuelve 1.

a. Cuando usar bucles while o do-while

En la práctica, puede ser difícil determinar cuándo se debe utilizar un bucle while y un bucle do-while. No importa cual se utilice ya que cualquier bucle while se puede convertir en un bucle do-while y viceversa. Comparar estos dos bucles:

```
while(x > 10) {
  x--;
}
if(x > 10) {
  do {
    x--;
  }while(x > 10);
}
```

Aunque uno de los bucles es más fácil de leer, son funcionalmente iguales. Java recomienda utilizar los bucles while cuando no sea necesario ejecutar sus instrucciones, y utilizar los bucles do-while cuando sea imprescindible que se ejecute como mínimo una vez, pero, en la práctica, elegir entre uno u otro es algo personal.

Por ejemplo, pese a que la primera declaración es más corta, la segunda tiene una ventaja que permite hacer uso de la sentencia if-then y realizar otras operaciones en la rama del else, como se muestra en el ejemplo:

```
if(x > 10) {
  do {
    x--;
  }while(x > 10);
}

else {
  x++;
}
```

4.5.7. La sentencia for

Ahora se extenderá el conocimiento con otra estructura de control repetición llamada "for". Existen dos tipos de sentencia for, la primera se refiere al bucle simple for, la segunda se llama for-each que es una mejora de la sentencia for.

a. La sentencia for simple

Un bucle for simple tiene la misma condición booleana o bloque de instrucciones que los otros bucles que hemos visto, e incluye dos nuevas secciones: la inicialización y la actualización. El siguiente código muestra como estos componentes están organizados:

```
for(initialization; booleanExpression; updateStatement) {
  // Body
}
```

El código puede parecer algo confusa y arbitraria al principio, la organización de los componentes y el flujo permite crear poderosas sentencias en poco espacio al contrario de otros bucles que necesitarían múltiples líneas. Fijarse que cada sección está separada por punto y coma (;) y la inicialización y la actualización pueden contener varias sentencias separadas por comas.

Las variables creadas en el bloque de inicialización tienen un ámbito limitado y solo serán accesibles dentro del bucle for. Alternativamente, las variables declaradas antes del for y modificadas en el bloque de inicialización pueden ser utilizadas fuera del bucle for ya que su ámbito es anterior a la creación del bucle for.

A continuación, este ejemplo imprime en pantalla los números del 0 al 9:

```
for(int i = 0; i < 10; i++) {
  System.out.print(i + " ");
}
```

La variable local i se inicializa a 0. Esta variable tiene un ámbito dentro del bucle y no es accesible desde fuera del bucle una vez el bucle haya terminado. Como en los bucles while, la expresión booleana es

evaluada en cada iteración del bucle antes de que se ejecuten las instrucciones del este. Si devuelve un true, el bucle se ejecuta e imprime el 0 seguido de un espacio en blanco. Luego el bucle ejecuta el bloque de actualización, que en este caso incrementa el valor de i a 1. Entonces el bucle evalúa otra vez la expresión booleana y el proceso se repite varias veces, imprimiendo lo siguiente:

```
0 1 2 3 4 5 6 7 8 9
```

En la décima iteración del bucle, el valor de i alcanza el 9 y se incrementa a 10. En la undécima iteración del bucle la expresión booleana será evaluada a falso ya que 10 no es menor que 10, el bucle terminará sin ejecutar el bloque de instrucciones.

Ejemplos con los que familiarizarse:

- Creación de un bucle infinito

```
for( ; ; ) {
  System.out.println("Hello World");
}
```

Puede parecer que este bucle for dará problemas de compilación, pero compilará y funcionará sin problemas. Es un bucle infinito que imprimirá el mismo mensaje repetidas veces. Este ejemplo refuerza la idea de que los componentes del bucle for son opcionales, y recuerda que el punto y coma que separa las secciones son necesarios, for() no compilará.

- Adición de varios términos a la sentencia for
-

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(y + " ");
}
System.out.print(x);
```

Este código demuestra tres variaciones del bucle for que puede que aún no se hayan visto. Primero, se puede declarar una variable, como x en el ejemplo, antes de que el bucle empiece y usarla después de que termine. Segundo, el bloque de inicialización, la expresión booleana, y el bloque de actualización, pueden incluir variables extra que pueden no referenciarse entre ellas. Por ejemplo, z está definida en el bloque de inicialización y nunca se usa. Finalmente, el bloque de actualización puede modificar múltiples variables. El código imprimirá:

```
0 1 2 3 4
```

- Re-declarando una variable en el bloque de inicialización

```
int x = 0;
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) { // DOES NOT COMPILE
  System.out.print(x + " ");
}
```

Este ejemplo parece similar al anterior, pero no compila debido al bloque de inicialización. La diferencia es que x se repite en el bloque de inicialización después de haber sido declarado ya antes del loop, dando lugar a que el compilador se detenga debido a una declaración de variable duplicada. Se puede fijar este bucle cambiando la declaración de x y y como sigue:

```
int x = 0;
long y = 10;
for(y = 0, x = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(x + " ");
}
```

Tener en cuenta que esta variación si compilará porque el bloque de inicialización simplemente asigna un valor a x y no lo declara.

- Utilizando tipos de datos incompatibles en el bloque de inicialización

```
for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) { // DOES NOT COMPILE
  System.out.print(x + " ");
}
```

Este ejemplo también se parece mucho al segundo ejemplo, pero como en el tercer ejemplo no compilará, aunque esta vez por una razón diferente. Las variables del bloque de inicialización deben ser del mismo tipo. En el primer ejemplo, y y z eran ambos long, así que el código compilará sin problema, pero en este ejemplo tienen tipos diferentes, por lo que el código no se compilará.

- Usando variables del bucle fuera de este

```
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(y + " ");
}
System.out.print(x); // DOES NOT COMPILE
```

La variación final del segundo ejemplo no compilará por una razón diferente a la de los ejemplos anteriores. Fijarse que x se define en el bloque de inicialización del bucle, y luego se usa después de que el bucle termine. Puesto que x sólo tiene ámbito para el bucle, usarlo fuera del bucle lanzará un error de compilación.

4.5.8. La sentencia for-each

A partir de Java 5.0, los desarrolladores han tenido una mejora para el bucle for a su disposición, uno específicamente diseñado para iterar sobre arrays y colección de objetos. Esto mejora el bucle for, que para mayor claridad se hará referencia a él como un bucle for-each, como se muestra en el siguiente código:

```
for(datatype instance : collection){
  //Body
}
```

La declaración del bucle for-each está compuesta por una sección de inicialización y un objeto para utilizarlo repetidas veces. El lado derecho de la sentencia for-each debe ser un array o un objeto que implemente la clase Java.lang.Iterable, que incluye la mayor parte de los frameworks de Java Collections. El lado izquierdo del bucle for-each debe incluir una declaración para una instancia de una variable, cuyo tipo coincida con el tipo de un miembro del array o de la colección en el lado derecho de la sentencia.

Ejemplos:

```
final String[] names = new String[3];
names[0] = "Lisa";
names[1] = "Kevin";
names[2] = "Roger";
for(String name : names) {
  System.out.print(name + ", ");
}
```

Este código compilará e imprimirá: Lisa, Kevin, Roger,

```
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
for(String value : values) {
  System.out.print(value + ", ");
}
```

Este código compilará e imprimirá: Lisa, Kevin, Roger,

```
String names = "Lisa";
for(String name : names) { // DOES NOT COMPILE
  System.out.print(name + " ");
}
```

En este ejemplo, el String names no es un array y no implementa la clase java.lang.iterable, así que el compilador lanzará una excepción ya que no sabe como iterar un String.

```
String[] names = new String[3];
for(int name : names) { // DOES NOT COMPILE
  System.out.print(name + " ");
}
```

Este código no compilará porque en el lado izquierdo del bucle no se define una instancia de String. Nótese que, en este último ejemplo, el array se inicializa con tres valores de puntero cero. En sí mismo, eso no causará un fallo de compilación, sólo imprimirá tres veces cero.

a. Comparación del bucle for y el bucle for-each

Dado que for y for-each usan la misma palabra clave, es posible preguntarse cómo están relacionados. Tomar un momento para explorar cómo el compilador convierte cada for-each en bucles for.

Cuando se introdujo el bucle for-each en Java 5, se agregó como una mejora de compilación. Esto significa que Java realmente convierte el bucle for-each en un bucle for estándar durante compilación. Por ejemplo, asumiendo que names es un array de String[] como se vio en la primera parte. Ejemplo, los dos bucles siguientes son equivalentes:

```
for(String name : names) {
  System.out.print(name + ", ");
}
for(int i=0; i < names.length; i++) {
  String name = names[i];
  System.out.print(name + ", ");
}
```

Para los objetos que heredan de Java.lang.iterable, hay una conversión diferente, pero similar. Por ejemplo, asumiendo que los valores son una instancia de List<Integer>, como vimos en el segundo caso ejemplo, los dos bucles siguientes son equivalentes:

```
for(int value : values) {
  System.out.print(value + ", ");
}
for(java.util.Iterator<Integer> i = values.iterator(); i.hasNext(); ) {
  int value = i.next();
  System.out.print(value + ", ");
}
```

Observar que, en la segunda versión, no hay declaración de actualización ya que no se requiere cuando se usa la clase java.util.Iterator.

Es posible que se haya notado que, en los ejemplos anteriores, había una coma extra impresa al final de la lista:

```
Lisa, Kevin, Roger,
```

Mientras que la declaración for-each es conveniente para trabajar con listas en muchos casos, oculta el acceso a la variable iterator del bucle. Si se quisiera imprimir sólo la coma entre nombres, se podría convertir el ejemplo en un bucle for estándar, como en el ejemplo siguiente:

```
java.util.List<String> names = new java.util.ArrayList<String>();
names.add("Lisa");
names.add("Kevin");
names.add("Roger");
for(int i=0; i<names.size(); i++) {
  String name = names.get(i);
  if(i>0) {
    System.out.print(", ");
  }
  System.out.print(name);
}
```

Este código de muestra produciría lo siguiente:

```
Lisa, Kevin, Roger
```

También es común utilizar un bucle for sobre un bucle for-each si se comparan múltiples elementos en un bucle dentro de una sola iteración, como en el ejemplo siguiente. Observar que se salta la ejecución del primer bucle, ya que el valor[-1] no está definido y arrojaría un IndexOutOfBoundsException error.

```
int[] values = new int[3];
values[0] = 10;
values[1] = new Integer(5);
values[2] = 15;
for(int i=1; i<values.length; i++) {
  System.out.print(values[i]-values[i-1]);
}
```

Este código devolverá lo siguiente:

```
-5, 10,
```

A pesar de estos ejemplos, los bucles mejorados for-each son bastante útiles en Java en una gran variedad de circunstancias. Como desarrollador, sin embargo, siempre se puede volver a un for estándar si se necesita un control más fino.

4.6. Comprendiendo el control de flujo avanzado

Hasta ahora, se ha tratado con bucles simples que sólo terminaban cuando su expresión booleana es evaluada como falsa. Ahora se mostrará otras formas en la que los bucles podrían terminar, o ramificar, y se verá que el camino tomado durante el tiempo de ejecución puede no ser tan sencillo como en los ejemplos anteriores.

4.6.1. Bucles anidados

En primer lugar, los bucles pueden contener otros bucles. Por ejemplo, considerar el siguiente código que itera sobre una matriz bidimensional, una matriz que contiene otras matrices como sus miembros. Se cubrirá los arreglos multidimensionales en detalle en el Capítulo 3, pero por ahora se supondrá que la siguiente es la forma de declarar un arreglo bidimensional.

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
for(int[] mySimpleArray : myComplexArray) {
  for(int i=0; i<mySimpleArray.length; i++) {
    System.out.print(mySimpleArray[i]+"\t");
  }
  System.out.println();
}
```

Nótese que en este ejemplo se mezcla intencionalmente un bucle for y uno for-each. Los bucles externos se ejecutarán un total de tres veces. Cada vez que se ejecuta el bucle exterior, el bucle interior se ejecutará cuatro veces. Cuando se ejecuta este código, se ve la siguiente salida:

```
5             2             1             3
3             9             8             9
5             7             12            7
```

Los bucles anidados pueden incluir bucles while y do-while, como se muestra en este ejemplo. Ver si se puede determinar lo que este código saldrá.

```
int x = 20;
while(x>0) {
  do {
    x -= 2
  }while (x>5);
  x--;
  System.out.print(x+"\t");
}
```

La primera vez que este bucle se ejecuta, el bucle interno se repite hasta que el valor de x es 4. El valor será entonces decrementado a 3 y será la salida, al final de la primera iteración del bucle exterior. En la segunda iteración del bucle exterior, el do-while interno se ejecutará una vez, aunque x no sea mayor que 5. Recordar que las sentencias do-while siempre se ejecutan al menos una vez. Esto reducirá el valor a 1, que será decrementado aún más por el operador de decremento en el bucle exterior a 0. Una vez que el valor alcance 0, el bucle externo se termina. El resultado es que el código emitirá lo siguiente:

```
3             0
```

4.6.2. Añadiendo etiquetas opcionales

Una cosa que se omite cuando se presentan las sentencias if-then, las sentencias switch y bucles es que todos ellos pueden tener etiquetas opcionales. Una etiqueta es un puntero opcional a la cabeza de una instrucción que permite a la aplicación saltar a ella o acabar. Es una sola palabra que procede de dos puntos (:). Por ejemplo, se puede añadir etiquetas opcionales a uno de los ejemplos anteriores:

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
OUTER_LOOP: for(int[] mySimpleArray : myComplexArray) {
  INNER_LOOP: for(int i=0; i<mySimpleArray.length; i++) {
    System.out.print(mySimpleArray[i]+"\t");
  }
  System.out.println();
}
```

Cuando se trata de un solo bucle, no añaden ningún valor, pero como se verá en la siguiente sección, son extremadamente útiles en entornos anidados. Las etiquetas opcionales a menudo sólo se utilizan en estructuras de bucle. Dicho esto, rara vez se considera una buena práctica de codificación hacerlo.

Las etiquetas siguen las mismas reglas para los identificadores. En cuanto a la legibilidad, se expresan comúnmente en mayúsculas, con barra baja entre palabras, para distinguirlas de las variables regulares.

4.6.3. La declaración break

Como se vio al trabajar con las sentencia switch, una declaración break transfiere el flujo de control a la declaración adjunta. Lo mismo es válido para las sentencias break que aparecen dentro de bucles while, do-while, y bucles for, ya que terminaran el bucle temprano, como se muestra en el siguiente código:

```
optionalLabel: while(booleanExpression) {
  //

  Body
  //

  Somewhere in loop
  break optionalLabel;
}
```

Observar en el código que la sentencia break puede tomar un parámetro opcional de etiqueta. Sin un parámetro de etiqueta, la sentencia break terminará el bucle interno más cercano que esté ejecutando actualmente. El parámetro de etiqueta opcional permite salir de un bucle exterior de nivel superior. En el siguiente ejemplo, se busca la primera posición del índice de matriz (x, y) de un número dentro de una matriz bidimensional no clasificada:

```
public class SearchSample {
  public static void main(String[] args) {
    int[][] list = {{1,13,5},{1,2,5},{2,7,2}};
```

```
    int searchValue = 2;
    int positionX = -1;
    int positionY = -1;
    PARENT_LOOP: for(int i=0; i<list.length; i++) {
      for(int j=0; j<list[i].length; j++) {
        if(list[i][j]==searchValue){
          positionX = i;
          positionY = j;
          break PARENT_LOOP;
        }
      }
    }
    if(positionX==-1 || positionY==-1) {
      System.out.println("Value "+searchValue+" not found");
    } else {
      System.out.println("Value "+searchValue+" found at: " + "("+positionX+","+positionY+
")");
    }
  }
}
```

Cuando se ejecute, este código saldrá:

```
Value 2 found at: (1,1)
```

En particular, échese un vistazo a la sentencia break PARENT_LOOP. Esta declaración romperá de toda la estructura del bucle tan pronto como se encuentre el primer valor que se ajuste. Ahora, imaginar lo que pasaría si se reemplazara el cuerpo del bucle interno por el siguiente:

```
if(list[i][j]==searchValue) {
  positionX = i;
  positionY = j;
  break;
}
```

¿Cómo cambiaría esto la salida? En lugar de salir cuando se encuentra el primer valor que coincida, el programa sólo saldrá del bucle interno cuando se cumpla la condición. En otras palabras, la estructura ahora encontrará el primer valor que se ajuste del último bucle interno que contenga el valor, resultando en la siguiente salida:

```
Value 2 found at: (2,0)
```

Finalmente, ¿qué tal si se elimina el break por completo?

```
if(list[i][j]==searchValue) {
  positionX = i;
```

```
  positionY = j;
}
```

En este caso, el código buscará el último valor de toda la estructura que tenga el valor equivalente. La salida se verá así:

```
Value 2 found at: (2,2)
```

Se puede ver en este ejemplo que el uso de una etiqueta en una sentencia break en un bucle anidado, o el no uso de la sentencia break en absoluto, puede hacer que el bucle se comporte de forma muy diferente.

4.6.4. La sentencia continue

Se completará ahora la discusión sobre el control avanzado del bucle con la sentencia continue, una declaración que causa que el flujo de datos finalice la ejecución del bucle actual, como se muestra en el siguiente código:

```
optionalLabel: while(booleanExpression) {
  //

  Body
  //

  Somewhere in loop
  continue optionalLabel;
}
```

Se puede notar que la sintaxis de la declaración continue refleja la declaración break. De hecho, las sentencias son similares en cuanto a su uso, pero con resultados diferentes. Mientras que la sentencia break transfiere el control a la sentencia adjunta, la sentencia continue transfiere el control a la expresión booleana que determina si el bucle debe continuar. En otras palabras, termina la iteración actual del bucle. También, al igual que la declaración break, la declaración continue se aplica al bucle interno más cercano en ejecución usando instrucciones de etiqueta opcionales para anular este comportamiento. Ejemplo:

```
public class SwitchSample {
  public static void main(String[] args) {
    FIRST_CHAR_LOOP: for (int a = 1; a <= 4; a++) {
      for (char x = 'a'; x <= 'c'; x++) {
        if (a == 2 || x == 'b')
        continue FIRST_CHAR_LOOP;
        System.out.print(" " + a + x);
      }
    }
```

```
  }
}
```

Con la estructura definida, el bucle devolverá el control al bucle padre cada vez que el primer valor sea 2 o el segundo valor sea b. Esto resulta en una ejecución del bucle interno para cada una de las tres llamadas del bucle exterior. La salida se ve así:

```
1a  3a  4a
```

Ahora, imaginar que se elimina la etiqueta FIRST_CHAR_LOOP en la declaración continue para que el control sea devuelto al bucle interno en vez del externo. Mirar cómo cambiará la salida:

```
1a  1c  3a  3c  4a  4c
```

Por último, si se elimina la sentencia continue y la sentencia if-then asociada a ésta, se llega a una estructura que produce todos los valores, tales como:

```
1a  1b  1c  2a  2b  2c  3a  3b  3c  4a  4b  4c
```

La siguiente tabla ayudará a recordar cuando se permiten etiquetas, break, y sentencias continue en Java. Aunque para fines ilustrativos los ejemplos han incluido el uso de estas sentencias en bucles anidados, también se pueden usar dentro de bucles simples.

	Permite etiquetas opcionales	Permite sentencias break	Permite sentencias continue
if	Si	No	No
while	Si	Si	Si
do while	Si	Si	Si
for	Si	Si	Si
switch	Si	Si	No

4.6.1. Resumen

En este capítulo, se ha visto que:

Entender los operadores de Java

- Cómo utilizar todos los operadores Java requeridos que se describen.

- Saber cómo influye la precedencia del operador en la forma en que se interpreta una expresión determinada.

Usando operadores binarios

- Los operadores de asignación son operadores binarios que modifican la variable con el valor del lado derecho de la ecuación.

- El "casting" es necesario siempre que se pase de un dato numérico más grande a un tipo de datos numéricos más pequeños, o la conversión de un número float a un valor int.

- Los operadores complejos son operadores de asignación con una operación aritmética o lógica incorporada que se aplican de izquierda a derecha de la expresión y almacena el valor resultante en la variable de la parte izquierda de la pantalla.

- Los operadores relacionales son aquellos que comparan dos expresiones y devuelven un valor booleano.

- Los operadores lógicos, (&), (|) y (^), se pueden aplicar a datos de tipo numéricos y booleanos.

- Los operadores de igualdad comparan dos operandos y devuelven un valor booleano si las expresiones o valores son iguales, o no iguales, respectivamente.

Comprender las sentencias de Java

- Las estructuras de control: las estructuras de control de la toma de decisiones, incluidas las de tipo if-then, if-then-else y switch, así como las estructuras de control de la repetición, incluidas for, for-each, while y do-while.

- La mayoría de estas estructuras requieren la evaluación de una expresión booleana en particular, ya sea para las decisiones de ramificación o una vez por repetición.

- La sentencia switch es la única que soporta una variedad de tipos de datos, incluyendo variables String a partir de Java 7.

- Con un for-each no es necesario escribir explícitamente una expresión booleana, ya que el compilador las construye implícitamente. Para mayor claridad, se refiere a un bucle for mejorado como un bucle for-each, pero sintácticamente están escritos como una declaración for.

Comprendiendo el control de flujo avanzado

- Los bucles pueden contener otros bucles, incluidos bucles while y do-while.

- Una etiqueta es un puntero opcional a la cabeza de una instrucción que permite a la aplicación saltar a ella o acabar.

- Cómo se puede mejorar el flujo a través de los bucles anidados, de sentencias break y de sentencias continue.

5. API de Java

5.1. Creando y Manipulando Strings

La clase String es una clase imprescindible de Java que, no obstante, deberá utilizarse lo mínimo posible en el código de una aplicación real. Después de todo, ni siquiera se puede llamar al método main() sin la clase String. Un string es, básicamente, una secuencia de caracteres. Por ejemplo:

```
String name = “Botella”;
```

Hasta ahora se había visto que para crear una variable de tipo referencia era necesario la palabra **new**; no obstante, *name* es una variable referencia y se ha creado sin hacer uso de **new**. En Java, el siguiente código es equivalente:

```
String name = “Botella”;
String name = new String(“Botella”);
```

5.1.1. Concatenación

No es lo mismo hacer (1 + 2) que (“1” + “2”), ya que el primero hará una operación aritmética (3) mientras que el resultado de la segunda será la concatenación de dos String (“12”).

Reglas básicas:

1. Si los dos operadores son numéricos, “+” los sumará.
2. Si uno de los dos operadores es un String, “+” los concatenará.
3. Las expresiones se tratan de izquierda a derecha.

Ejemplos:

```
System.out.println(1 + 2);  // 3
System.out.println("a" + "b");  // ab
System.out.println("a" + "b" + 3);  // ab3
System.out.println(1 + 2 + "c");  // 3c
```

Primer caso: ambos son números por lo que los suma.

Segundo caso: ambos son String, por lo que los concatena.

Tercer caso: debido a la tercera regla, los dos primeros operadores que son String se convierten en un solo String (“ab”) y como la siguiente operación es un String más un número, debido a la segunda regla, se concatenarán todos los operandos.

Cuarto caso: En este caso, los dos primeros elementos son números, por lo que se sumarán, y al pasar a la siguiente operación (un número "3" y un String) se concatenarán.

Con el fin de complicarlo un poco más, véase el siguiente ejemplo:

```
int three = 3;
String four = "4";
System.out.println(1 + 2 + three + four);
```

En este caso, tenemos una variable de tipo int que vale 3 y otra de tipo String que vale "4". Puesto que los tres primeros elementos son números enteros, estos se sumarán, dando como resultado 6. Por último, se concatenará el 6 al String "4", por lo que por consola se imprimirá "64".

También hay que tener en cuenta en este ámbito la existencia del operador +=.

s += "2" significa lo mismo que s = s + "2".

Ejemplo:

```
4:  String s = "1";  // s almacena "1"
5:  s += "2";  // s almacena "12"
6:  s += 3;  // s almacena "123"
7:  System.out.println(s);  // imprime 123
```

5.1.2. Inmutabilidad

Una vez un objeto de tipo String es creado, este es inmutable, y no se puede modificar. No es posible hacerlo más grande o más pequeño, o modificar alguno de los caracteres que lo conforman.

En caso de que se añada un carácter o se concatenen dos String, un objeto String nuevo es creado.

5.1.3. La String Pool

Los String son una parte tan relevante de Java que pueden llegar a ocupar de un 25% a un 40% de la memoria de una aplicación. Java es capaz de saber si hay Strings que se repiten a lo largo del programa, y los reutiliza. La String pool o la pool interna es el lugar en la máquina virtual de Java que almacena estos String.

La String Pool contiene valores de String literales que haya en el programa. Por ejemplo, "name" es un String literal e iría a la String Pool, mientras que myObject .toString() es un String no literal, por lo que no iría a la String Pool.

5.1.4. Métodos importantes de String

La clase String tiene docenas de métodos. A la hora de trabajar con ellos, es necesario tener en cuenta que Java almacena los String como cadenas de caracteres cuya primera posición es la número 0.

Ejemplo:

a	n	i	m	a	l	s
0	1	2	3	4	5	6

a. length()

El método length() devuelve el número de caracteres de un String.

```
int length()
```

Ejemplo:

```
String string = "animals";
System.out.println(string.length());  // 7
```

En este caso, el carácter "0" es el primero, pero el String tiene 7 caracteres.

b. charAt()

El método charAt() permite recorrer un String para conocer el valor de un carácter.

```
char charAt(int index)
```

Ejemplo:

```
String string = "animals";
System.out.println(string.charAt(0));  // a
System.out.println(string.charAt(6));  // s
System.out.println(string.charAt(7));  // devuelve una excepción porque no existe ese dígi
to
```

En este caso, la posición "0" corresponde al primer carácter del String. Cuando no exista ningún carácter en una determinada posición, se lanzará una excepción como se ve a continuación:

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 7
```

c. indexOf()

El método indexOf() devuelve la posición exacta de un carácter dentro de un String.

```
int indexOf(char ch)
int indexOf(char ch, index fromIndex)
int indexOf(String str)
int indexOf(String str, index fromIndex)
```

Ejemplo:

```
String string = "animals";
System.out.println(string.indexOf('a'));  // 0
System.out.println(string.indexOf("al"));  // 4
System.out.println(string.indexOf('a', 4));  // 4
System.out.println(string.indexOf("al", 5));  // -1
```

La primera sentencia encuentra una concordancia en la posición 0. La segunda busca una concordancia más específica (2 letras). La tercera busca a partir del carácter 4. La última no encuentra nada porque empieza a buscar después de que haya concordancias en el String (por eso devuelve -1). En este caso no se devuelve 0 porque los caracteres empiezan en 0 y no devuelve una excepción (ya que la posición -1 no existe).

d. substring()

El método substring() también busca caracteres en un String y devuelve parte del mismo. El primer parámetro que se le dará será la posición en la que debe de empezar a leer el String, pero además existe un segundo parámetro opcional que se le puede dar, y es aquel donde debe parar de leer. Se debe entender este segundo parámetro como donde detenerse y no como cuál es el último carácter a incluir. Todo esto basándose en la premisa de que el String empieza con la posición "0".

```
int substring(int beginIndex)
int substring(int beginIndex, int endIndex)
```

Ejemplo:

```
String string = "animals";
System.out.println(string.substring(3));  // mals
System.out.println(string.substring(string.indexOf('m')));  // mals
System.out.println(string.substring(3, 4));  // m
System.out.println(string.substring(3, 7));  // mals
```

En el primer caso, toma los caracteres desde la posición 3 hasta el final. La segunda hace lo mismo pero añadiendo un indexOf() de la letra m, que es un buena práctica porque no siempre se sabrá en qué posición se encuentra lo que buscamos. En el tercer caso, toma los datos desde la posición 3 hasta la posición 4 sin incluir esta última. En el último caso, empieza en la posición 3 y termina en la 7, que como es el final del String no sería un problema.

Otros ejemplos:

```
System.out.println(string.substring(3, 3));  // string vacío
System.out.println(string.substring(3, 2));  // devuelve una excepción
System.out.println(string.substring(3, 8));  // devuelve una excepción
```

Tal y como se ve en los ejemplos anteriores, si el origen y el fin son el mismo, devolverá un String vacío. En el segundo ejemplo se lanzará una excepción porque el final de lectura es anterior al de inicio. Y, por último, si el final va más allá del final del String se tendrá otra excepción (la posición final del String string, en este caso, es la posición 7).

e. toLowerCase() y toUpperCase()

Estos métodos permiten convertir la cadena de caracteres a minúsculas y a mayúsculas, respectivamente.

```
String toLowerCase(String str)
String toUpperCase(String str)
```

Ejemplos:

```
String string = "animals";
System.out.println(string.toUpperCase());  // ANIMALS
System.out.println("Abc123".toLowerCase());  // abc123
```

El método toUpperCase() convierte los caracteres de un String en mayúsculas. Mientras que toLowerCase() convierte los caracteres del String en minúsculas. A pesar de esto, el String original sigue siendo inmutable, no cambia, lo único es que se imprime de manera diferente.

f. equals() y equalsIgnoreCase()

El método equals() comprueba que dos String contengan exactamente los mismos caracteres y en el mismo orden. El método equalsIgnoreCase() también comprueba ambos String con la excepción de que convertirá los caracteres si fuera necesario.

```
boolean equals(String str)
boolean equalsIgnoreCase(String str)
```

Ejemplo:

```
System.out.println("abc".equals("ABC"));  // false
System.out.println("ABC".equals("ABC"));  // true
System.out.println("abc".equalsIgnoreCase("ABC"));  // true
```

En los casos anteriores, el método compara los dos String y devuelve un boolean (true o false). En los dos primeros casos, compara exactamente los caracteres, mientras que en el tercero compara sin tener en cuenta mayúsculas o minúsculas.

g. startsWith() y endsWith()

Los métodos startsWith() y endsWith() comprueban que el valor dado concuerda con una parte del String (al principio o al final).

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

Ejemplo:

```
System.out.println("abc".startsWith("a"));  // true
System.out.println("abc".startsWith("A"));  // false
System.out.println("abc".endsWith("c"));  // true
System.out.println("abc".endsWith("a"));  // false
```

h. contains()

El método contains() también busca coincidencias en un String, puede estar en cualquier parte del mismo.

```
boolean contains(String str)
```

Ejemplo:

```
System.out.println("abc".contains("b"));  // true
System.out.println("abc".contains("B"));  // false
```

De nuevo, se tiene un método que distingue entre mayúsculas y minúsculas.

i. replace()

El método replace() hace una búsqueda simple y reemplaza en el String. Se puede realizar mediante parámetros char o mediante parámetros CharSequence. Un CharSequence es una versión generalizada de representar varias clases, incluyendo String y StringBuilder.

```
String replace(char oldChar, char newChar)
String replace(CharSequence oldChar, CharSequence newChar)
```

Ejemplo:

```
System.out.println("abcabc".replace('a', 'A')); // AbcAbc
System.out.println("abcabc".replace("a", "A")); // AbcAbc
```

El primer ejemplo utiliza el método con parámetros char, mientras que en el segundo utiliza el método con parámetros charSequence.

j. trim()

El método trim() elimina espacios en blanco por delante y por detrás del String.

```
public String trim()
```

Ejemplo:

```
System.out.println("abc".trim());  // abc
System.out.println("\t     a b c\n".trim());  // a b c
```

El primer ejemplo muestra el String original porque no tiene espacios en blanco al principio o al final. El segundo se deshace de la tabulación inicial, los espacios subsiguientes y el "nueva línea". Deja los espacios entre las letras.

5.1.5. Encadenado de Métodos

Es muy común llamar a varios métodos desde un mismo String:

```
String start = "AniMaL";
String trimmed = start.trim();  // "AniMaL"
String lowercase = trimmed.toLowerCase();  // "animal"
String result = lowercase.replace('a', 'A');  // "Animal"
System.out.println(result);
```

En este caso, en cada línea de código se ejecuta un nuevo método almacenando el "cambio" realizado. En la práctica se tiende a eliminar líneas de código usando lo que se llama **encadenado de métodos**.

Ejemplo:

```
String result = "AniMaL".trim().toLowerCase().replace('a', 'A');
System.out.println(result);
```

Este código es exactamente igual al anterior, pero con menor número de líneas. Para leer código encadenado se debe empezar por la izquierda y evalúa el primer método. Después, evalúa el siguiente método sobre el valor de retorno del anterior. Y así sucesivamente.

Un ejemplo práctico:

```
5: String a = "abc";
6: String b = a.toUpperCase();
7: b = b.replace("B", "2").replace('C', '3');
8: System.out.println("a=" + a);
9: System.out.println("b=" + b);
```

En el caso de la variable "a", no se verá modificada hasta el final (línea 8) en que se verá impresa. Para la variable "b", es algo más complicado porque se basa en una modificación de la variable "a" y después una serie de métodos encadenados dando un resultado de "b=A23".

5.2. Usando la Clase StringBuilder

En un programa Java, por pequeño que sea, puede haber gran cantidad de objetos String. Por ejemplo:

```
10: String alpha = "";
11: for(char current = 'a'; current <= 'z'; current++)
12:   alpha += current;
13: System.out.println(alpha);
```

Se instancia un String vacío en la línea 10, y en la línea 12 agrega una "a" por lo que el objeto "" pasa a la *garbage collection*. En el siguiente bucle for, el objeto pasa a ser "ab" por lo que el objeto anterior "a" pasa a la garbage collection. Y así sucesivamente mientras se mantenga el bucle, por lo que al final, después de 26 iteraciones, se habrán creado 27 objetos.

Esto es muy ineficiente. Por suerte, Java tiene la clase StringBuilder, que crea un String sin tener que almacenar todos los valores anteriores del String. Sin embargo, esta clase no es inmutable.

```
15: StringBuilder alpha = new StringBuilder();
16: for(char current = 'a'; current <= 'z'; current++)
17:   alpha.append(current);
18: System.out.println(alpha);
```

En la línea 15, un objeto StringBuilder queda instanciado. La llamada append() en la línea 17 añade una carácter al Stringbuilder por cada vuelta del bucle for y coloca el valor al final de "alpha". Este código reutiliza el mismo StringBuilder sin crear un String a cada vez.

5.2.1. Mutabilidad y Encadenado

Tal y como se ha visto en el ejemplo anterior, la clase StringBuilder no es inmutable, ya que se le han asignado 27 valores diferentes (desde el valor en blanco hasta todas las letras del abecedario).

Cuando se encadenan varios métodos String, se tiene como resultado un nuevo String con la respuesta. Cuando se encadenan objetos StringBuilder no funciona igual, sino que cambia su estado y devuelve una referencia a sí mismo. Ejemplo:

```
4: StringBuilder sb = new StringBuilder("start");
5: sb.append("+middle");  // sb = "start+middle"
6: StringBuilder same = sb.append("+end");  // "start+middle+end"
```

La línea 5 añade texto al final de "sb". Devolviendo además una referencia a "sb" que se ignora. La línea 6 añade texto al final de "sb" y devuelve una referencia a "sb". Esta vez la referencia se almacena en "same" (lo que significa que "sb" y "same" apuntan exactamente al mismo objeto e imprimirán el mismo valor.

Ejemplo práctico:

```
4: StringBuilder a = new StringBuilder("abc");
5: StringBuilder b = a.append("de");
6: b = b.append("f").append("g");
7: System.out.println("a=" + a);
8: System.out.println("b=" + b);
```

En este caso, el objeto StringBuilder se llama solo una vez, pero al final, ambos "println" tendrán el mismo valor (abcdefg)

5.2.2. Creando un StringBuilder

Existen 3 formas de construir un StringBuilder:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```

La primera contiene una secuencia de caracteres vacía y se asigna a sb1. La segunda se crea conteniendo un valor específico. Por último tenemos la versión en que se crea con un parámetro que le dice a Java de reservar una cantidad específica de caracteres para ese StringBuilder.

Tamaño vs. Capacidad

Tamaño es el número de caracteres en una secuencia, y la capacidad es el número de caracteres que puede contener la secuencia. En un String (que es inmutable) el tamaño y la capacidad son los mismos. En Java un StringBuilder puede cambiar así que, cuando se construye un StringBuilder, puede contener un valor por defecto (16 caracteres) o un valor elegido por el programador.

En el ejemplo práctico siguiente, se puede ver que se empieza llamando al StringBuilder con una capacidad de 5 caracteres, después se añade una serie de caracteres lo que termina teniendo un tamaño de 4 y una capacidad de 5. Finalmente se le añaden 3 más, lo que hace que el tamaño sea de 7, pero como no había capacidad suficiente Java la ha aumentado automáticamente.

StringBuilder sb = new StringBuilder(5);

0	*1*	*2*	*3*	*4*

sb.append("anim");

a	*n*	*i*	*m*	
0	*1*	*2*	*3*	*4*

sb.append("als");

a	*n*	*i*	*m*	*a*	*l*	*s*		
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	...

5.2.3. Métodos importantes de StringBuilder

Los métodos más utilizados de la clase StringBuilder son:

a. charAt(), indexOf(), length(), y substring()

Estos cuatro métodos funcionan igual que los métodos homónimos de la clase String.

Ejemplo práctico:

```
StringBuilder sb = new StringBuilder("animals");
String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));
int len = sb.length();
char ch = sb.charAt(6);
System.out.println(sub + " " + len + " " + ch);
```

La respuesta correcta es "anim 7 s". Los indexOf() devuelven las posiciones 0 y 4 respectivamente, por lo que se imprimirán las letras que se encuentran en las posiciones de 0 a 3 (anim). length() devuelve un 7 por el número de caracteres que tiene el StringBuilder sb. Y por último, el charAt() devuelve el carácter que se encuentra en la posición 6 (en base 0).

Es necesario tener en cuenta que substring() devuelve un String y no un StringBuilder, es por esto que sb no cambia.

b. append()

El método append() es el método más usado de StringBuilder, ya que añade un String a un StringBuilder y devuelve una referencia del StringBuilder en cuestión. Una de las declaraciones del método es la siguiente:

```
StringBuilder append(String str)
```

Notese que se ha dicho una de las formas. Existen más de 10 formas de hacerlo, dependiendo del tipo de parámetros que se le suministre al método.

```
StringBuilder sb = new StringBuilder().append(1).append('c');
sb.append("-").append(true);
System.out.println(sb);  // 1c-true
```

append() es llamado justo después del constructor. Se le puede llamar sin tener que convertir los parámetros a un String primero.

c. insert()

El método insert() añade caracteres a un StringBuilder en el índice dado y devuelve una referencia del mismo StringBuilder. Ejemplo:

```
StringBuilder insert(int offset, String str)
```

Ejemplos prácticos:

```
3: StringBuilder sb = new StringBuilder("animals");
4: sb.insert(7, "-");  // sb = animals-
5: sb.insert(0, "-");  // sb = -animals-
6: sb.insert(4, "-");  // sb = -ani-mals-
7: System.out.println(sb);
```

La línea 4 inserta un guion antes de la posición 7 (final de caracteres). La línea 5 inserta un guion antes de la posición 0 (primer carácter). La línea 6 inserta un un guion antes de la posición 4.

d. delete() y deleteCharAt()

El método delete() es lo opuesto al método insert(), elimina caracteres de la secuencia y devuelve una referencia del mismo Stringbuilder. el método deleteCharAt() es conveniente utilizarlo cuando solo se quiere eliminar un carácter. Se escribe de la siguiente manera:

```
StringBuilder delete(int start, int end)
StringBuilder deleteCharAt(int index)
```

Ejemplo:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1,3);  // sb = adef
sb.deleteCharAt(5);  // devuelve una excepción
```

Primero elimina los caracteres que se encuentran entre las posiciones 1 y 3 (comenzando por 0, es decir, "b" y "c" ya que el parámetro 3 no se incluye en la eliminación). Después de esto elimina el carácter que se encuentra en la posición 5 (de nuevo comenzando por 0) pero se lanza una excepción, ya que como ya se ha reducido el String con el método en la línea anterior, no existe la posición 5 (nombre de la excepción: StringIndexOutOfBoundsException).

e. reverse()

El método reverse() invierte los caracteres en la secuencia y devuelve una referencia del mismo StringBuilder.

```
StringBuilder reverse()
StringBuilder sb = new StringBuilder("ABC");
sb.reverse();
System.out.println(sb);
```

Tal y como se espera, imprimirá "CBA".

f. toString()

El último método convierte un StringBuilder en un String. Se escribirá de la siguiente manera:

```
String toString()
String s = sb.toString();
```

Usado habitualmente para devolver un String a otra función.

g. StringBuilder vs. StringBuffer

StringBuilder se implement en Java 5, pero en versiones anteriores se utilizaba StringBuffer, que es más lento que StringBuilder.

5.3. Entender Igualdad

En el Capítulo 4 se estudió el uso de == para comparar números y referencias a objetos con la misma posición en memoria.

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two);  // false
System.out.println(one == three);  // true
```

En el ejemplo anterior no se está tratando con primitivas de datos, "one" y "two" son dos StringBuilders completamente diferentes, por lo que son dos objetos distintos. Por consiguiente, en el primer System.out.println se obtiene un false.

Por otra parte, en la siguiente comparación, hay que tener en cuenta que el método append() ha añadido un carácter al objeto "one" y que, después, este objeto se ha asociado a la variable "three". Esto devolverá true a que "one" y "three" son iguales, ya que hacen referencia al mismo objeto.

Comparación de objetos String:

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y);  // true
```

En este caso, "x" e "y" apuntan a la misma localización de memoria, por lo que la comparación da true. Esto se debe a que ambos String son literales y, por lo tanto, almacenados en el String Pool una sola vez. Son el mismo objeto.

En el siguiente caso; no obstante, ocurre algo diferente:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x == z);  // false
```

En este caso no se tiene el mismo String literal. Aunque "x" e "z" evalúan el mismo String, uno es computado en tiempo de ejecución. Como no es lo mismo en el tiempo de compilación, se crea un nuevo objeto String.

Otro caso:

```
String x = new String("Hello World");
String y = "Hello World";
System.out.println(x == y);  // false
```

Como el primero no es un String literal y no se almacena en la String Pool, también devolverá un false.

MORALEJA: no usar nunca == para comparar objetos String.

Es mejor usar igualdad lógica y no igualdad de objetos:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x.equals(z));  // true
```

El método equals de la clase String permite comparar el contenido de dos String porque los creadores de la clase así lo implementaron. Si una clase no tiene implementado el método equals, por defecto Java compara las referencias a las que apuntan los objetos (exactamente lo mismo que hace ==). Si se llama a equals() entre dos instancias StringBuilder, por ejemplo, se comprobarán sus referencias.

Véase otro ejemplo:

```
1: public class Tiger {
2:   String name;
3:   public static void main(String[] args) {
4:     Tiger t1 = new Tiger();
5:     Tiger t2 = new Tiger();
6:     Tiger t3 = t1;
7:     System.out.println(t1 == t1); // true
8:     System.out.println(t1 == t2); // false
9:     System.out.println(t1.equals(t2)); // false
10:  }

11:}
```

Las dos primeras comparaciones comprueban la igualdad de referencia de objetos. La línea 7 devuelve true porque se está comparando referencias al mismo objeto. La Línea 8 devuelve un false porque las referencias a objetos son diferentes. La línea 9 imprime false porque Tiger no implementa equals().

5.4. Entender Arrays de Java

Hasta el momento, en secciones anteriores del capítulo, se ha estado haciendo referencia a los objetos de tipo String o StringBuilder como "secuencias de caracteres". Literalmente, esto es cierto para Java, ya que ambos tipos de datos están implementados como un array o un vector caracteres (datos tipo char) en memoria. Un String es un array de longitud fija, y un StringBuilder es un array que es reemplazado por un array de mayor tamaño en caso de ser necesario.

Un array en Java puede ser de cualquier tipo de dato. Si por alguna razón no se quisiera usar un String, se podría usar un array de tipo primitivo char directamente:

```
char[] letters;
```

Esto no sería muy conveniente porque se perdería las propiedades especiales del String (todos sus métodos).

En otras palabras, un Array es una lista de objetos o de primitivas de datos que puede contener elementos duplicados.

5.4.1. Creando un array de primitivas de datos

La manera más común de crear un array es la siguiente:

```
int[] numbers1 = new int[3];
```

int: tipo de dato del array

[]: símbolo del array

3: tamaño del array

Utilizar esta forma de instanciar un array establecerá todos los elementos del array con valores por defecto. El valor por defecto para un int es 0. Los array también tienen su primer dato en la posición 0.

Ejemplo numbers1:

Elemento	0	0	0
índice	0	1	2

Otra manera de crear una array es especificando todos los elementos con los que debe inicializarse.

```
int[] numbers2 = new int[] {42, 55, 99};
```

En este ejemplo se crea un array de tamaño 3, pero especificando los valores iniciales.

Elemento	42	55	99
índice	0	1	2

Esta última forma es redundante porque Java ya sabe qué tipo de array estás creando, por lo que se puede hacer de esta otra manera para simplificar:

```
int[] numbers2 = {42, 55, 99};
```

Esto se llama un array anónimo, y se llama así porque no se especifica ni el tipo ni el tamaño.

Finalmente, se puede poner los [] antes o después del nombre, y añadir espacios es opcional.

```
int[] numAnimals;
int []  numAnimals2;
int numAnimals3[];
int numAnimals4  [];
```

a. Múltiples Arrays en una declaración

Varios ejemplos. Primero:

```
int[] ids, types;
```

Esto devuelve 2 variables de tipo int[]. Segunda:

```
int ids[], types;
```

Solo se ha movido los [] pero el comportamiento ha cambiado. En este caso se tiene una variable de tipo int[] y una de tipo int. La primera llamada ids[] y la segunda simplemente llamada types, sin corchetes, con lo que es un solo número entero.

5.4.2. Crear un array con variables referencia

Un array puede ser de cualquier tipo de dato existente en Java, incluyendo objetos de las clases que crea el mismo programador.

El siguiente ejemplo muestra arrays del tipo de dato String, ya predefinido en el lenguaje Java:

```
public class ArrayType {
  public static void main(String args[]) {
    String [] bugs = { "cricket", "beetle", "ladybug" };
    String [] alias = bugs;
    System.out.println(bugs.equals(alias));  // true
    System.out.println(bugs.toString());  // [Ljava.lang.String;@160bc7c0
  }

}
```

Se puede llamar al método equals() porque el array es un objeto. Devuelve true debido a la igualdad de referencias. El método de equals() en arrays no compara los elementos del array. Si el array fuera de tipo int[], el código funcionaría igual, ya que int[] es un objeto, no la primitiva de dato int.

Analizando lo que devuelve el método .toString: [Ljava.lang.String;@160bc7c0. [L significa que se trata de un array, java.lang.String es el tipo de dato, y 160bc7c0 es el código hash.

Véase otro ejemplo con el array siguiente:

```
class Names {
  String names[];
}
```

Lo anterior devuelve un null. El código no ha instanciado en ningún momento el array con lo que es solo una variable de referencia a un null. Siguiente paso:

```
class Names {
  String names[] = new String[2];
}
```

Es un array porque tiene un [], y es de tipo String porque está especificado en la declaración. Tiene dos elementos porque tiene una longitud de 2, y aunque cada uno de los elementos es null, tiene potencial para apuntar a dos objetos String.

Otro ejemplo:

```
3: String[] strings = { "stringValue" };
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder();  // NO COMPILA
7: objects[0] = new StringBuilder();  // cuidado!
```

La linea 3 crea un array de tipo String.

La línea 4 no requiere de "casting" ya que String es un tipo de dato que desciende de Object (como todos en Java).

En la línea 5, se requiere de un "casting" porque se está transformando a un tipo de dato más específico.

La línea 6 no compila porque el String[] solo permite objetos String y StringBuilder no es un String.

La línea 7 es algo diferente, desde el punto de vista de un compilador no hay problema, ya que un objeto StringBuilder puede ir dentro de un Objeto[]. El problema es que no se tiene un Objeto[], se tiene un String[] referido a partir de una variable Objeto []. En el momento de ejecución, el código devuelve una excepción (ArrayStoreException).

5.4.3. Hacer uso de un array

Ahora que se sabe cómo crear un array, se va a aprender a acceder a sus elementos:

```
4: String[] mammals = {"monkey", "chimp", "donkey"};
5: System.out.println(mammals.length);  // 3
6: System.out.println(mammals[0]);  // monkey
7: System.out.println(mammals[1]);  // chimp
8: System.out.println(mammals[2]);  // donkey
```

La línea 4 declara e inicializa el array.

La línea 5 muestra por consola cuántos elementos puede contener el array.

El resto del código imprime el contenido del array por consola.

Merece la pena remarcar que en los arrays el indexado comienza en la posición 0, al igual que en objetos String o StringBuilder.

Aunque los elementos de un array sean null, se contabilizan como elementos, por eso un array con [6] tendrá una longitud de 6.

Es muy común usar un bucle cuando se está leyendo o escribiendo un array. Ejemplo:

```
5: int[] numbers = new int[10];
6: for (int i = 0; i < numbers.length; i++)
7:    numbers[i] = i + 5;
```

En la línea 5 solo se instancia una array de 10 huecos.

En la línea 6 se muestra un bucle for con un formato estándar. Empieza en el índice 0 y continúa recorriendo todo el array.

En la línea 7 se modifica el valor del elemento que esté en la posición i del array numbers asignándosele un número.

Ejemplos que devuelven la excepción ArrayIndexOutOfBoundException:

```
numbers[10] = 3;
```

numbers es un array de tamaño 10, lo que significa que solo contiene valores desde numbers[0] hasta numbers[9].

```
numbers[numbers.length] = 5;
```

De nuevo, se intenta acceder a numbers[10], lo cual no existe, ya que la posición máxima que tiene el array numbers es 9.

```
for (int i = 0; i <= numbers.length; i++) numbers[i] = i + 5;
```

Finalmente, el bucle for usa incorrectamente <= en lugar de <, lo que es también una forma de referirse al décimo elemento.

5.4.4. Sorting

Se pueden ordenar los array de manera fácil mediante el método sort (Arrays.sort()). Para ello se deberá de importar las siguientes declaraciones:

```
import java.util.* // importa el paquete completo incluyendo Arrays
import java.util.Arrays; // importa solo Arrays
```

En el siguiente ejemplo se ordena 3 números:

```
int[] numbers = { 6, 9, 1 };
Arrays.sort(numbers);
for (int i = 0; i < numbers.length; i++)
  System.out.print (numbers[i] + " ");
```

El resultado es 1 6 9, como se espera. Es preciso remarcar que para imprimir los valores del array se ha implementado un bucle que lo recorre e imprime cada uno de sus valores por consola. Imprimiendo el método .toString() del array dará el ya visto hash de [I@2bd9c3e7.

Véase otro ejemplo:

```
String[] strings = { "10", "9", "100" };
Arrays.sort(strings);
for (String string : strings)
  System.out.print(string + " ");
```

En este caso el resultado no será lo que se espera. Este código devuelve 10 100 9. El problema es que el ordenado de String se hace por orden alfabético, y 1 sale antes que 9 (los números se ordenan antes que las letras y las mayúsculas antes que las minúsculas).

5.4.5. Búsquedas en arrays

Java también ofrece por defecto la posibilidad de realizar búsquedas de elementos en arrays. Una de estas búsquedas es la búsqueda binaria (binary search), la cual funciona de la siguiente manera:

Escenario	Resultado
Elemento buscado encontrado en array ordenado	Índice de acierto
Elemento buscado NO encontrado en array ordenado	Valor negativo
Array no ordenado	Este resultado no es previsible, la búsqueda binaria únicamente puede ser utilizada con arrays ordenados.

Véase el ejemplo siguiente:

```
3: int[] numbers = {2,4,6,8};
4: System.out.println(Arrays.binarySearch(numbers, 2));  // 0
5: System.out.println(Arrays.binarySearch(numbers, 4));  // 1
6: System.out.println(Arrays.binarySearch(numbers, 1));  // -1
7: System.out.println(Arrays.binarySearch(numbers, 3));  // -2
8: System.out.println(Arrays.binarySearch(numbers, 9));  // -5
```

En la línea 3 se puede ver que numbers es un array ordenado, si no fuera así no se podría aplicar la búsqueda binaria.

La línea 4 busca el índice de 2. La respuesta es índice 0.

La línea 5 busca el índice de 4 que es 1.

La línea 6 busca el índice de 1. A pesar de que 1 no está en la lista, el algoritmo de binarySearch puede determinar que debería ser insertado en el elemento 0 para conservar el orden. Cómo el 0 es una posición válida para un array, Java tiene que sustraer 1 y por eso da -1.

La línea 7 es similar, aunque 3 no está en la lista, se insertaría en el elemento 1, lo que daría -1-1 que es el -2 que se obtiene.

Por último, en la línea 8 se quiere buscar el índice de 9 que se insertaría en el índice 4. De nuevo se convierte ese 4 a negativo y se le resta 1, lo que da -4-1 = -5.

¿Qué ocurrirá en el siguiente ejemplo?

```
5: int numbers = new int[] {3,2,1};
6: System.out.println(Arrays.binarySearch(numbers, 2));
7: System.out.println(Arrays.binarySearch(numbers, 3));
```

El array numbers **no** está ordenado. Esto significa que la salida no es previsible. Cuando se prueba este ejemplo, la línea 6 devuelve correctamente 1. Sin embargo, la línea 3 devuelve una respuesta incorrecta.

5.4.6. Varargs

Existen diferentes maneras de pasar un array como parámetro a un método. He aquí tres ejemplos del paso de un array como parámetro con el método main():

```
public static void main(String[] args)
public static void main(String args[])
public static void main(String... args) // varargs
```

El tercer ejemplo usa una sintaxis llamada varargs (argumentos variables), vista en el Capítulo 3, y que se ampliará en el Capítulo 6. Por ahora solo se necesita saber que se puede usar una variable definida

usando varargs como si fuera un array normal. Por ejemplo: args.length y args[0] son tan válidos como con cualquier otro array.

5.4.7. Arrays Multidimensionales

Los array son objetos, y el contenido de los array puede ser también un conjunto de objetos.

a. Crear un array multidimensional

Crear un array multidimensional es tan sencillo como poner tantos [] como dimensiones se deseen en el array. Por ejemplo:

```
int[][] vars1;  // 2D array
int vars2 [][];  // 2D array
int[] vars3[];  // 2D array
int[] vars4 [], space [][];  // a 2D AND a 3D array
```

Los primeros dos ejemplos son formas de crear un array de 2 dimensiones.

El tercero crea un array vars4 de 2 dimensiones y un array space de 3 dimensiones, lo cual es muy poco habitual.

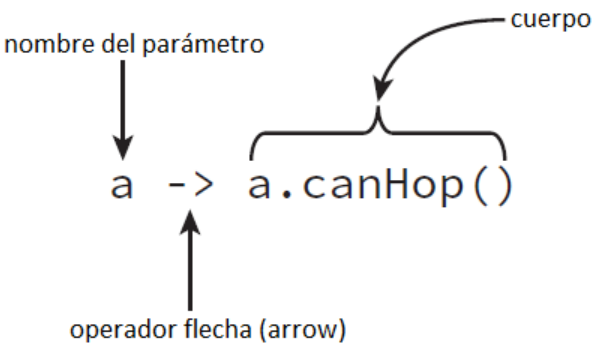
Se puede especificar el tamaño del array multidimensional tal que así:

```
String [][] rectangle = new String[3][2];
```

El resultado de esta declaración es un array rectangular con 3 elementos, cada uno de ellos contiene un array con 2 elementos. Se puede pensar en el rango direccionable como de [0][0] a [2][1].

Vease un ejemplo de la forma en la que se guardan objetos en las posiciones de un array multidimensional.

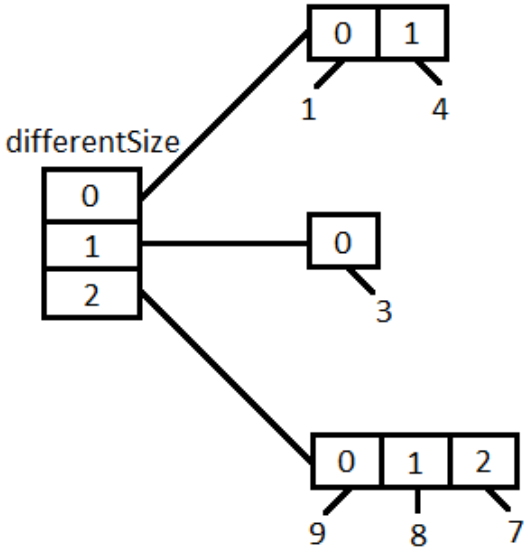
```
rectangle[0][1] = "set";
```



Este array está escasamente poblado porque tiene muchos valores nulos. Se puede ver que "rectangle" sigue apuntando a un array de 3 elementos y que cada uno de estos elementos es un array de 2 elementos. Es en la posición 1 del array que está en la posición 0 de rectangle donde se encuentra el String "set".

Considérese también el siguiente ejemplo, que muestra la inicialización de un array multidimensional asimétrico (los componentes del array principal son arrays de diferentes dimensiones).

```
int[][] differentSize = {{1, 4}, {3}, {9,8,7}};
```



Se crea un array differentSizes de 3 elementos. Sin embargo, ahora los elementos del siguiente nivel tienen todos diferentes dimensiones. Uno es de longitud 2, el siguiente 1, y el ultimo es de longitud 3. Esta vez el array es de primitivas de datos, con lo que se muestran como si estuvieran el array en sí.

Otra manera de crear un array asimétrico es inicializando solo la primera dimensión del array, y definiendo el tamaño de cada componente del array en sentencias separadas:

```
int [][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```

b. Usar un array multidimensional

La operación más habitual de los array multidimensionales es la de hacer un bucle a través de este. El siguiente ejemplo imprime un array de 2D:

```
int[][] twoD = new int[3][2];
for (int i = 0; i < twoD.length; i++) {
  for (int j = 0; j < twoD[i].length; j++)
```

```
    System.out.print(twoD[i][j] + " "); // print element
  System.out.println(); // time for a new row
}
```

Se tienen dos bucles aquí. El primero usa el índice "i" y recorre el primer subarray para twoD. El segundo usa una variable de bucle diferente, "j". Esto es muy importante, porque sino podrían mezclarse. El bucle interno comprueba cuanto elementos hay en el segundo nivel del array, y lo imprime por consola. Cuando este se completa, se imprime un sato de línea y el bucle exterior repite el proceso en el próximo elemento.

Otro ejemplo de recorrido de un array de dos dimensiones:

```
for (int[] inner : twoD) {
  for (int num : inner)
    System.out.print(num + " ");
  System.out.println();
}
```

No son menos líneas, pero cada línea es menos compleja y no hay variables de bucle o condiciones de finalización.

5.5. Entender un ArrayList

Al igual que un array, una ArrayList es una secuencia de elementos que permite duplicados.

Un array tiene una deficiencia evidente: hay que saber cuántos elementos estarán dentro cuando este se crea, y su tamaño es inmutable. Como un StringBuilder, un ArrayList puede cambiar de tamaño en tiempo de ejecución según sea necesario.

La clase ArrayList necesita ser importada en las clases donde se quiera utilizar.

```
import java.util.*  // importa todo el paquete que incluye ArrayList
import java.util.ArrayList;  // importa solo ArrayList
```

5.5.1. Crear un ArrayList

Como con StringBuilder, existen 3 formas de crear un ArrayList:

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```

La primera forma crea un ArrayList que contiene espacio para el número predeterminado de elementos que tenga la clase ArrayList, pero no rellena ninguno de estos elementos todavía.

La segunda forma crea un ArrayList que contiene un espacio específico para 10 elementos, pero, de nuevo, sin rellenar.

El ejemplo final hace una copia de otro ArrayList. Se copia tanto el tamaño como el contenido de ese ArrayList. En este caso, list2 está vacío, por lo que list3 estará también vacío.

A pesar de que por el momento con saber estos tres constructores es suficiente, es necesario tener en cuenta algunas variantes. Los ejemplos anteriores eran la antigua forma de crear un ArrayList antes de Java 5. Todavía funcionan y es necesario conocerlos; no obstante, Java 5 introdujo los **tipos genéricos**, lo que permite especificar el tipo de dato de los elementos que contendrá el ArrayList.

```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
```

Java 5 permite decirle al compilador cuál sería el tipo de dato de los elementos del ArrayList especificándolo entre < >. A partir de Java 7, incluso se puede omitir ese tipo en el lado derecho de la expresión. Los <> (también llamado operador diamante) siguen siendo necesarios.

ArrayList implementa una interfaz llamada List. En otras palabras, ArrayList es una List. Se aprenderá todo sobre interfaces en el Capítulo 7. Mientras tanto, se debe saber que se puede almacenar un ArrayList en una variable de referencia List pero no viceversa. La razón es que List es una interfaz y las interfaces no pueden ser instanciadas.

```
List<String> list6 = new ArrayList<>();
ArrayList<String> list7 = new List<>();  // NO COMPILA
```

5.5.2. Usar un ArrayList

ArrayList tiene gran cantidad de métodos. A continuación, se muestran los más utilizados.

En primer lugar, cabe mencionar algo que se puede observar en las cabeceras de los métodos que aparecen a partir de ahora, y es la "clase" llamada E. E, en realidad, no es una clase, sino que se usa por convención para hacer referencia a "cualquier clase que este ArrayList pueda contener". Si no se especificó un tipo al crear el ArrayList, E es del tipo Objet. De lo contrario, será de la clase que se indica entre <>.

También es interesante saber que ArrayList implementa un método toString() que muestra el contenido del propio ArrayList.

5.5.3. Métodos útiles de ArrayList

a. add()

Los métodos add() insertan un nuevo valor en un ArrayList. Las cabeceras del método son las siguientes:

```
boolean add(E element)
void add(int index, E element)
```

El primer método siempre devuelve true. Esto se debe a que existen clases entre las colecciones de Java que necesitan este valor de retorno al añadir un elemento, lo cual no es relevante por el momento.

Se muestran a continuación algunos ejemplos:

```
ArrayList list = new ArrayList();
list.add("hawk");  // [hawk]
list.add(Boolean.TRUE);  // [hawk, true]
System.out.println(list);  // [hawk, true]
```

En primer lugar, add( ) almacena el String "hawk" en el ArrayList list. Después, almacena la constante boolean TRUE. Esto es correcto puesto que el tipo de list es Object, y todos los objetos Java descienden de Object (excepto las primitivas de datos, que no son objetos).

A continuación, se muestra un ejemplo con genéricos:

```
ArrayList<String> safer = new ArrayList<>();
safer.add("sparrow");
safer.add(Boolean.TRUE);  // NO COMPILA
```

Esta vez el compilador sabe que sólo están permitidos los objetos String y evita el intento de añadir un boolean.

Continuando con el ejemplo:

```
4: List<String> birds = new ArrayList<>();
5: birds.add("hawk");  // [hawk]
6: birds.add(1, "robin");  // [hawk, robin]
7: birds.add(0, "blue jay");  // [blue jay, hawk, robin]
8: birds.add(1, "cardinal");  // [blue jay, cardinal, hawk, robin]
9: System.out.println(birds);  // [blue jay, cardinal, hawk, robin]
```

En este ejemplo, la línea 5 añade "hawk" al final del ArrayList birds.

La línea 6 agrega "robin" al índice 1 de birds, que resulta ser el final.

La línea 7 agrega "blue jay" al índice 0, que es el la posición inicial del array birds.

Finalmente, la línea 8 agrega "cardinal" al índice 1.

b. remove()

Los métodos remove() eliminan el primer valor que coincida con el parámetro que se pasa al método o eliminan el elemento en un índice especificado. Las cabeceras del método son las siguientes:

```
boolean remove(Object object)
E remove(int index)
```

Esta vez el valor boolean de retorno dice si se eliminó el elemento del ArrayList.

El tipo de retorno E es el elemento que realmente se eliminó.

A continuación se muestra cómo utilizar estos métodos:

```
3: List<String> birds = new ArrayList<>();
4: birds.add("hawk");  // [hawk]
5: birds.add("hawk");  // [hawk, hawk]
6: System.out.println(birds.remove("cardinal"));  // prints false
7: System.out.println(birds.remove("hawk"));  // prints true
8: System.out.println(birds.remove(0));  // prints hawk
9: System.out.println(birds);  // []
```

La línea 6 intenta eliminar un elemento que no está en birds. Devuelve false porque no se encuentra tal elemento.

La línea 7 trata de eliminar un elemento que se encuentra en birds, elimina uno de los String "hawk" y devuelve true. Nótese que elimina sólo una coincidencia.

La línea 8 elimina el elemento en el índice 0, que es el último elemento restante en el ArrayList. Como llamar a remove() con un int usa el índice, un índice que no existe lanzará una excepción.

Por ejemplo, birds.remove(100) lanza un IndexOutOfBoundsException. También hay un método removeIf(). Se cubrirá en el siguiente capítulo porque usa expresiones lambda.

c. set()

El método set() cambia uno de los elementos de ArrayList sin variar el tamaño. La cabecera del método es la siguiente:

```
E set(int index, E newElement)
```

El tipo de retorno E es el elemento que se reemplazó.

A continuación, se muestra cómo utilizar el método:

```
15: List<String> birds = new ArrayList<>();
16: birds.add("hawk");  // [hawk]
17: System.out.println(birds.size());  // 1
18: birds.set(0, "robin");  // [robin]
19: System.out.println(birds.size());  // 1
20: birds.set(1, "robin");  // IndexOutOfBoundsException
```

La línea 16 añade un elemento al array, aumentando el tamaño de este en 1.

La línea 18 reemplaza a ese elemento y el tamaño se mantiene en 1.

La línea 20 intenta reemplazar un elemento que no está en la ArrayList. Dado que el tamaño es 1, el único índice válido es 0. Java lanza una excepción porque esto no está permitido.

d. isEmpty() y size()

Los métodos isEmpty() y size() observan cuántas de las posiciones del ArrayList están en uso. Las cabeceras del método son las siguientes:

```
boolean isEmpty()
int size()
```

Véase un ejemplo de su uso:

```
System.out.println(birds.isEmpty()); // true
System.out.println(birds.size()); // 0
birds.add("hawk"); // [hawk]
birds.add("hawk"); // [hawk, hawk]
System.out.println(birds.isEmpty()); // false
System.out.println(birds.size()); // 2
```

Al principio, birds tiene un tamaño de 0 y está vacío. Tiene; no obstante, una capacidad superior a 0. Sin embargo, al igual que con StringBuilder, no se utiliza la capacidad para determinar el tamaño o la longitud. Después de añadir elementos, el tamaño se vuelve positivo y deja de estar vacío.

e. clear()

El método clear() proporciona una manera fácil de descartar todos los elementos de la ArrayList. La cabecera del método es la siguiente:

```
void clear()
```

Véase un ejemplo de su uso:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");  // [hawk]
birds.add("hawk");  // [hawk, hawk]
System.out.println(birds.isEmpty());  // false
System.out.println(birds.size());  // 2
birds.clear();  // []
```

```
System.out.println(birds.isEmpty());  // true
System.out.println(birds.size());  // 0
```

Después de llamar a clear(), birds vuelve a ser una ArrayList de tamaño 0.

f. contains()

El método contains() comprueba si un valor determinado se encuentra en un ArrayList. La cabecera del método es la siguiente:

```
boolean contains(Object object)
```

Véase un ejemplo de su uso:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");  // [hawk]
System.out.println(birds.contains("hawk"));  // true
System.out.println(birds.contains("robin"));  // false
```

Este método llama a equals() en cada elemento de la ArrayList para ver si hay coincidencias. Como String implementa equals(), el código anterior funciona correctamente.

g. equals()

Finalmente, ArrayList tiene una implementación personalizada de equals() que compara dos listas para ver si contienen los mismos elementos en el mismo orden.

```
boolean equals(Object object)
```

Véase un ejemplo de su uso:

```
31: List<String> one = new ArrayList<>();
32: List<String> two = new ArrayList<>();
33: System.out.println(one.equals(two));  // true
34: one.add("a");  // [a]
35: System.out.println(one.equals(two));  // false
36: two.add("a");  // [a]
37: System.out.println(one.equals(two));  // true
38: one.add("b");  // [a,b]
39: two.add(0, "b");  // [b,a]
40: System.out.println(one.equals(two));  // false
```

En la línea 33, los dos objetos ArrayList son iguales. Una lista vacía es ciertamente los mismos elementos en el mismo orden que otra lista vacía.

En la línea 35, los objetos ArrayList no son iguales porque el tamaño es diferente.

En la línea 37, vuelven a ser iguales porque en cada uno de ellos hay el mismo elemento.

En la línea 40, no son iguales. El tamaño y los valores son los mismos, pero no en el mismo orden.

5.5.4. Clases envolventes (Wrapper classes)

Hasta ahora, sólo se ha trabajado con objetos String en un ArrayList. ¿Qué pasa si se quiere un ArrayList de primitivas de datos?

Cada tipo primitivo tiene una clase envolvente, que es un tipo de objeto que corresponde a la primitiva de dato. La siguiente tabla enumera todas las clases envolventes junto con el constructor para cada una de ellas.

Tipos primitivos	Clases envolventes	Ejemplo de construcción
boolean	Boolean	new Boolean(true)
byte	Byte	new Byte((byte) 1)
short	Short	new Short((short) 1)
int	Integer	new Integer(1)
long	Long	new Long(1)
float	Float	new Float(1.0)
double	Double	new Double(1.0)
char	Character	new Character('c')

Las clases envolventes tienen métodos que permiten transformar el objeto en primitiva de dato.

También hay métodos para convertir un String en una primitiva de datos o en una clase envolvente, y es necesario conocerlos.

Los métodos como parseInt() devuelven una primitiva de datos, y los métodos valueOf() devuelven una clase envolvente. Por ejemplo:

```
int primitive = Integer.parseInt("123");
Integer wrapper = Integer.valueOf("123");
```

La primera línea convierte un String a un int (primitiva de datos). El segundo convierte un String en una clase envolvente de tipo Integer. Si el String introducido no es válido para el tipo dado, Java lanza una excepción.

En estos ejemplos, las letras y los puntos no son válidos para un valor entero:

```
int bad1 = Integer.parseInt("a");  // throws NumberFormatException
Integer bad2 = Integer.valueOf("123.45");  // throws NumberFormatException
```

La siguiente tabla enumera los métodos que se necesitan conocer para crear un objeto de tipo primitiva de datos o clase envolvente a partir de un String. A la hora de escribir código, no se estará tan preocupado por lo que se devuelve en cada método debido al *autoboxing* (sección siguiente).

Clases envolventes	De String a primitivo	De String a clase envolvente
Boolean	Boolean.parseBoolean(“true”);	Boolean.valueOf(“TRUE”);
Byte	Byte.parseByte("1");	Byte.valueOf("2");
Short	Short.parseShort("1");	Short.valueOf("2");
Integer	Integer.parseInt("1");	Integer.valueOf("2");
Long	Long.parseLong("1");	Long.valueOf("2");
Float	Float.parseFloat("1");	Float.valueOf("2.2");
Double	Double.parseDouble("1");	Double.valueOf("2.2");
Character	Ninguno	Ninguno

5.5.5. Autoboxing

Desde Java 5, solo se tiene que escribir el valor primitivo y Java lo convertirá a la clase envolvente que corresponda de manera automática. Esto se llama autoboxing.

Ejemplo:

```
4: List<Double> weights = new ArrayList<>();
5: weights.add(50.5);  // [50.5]
6: weights.add(new Double(60));  // [50.5, 60.0]
7: weights.remove(50.5);  // [60.0]
8: double first = weights.get(0);  // 60.0
```

La línea 5 realiza autoboxing con la primitiva de datos double y la transforma en un objeto Double automáticamente. Después, añade el objeto al ArrayList.

La línea 6 muestra que todavía se puede escribir el código de la manera larga y pasar a un objeto de clase envolvente.

La línea 7 vuelve a realizar autoboxing igual que en la línea 5, solo que esta vez, se elimina el objeto del ArrayList.

La línea 8 recupera el Double y lo convierte en una primitiva de datos double.

Otro ejemplo:

```
3: List<Integer> heights = new ArrayList<>();
4: heights.add(null);
5: int h = heights.get(0);  // NullPointerException
```

En la línea 4, se añade un null a la lista. Esto es legal porque se puede asignar una referencia null a cualquier variable de referencia.

En la línea 5, se trata de convertir ese valor null a una primitiva de datos. Esto no es posible. Java intenta obtener el valor int de un null. Como llamar a cualquier método en null se obtendrá una excepción NullPointerException.

Cuidado al desempaquetar a un Integer:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.remove(1);
System.out.println(numbers);
```

El resultado de la impresión por consola de numbres muestra 1. Después de añadir los dos valores, el ArrayList contiene [1,2]. A continuación, se solicita que se elimine el elemento con índice 1. Así es: índice 1. Debido a que ya existe un método remove() que toma un parámetro int, Java llama a ese método en lugar de realizar *autoboxing*. Si se desea eliminar el 2, se puede escribir numbers.remove(new Integer(2)) para forzar el uso de la clase envolvente.

5.5.6. Conversión entre array y List

Hay que saber cómo convertir entre un array y un ArrayList. Se empezará con convertir una ArrayList en un array:

```
3: List<String> list = new ArrayList<>();
4: list.add("hawk");
5: list.add("robin");
6: Object[] objectArray = list.toArray();
7: System.out.println(objectArray.length);  // 2
8: String[] stringArray = list.toArray(new String[0]);
9: System.out.println(stringArray.length);  // 2
```

La línea 6 muestra que un ArrayList sabe cómo convertirse a un array. El único problema es que está predeterminado para una matriz de clase Objeto. Esto no es normalmente lo que se queire.

La línea 8 especifica el tipo de array y hace lo que realmente se quiere. La ventaja de especificar un tamaño de 0 para el parámetro es que Java creará un nuevo array del tamaño adecuado para el

parámetro valor de retorno. Si se desea, se predefinir un tamaño directamente. Si el ArrayList encaja en ese tamaño, se hará uso de ese array. De lo contrario, se creará uno nuevo.

La conversión de un array a List es menos trivial. Se crea una lista de tamaño fijo, que no un ArrayList, con el tamaño del array. Tanto el array inicial como la lista que se ha creado a partir del array están enlazadas, y lo cambios que se hagan sobre un elemento se podrán ver en el otro, tal y como se muestra en el ejemplo siguiente:

```
20: String[] array = { "hawk", "robin" };  // [hawk, robin]
21: List<String> list = Arrays.asList(array);  // devuelve lista de dimension fija
22: System.out.println(list.size());  // 2
23: list.set(1, "test");  // [hawk, test]
24: array[0] = "new";  // [new, test]
25: for (String b : array) System.out.print(b + " ");  // nuevo test
26: list.remove(1);  // devuelve UnsupportedOperation Exception
```

La Línea 21 convierte el array a una List. Nótese que no es una java.util.ArrayList.

La línea 23 es correcta, porque set() simplemente reemplaza un valor existente. Actualiza tanto el array como la lista porque apuntan al mismo almacén de datos.

La línea 24 también cambia tanto el array como la lista.

La línea 25 muestra que el array ha cambiado, imprimiendo "nuevo test" .

La línea 26 lanza una excepción porque no se permite cambiar el tamaño de la lista.

5.5.7. Sorting

Ordenar un ArrayList es muy similar a ordenar un array.

```
List<Integer> numbers = new ArrayList<>();
numbers.add(99);
numbers.add(5);
numbers.add(81);
Collections.sort(numbers);
System.out.println(numbers);  // [5, 81, 99]
```

Como se puede ver, los números se han ordenado, como se esperaba.

5.6. Trabajando con Fechas y Horas

En Java 8, Oracle renovó completamente la forma de trabajar con fechas y horas. Todavía se puede escribir el código a la "vieja usanza", y a pesar de estar aprendiendo Java a partir de la versión 8, es necesario conocerla tanto para comprender código antiguo como para que resulte más sencilla la comprensión general de cómo funcionan las fechas y horas en Java 8.

Al igual que con ArrayList, se necesitan importar ciertas clases Java para trabajar con fechas y horas. La mayoría de ellas están en el paquete java.time.

```
import java.time.*; // import time classes
```

En las siguientes secciones se verá cómo crear, manipular y dar formato a las fechas y horas.

5.6.1. Creando fechas y horas

En el mundo real, se suele hablar de fechas y zonas horarias como si la otra persona estuviera cerca. Por ejemplo, si se dice "Te llamaré a las 11:00 el martes por la mañana", se asume que las 11:00 significa lo mismo para ambos. Pero si uno vive en Nueva York y el otro en California, es necesario especificar. En California es tres horas más temprano que en Nueva York porque los estados están en diferentes zonas horarias. En vez de eso se dirá: "Te llamaré a las 11:00 EST el martes por la mañana". Cuando se trabaje con fechas y horas, lo primero que debe hacer es decidir cuánta información se necesita. Hay tres opciones:

LocalDate Contiene sólo una fecha, sin hora y sin zona horaria. Un ejemplo de uso de LocalDate es para hacer referencia, por ejemplo, a un cumpleaños. El cumpleaños de alguien es un día completo, sin importar la hora que sea.

LocalTime Contiene sólo una hora, sin fecha y sin zona horaria. Un ejemplo de uso de LocalTime es para hacer referencia a la medianoche. Es medianoche a la misma hora todos los días.

LocalDateTime Contiene una fecha y hora, pero no una zona horaria. Un ejemplo de uso de LocalDateTime es para hacer referencia a la media noche de Año Nuevo., que debe ser un día a una hora concreta.

Oracle recomienda evitar las zonas horarias a menos que realmente se necesiten. Si es necesario hacer uso de zonas horarias existe la clase ZonedDateTime.

Ejemplos:

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
```

Cada una de las tres clases tiene un método estático llamado now() que muestra la fecha y hora actual. Esto dependerá de la fecha y la hora en la que esté el dispositivo desde el que se lance el programa. Por ejemplo, suponiendo que estos métodos se lanzaron desde un ordenador en los Estados Unidos, el 20 de enero a las 12:45 p. m:

```
2015-01-20
12:45:18.401
2015-01-20T12:45:18.401
```

La primera salida contiene sólo una fecha y ninguna hora. La segunda contiene sólo una hora y ninguna fecha, muestra horas, minutos, segundos y nanosegundos. La tercera contiene la fecha y la hora. Java utiliza T para separar la fecha y la hora al convertir LocalDateTime en una cadena.

El cambio de formato en las fechas

En los Estados Unidos, el mes se escribe antes que el día. Los ejemplos usan formatos de fecha y hora de los Estados Unidos (esto es muy común a la hora de buscar información en Internet, ya que pocas veces se podrá encontrar en español). Sólo hay que recordar que el mes viene antes que el día. Además, Java tiende a utilizar un reloj de 24 horas aunque Estados Unidos usa un reloj de 12 horas con a. m. /p. m.

Véanse más ejemplos:

```
LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);
LocalDate date2 = LocalDate.of(2015, 1, 20);
```

Ambas líneas crean objetos de tipo LocalDate(fechas sin horas), pasándoles como parámetro el año, mes y día. Aunque es una buena práctica usar las constantes predefinidas para los meses (para hacer el código más fácil de leer), se puede pasar el número int del mes directamente.

Las cabeceras del método son las siguientes:

```
public static LocalDate of(int year, int month, int dayOfMonth)
public static LocalDate of(int year, Month month, int dayOfMonth)
```

El mes es de un tipo especial de clase llamada **enum** (que queda fuera del ámbito de estudio del curso de Java Básico).

Al crear una hora, se puede elegir el grado de detalle que se desea obtener. Se puede especificar sólo la hora y el minuto, o se puede añadir el número de segundos. Incluso se pueden añadir nanosegundos si se quiere ser muy preciso. (Un nanosegundo es la milmillonésima parte de un segundo, probablemente no habrá que ser tan específico).

```
LocalTime time1 = LocalTime.of(6, 15); // hour and minute
LocalTime time2 = LocalTime.of(6, 15, 30); // + seconds
LocalTime time3 = LocalTime.of(6, 15, 30, 200); // + nanoseconds
```

Las cabeceras del método son las siguientes:

```
public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second, int nanos)
```

Finalmente, para trabajar con fechas y horas:

```
LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 20, 6, 15, 30);
LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);
```

La primera línea de código muestra cómo especificar toda la información sobre LocalDateTime en la misma línea. Sin embargo, tener tantos números seguidos es difícil de leer. La segunda línea de código muestra cómo pasar objetos LocalDate y LocalTime como parámetros al método para crear un objeto LocalDateTime.

Las cabeceras del método son las siguientes:

```
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute,
int second)
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute,
int second, int nanos)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute
)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute
, int second)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute
, int second, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime)
```

¿Se ha notado que no se usa un constructor en ninguno de los ejemplos? Las clases de fecha y hora tienen constructores privados para obligar a utilizar los métodos estáticos.

```
LocalDate d = new LocalDate(); // DOES NOT COMPILE
```

No hay que hacer esto. No se permite construir una fecha o un objeto de hora directamente.

Otro dato de interés es ver lo que pasa cuando se pasa números inválidos a of(). Por ejemplo:

```
LocalDate.of(2015, Month.JANUARY, 32) // throws DateTimeException
```

No se necesita conocer la excepción exacta que se lanza, pero esta es bastante clara:

```
java.time.DateTimeException: Invalid value for DayOfMonth
```

(valores válidos 1 - 28/31): 32

Creación de fechas en Java 7 y anteriores

Puede ver algunos de los problemas con la "vieja manera" de trabajar con fechas y horas en la siguiente tabla. No había forma de especificar una fecha sin la hora. La clase Date representaba tanto la fecha como la hora, tanto si se deseaba como si no. Los índices mensuales se basaban en 0 en lugar de 1, lo que era confuso. En Java 1.1, se creaba una Date especificada con esto: Date jan = nueva Fecha (2015, Calendar. ENERO, 1);. Se podía utilizar la clase Calendar a partir de Java 1.2.

	Vieja manera	*Nueva manera (Java 8 ...)*
Importaciones	*import java.util.*;*	*Impot java.time.*;*
Creando un objeto con la fecha actual	*Date d = new Date();*	*LocalDate d = LocalDate.now();*
Creando un objeto con la fecha y la hora actual	*Date d = new Date();*	*LocalDateTime dt = LocalDateTime.now();*
Creando un objeto representando el 1 de Enero del 2015	*Calendar c = Calendar.getInstance();* *c.set(2015, Calendar.JANUARY, 1);* *Date jan = c.getTime;* *O* *Calendar c = new GregorianCalendar(2015,* *Calendar. JANUARY, 1);* *Date jan = c.getTime();*	*LocalDate jan = LocalDate.of(2015, Month.JANUARY,1);*
Creando el 1 de Enero de 2015 sin constantes	*Calendar c = Calendar.getInstance();* *c.set(2015, 0, 1);* *Date jan = c.getTime();*	*LocalDate jan =* *LocalDate.of(2015,* *1, 1)*

5.6.2. Manipulando fechas y horas

Las clases de fecha y hora son inmutables, como lo era String. Esto significa que se debe recordar asignar los resultados de estos métodos a una variable de referencia para que no se pierdan.

```
12: LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
13: System.out.println(date); // 2014-01-20
```

```
14: date = date.plusDays(2);
15: System.out.println(date); // 2014-01-22
16: date = date.plusWeeks(1);
17: System.out.println(date); // 2014-01-29
18: date = date.plusMonths(1);
19: System.out.println(date); // 2014-02-28
20: date = date.plusYears(5);
21: System.out.println(date); // 2019-02-28
```

Este código hace exactamente lo que parece. Se empieza con el 20 de enero de 2014.

En la línea 14, se añaden dos días y se reasigna a una variable de referencia.

En la línea 16, se suma una semana. Este método permite escribir un código más claro que plusDays(7).

En la línea 18, se añade un mes. Esto llevaría al 29 de febrero de 2014. Sin embargo, 2014 no es un año bisiesto. (2012 y 2016 son años bisiestos.) Java es lo suficientemente inteligente como para darse cuenta de que el 29 de febrero de 2014 no existe y da 28 de febrero de 2014 en su lugar.

Finalmente, la línea 20 añade años.

También existen métodos para restar fechas y horas. Esta vez, se trabajará con LocalDateTime.

```
22: LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
23: LocalTime time = LocalTime.of(5, 15);
24: LocalDateTime dateTime = LocalDateTime.of(date, time);
25: System.out.println(dateTime); // 2020-01-20T05:15
26: dateTime = dateTime.minusDays(1);
27: System.out.println(dateTime); // 2020-01-19T05:15
28: dateTime = dateTime.minusHours(10);
29: System.out.println(dateTime); // 2020-01-18T19:15
30: dateTime = dateTime.minusSeconds(30);
31: System.out.println(dateTime); // 2020-01-18T19:14:30
```

La línea 25 imprime la fecha del 20 de enero de 2020 a las 5:15 a. m.

La línea 26 resta un día entero, imprimiendo 19 de enero de 2020 a las 5:15 a. m.

La línea 28 resta 10 horas, mostrando que la fecha cambiará si es necesario, e imprime 18 de enero de 2020 a las 19:15 (7:15 p. m.).

Finalmente, la línea 30 resta 30 segundos. Se ve que, de repente, el valor empieza a mostrar segundos. Java es lo suficientemente inteligente como para ocultar los segundos y nanosegundos cuando no se usan.

Es común que los métodos de fecha y hora estén encadenados. Por ejemplo, sin las instrucciones de impresión, el ejemplo anterior podría volver a escribirse del siguiente modo:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date2, time).minusDays(1).minusHours(10).minusSe
conds(30);
```

Cuando se tiene muchas manipulaciones que hacer, este encadenamiento es útil. ¿Qué imprimirá esto?

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date.plusDays(10);
System.out.println(date);
```

Imprime el 20 de enero de 2020. Agregar 10 días fue inútil porque se ignora el resultado. Siempre que se vean tipos inmutables, hay que asegurarse de que el valor de retorno de una llamada de método se guarda en alguna parte, pero esta vez, no es el caso.

¿Qué está mal aquí?

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date = date.plusMinutes(1); // DOES NOT COMPILE
```

LocalDate no contiene la hora, por lo que que no se le pueden añadir minutos. Esto puede ser complicado en una secuencia encadenada de operaciones de sumas/restas, así que hay que asegurarse de saber qué métodos de la Tabla 3.4 pueden ser llamados en cuál de los tres objetos.

	Se puede llamar desde LocalDate?	Se puede llamar desde LocalTime?	Se puede llamar desde LocalDateTime?
plusYears/minusYears	Yes	No	Yes
plusMonths/minusMonths	Yes	No	Yes
plusWeeks/minusWeeks	Yes	No	Yes
plusDays/minusDays	Yes	No	Yes
plusHours/minusHours	No	Yes	Yes
plusMinutes/minusMinutes	No	Yes	Yes
plusSeconds/minusSeconds	No	Yes	Yes
plusNanos/minusNanos	No	Yes	Yes

Manipulación de fechas en Java 7 y anteriores

Al mirar todo el código en la siguiente tabla para hacer cálculos de tiempo a la "vieja manera", se puede ver por qué Java necesitaba renovar las APIs de fecha y hora, La "vieja manera" requería mucho código para hacer algo simple.

	Vieja manera	*Nueva manera (Java 8 ...)*
Añadiendo un día	*public Date addDay(Date date) {* *Calendar cal = Calendar* *. getInstance();* *cal.setTime(date);* *cal.add(Calendar.DATE, 1);* *return cal.getTime();* *}*	*public LocalDate* *addDay(LocalDate date) {* *return date.* *plusDays(1);* *}*
Quitando un día	*public Date subtractDay(Date date)* *{* *Calendar cal = Calendar* *. getInstance();* *cal.setTime(date);* *cal.add(Calendar.DATE, -1);* *return cal.getTime();* *}*	*public LocalDate* *subtractDay(LocalDate* *date) {* *return date.* *minusDays(1);* *}*

5.6.3. Trabajar con Periodos

Supóngase el siguiente ejemplo: un zoológico realiza actividades de ocio para darles a los animales algo de diversión. El jefe del zoológico ha decidido cambiar los juguetes cada mes. Este sistema continuará durante tres meses para ver cómo funciona.

```
public static void main(String[] args) {
  LocalDate start = LocalDate.of(2015, Month.JANUARY, 1);
  LocalDate end = LocalDate.of(2015, Month.MARCH, 30);
  performAnimalEnrichment(start, end);
```

```
}
private static void performAnimalEnrichment(LocalDate start, LocalDate end) {
  LocalDate upTo = start;
  while (upTo.isBefore(end)) { // check if still before end
    System.out.println("give new toy: " + upTo);
    upTo = upTo.plusMonths(1); // add a month
  }
}
```

Este código funciona correctamente. Añade un mes a la fecha hasta que alcanza la fecha final.

a. Convertir a un long

LocalDate y LocalDateTime tienen un método para convertirlos en equivalentes long en relación a 1970. ¿Qué tiene de especial 1970? Eso es la fecha que UNIX empezó a usar para los estándares de fecha, así que Java lo reutilizó.

- LocalDate tiene toEpochDay(), que es el número de días desde el 1 de enero de 1970.
- LocalDateTime tiene toEpochTime(), que es el número de segundos desde el 1 de enero, 1970.
- LocalTime no tiene este método. Como representa un tiempo que ocurre en cualquier fecha, no tiene sentido basarse en 1970.

Java tiene una clase de Period para determinar periodos de tiempo. Este código hace lo mismo que el ejemplo anterior:

```
public static void main(String[] args) {
  LocalDate start = LocalDate.of(2015, Month.JANUARY, 1);
  LocalDate end = LocalDate.of(2015, Month.MARCH, 30);
  Period period = Period.ofMonths(1); // create a period
  performAnimalEnrichment(start, end, period);
}
private static void performAnimalEnrichment(LocalDate start, LocalDate end, Period period)
{ // uses the generic period
  LocalDate upTo = start;
  while (upTo.isBefore(end)) {
    System.out.println("give new toy: " + upTo);
    upTo = upTo.plus(period); // adds the period
  }
}
```

El método añade un Period como parámetro. Esto permite reutilizar el mismo método para diferentes periodos de tiempo a medida que el jefe del zoo cambia de opinión. Hay cinco maneras de crear una clase Period:

```
Period annually = Period.ofYears(1); // every 1 year
Period quarterly = Period.ofMonths(3); // every 3 months
Period everyThreeWeeks = Period.ofWeeks(3); // every 3 weeks
Period everyOtherDay = Period.ofDays(2); // every 2 days
Period everyYearAndAWeek = Period.of(1, 0, 7); // every year and 7 days
```

Hay que tener en cuenta lo siguiente: no se pueden encadenar métodos al crear un Period. El código siguiente parece que es equivalente al ejemplo de cada año y semana, pero no lo es. Sólo se utiliza el último método porque los métodos Period.ofXXX son métodos estáticos.

```
Period wrong = Period.ofYears(1).ofWeeks(1); // every week
```

Esto es lo mismo que escribir lo siguiente:

```
Period wrong = Period.ofYears(1);
wrong = Period.ofWeeks(7);
```

Está claro que esto no es lo que se pretendía. Es por eso que el método of() nos permite pasar en el número de años, meses y días.

Period sirve para crear periodos de, como mínimo, un día. También existe la clase Duration, que crea periodos de unidades de tiempo más pequeñas. Para Duration, se puede especificar el número de días, horas, minutos, segundos o nanosegundos. Y sí, se podría pasar 365 días para ganar un año, pero no es lo correcto, para eso está la clase Period.

Lo último que hay que saber sobre Period es con qué objetos se puede utilizar. Ejemplos:

```
3: LocalDate date = LocalDate.of(2015, 1, 20);
4: LocalTime time = LocalTime.of(6, 15);
5: LocalDateTime dateTime = LocalDateTime.of(date, time);
6: Period period = Period.ofMonths(1);
7: System.out.println(date.plus(period)); // 2015-02-20
8: System.out.println(dateTime.plus(period)); // 2015-02-20T06:15
9: System.out.println(time.plus(period)); // UnsupportedTemporalTypeException
```

Las líneas 7 y 8 añaden un mes más al 20 de enero de 2015, imprimiendo por consola el 20 de febrero de 2015. La primera linea tiene sólo la fecha y la segunda la fecha y la hora.

La línea 9 intenta añadir un mes a un objeto que sólo tiene tiempo. Esto no funcionará. Java lanza una excepción.

5.6.4. Formatear Fechas y Tiempos

Las clases de fecha y hora tienen gran cantidad de métodos que ofrecen multitud de utilidades:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
System.out.println(date.getDayOfWeek()); // MONDAY
System.out.println(date.getMonth()); // JANUARY
System.out.println(date.getYear()); // 2020
System.out.println(date.getDayOfYear()); // 20
```

Se podrían utilizar los métodos que se ven en esta impresión por consola para mostrar información sobre la fecha; sin embargo, sería más trabajo del necesario. Java proporciona una clase llamada DateTimeFormatter para ayudar. A diferencia de la clase LocalDateTime, DateTimeFormatter se puede utilizar para dar formato a cualquier tipo de objeto de fecha y/o hora. Lo que cambia es únicamente el formato. DateTimeFormatter está en el paquete java.time.format.

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);

System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

ISO es una norma para las fechas. La salida del código anterior se ve así:

```
2020-01-20
11:12:34
2020-01-20T11:12:34
```

Esta es una manera razonable para que las máquinas se comuniquen, pero probablemente no es la forma en la que se quiere que aparezca la fecha y la hora en el programa. Afortunadamente hay algunos formatos predefinidos que son más útiles:

```
DateTimeFormatter shortDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(shortDateTime.format(dateTime)); // 1/20/20
System.out.println(shortDateTime.format(date)); // 1/20/20
System.out.println(shortDateTime.format(time)); // UnsupportedTemporalTypeException
```

Aquí se especifica que se quiere un formateo en el formato corto predefinido. La última línea lanza una excepción porque un LocalTime no se puede formatear como una fecha. El método format() se puede utilizar tanto en los objetos Formatter como en los objetos de fecha/hora, lo que le permite actuar sobre los objetos en cualquier orden.

Las impresiones por consola del código siguiente imprime exactamente lo mismo que el código anterior:

```
DateTimeFormatter shortDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(dateTime.format(shortDateTime));
System.out.println(date.format(shortDateTime));
System.out.println(time.format(shortDateTime));
```

La tabla siguiente muestra los métodos de formateo legal e ilegal localizados.

DateTimeFormatter f = DateTime Formatter._____ (FormatStyle.SHORT);	Llamando a f.format(localDate)	Llamando a f.format(localDateTime)	Llamando a f.format(localTime)
ofLocalizedDate	Legal - muestra todo el objeto	Legal – muestra solo la parte de la fecha	Lanza una runtime exception
ofLocalizedDateTime	Lanza una runtime exception	Legal – muestra todo el objeto	Lanza una runtime exeption
ofLocalizedTime	Lanza una runtime exception	Legal – muestra solo la parte de la hora	Legal muestra todo el objeto

Hay dos formatos predefinidos: SHORT y MEDIUM. Los otros formatos predefinidos se refieren a zonas horarias.

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
DateTimeFormatter shortF = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter mediumF = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(shortF.format(dateTime)); // 1/20/20 11:12 AM
System.out.println(mediumF.format(dateTime)); // Jan 20, 2020 11:12:34 AM
```

Si no se desea utilizar uno de los formatos predefinidos, se puede crear uno personalizado. Por ejemplo, este código escribe el nombre completo del mes (sin abreviar):

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
System.out.println(dateTime.format(f)); // January 20, 2020, 11:12
```

MMMM - M representa el mes. Cuanta más Ms se tenga, más verbosa será la salida en Java. Por ejemplo, M imprimirá 1, MM imprimirá 01, MMM imprimirá Jan y MMMM imprimirá Enero.

dd - d representa el número del día. Al igual que con M, mientras más des se tenga, más verbosa será la salida Java. dd incluirá el cero inicial para un día de un solo dígito.

yyyy - y representa el año. yy imprimirá un año de dos dígitos e yyyy imprimirá un año de cuatro dígitos.

hh - h representa la hora. Utilizar hh para incluir incluirá el cero inicial para una hora de un solo dígito.

mm - m representa el minuto. mm incluirá el cero inicial para un minuto de un solo dígito.

Formateo de fechas en Java 7 y anteriores

En vez de utilizar Formatter, se utilizaba otra clase.

	Vieja manera	*Nueva manera*
Formateando las horas	*SimpleDateFormat sf = new SimpleDateFormat("hh:mm"); sf.format(jan3);*	*DateTimeFormatter f = DateTimeFormatter. ofPattern("hh:mm"); dt.format(f);*

Véase un ejemplo más referido al formato de fechas. ¿Alguna parte del código siguiente lanzará una excepción?

```
4: DateTimeFormatter f = DateTimeFormatter.ofPattern("hh:mm");
5: f.format(dateTime);
6: f.format(date);
7: f.format(time);
```

Se debe reparar en lo que representan los símbolos hh y mm. Se tiene h para hora y m para minuto. Recordar que M (mayúsculas) es un mes y m (en minúsculas) es un minuto. Sólo se puede utilizar este formato con objetos que contengan horas. Por lo tanto, la línea 6 lanzará una excepción.

5.6.5. Parseando fechas y horas

Ahora que ya se sabe cómo convertir una fecha u hora a un String formateado, resultará fácil convertir un String a una fecha u hora. Al igual que el método format(), el método parse() también toma un Formatter. Si no se especifica uno, utiliza el valor predeterminado para ese tipo.

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);
LocalTime time = LocalTime.parse("11:22");
System.out.println(date); // 2015-01-02
System.out.println(time); // 11:22
```

Aquí se muestra el uso de un formateador personalizado y un valor por defecto. Esto no es común. El *parsing* es consistente en que si algo sale mal, Java lanza una Runtime Exception. Puede ser un formato que no coincida con el formato de la cadena al ser *parseado* o una fecha inválida.

5.7. Resumen

En este capítulo, se ha aprendido que:

String

- Los Strings son secuencias inmutables de caracteres.

- El operador new es opcional.

- El operador de concatenación (+) crea un nuevo String con el contenido del primer String seguido del contenido del segundo String. Si cualquiera de los operandos involucrados en la expresión + es una cadena, se utiliza concatenación; de lo contrario, se utiliza la suma.

- Los String literales se almacenan en la String Pool.

- La clase String tiene muchos métodos. Los más habituales son: charAt(), concat(), endsWith(), equals(), equalsIgnoreCase(), indexOf(), length(), replace(), startWith(), substring(), toLowerCase(), toUpperCase() y trim().

StringBuilder

- Los StringBuilder son secuencias mutables de caracteres.

- La clase StringBuilder tiene muchos métodos. Los más habituales son: append(), charAt(), delete(), deleteCharAt(), indexOf(), insert(), length(), reverse(), substring() y toString().

- La mayoría de los métodos devuelven una referencia al objeto actual para permitir el encadenamiento de métodos.

Igualdad

- Llamar == en objetos String comprobará si apuntan al mismo objeto en el Pool.

- Llamando == en las referencias de StringBuilder se comprobará si están apuntando al mismo objeto StringBuilder.

- Llamar a equals() en objetos String comprobará si la secuencia de caracteres es la misma.

- Llamar a equals() en los objetos StringBuilder comprobará si están apuntando al mismo objeto en lugar de mirar el valor de los caracteres que lo componen.

Array

- Un array es un área de memoria de tamaño fijo que tiene espacio para primitivas de datos o punteros a objetos. Se especifica el tamaño al crearlo; por ejemplo, int[] a = new int[6];.

- Los índices de los arrays comienzan con 0 y los elementos son referidos usando a[0].

- El método Arrays.sort() ordena un array.

- Arrays.binarySearch() busca en un array ordenada y devuelve el índice de la primera coincidencia. Si no se encuentra ninguna coincidencia, niega la posición donde el elemento debe ser insertado y resta 1.

- Los métodos que son pasados como varargs (...) pueden ser usados como un array normal.

- En un array multidimensional, los arrays de segundo nivel y posteriores pueden ser de diferentes tamaños.

ArrayList

- Un ArrayList puede cambiar de tamaño durante su vida útil.

- Se puede almacenar en un ArrayList o en una referencia de List.

- Los genéricos pueden especificar el tipo de dato que va en la ArrayList.

- Los métodos más habituales de ArrayList son: add(), clear(), contains(), contains(), equals(), isEmpty(), remove(), set(), y size().

- Aunque no se permite que una ArrayList contenga primitivas de datos, Java realizará *autoboxing* a los parámetros pasados al tipo apropiado.

- Collections.sort() ordena un ArrayList.

Fechas y horas

- Un LocalDate contiene sólo una fecha, un LocalTime contiene sólo una hora y un LocalDateTime contiene una fecha y una hora. Los tres tienen constructores privados y se crean usando LocalDate.now () o LocalDate.of() (o los equivalentes para esa clase).

- Las fechas y las horas se pueden manipular utilizando los métodos plusXXX o minusXXX.

- La clase Period representa un número de días, meses o años para sumar o restar de un LocalDate o LocalDateTime.

- DateTimeFormatter se utiliza para imprimir fechas y horas en el formato deseado. Las clases de fecha y hora son todas inmutables, lo que significa que se debe utilizar el valor de retorno.

6. Métodos y encapsulamiento

6.1. Diseñando Métodos

La mayoría de programas Java poseen un método main(). Por supuesto, se pueden escribir en un programa Java tantos métodos como se desee, por ejemplo, se puede escribir un método básico como el siguiente:

```
public final void nap(int minutes) throws InterruptedException {
  // take a nap
}
```

El código anterior muestra la declaración de un método llamado nap, donde se especifica toda la información necesaria para llamarlo y para implementar su funcionalidad. Un método está conformado por diferentes partes.

La tabla siguiente es una breve muestra de los elementos que se usan en la declaración del método nap.

Elemento	Valor en el ejemplo	Necesario?
Modificador de acceso	public	No
Especificador opcional	final	No
Tipo de retorno	void	Si
Nombre del método	nap	Si
Lista de parámetros	(int minutes)	Sí, pero puede estar vacía
Lista de excepciones opcionales	throws InterruptedException	No

Cuerpo del método	{
	// take a nap
	}
		Sí, pero puede estar vacío

Para llamar a este método, solo hace falta escribir su nombre, seguido de un valor int entre paréntesis:

```
nap(10);
```

6.1.1. Modificadores de acceso

Java ofrece cuatro opciones de modificadores de acceso:

public: El método puede ser llamado desde cualquier clase.

private: El método solo puede ser llamado desde la misma clase.

protected: El método solo puede ser llamado por clases del mismo paquete o subclases.

***default* (paquete privado)**: El método sólo se puede llamar desde clases del mismo paquete. No hay ninguna palabra clave para el acceso por defecto. Simplemente se omite el modificador de acceso (es decir, no se pone nada).

Se explorará el ámbito de los distintos modificadores de acceso más adelante en este capítulo.

A continuación se muestran una serie de ejemplos referentes a la declaración de un método en Java con el fin de ver si son válidas o no lo son. Es necesario prestar atención a los modificadores de acceso y dónde están situados.

```
public void walk1() {}
default void walk2() {} // DOES NOT COMPILE
void public walk3() {} // DOES NOT COMPILE
void walk4() {}
```

walk1() es una declaración de método válida con acceso público.

walk4() es una declaración de método válida con acceso por defecto.

walk2() no compila porque default no es un modificador de acceso válido.

walk3() no compila porque el modificador de acceso se especifica después del tipo de retorno.

6.1.2. Especificadores opcionales

Hay una serie de especificadores opcionales en Java. A diferencia de los modificadores de acceso, se puede tener múltiples especificadores en el mismo método (aunque no todas las combinaciones son legales). Cuando esto sucede, se pueden especificar en cualquier orden. Y como es opcional, también puede no tenerse ninguno de ellos. Esto significa que se puede tener cero o más especificadores en una declaración de método.

static: Usado para métodos de clase.

abstract: Usado cuando no se especifica el cuerpo del método.

final: Se utiliza para indicar que un método no puede ser sobreescrito.

native: Se utiliza al interactuar con código escrito en otro idioma como C++.

strictfp: Se utiliza para hacer que los cálculos de coma flotante sean portables.

synchronized

De nuevo, sólo hay que concentrarse en la sintaxis por ahora. ¿Por qué las siguientes líneas de código compilan o no compilan?

```
public void walk1() {}
public final void walk2() {}
public static final void walk3() {}
public final static void walk4() {}
public modifier void walk5() {} // DOES NOT COMPILE
```

```
public void final walk6() {} // DOES NOT COMPILE
final public void walk7() {}
```

walk1 () es una declaración de método válida sin especificador opcional.

walk2 () es una declaración de método válida, con final como especificador opcional.

walk3 () y walk4 () son declaraciones de método válidas con final y static como especificadores opcionales. El orden de estas dos palabras clave no importa.

walk5 () no compila porque el modificador no es un especificador opcional válido.

walk6 () no compila porque el especificador opcional está después del tipo de retorno.

walk7() compila. Java permite que los especificadores opcionales aparezcan antes de la modificación de acceso.

6.1.3. Tipo de retorno

El siguiente elemento en una declaración de método es el tipo de retorno. El tipo de retorno puede ser un tipo de dato Java como String o int, o un tipo de dato propio. Si no hay ningún tipo de devolución, se utiliza la palabra clave **void**. Este tipo especial return proviene del idioma inglés: void significa vacío, sin contenido.

Cuando se comprueban los tipos de retorno, también hay que mirar dentro del cuerpo del método. Los métodos con un tipo de retorno que no sea nulo deben tener una declaración **return** dentro del cuerpo del método. Esta declaración de retorno debe incluir el primitivo u objeto a devolver. A los métodos que tienen un tipo de devolución void se les permite tener una declaración return sin ningún valor devuelto u omitir completamente la declaración del return.

¿Por qué estos métodos compilan o no?

```
public void walk1() { }
public void walk2() { return; }
public String walk3() { return ""; }
public String walk4() { } // DOES NOT COMPILE
public walk5() { } // DOES NOT COMPILE
String walk6(int a) { if (a == 4) return ""; } // DOES NOT COMPILE
```

Puesto que el tipo de vuelta de walk1() es void, el return es opcional.

walk2() muestrael return opcional que no devuelve nada correctamente.

walk3() es un método válido con un tipo de devolución String y un return que devuelve un String.

walk4() no compila porque falta el return.

walk5 () no compila porque falta el tipo de retorno.

walk6 () es un poco más complicado. Hay un return, pero no siempre se ejecuta debido a la condición. Si a es 6, por ejemplo, el return no se ejecuta. Como el método siempre ha de devolver un String, hay un error de compilación.

Al devolver un valor, debe ser asignable al tipo de retorno.

```
int integerMethod() {
  return 9;
}
int longMethod(){
  return 9L; // DOES NOT COMPILE
}
```

Es posible utilizar variables locales dentro de un método, como se muestra en el siguiente código. El tipo de la variable local coincide con el tipo de retorno del método. A continuación, devuelve esa variable local en lugar del valor directamente:

```
int integerExpanded() {
  int temp = 9;
  return temp;
}


int longExpanded() {
  int temp = 9L; // DOES NOT COMPILE
  return temp;
}
```

Esto muestra más claramente por qué no se puede devolver un long primitivo en un método que devuelve un int. No se puede asignar un long a una variable int, por lo que tampoco se puede devolver directamente.

6.1.4. Nombre del método

Los nombres de métodos siguen las mismas reglas que los nombres de variables en "Bloques de construcción Java". Para revisar, un identificador sólo puede contener letras, números, $ o _. Además, no se permite que el primer carácter sea un número y no se permiten las palabras reservadas. Por convención, los métodos comienzan con una letra minúscula, pero no es obligatorio.

Véanse algunos ejemplos:

```
public void walk1() { }
public void 2walk() { } // DOES NOT COMPILE
public walk3 void() { } // DOES NOT COMPILE
public void Walk_$() { }
public void() { } // DOES NOT COMPILE
```

walk1() es una declaración de método válida con un nombre tradicional.

2walk () no compila porque los identificadores no pueden comenzar con números.

walk3() no compila porque el nombre del método esta antes que el tipo de retorno.

Walk_$() es una declaración de método válida. Aunque ciertamente no es una buena práctica comenzar un nombre de método con letra mayúscula y terminar con puntuación, es legal.

La última línea de código no se compila porque falta el nombre del método.

6.1.5. Lista de parámetros

Aunque la lista de parámetros es necesaria, no tiene por qué contener ningún parámetro. Esto significa que se puede tener sólo un par de paréntesis vacíos después del nombre del método, como por ejemplo void nap(){}.

Si se tienen varios parámetros, se separan con una coma. Hay un par de reglas más para la lista de parámetros que se verán cuando se cubra "varargs" en el apartado siguiente del capítulo.

```
public void walk1() { }
public void walk2 { } // DOES NOT COMPILE
public void walk3(int a) { }
public void walk4(int a; int b) { } // DOES NOT COMPILE
public void walk5(int a, int b) { }
```

walk1() es una declaración de método válida sin ningún parámetro.

walk2() no compila porque le falta los paréntesis de la lista de parámetros.

walk3() es una declaración de método válida con un parámetro.

walk4() no compila porque los parámetros están separados por un punto y coma en lugar de una coma. Los puntos y coma son para separar sentencias, no listas de parámetros.

walk5() es una declaración de método válida con dos parámetros.

6.1.6. Lista de excepciones opcionales

En Java, el código puede indicar que algo salió mal lanzando una excepción. Se cubrirá en el Capítulo 8, "Excepciones". Por ahora, sólo se necesita saber que es opcional el propagar excepciones en la declaración de un método.

En el ejemplo, InterruptedException es un tipo de excepción. Las excepciones que se provocan en un método se propagan añadiendo la palabra reservada **throws** después de los argumentos del método. Se pueden propagar tantos tipos de excepciones como se desee, separadas por comas. Por ejemplo:

```
public void zeroExceptions() { }
public void oneException() throws IllegalArgumentException {}
public void twoExceptions() throws IllegalArgumentException, InterruptedException { }
```

6.1.7. Cuerpo del método

La parte final de una declaración de un método es el cuerpo del método (excepto para los métodos abstractos e interfaces, pero no es necesario conocer ninguno de ellos hasta el próximo capítulo). Un cuerpo de método es simplemente un bloque de código. Tiene llaves que contienen cero o más sentencias Java.

Véase el siguiente código:

```
public void walk1() { }
public void walk2; // DOES NOT COMPILE
public void walk3(int a) { int name = 5; }
```

walk1() es una declaración de método válida con un cuerpo de método vacío.

walk2() no compila porque le faltan los paréntesis de los parámetros.

walk3() es una declaración de método válida con una sentencia en el cuerpo del método.

6.2. Trabajando con Varargs

Como se vio en secciones anteriores del capítulo, un método puede utilizar un parámetro vararg (argumento de tamaño variable) como si fuera un array.

Un parámetro vararg debe ser el último elemento de la lista de parámetros de un método. Esto implica que sólo se permite tener un parámetro vararg por método.

Véase por qué cada de las siguientes líneas de código compila o no compila:

```
public void walk1(int... nums) { }
public void walk2(int start, int... nums) { }
public void walk3(int... nums, int start) { } // DOES NOT COMPILE
public void walk4(int... start, int... nums) { } // DOES NOT COMPILE
```

walk1() es una declaración de método válida con un parámetro vararg.

walk2() es una declaración de método válida con un parámetro int y un parámetro vararg.

walk3() y walk4() no compilan porque tienen un parámetro vararg en una posición que no es la última.

Cuando se llama a un método con un parámetro vararg, se puede elegir pasar sus valores en un array, o se pueden pasar los elementos del array y dejar que Java lo cree automáticamente. Incluso se pueden omitir los valores de vararg en la llamada al método y Java creará un array de tamaño cero.

Véase el siguiente código con un ejemplo de vararg:

```
15: public static void walk(int start, int... nums) {
16:   System.out.println(nums.length);
17: }
18: public static void main(String[] args) {
```

```
19:   walk(1); // 0
20:   walk(1, 2); // 1
21:   walk(1, 2, 3); // 2
22:   walk(1, new int[] {4, 5}); // 2
23: }
```

La línea 19 pasa 1 como parámetro, pero nada más. Esto significa que Java crea un array de tamaño 0 para nums.

La línea 20 pasa 1 como parámetro, y un valor más. Java convierte este valor a un array de tamaño 1.

La línea 21 pasa 1 como parámetro, y dos valores más. Java convierte estos dos valores a un array de tamaño 2.

La línea 22 pasa 1 como parámetro, y un array de tamaño 2 directamente.

Se ha visto que Java creará un array vacío si no se pasa ningún parámetro para un vararg. ¿Qué ocurre si se pasa explícitamente null?

```
walk(1, null); // throws a NullPointerException
```

Puesto que null no es un int, Java lo trata como una referencia a un array que resulta ser null. Entonces el método walk() lanza una excepción porque intenta determinar la longitud de null.

Acceder a un parámetro vararg es también como acceder a un array. Por ejemplo:

```
16: public static void run(int... nums) {
17:   System.out.println(nums[1]);
18: }
19: public static void main(String[] args) {
20:   run(11,22);  // 22
21: }
```

La línea 20 llama a un parámetro vararg de dos parámetros. Cuando el método es llamado, ve una matriz de tamaño 2. Puesto que los índices están basados en 0, se imprime 22.

6.3. Aplicando Modificadores de Acceso

Ya se ha visto que hay cuatro modificadores de acceso: público, privado, protegido y por defecto. A continuación se muestran de más restrictivo a menos restrictivo:

private: Accesible solo desde la misma clase.

default (package private) access: private y otras clases en el mismo package.

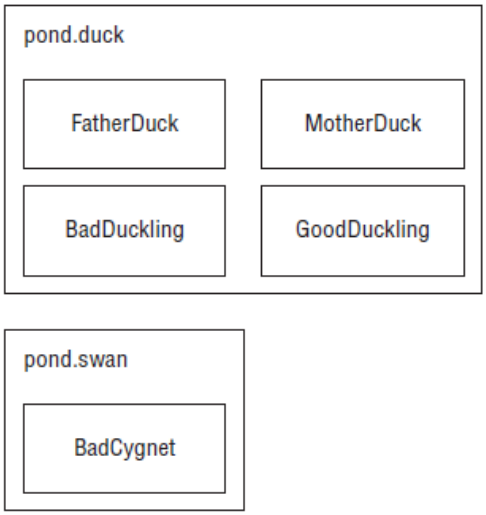
protected: default access y clases hijas.

public: protected y clases de otros packages.

6.3.1. Acceso private

El acceso privado es sencillo. Sólo el código de la misma clase puede llamar a métodos privados o acceder a variables privadas.

Con respecto a la figura siguiente, se pueden ver las clases que se usarán para mostrar ejemplos del acceso privado y predeterminado (default). Las cajas grandes son los nombres de los paquetes. Las cajas más pequeñas dentro de ellas son las clases en cada paquete. Se puede consultar esta figura si se desea ver rápidamente cómo se relacionan las clases de los siguientes ejemplos.



Este es un código sin errores de compilación ya que todo está en una clase:

```
1: package pond.duck;
2: public class FatherDuck {
3:   private String noise = "quack";
4:   private void quack() {
5:     System.out.println(noise); // private access is ok
6:   }
7:   private void makeNoise() {
8:     quack(); // private access is ok
9:   } }
```

Hasta ahora, todo bien. FatherDuck hace una llamada al método privado quack() en la línea 8 y utiliza la variable de instancia privada noise en la línea 5.

Ahora se añadirá otra clase:

```
1: package pond.duck;
2: public class BadDuckling {
3:   public void makeNoise() {
4:     FatherDuck duck = new FatherDuck();
```

```
5:     duck.quack(); // DOES NOT COMPILE
6:     System.out.println(duck.noise); // DOES NOT COMPILE
7:   } }
```

En la línea 5, se intenta acceder a un método privado de otra clase.

En la línea 6, intenta acceder a una variable de instancia privada de otra clase. Ambos generan errores en el compilador.

El acceso a componentes privados de otras clases no está permitido y se necesita usar otro tipo de acceso.

6.3.2. Acceso default (Package Private)

Afortunadamente, MotherDuck es más complaciente con lo que sus patitos pueden hacer. Ella permite a las clases del mismo paquete acceder a sus miembros. Cuando no hay ningún modificador de acceso, Java usa el predeterminado, que es el acceso privado de paquetes. Esto significa que el miembro es "privado" a las clases en el mismo paquete. En otras palabras, sólo las clases del paquete pueden acceder a él.

```
package pond.duck;
public class MotherDuck {
  String noise = "quack";
  void quack() {
    System.out.println(noise); // default access is ok
  }
  private void makeNoise() {
    quack(); // default access is ok
  }
}
```

MotherDuck puede llamar a quack () y referirse a noise. Después de todo, los métodos y variables de la misma clase están ciertamente en el mismo paquete. La gran diferencia es que MotherDuck permite a otras clases del mismo paquete tener acceso a sus métodos o variables(debido a que son default) mientras que FatherDuck no lo tiene (debido a que son private). GoodDuckling tiene una experiencia mucho mejor que BadDuckling:

```
package pond.duck;
public class GoodDuckling {
  public void makeNoise() {
    MotherDuck duck = new MotherDuck();
    duck.quack(); // default access
    System.out.println(duck.noise); // default access
  }
}
```

GoodDuckling logra llamar a quack() e imprimir a noise copiando a su padre. Nótese que todas las clases que se han cubierto hasta ahora están en el mismo paquete pond.duck. Esto permite el acceso predeterminado (default) al trabajo.

En este mismo estanque, un cisne acaba de dar a luz a un cisne bebé. Un cisne bebé se llama BadCygnet. BadCygnet quiere acceder a las variables y métodos de MotherDuck.

```
package pond.swan;
import pond.duck.MotherDuck; // import another package
public class BadCygnet {
  public void makeNoise() {
    MotherDuck duck = new MotherDuck();
    duck.quack(); // DOES NOT COMPILE
    System.out.println(duck.noise); // DOES NOT COMPILE
  }
}
```

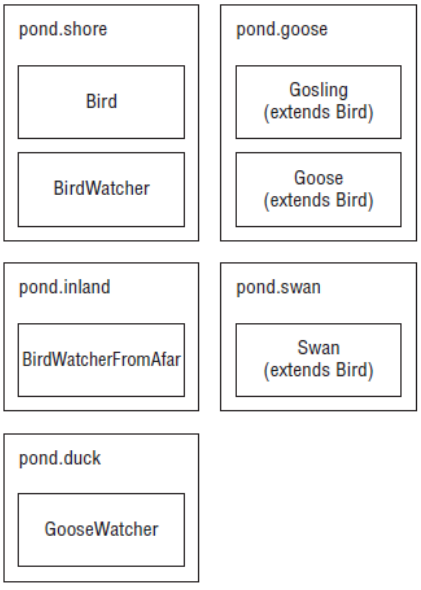
MotherDuck sólo permite acceder a sus métodos y variables a aquellas clases que estén en el mismo paquete que ella (pond.duck). BadCygnet; no obstante, está en el paquete pond.swan, por lo que no tiene acceso a las variables ni métodos de MotherDuck.

Se remarca que cuando no hay modificación de acceso en un método o variable, sólo las clases del mismo paquete pueden acceder dichos métodos o variables.

6.3.3. Acceso protected

El acceso protegido permite todo lo que el acceso default (paquete privado) permite y más. El modificador de acceso protegido agrega la posibilidad de acceder a los componentes de una clase padre. Se abordará la creación de subclases en profundidad en el Capítulo 7.

La siguiente figura muestra las clases que se van a crear en esta sección.



Primero, se crea una clase Bird y se da acceso protected a sus variables y métodos:

```
package pond.shore;
public class Bird {
  protected String text = "floating"; // protected access
  protected void floatInWater() { // protected access
    System.out.println(text);
  }
}
```

A continuación, se crea una subclase:

```
package pond.goose;
import pond.shore.Bird; // in a different package
public class Gosling extends Bird { // extends means create subclass
  public void swim() {
    floatInWater(); // calling protected member
    System.out.println(text); // calling protected member
  }
}
```

Es una subclase muy sencilla. Extiende la clase Bird. Extender significa crear una subclase que tenga acceso a cualquier método o variable protected o public de la clase padre. Al ejecutar este código se imprime "floating" dos veces: una vez desde floatInWater() y una vez desde la sentencia print en swim(). Dado que Gosling es una subclase de Bird, puede acceder a estos métodos y variables, aunque esté en un paquete diferente.

Protected también da acceso a todo lo que hace el acceso default. Esto significa que una clase del mismo paquete que Bird puede acceder a sus métodos y variables también.

```
package pond.shore; // same package as Bird
public class BirdWatcher {
  public void watchBird() {
    Bird bird = new Bird();
    bird.floatInWater(); // calling protected member
    System.out.println(bird.text); // calling protected member
  }
}
```

Ya que Bird y BirdWatcher están en el mismo paquete, BirdWatcher puede acceder a variables y métodos protected de Bird. La definición de protected permite el acceso a subclases y clases dentro del mismo paquete.

Ahora se va a intentar lo mismo desde un paquete diferente:

```
package pond.inland;
import pond.shore.Bird; // different package than Bird
public class BirdWatcherFromAfar {
  public void watchBird() {
    Bird bird = new Bird();
    bird.floatInWater(); // DOES NOT COMPILE
    System.out.println(bird.text); // DOES NOT COMPILE
  }
}
```

BirdWatcherFromAfar no está en el mismo paquete que Bird y no hereda de Bird. Esto significa que no se permite el acceso los variables o métodos protected de Bird.

Las subclases y clases del mismo paquete son las únicas que pueden acceder a elementos protected.

Considerar esta clase:

```
1: package pond.swan;
2: import pond.shore.Bird; // in different package than Bird
3: public class Swan extends Bird { // but subclass of bird
4:   public void swim() {
5:     floatInWater(); // package access to superclass
6:     System.out.println(text); // package access to superclass
7:   }
8:   public void helpOtherSwanSwim() {
9:     Swan other = new Swan();
10:    other.floatInWater(); // package access to superclass
11:    System.out.println(other.text); // package access to superclass
12:  }
13:  public void helpOtherBirdSwim() {
14:    Bird other = new Bird();
15:    other.floatInWater(); // DOES NOT COMPILE
16:    System.out.println(other.text); // DOES NOT COMPILE
17:  }
18:}
```

Esto es interesante. Swan no está en el mismo paquete que Bird, pero lo extiende -lo que implica que tiene acceso a los miembros protected de Bird ya que es una subclase. Y lo hace. Las líneas 5 y 6 se refieren a miembros protegidos a través de su herencia.

Las líneas 10 y 11 también utilizan con éxito a los elementos protected de Bird. Esto está permitido porque estas líneas se refieren a un objeto Swan. Swan hereda de Bird, así que está bien. El objeto Swan swan creado en la línea 9 puede acceder por lo tanto a métodos de Bird.

Las líneas 15 y 16 no compilan. Hay una diferencia clave con las líneas 10 y 11. Esta vez se utiliza una referencia de Bird. Se crea en la línea 14. Bird está en un paquete diferente, y este código no es heredado de Bird, por lo que no pueden usar elementos protected. ¿Qué ha pasado? Se acaba de decir repetidamente que Swan hereda de Bird. Y lo hace. Sin embargo, la variable de referencia no es un Swan. El código está en la clase Swan.

Desde un punto de vista diferente, las normas protected se aplican en dos escenarios:

- Un elemento se utiliza sin referirse a una variable. Este es el caso de las líneas 5 y 6. En este caso, se está aprovechando la herencia y el acceso protegido está permitido.
- Un miembro se utiliza a través de una variable. Este es el caso de las líneas 10,11,15 y 16. En este caso, las reglas para la clase de referencia de la variable son las que importan. Si se trata de una subclase, se permite el acceso protegido. Esto funciona para referencias a la misma clase o a una subclase.

Véase otro ejemplo similar:

```
package pond.goose;
import pond.shore.Bird;
public class Goose extends Bird {
  public void helpGooseSwim() {
    Goose other = new Goose();
    other.floatInWater();
    System.out.println(other.text);
  }
  public void helpOtherGooseSwim() {
    Bird other = new Goose();
    other.floatInWater(); // DOES NOT COMPILE
    System.out.println(other.text); // DOES NOT COMPILE
  }
}
```

El primer método es seguro. De hecho, es equivalente al ejemplo del Cisne. Goose extiende de Bird. Ya que se está en la subclase Goose y al referirse a una referencia de Goose, se puede acceder a elementos protected.

El segundo método hay un problema. Aunque el objeto resulta ser un Goose, se almacena en una referencia de Bird. No se permite referirse a elementos de la clase Bird ya que no se está en el mismo paquete y Bird no es una subclase de Bird.

¿Qué hay de este?

```
package pond.duck;
import pond.goose.Goose;
```

```
public class GooseWatcher {
  public void watch() {
    Goose goose = new Goose();
    goose.floatInWater(); // DOES NOT COMPILE
  }
}
```

Este código no compila porque no se está en la clase Goose. El método floatInWater() se declara en Bird. GooseWatcher no está en el mismo paquete que Bird, ni extiende Bird. Goose extiende de Bird. Eso sólo permite a Goose referirse a floatInWater() y no a los que llaman a Goose.

6.3.4. Acceso public

El acceso protected no es un concepto sencillo. Afortunadamente, el último tipo de modificador de acceso es fácil: public significa que cualquiera puede acceder a la variable o método public desde cualquier lugar.

```
package pond.duck;
public class DuckTeacher {
  public String name = "helpful"; //

  public access public void swim() { // public access
    System.out.println("swim");
  }
}
```

DuckTeacher permite el acceso a cualquier clase que quiera. Ahora se puede probar:

```
package pond.goose;
import pond.duck.DuckTeacher;
public class LostDuckling {
  public void swim() {
    DuckTeacher teacher = new DuckTeacher();
    teacher.swim(); // allowed
    System.out.println("Thanks" + teacher.name); // allowed
  }
}
```

LostDuckling es capaz de referirse a swim() y a name en DuckTeacher porque son públicos.

Para revisar los modificadores de acceso:

Accesible desde	Método o variable private	Método o variable default	Método o variable protected	Método o variable public
Su misma clase	Yes	Yes	Yes	Yes
Otra clase del mismo paquete	No	Yes	Yes	Yes
Una superclase en un paquete distinto	No	No	Yes	Yes
Una no-superclase en un paquete distinto	No	No	No	Yes

6.3.5. Diseñando métodos y variables estáticas

Los métodos estáticos no requieren una instancia de clase. Se comparten entre todas instancias de la clase. Se puede pensar en los elementos estáticos como algo que existe una sola vez independientemente de las veces que se instancie la clase.

¿Tiene cada clase su propia copia del código?

Cada clase tiene una copia de las variables de instancia. Sólo hay una copia del código para los métodos de instancia. Cada instancia de la clase puede llamarse tantas veces como se necesite. Sin embargo, cada llamada de un método de instancia (o cualquier método) obtiene espacio en la pila para parámetros de método y variables locales.

Lo mismo sucede con los métodos estáticos. Hay una copia del código. Los parámetros y variables locales van en la pila.

Sólo recordar que sólo los datos tienen su propia copia. No es necesario duplicar copias del código en sí.

Se ha visto un método estático desde el Capítulo 3. El método main() es un método estático. Eso significa que se le puede llamar por el nombre de la clase.

```
public class Koala {
  public static int count = 0; // static variable
  public static void main(String[] args) { // static method
    System.out.println(count);
  }
}
```

La JVM básicamente llama a Koala.main() para iniciar el programa. Se puede tener un KoalaTester que no hace nada más que llamar al método main().

```
public class KoalaTester {
  public static void main(String[] args) {
    Koala.main(new String[0]); // call static method
  }
}
```

Es una forma bastante complicada de imprimir 0, ¿no? Cuando se ejecuta KoalaTester, hace una llamada al método main() de Koala, que imprime el valor del conteo. El propósito de todos estos ejemplos es mostrar que main() pueda ser llamado como cualquier otro método estático.

Además de los métodos main(), los métodos estáticos tienen dos propósitos principales:

- Para métodos utilitarios o de ayuda que no requieren ningún estado de objeto. Como no hay necesidad de acceder a las variables de instancia, tener métodos estáticos elimina la necesidad de instanciar el objeto sólo para llamar al método.
- Para un estado que es compartido por todas las instancias de una clase, como un contador. Todas las instancias deben compartir el mismo estado. Los métodos que se limitan a utilizar ese estado también deben ser estáticos.

Se verán algunos ejemplos que cubren otros conceptos estáticos.

a. Llamando a variables o métodos estáticos

Normalmente, acceder a un miembro estático es fácil. Sólo se tiene que poner el nombre de clase antes del método o variable y listo. Por ejemplo:

```
System.out.println(Koala.count);
Koala.main(new String[0]);
```

Se puede utilizar también una instancia del objeto para llamar a un método estático. El compilador comprueba el tipo de la referencia y usa eso en lugar del objeto. Este código es perfectamente legal:

```
5: Koala k = new Koala();
6: System.out.println(k.count); // k is a Koala
7: k = null;
8: System.out.println(k.count); // k is still a Koala
```

Este código produce 0 dos veces. La línea 6 ve que k es un Koala y count es una variable estática, por lo que lee esa variable estática. La línea 8 hace lo mismo. A Java no le importa que k sea null.

Una vez más porque esto es realmente importante: ¿qué imprime el siguiente código?

```
Koala.count = 4;
Koala koala1 = new Koala();
Koala koala2 = new Koala();
koala1.count = 6;
koala2.count = 5;
System.out.println(Koala.count);
```

La respuesta es 5. Sólo hay una variable de conteo ya que es estática. Adquiere el valor 4, luego 6, y finalmente 5.

6.3.6. Instancia vs Estático

Un elemento estático no puede llamar a un elemento de instancia. Esto no debería ser una sorpresa, ya que los elementos estáticos no requieren de ninguna instancia de la clase.

Lo siguiente es un error común que los programadores novatos pueden cometer:

```
public class Static {
  private String name = "Static class";
  public static void first() { }
  public static void second() { }
  public void third() { System.out.println(name);}
  public static void main(String args[]) {
    first();
    second();
    third(); // DOES NOT COMPILE
  }
}
```

El compilador dará un error al hacer una referencia estática a un método no estático. Si se soluciona esto añadiendo static a third(), se creará un nuevo problema.

Todo lo que esto hace es mover el problema. Ahora, third() se refiere al name no estático. Añadir static a name solucionaría el problema. Otra solución habría sido llamar a third() como un método de instancia- por ejemplo, new Static().third();.

Tanto métodos estáticos como métodos de instancia pueden llamar a un método estático porque los métodos estáticos no requieren de un objeto para su uso. Sólo un método de instancia puede llamar otro método de instancia en la misma clase sin utilizar una variable de referencia, porque los métodos de instancia requieren un objeto. Se aplica una lógica similar para las variables estáticas y de instancia. Véase un resumen de lo anterior en la siguiente tabla:

Tipo	Llamada	¿Legal?	¿Cómo?
Método estático	Otro método o variable estático	Si	Usando el nombre de la clase
Método estático	Un método o variable de instancia	No	
Método de instancia	Un método o variable estático	Si	Usando el nombre de la clase o la variable referencia
Método de instancia	Otro método o variable de instancia	Si	Usando una referencia a una variable

Se probará un ejemplo más para tener más práctica en reconocer este escenario. ¿Se entiende por qué las siguientes líneas no se compilan?

```
1: public class Gorilla {
2:   public static int count;
3:   public static void addGorilla() { count++; }
4:   public void babyGorilla() { count++; }
5:   public void announceBabies() {
6:     addGorilla();
7:     babyGorilla();
8:   }
9:   public static void announceBabiesToEveryone() {
10:    addGorilla();
11:    babyGorilla(); // DOES NOT COMPILE
12:  }
13:  public int total;
14:  public static average = total / count; // DOES NOT COMPILE
15:}
```

Las líneas 3 y 4 están bien porque tanto los métodos estáticos como de instancia pueden referirse a una variable estática.

Las líneas 5-8 están bien porque un método de instancia puede llamar a un método estático.

La línea 11 no compila porque un método estático no puede llamar a un método de instancia.

Del mismo modo, la línea 14 no compila porque una variable estática está intentando usar una variable de instancia.

Un uso común para las variables estáticas es contar el número de instancias:

```
public class Counter {
  private static int count;
  public Counter() { count++; }
  public static void main(String[] args) {
    Counter c1 = new Counter();
    Counter c2 = new Counter();
    Counter c3 = new Counter();
    System.out.println(count); // 3
  }
}
```

Cada vez que se llama al constructor, se incrementa la cuenta por 1. Este ejemplo se basa en el hecho de que las variables estáticas (y de instancia) se inicializan automáticamente al valor predeterminado para ese tipo, que es 0 para int. Consultar el Capítulo 3 para revisar los valores predeterminados. También nótese que no se escribe Counter.count. Se podría haberlo hecho. No es necesario porque ya se está en esa clase para que el compilador pueda inferirlo.

6.3.7. Variables estáticas

Algunas variables estáticas están destinadas a cambiar a medida que se ejecuta el programa. Los contadores son un ejemplo común de esto. Se quiere que el conteo aumente con el tiempo. Al igual que con las variables de instancia, se puede inicializar una variable estática en la línea que se declara:

```
public class Initializers {
  private static int counter = 0; // initialization
}
```

Otras variables estáticas están pensadas para no cambiar nunca durante el programa. Este tipo de variable se conoce como constante. Se utiliza para las constantes el modificador **final,** para asegurar que la variable nunca cambie. Las constantes finales estáticas utilizan una convención de nomenclatura diferente a otras variables: usan letras mayúsculas con barra baja entre palabras. Por ejemplo:

```
public class Initializers {
  private static final int NUM_BUCKETS = 45;
  public static void main(String[] args) {
    NUM_BUCKETS = 5; // DOES NOT COMPILE
  }
}
```

El compilador se asegurará de que no se intente actualizar accidentalmente una variable final.

¿Las siguientes líneas compilan?

```
private static final ArrayList<String> values = new ArrayList<>();
public static void main(String[] args) {
  values.add("changed");
}
```

Sí, compila, values es una variable de referencia. Se permite llamar a métodos sobre variables de referencia. Todo lo que el compilador puede hacer es comprobar que no intentamos reasignar los valores finales para apuntar a un objeto diferente.

6.3.8. Inicialización estática

En el Capítulo 3, se cubrieron los inicializadores de instancias que parecían métodos sin nombre. Sólo código dentro de llaves. Los inicializadores estáticos parecen similares. Añaden la palabra clave static para especificar que deben ejecutarse cuando se utiliza la clase por primera vez. Por ejemplo:

```
private static final int NUM_SECONDS_PER_HOUR;
static {

  int numSecondsPerMinute = 60;
  int numMinutesPerHour = 60;
  NUM_SECONDS_PER_HOUR = numSecondsPerMinute * numMinutesPerHour;
}
```

El inicializador estático funciona cuando la clase se utiliza por primera vez. Las sentencias en él se ejecutan y asignan cualquier variable estática según sea necesario. Hay algo interesante en este ejemplo. Se acaba de decir que las variables finales no pueden ser reasignadas. La clave aquí es que el inicializador estático es la primera asignación. Y como ocurre por adelantado, está bien.

Se ve a continuación otro ejemplo:

```
14: private static int one;
15: private static final int two;
16: private static final int three = 3;
17: private static final int four; // DOES NOT COMPILE
18: static {
19:   one = 1;
20:   two = 2;
21:   three = 3; // DOES NOT COMPILE
22:   two = 4; // DOES NOT COMPILE
23: }
```

La línea 14 declara una variable estática que no es final. Puede ser asignada tantas veces como queramos.

La línea 15 declara una variable final sin inicializar. Esto significa que se puede inicializarla exactamente una vez en un bloque estático.

La línea 22 no compila porque es el segundo intento de inicializar una constante.

La línea 16 declara una variable final y la inicializa al mismo tiempo.

No es posible reasignar una constate, por lo que la línea 21 no compila.

La línea 17 declara una variable final que nunca se inicializa. El compilador da un error de compilación porque sabe que los bloques estáticos son el único lugar donde la variable podría ser inicializada.

Evitar los inicializadores estáticos y de instancia

El uso de inicializadores estáticos y de instancia puede hacer que el código sea mucho más difícil de leer. Todo lo que se puede hacer en un inicializador de instancia se puede hacer en un constructor. El enfoque constructor es más fácil de leer. Hay un caso común para usar un inicializador estático: cuando se necesita inicializar un campo estático y el código para hacerlo requiere más de una línea. Esto ocurre a menudo cuando se desea inicializar una colección como ArrayList. Cuando se necesita utilizar un inicializador estático, poner toda la inicialización estática en el mismo bloque. Así, el orden es obvio.

6.3.9. *Imports* estáticos

En el Capítulo 3, se vio que se podía importar una clase específica o todas las clases de un paquete:

```
import java.util.ArrayList;
import java.util.*;
```

Se podría usar esta técnica para importar:

```
import java.util.List;
import java.util.Arrays;
public class Imports {
  public static void main(String[] args) {
    List<String> list = Arrays.asList("one", "two");
  }
}
```

Los imports son convenientes porque no se necesita especificar de dónde viene cada clase cada vez que se usa. Hay otro tipo de importación llamada **static imports**. Las importaciones regulares son para clases de importación. Las importaciones estáticas son para importar elementos estáticos de clases. Al igual que las importaciones regulares, se puede usar un comodín o importar un elemento específico. La idea es que no se debería tener que especificar de dónde viene cada método estático o variable cada vez que se use.

El método anterior tiene una llamada de método estático: Arrays.asList. Reescribir el código para usar una importación estática produce lo siguiente:

```
import java.util.List;
import static java.util.Arrays.asList; // static import
```

```
public class StaticImports {
  public static void main(String[] args) {
    List<String> list = asList("one","two"); // no Arrays.
  }
}
```

En este ejemplo, se está importando específicamente el método asList. Esto significa que cada vez que se refiere a asList en la clase, llamará a Arrays.asList().

Un caso interesante es lo que pasaría si se creara un método asList en la clase StaticImports. Java le daría preferencia sobre el importado y se usaría el método que se codificara.

Este ejemplo muestra casi todo lo que se puede hacer mal con imports estáticos ¿Por qué?

```
1: import static java.util.Arrays; // DOES NOT COMPILE
2: import static java.util.Arrays.asList;
3: static import java.util.Arrays.*; // DOES NOT COMPILE
4: public class BadStaticImports {
5:   public static void main(String[] args) {
6:     Arrays.asList("one"); // DOES NOT COMPILE
7:   }}
```

La línea 1 intenta utilizar una importación estática para importar una clase. Recordar que las importaciones estáticas son sólo para importar variables y métodos estáticos. Las importaciones regulares son para importar una clase.

La línea 3 intenta ver si se está prestando atención al orden de las palabras clave. La sintaxis es import static y no viceversa.

En la línea 6 se usa el método asList imporstado en la línea 2. Sin embargo, no se importa la clase Arrays en ninguna parte. Esto hace que sea correcto escribir asList("one"); pero no Arrays.asList ("one");.

Sólo hay un escenario más con importaciones estáticas. En el Capítulo 3, se aprendió que importar dos clases con el mismo nombre da un error de compilador. Esto se aplica también a las importaciones estáticas. El compilador mostrará si se intenta hacer explícitamente una importación estática de dos métodos con el mismo nombre o dos variables estáticas con el mismo nombre. Por ejemplo:

```
import static statics.A.TYPE;
import static statics.B.TYPE; // DOES NOT COMPILE
```

Afortunadamente, cuando esto sucede, se puede hacer referencia a los elementos estáticos a través de su nombre de clase en el código en lugar de intentar usar una importación estática.

6.4. Pasando Datos a Través de Métodos

Java es un lenguaje "pass-by-value". Esto significa que se realiza una copia de la variable y el método recibe esa copia. Las asignaciones realizadas en el método no afectan a la llamada. Ejemplo:

```
2: public static void main(String[] args) {
3:   int num = 4;
4:   newNumber(5);
5:   System.out.println(num); // 4
6: }
7: public static void newNumber(int num) {
8:   num = 8;
9: }
```

En la línea 3, a num se le asigna el valor de 4.

En la línea 4, se llama a un método.

En la línea 8, el parámetro numérico del método se ajusta a 8. Aunque este parámetro tiene el mismo nombre que la variable de la línea 3, es una coincidencia. El nombre podría ser cualquier cosa. La variable de la línea 3 nunca cambia porque no se realizan asignaciones.

Véase un ejemplo ahora en vez de con una primitiva de datos, con una variable referencia ¿Cuál será el resultado del siguiente código?

```
public static void main(String[] args) {
  String name = "Webby";
  speak(name);
  System.out.println(name);
}
public static void speak(String name) {
  name = "Sparky";
}
```

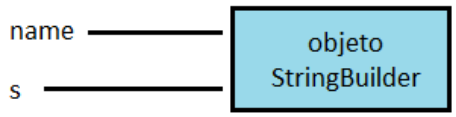
La respuesta correcta es Webby.

Véase otro ejemplo diferente:

```
public static void main(String[] args) {
  StringBuilder name = new StringBuilder();
  speak(name);
  System.out.println(name); // Webby
}
public static void speak(StringBuilder s) {
  s.append("Webby");
}
```

En este caso, la salida es Webby porque el método speak tiene en su cuerpo un método append de la clase StringBuilder. No reasigna el nombre a un objeto diferente, como en el ejemplo anterior.

En la figura siguiente, se puede ver cómo se sigue utilizando el valor pass-by-value. s es una copia del nombre de la variable. Ambos apuntan al mismo StringBuilder, lo que significa que los cambios realizados en el StringBuilder están disponibles para ambas referencias.



Pass-by-Value vs. Pass-by-Reference

Diferentes idiomas manejan los parámetros de diferentes maneras. Pass-by-value es utilizado por muchos idiomas, incluyendo Java. En este ejemplo, el método swap no modifica los valores originales. Sólo cambia a y b dentro del método.

```
public static void main(String[] args) {
  int original1 = 1;
  int original2 = 2;
  swap(original1,original2);
  System.out.println(original1); // 1
  System.out.println(original2); // 2
}
public static void swap(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

El otro enfoque es pass-by-reference. Se utiliza por defecto en algunos idiomas, como Perl. No vamos a mostrar el código Perl aquí porque se está estudiando Java. El siguiente ejemplo es en un lenguaje inventado que muestra el pass-by-reference:

```
original1 = 1;
original2 = 2;
swapByReference(original1, original2);
print(original1); // 2 (not in Java)
print(original2); // 1 (not in Java)
swapByReference(a, b) {
  temp = a;
  a = b;
  b = temp;
}
```

¿Se ve la diferencia? En el lenguaje inventado, la persona que llama se ve afectada por las asignaciones variables realizadas en el método.

Para repasar, Java utiliza pass-by-value para obtener datos en un método. Asignar una nueva referencia o primitiva de datos a un parámetro no cambia a la variable que esté fuera del método.

Véase con especial atención el siguiente código:

```
1: public class ReturningValues {
2:   public static void main(String[] args) {
3:     int number = 1; // 1
4:     String letters = "abc"; // abc
5:     number(number); // 1
6:     letters = letters(letters); // abcd
7:     System.out.println(number + letters); // 1abcd
8:   }
9:   public static int number(int number){
10:    number++;
11:    return number;
12:  }
13:  public static String letters(String letters) {
14:    letters += "d";
15:    return letters;
16:  }
17:}
```

Las líneas 3 y 4 son asignaciones directas.

La línea 5 llama a un método.

La línea 10 incrementa el parámetro del método a 2, pero deja la variable numérica en el método main() como 1.

Mientras que la línea 11 devuelve el valor, este es ignorado en el método main.

La llamada al método en la línea 6; no obstante, no ignora el resultado para que letters se convierta en "abcd".

6.5. Sobrecargando Métodos

La sobrecarga de métodos se produce cuando hay métodos diferentes con el mismo nombre, pero con diferentes parámetros.

Se ha estado llamando a métodos sobrecargados durante los capítulos anteriores. Los métodos System.out.println o append de StringBuilder proporcionan muchas versiones sobrecargadas para que se les pueda pasar casi cualquier cosa como parámetro. En ambos ejemplos, el único cambio fue el tipo de parámetro. La sobrecarga también permite diferentes números de parámetros.

Todo lo que no sea el nombre del método y sus parámetros puede variar para los métodos sobrecargados. Esto significa que puede haber diferentes modificadores de acceso, especificadores (como estáticos), tipos de retorno y listas de excepciones.

Todos estos son métodos sobrecargados válidos:

```
public void fly(int numMiles) { }
public void fly(short numFeet) { }
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) { }
public void fly(short numFeet, int numMiles) throws Exception { }
```

Como se puede ver, se puede sobrecargar cambiando cualquier cosa en la lista de parámetros. Se puede tener un tipo diferente, más tipos o los mismos tipos en un orden diferente. Obsérvese también que las modificaciones de acceso y la lista de excepciones son irrelevantes para la sobrecarga.

Ejemplo que no es una sobrecarga válida:

```
public void fly(int numMiles) { }
public int fly(int numMiles) { } // DOES NOT COMPILE
```

Este método no compila porque sólo difiere del original por el tipo de devolución. Las listas de parámetros son las mismas, por lo que son métodos duplicados en cuanto a Java se refiere. ¿Qué hay de estos dos? ¿Por qué el segundo no compila?

```
public void fly(int numMiles) { }
public static void fly(int numMiles) { } // DOES NOT COMPILE
```

De nuevo, la lista de parámetros es la misma. La única diferencia es que uno es un método de instancia y el otro es un método estático.

Llamar a métodos sobrecargados es fácil. Por ejemplo:

```
public void fly(int numMiles) {
  System.out.println("int");
}
public void fly(short numFeet) {
  System.out.println("short");
}
```

La llamada fly((short) 1); imprime short.

6.5.1. Sobrecargas y Varargs

¿A qué método se llama si se le pasa un int[]?

```
public void fly(int[] lengths) { }
public void fly(int... lengths) { } // DOES NOT COMPILE
```

Java trata a los varargs como si fueran un array. Esto significa que la firma del método es la misma para ambos métodos. Como no se permite sobrecargar los métodos con la misma lista de parámetros, este código no compila. Aunque el código no se vea igual, compila la misma lista de parámetros.

Se pueda llamar a cualquiera de los dos métodos anteriores pasando un array:

```
fly(new int[] { 1, 2, 3 });
```

Sin embargo, sólo se puede llamar a la versión de varargs con parámetros independientes:

```
fly(1, 2, 3);
```

6.5.2. Autoboxing

En el capítulo anterior, se vio cómo Java convertía una primitiva de datos int a un objeto Integer para añadirlo a un ArrayList a través del *autoboxing*. Esto funciona para el siguiente código también.

```
public void fly(Integer numMiles) { }
```

Esto significa que llamar a fly(3); llamará al método anterior como se esperaba. Sin embargo, ¿qué sucede si se tiene una versión primitiva y otra Integer?

```
public void fly(int numMiles) { }
public void fly(Integer numMiles) { }
```

Java coincidirá con la versión int numMiles. Java intenta utilizar la lista de parámetros más específica que puede encontrar. Cuando la versión primitiva de int no está presente, se hará *autoboxing*. Sin embargo, cuando se proporciona la versión primitiva de int, no hay razón para que Java haga el trabajo extra de *autoboxing*.

6.5.3. Tipos de referencias

Dada la regla de que Java escoge la versión más específica de un método para ejecutarlo, ¿Qué imprime por consola este código?

```
public class ReferenceTypes {
  public void fly(String s) {
    System.out.print("string");
```

```
  }
  public void fly(Object o) {
    System.out.print("object");
  }
  public static void main(String[] args) {
    ReferenceTypes r = new ReferenceTypes();
    r.fly("test");
    r.fly(56);
  }
}
```

La respuesta es "string object".

La primera llamada es un String y encuentra una coincidencia directa. No hay razón para usar la versión Object cuando hay una lista de parámetros String esperando a ser llamada.

La segunda llamada busca una lista de parámetros int. Cuando no lo encuentra, se *autoboxea* a Integer. Como todavía no encuentra ninguna coincidencia, va al Object.

6.5.4. Primitivas de datos

Las primitivas de datos funcionan de manera similar a las variables de referencia. Java intenta encontrar el método más específico de comparación sobrecargado. ¿Que pasará aquí?

```
public class Plane {
  public void fly(int i) {
    System.out.print("int ");
  }
  public void fly(long l) {
    System.out.print("long ");
  }
  public static void main(String[] args) {
    Plane p = new Plane();
    p.fly(123);
    p.fly(123L);
  }
}
```

La respuesta es int long.

La primera llamada pasa un int, con lo cual, hay una coincidencia exacta con el primer método fly.

La segunda llamada pasa un long y también hay una coincidencia exacta, esta vez con el segundo método fly.

Si se comenta el método sobrecargado con la lista de parámetros int (el primer método fly), la salida será long long. Java no tiene ningún problema en llamar a un primitivo más grande. Sin embargo, no lo hará a menos que no se encuentre una mejor coincidencia.

Java sólo puede aceptar tipos más amplios. Un int se puede pasar a un método que toma un parámetro long. Java no convertirá automáticamente. Si se quiere pasar un long a un método que toma un parámetro int, se tiene que añadir un **casting**.

6.5.5. Juntándolo todo

Java llama al método más específico que puede. Cuando algunos de los tipos interactúan, las reglas Java se centran en la compatibilidad. En Java 1.4 y antes, el *autoboxing* y varargs no existían. Aunque eso fue hace mucho tiempo, el antiguo código todavía necesita funcionar, lo que significa que el *autoboxing* y los varargs son lo último cuando Java examina los métodos sobrecargados.

Regla	Ejemplo de los que será elegido para glide(1,2)
Coincidencia exacta de tipo	pubic String glide(int i, int j){}
Tipo primitive largo	public String glide(long I, long j){}
Tipo autoboxeado	public String glide(Integer I, Integer j){}
Varargs	public String glide(int… nums){}

Ejemplos:

```
public class Glider2 {
  public static String glide(String s) {
    return "1";
  }
  public static String glide(String... s) {
    return "2";
  }
  public static String glide(Object o) {
    return "3";
  }
  public static String glide(String s, String t){
    return "4";
  }
  public static void main(String[] args) {
    System.out.print(glide("a"));
    System.out.print(glide("a", "b"));
```

```
    System.out.print(glide("a", "b", "c"));
  }
}
```

Imprime 142.

La primera llamada coincide con la cabecera que tiene como parámetro un solo String, porque es el más específico.

La segunda llamada coincide con la cabecera que tiene como parámetros dos String, ya que es también una coincidencia exacta.

No es hasta la tercera llamada que se utiliza la versión de varargs, ya que no hay mejores coincidencias.

Véase un ejemplo diferente:

```
public class TooManyConversions {
  public static void play(Long l) { }
  public static void play(Long... l) { }
  public static void main(String[] args) {
    play(4); // DOES NOT COMPILE
    play(4L); // calls the Long version
  }
}
```

Aquí hay un problema. Java puede convertir el int 4 a un long 4 o un Integer 4. Pero no puede manejar la conversión en dos pasos a un Integer y luego a un long.

Si se tuviera un public stativ void play(Object o) { }, coincidiría porque sólo una conversión sería necesaria: de int a Integer. Un Integer es un Object, como se verá en el Capítulo 7.

6.6. Creando Constructores

Como se aprendió en el Capítulo 3, un constructor es un método especial que coincide con el nombre de la clase y no tiene ningún tipo de devolución. Aquí hay un ejemplo:

```
public class Bunny {
  public Bunny() {
    System.out.println("constructor");
  }
}
```

El nombre del constructor, Bunny, coincide con el nombre de la clase, Bunny, y no hay ningún tipo de devolución, ni siquiera void. Eso lo hace un constructor.

¿Por qué estos dos no son constructores válidos para la clase Bunny?

```
public bunny() { } // DOES NOT COMPILE
public void Bunny() { }
```

El primero no coincide con el nombre de la clase porque Java distingue entre mayúsculas y minúsculas. Como no coincide, Java sabe que no puede ser un constructor y se supone que es un método regular. Sin embargo, le falta el tipo de devolución y no compila.

El segundo método es un método perfectamente bueno, pero no es constructor porque tiene un tipo de retorno.

Los constructores se utilizan al crear un nuevo objeto. Este proceso se llama instanciar, porque crea una nueva instancia de la clase. Un constructor es llamado cuando se escribe **new** seguido del nombre de la clase que se quiere instanciar. Por ejemplo:

```
new Bunny();
```

Cuando Java ve la palabra clave **new**, asigna memoria para el nuevo objeto.

Un constructor se utiliza normalmente para inicializar variables de instancia. La palabra clave **this** le dice a Java que desea referenciar una variable de instancia (o atributo) de la clase. La mayoría de las veces, esto es opcional. El problema es que a veces hay dos variables con el mismo nombre. En un constructor, uno es un parámetro y el otro una variable de instancia.

Si no se dice lo contrario, Java interpreta que la variable que se está utilizando es la que tiene un alcance más específico, que es el parámetro. Usando this.name le dice a Java que se quiere la variable de instancia.

Esta es una forma común de escribir a un constructor:

```
1: public class Bunny {
2:   private String color;
3:   public Bunny(String color) {
4:     this.color = color;
5:   } }
```

En la línea 4, se asigna el parámetro color al color de la variable de instancia. El lado derecho de la asignación se refiere al parámetro porque no se especifica nada especial. El lado izquierdo de la asignación usa **this** para decirle a Java que se quiere usar la variable de instancia.

Ahora se verán algunos ejemplos que no son comunes en Java:

```
1: public class Bunny {
2:   private String color;
3:   private int height;
4:   private int length;
5:   public Bunny(int length, int theHeight) {
6:     length = this.length; // backwards – no good!
```

```
7:     height = theHeight; // fine because a different name
8:     this.color = "white"; // fine, but redundant
9:   }
10:  public static void main(String[] args) {
11:    Bunny b = new Bunny(1, 2);
12:    System.out.println(b.length + " " + b.height + " " + b.color);
13:  }}
```

La línea 6 es incorrecta. El length de la variable de instancia comienza con un valor 0. Ese 0 se asigna al length del parámetro del método. La variable de instancia se mantiene en 0.

La línea 7 es más directa. El parámetro theHeight y el height de la variable de instancia tienen diferentes nombres. No hay ninguna colisión de nombres.

Por último, la línea 8 muestra que está permitido utilizarlo incluso cuando no hay duplicación de nombres de variables.

6.6.1. Constructor por defecto

Cada clase en Java tiene un constructor. Si no se incluye ningún constructor en la clase, Java creará uno sin ningún parámetro por defecto.

Este constructor creado en Java se denomina constructor predeterminado o constructor por defecto. A veces se le llama el constructor de los no-argumentos por defecto para mayor claridad. Aquí hay un ejemplo:

```
public class Rabbit {
  public static void main(String[] args) {
    Rabbit rabbit = new Rabbit(); // Calls default constructor
  }
}
```

En la clase Rabbit, Java no ve que ningún constructor haya sido creado y crea uno. Este constructor predeterminado es equivalente a escribir esto:

```
public Rabbit() {}
```

El constructor predeterminado tiene una lista de parámetros vacía y un cuerpo vacío. Sin embargo, puesto que no hace nada, Java puede proveerlo y ahorrarse algo de tiempo.

Un constructor predeterminado sólo se suministra si no hay constructores presentes. ¿Cuál de estas clases tiene un constructor por defecto?

```
class Rabbit1 {
}
class Rabbit2 {
```

```
  public Rabbit2() { }
}
class Rabbit3 {
  public Rabbit3(boolean b) { }
}
class Rabbit4 {
  private Rabbit4() { }
}
```

Solo Rabbit1 obtiene un constructor de no-argumento por defecto. No tiene un constructor creado, por lo que Java genera un constructor sin argumento predeterminado. Rabbit2 y Rabbit3 ya tienen constructores públicos. Rabbit4 tiene un constructor privado. Puesto que estas tres clases tienen un constructor definido, el constructor de no-argumento predeterminado no se inserta.

¿Cómo llamar a estos constructores?

```
1: public class RabbitsMultiply {
2:   public static void main(String[] args) {
3:     Rabbit1 r1 = new Rabbit1();
4:     Rabbit2 r2 = new Rabbit2();
5:     Rabbit3 r3 = new Rabbit3(true);
6:     Rabbit4 r4 = new Rabbit4(); // DOES NOT COMPILE
7:   }
8: }
```

La línea 3 llama al constructor de no-argumento predeterminado generado. Las líneas 4 y 5 llaman a los constructores provistos por el usuario. La línea 6 no compila. Rabbit4 hizo al constructor privado para que otras clases no pudieran llamarlo.

Tener un constructor privado en una clase le dice al compilador que no proporcione un constructor de argumentos predeterminado. También evita que otras clases instancien la clase. Esto es útil cuando una clase sólo tiene métodos estáticos o la clase desea controlar todas las llamadas para crear nuevas instancias de sí misma.

6.6.2. Sobrecargando constructores

Hasta ahora, sólo se ha visto un constructor por clase. Se puede tener varios constructores en la misma clase siempre y cuando tengan cabeceras diferentes. Cuando se sobrecargan métodos, el nombre del método y la lista de parámetros son necesarios para que coincidan. En el caso de los constructores, el nombre es siempre el mismo, ya que tiene que ser igual al nombre de la clase. Esto significa que los constructores deben tener diferentes parámetros para ser sobrecargados.

Este ejemplo muestra dos constructores:

```
public class Hamster {
  private String color;
```

```
  private int weight;
  public Hamster(int weight) { // first constructor
    this.weight = weight;
    color = "brown";
  }
  public Hamster(int weight, String color) { // second constructor
    this.weight = weight;
    this.color = color;
  }
}
```

Uno de los constructores toma un único parámetro int. El otro toma un int y un String. Estas listas de parámetros son diferentes, por lo que los constructores se sobrecargan con éxito.

Pero hay un problema aquí. Hay un poco de duplicación. Lo que realmente se quiere es que el primer constructor llame al segundo constructor con dos parámetros:

```
public Hamster(int weight) {
  Hamster(weight, "brown"); // DOES NOT COMPILE
}
```

Esto no funcionará. Los constructores sólo pueden ser llamados escribiendo new antes del nombre del constructor. No son como los métodos normales que se pueden llamar simplemente. ¿Qué pasa si se pone new antes del nombre del constructor?

```
public Hamster(int weight) {
  new Hamster(weight, "brown"); // Compiles but does not do what we want
}
```

Este intento compila. Pero no hace lo que se quiere. Cuando se llama al constructor con un parámetro, crea un objeto con el weight y el color predeterminados. Luego construye un objeto diferente con el weight y el color deseado e ignora el nuevo objeto. Se quiere que el weight y el color se coloquen en el objeto que se está tratando de instanciar en primer lugar, no que se instancien dos objetos.

Java proporciona una solución: el uso de **this**, la misma palabra clave que se usa para referirse a las variables de instancia. Cuando esto se usa como si fuera un método, Java llama a otro constructor en la misma instancia de la clase.

```
public Hamster(int weight) {
  this(weight, "brown");
}
```

Ahora Java llama al constructor que toma dos parámetros y se instancia un objeto con los atributos weight y color inicializados.

this() tiene una regla especial que se necesita conocer. Si se elige llamarlo, la llamada this() debe ser la primera sentencia no comentada en el constructor.

```
3: public Hamster(int weight) {
4:   System.out.println("in constructor");
5:   // ready to call this
6:   this(weight, "brown"); // DOES NOT COMPILE
7: }
```

Aunque una sentencia de impresión en la línea 4 no cambia ninguna variable, sigue siendo una sentencia Java y no está permitido insertarla antes de la llamada a this(). El comentario en la línea 5 está bien. Los comentarios no ejecutan sentencias Java y están permitidos en cualquier lugar.

Encadenamiento en los constructores

Los constructores sobrecargados a menudo se llaman entre sí. Una técnica común es hacer que cada constructor agregue un parámetro hasta llegar al constructor que hace todo el trabajo. Este enfoque se llama encadenamiento constructor. En este ejemplo, los tres constructores están encadenados.

```
public class Mouse {
  private int numTeeth;
  private int numWhiskers;
  private int weight;
  public Mouse(int weight) {
    this(weight, 16); // calls constructor with 2 parameters
  }
  public Mouse(int weight, int numTeeth) {
    this(weight, numTeeth, 6); // calls constructor with 3 parameters
  }
  public Mouse(int weight, int numTeeth, int numWhiskers) {
    this.weight = weight;
    this.numTeeth = numTeeth;
    this.numWhiskers = numWhiskers;
  }
  public void print() {
    System.out.println(weight + " " + numTeeth + " " + numWhiskers);
  }
  public static void main(String[] args) {
    Mouse mouse = new Mouse(15);
    mouse.print();
  }
}
```

Este código imprime 15 16 6. El método main() llama al constructor con un parámetro. Ese constructor agrega un segundo valor de codificación y llama al constructor con dos parámetros. Ese constructor agrega un valor de codificación más y llama al constructor con tres parámetros. El constructor de tres parámetros asigna las variables de instancia.

6.6.3. Campos final

Como se vio anteriormente en el capítulo, a las variables de instancia **final** se les debe asignar un valor exactamente una vez. Esto sucedía en la línea de la declaración o en una instancia inicializadora. Hay una ubicación más donde se puede se puede inicializar una constante: el constructor.

```
public class MouseHouse {
  private final int volume;
  private final String name = "The Mouse House";
  public MouseHouse(int length, int width, int height) {
    volume = length * width * height;
  }
}
```

El constructor es parte del proceso de inicialización, por lo que se permite asignar en él variables de instancia final. Para cuando se complete el constructor, todas las variables de instancia final deben estar inicializadas.

6.6.4. Orden de inicialización

El Capítulo 3 se cubrió el orden de la inicialización. Ahora que se ha aprendido acerca de los inicializadores estáticos, es el momento de revisar eso. Desafortunadamente, se tiene que memorizar esta lista:

1. Si hay una superclase, inicializarla primero (se cubrirá en el siguiente capítulo).
2. Declaraciones de variables estáticas e inicializadores estáticos en el orden en que aparecen en el fichero.
3. Declaraciones de variables de instancias e inicializadores de instancias en el orden en que aparecen en el fichero.
4. El constructor

Ejemplo:

```
1: public class InitializationOrderSimple {
2:   private String name = "Torchie";
3:   { System.out.println(name); }
4:   private static int COUNT = 0;
5:   static { System.out.println(COUNT); }
6:   static { COUNT += 10; System.out.println(COUNT); }
7:   public InitializationOrderSimple() {
8:     System.out.println("constructor");
9:   }
10:}

1: public class CallInitializationOrderSimple {
2:   public static void main(String[] args) {
```

```
3:     InitializationOrderSimple init = new InitializationOrderSimple();
4:   }
5: }
```

La salida es la siguiente:

```
0
10
Torchie
constructor
```

La regla 1 no se aplica porque no hay superclase.

La regla 2 dice que deben ejecutarse las declaraciones de variables estáticas y los inicializadores estáticos: en este caso, las líneas 5 y 6, que imprimen 0 y 10.

La Regla 3 dice de ejecutar las declaraciones de variable de instancia y los inicializadores de instancia en las líneas 2 y 3, que imprimen Torchie.

Finalmente, la regla 4 dice que se debe ejecutar el constructor en las líneas 7-9, las cuales imprimen constructor.

El siguiente ejemplo es menos sencillo. Las cuatro reglas se aplican sólo si un objeto es instanciado. Si se hace referencia a la clase sin una nueva llamada, sólo se aplicarán las reglas 1 y 2. Las otras dos reglas se refieren a instancias y constructores. Tienen que esperar hasta que haya código para instanciar el objeto. ¿Qué pasará aquí?

```
1: public class InitializationOrder {
2:   private String name = "Torchie";
3:   { System.out.println(name); }
4:   private static int COUNT = 0;
5:   static { System.out.println(COUNT); }
6:   { COUNT++; System.out.println(COUNT);}
7:   public InitializationOrder() {
8:     System.out.println("constructor");
9:   }
10:  public static void main(String[] args) {
11:    System.out.println("read to construct");
12:    new InitializationOrder();
13:  }
14:}
```

La salida sería la siguiente:

```
0
read to construct
Torchie
1
constructor
```

De nuevo, la regla 1 no se aplica porque no hay superclase.

La regla 2 dice que deben ejecutarse las declaraciones de las variables estáticas e inicializadores estáticos-líneas 4 y 5, en ese orden. La línea 5 imprime 0.

Ahora, el método main() puede ejecutarse. A continuación, se puede usar la regla 3 para ejecutar las variables de instancia y los inicializadores de instancia. Aquí están las líneas 2 y 3, que imprimen Torchie.

Finalmente, la regla 4 dice que se debe ejecutar el constructor,en este caso, las líneas 7-9, las cuales imprimen constructor.

Este ejemplo es el más difícil:

```
1: public class YetMoreInitializationOrder {
2:   static { add(2); }
3:   static void add(int num) { System.out.print(num + " "); }
4:   YetMoreInitializationOrder() { add(5);}
5:   static { add(4); }
6:   { add(6); }
7:   static { new YetMoreInitializationOrder(); }
8:   { add(8); }
9:   public static void main(String[] args) { }
10:}
```

La respuesta correcta es 2 4 6 8 5. ¿Por qué?

No hay superclase, así que se salta directamente a la regla 2.

Hay tres bloques estáticos: en las líneas 2, 5 y 7. Se ejecutan en ese orden. El bloque estático en la línea 2 llama al método add(), que imprime 2.

El bloque estático en la línea 5 llama al método add(), que imprime 4.

El último bloque estático, en la línea 7, llama a new para instanciar el objeto. Esto significa que se puede pasar a la regla 3 para ver las variables de instancia y los inicializadores de instancia. Hay dos de ellas: en las líneas 6 y 8. Ambos llaman al método add() e imprimen 6 y 8, respectivamente.

Finalmente, se pasa a la regla 4 y se llama al constructor, que llama al método add() una vez más e imprime 5.

Este ejemplo es difícil por algunas razones. Hay mucho seguimiento que hacer. Además, el código tiene muchos bloques y métodos de código de una sola línea, lo que lo hace más difícil de visualizar.

6.7. Encapsulando Datos

Anteriormente se han visto ejemplos de clases con atributos no privados, como la siguiente:

```
public class Swan {
  int numberEggs; // instance variable
}
```

Puesto que existe un acceso predeterminado (paquete privado), esto significa que cualquier clase del paquete puede modificar numberEggs. Ya no se tiene el control de lo que se establece en la clase. Una llamada podría incluso escribir esto:

```
mother.numberEggs = -1;
```

Esto es una mala idea, no es una buena práctica permitir que los atributos de una clase se puedan modificar directamente en ninguna parte del código salvo en la propia clase.

El término **encapsulación** es una práctica Java que consiste en programar las clases de tal forma que sus atributos únicamente puedan ser accesibles haciendo uso de unos métodos específicos para su lectura y modificación. Veamos la ya conocida clase Swan:

```
1: public class Swan {
2:   private int numberEggs; // private
3:   public int getNumberEggs() { // getter
4:     return numberEggs;
5:   }
6:   public void setNumberEggs(int numberEggs) { // setter
7:     if (numberEggs >= 0) // guard condition
8:     this.numberEggs = numberEggs;
9:   }
10:}
```

Nótese que numberEggs es ahora privado en la línea 2. Esto significa que sólo el código dentro de la clase puede leer o escribir el valor de numberEggs. Supongamos que, por razones de diseño, numberEggs no debe tener nunca un valor negativo (pues un cisne no puede tener un número de huevos negativo).

En las líneas 3-5 se añade un método para leer el valor del atributo numberEggs desde fuera de la clase Swan, que se llama método de acceso o **getter**.

También se añade un método en las líneas 6-9 para actualizar el valor del atributo numberEggs , que se llama método modificador o **setter**. El setter tiene una sentencia if en este ejemplo para evitar que la variable de instancia se ajuste a un valor no válido. Esta condición protege la variable de instancia.

En la línea 8, se utilizará la palabra clave **this** que se vio en la sección de Constructores para diferenciar entre el parámetro de método numberEggs y la variable de instancia numberEggs.

La encapsulación consiste pues en recordar que las variables de instancia o atributos de una clase son siempre privados y que cada atributo tiene ligado a sí los métodos getters/setters pertinentes, que son públicos.

Java define una convención de nomenclatura que se usa en JavaBeans (que se verán con mayor profundidad al finalizar el curso de Java Básico). JavaBeans llama a una variable de instancia propiedad. Lo único que se necesita saber acerca de JavaBeans por el momento es lo que aparece en la siguiente tabla:

Regla	Ejemplo
Propiedades privadas	private int numEggs;

Los métodos getter empiezan con is si la propiedad es un boolean	public boolean isHappy(){
	return happy;
	}

Los métodos getter empiezan con get si la propiedad no es un boolean	public int getNumEggs(){
	return numEggs;
	}

Los métodos setter empiezan con set	public void setHappy(boolean happy){
	this.happy = happy;
	}

Los nombres de los métodos tienen que tener un prefijo set/get/is, seguido de la primera letra de la propiedad en mayúscula, seguido del resto del nombre de la propiedad	public void setNumEggs(int num){
	numEggs = num;
	}

Es hora de practicar. Véase si se puede determinar qué líneas siguen las convenciones de denominación de JavaBeans:

```
12: private boolean playing;
13: private String name;
14: public boolean getPlaying() { return playing; }
15: public boolean isPlaying() { return playing; }
16: public String name() { return name; }
17: public void updateName(String n) { name = n; }
18: public void setname(String n) { name = n; }
```

Las líneas 12 y 13 son buenas. Son variables de instancia privada.

La línea 14 no sigue las convenciones de nombres de JavaBeans. Puesto que playing es un boolean, el getter debe comenzar con is.

La línea 15 es la correcta para playing.

La línea 16 no sigue las convenciones de nombres de JavaBeans porque debería llamarse getName.

Las líneas 17 y 18 no siguen las convenciones de nombres de JavaBeans porque deberían llamarse setName. Recordar que Java distingue entre mayúsculas y minúsculas, por lo que el nombre de setname no es adecuado para cumplir con la convención de nombres.

6.7.1. Creando clases inmutables

Encapsular datos es necesario, ya que evita que las personas que hagan uso de la clase realicen cambios incontrolados en ella. Otra técnica común es hacer que las clases sean inmutables, por lo que no se pueden cambiar en absoluto.

En las clases inmutables los atributos siempre serán iguales. Esto ayuda a que los programas sean más fáciles de mantener. También ayuda con el rendimiento limitando el número de copias, como se vio con String en el Capítulo 3, "Core Java APIs".

Un paso para hacer una clase inmutable es omitir los métodos setters. Si se quiere que la persona que instancia la clase pueda especificar el valor inicial de sus atributos, pero no se quiere que estos cambien después de crear el objeto, se hará uso para ello de los constructores de la clase.

```
public class ImmutableSwan {
  private int numberEggs;
  public ImmutableSwan(int numberEggs) {
    this.numberEggs = numberEggs;
  }
  public int getNumberEggs() {
    return numberEggs;
  }

}
```

En este ejemplo, no se tiene un setter. Se tiene un constructor que permite establecer un valor inicial para el atributo numberEggs. Las clases inmutables pueden tener valores iniciales para sus atributos, pero estos no pueden cambiar después de la instanciación.

6.7.2. Tipos de retorno en la clases inmutables

Cuando se escriba una clase inmutable, hay que tener cuidado con los tipos de retorno.

Superficialmente, esta clase parece inmutable ya que no hay método setter:

```
public class NotImmutable {
  private StringBuilder builder;
  public NotImmutable(StringBuilder b) {
    builder = b;
  }
  public StringBuilder getBuilder() {
    return builder;
  }
}
```

El problema es que en realidad no lo es. Se considera este fragmento de código:

```
StringBuilder sb = new StringBuilder("initial");
NotImmutable problem = new NotImmutable(sb);
sb.append("added");
StringBuilder gotBuilder = problem.getBuilder();
gotBuilder.append("more");
System.out.println(problem.getBuilder());
```

Esto imprime "initial added more" claramente no es lo que se pretendía. El problema es que se está pasando el mismo StringBuilder por todas partes. La persona que llama tiene una referencia desde que se pasó al constructor. Cualquiera que llame al getter también consigue una referencia. Una solución es hacer una copia del objeto mutable.

```
public Mutable(StringBuilder b) {
  builder = new StringBuilder(b);
}
public StringBuilder getBuilder() {
  return new

  StringBuilder(builder);
}
```

Ahora la persona que llama puede hacer cambios en el objeto sb inicial. Mutable ya no se preocupa por ese objeto después de que el constructor se ejecute. Lo mismo ocurre con el getter: las personas que llaman pueden cambiar su StringBuilder sin afectar a Mutable.

Otro enfoque para el getter es devolver un objeto inmutable:

```
public String getValue() {
  return builder.toString();
}
```

No hay ninguna regla que diga que se tenga que devolver el mismo tipo que se está almacenando. El String es seguro para devolver porque es inmutable siempre.

Revisando lo visto, la encapsulación tiene como finalidad evitar que las personas que hacen uso de una clase cambien sus variables de instancia directamente. Inmutabilidad se refiere a evitar que las personas que hacen uso de una clase modifiquen las variables de instancia.

6.8. Escribiendo Lambdas Simples

Java es un lenguaje orientado a objetos. En Java 8, el lenguaje agregó la capacidad de escribir código usando ciertos elementos característicos de la programación funcional, la cual se basa en especificar lo que se desea hacer en lugar de ocuparse del estado de los objetos.

La programación funcional utiliza expresiones lambda para escribir código. Se puede pensar en una expresión lambda como un método anónimo. Tiene parámetros y un cuerpo igual que los métodos completos, pero no tiene un nombre como un método real. Las expresiones lambda a menudo se conocen como lambdas para abreviar.

En otras palabras, una expresión lambda es como un método sin nombre que se puede pasar como si fuera una variable.

En esta sección, se cubrirá un ejemplo de por qué las lambdas son útiles, la sintaxis de las lambdas y cómo usar predicados.

6.8.1. Ejemplo Lambda

El objetivo es imprimir todos los animales en una lista según algunos criterios. Se mostrará cómo hacer esto sin lambdas para ilustrar cómo las lambdas son útiles. Clase Animal:

```
public class Animal {
  private String species;
  private boolean canHop;
  private boolean canSwim;
  public Animal(String speciesName, boolean hopper, boolean swimmer) {
    species = speciesName;
    canHop = hopper;
    canSwim = swimmer;
  }
  public boolean canHop() { return canHop; }
  public boolean canSwim() { return canSwim; }
  public String toString() { return species; }
}
```

La clase Animal tiene tres variables de instancia, que se inicializan en el constructor. Tiene dos métodos que permiten saber si el animal puede saltar (hop) o nadar (swim). También tiene un método toString() para que se pueda identificar fácilmente al Animal en los programas.

Se plantea escribir un método que dada una lista de animales busque los animales que cumplan una condición (nadar, saltar, etc) y los imprima por pantalla, para ello se prepara la siguiente clase:

```
1: public class TraditionalSearch {
2:   public static void main(String[] args){
3:     List<Animal> animals = new ArrayList<Animal>(); // list of animals
4:     animals.add(new Animal("fish", false, true));
5:     animals.add(new Animal("kangaroo", true, false));
6:     animals.add(new Animal("rabbit", true, false));
7:     animals.add(new Animal("turtle", false, true));
8:
9:     listCanHop(animals);
10:   }
11:
12:  public static void listCanHop(List<Animal> animals) {
13:    for (Animal subject: animals) {
14:      if (subject.canHop())
15:              System.out.print(subject.toString() + " ");
16:    }
17:  }
```

La solución que se propone en el método listCanHop() es la realización de un bucle forEach, en este "subject" toma como valor un animal de la lista para cada iteración y comprueba si puede saltar o no, en el caso de que pueda saltar se imprime por pantalla el nombre del animal y un espacio, componiendo con cada iteración una lista de los animales que pueden saltar.

¿Qué pasa ahora si se quiere imprimir los Animales que nadan? Se necesita escribir otro método listCanSwim(), volver a crear el bucle y cambiar la comprobación. Aunque es relativamente sencillo, resulta pesado tener que repetir un método constantemente cuando existe otra alternativa, el uso de una expresión lambda como la siguiente:

```
a -> a.canHop();
```

6.8.2. Sintaxis Lambda

Una de las expresiones lambda más simples que se puede escribir es:

```
a -> a.canHop();
```

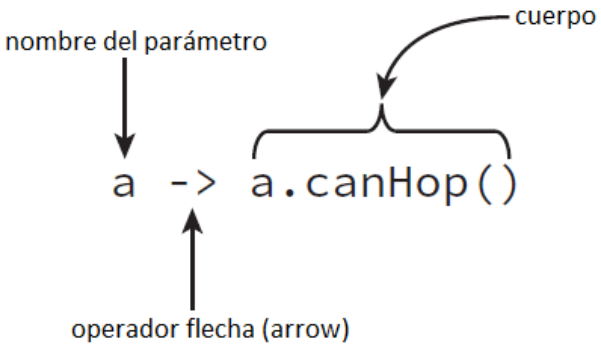
Esto significa que Java debería llamar a un método con un parámetro Animal ("a") que devuelve un valor booleano que es el resultado de a.canHop(). Se sabe todo esto porque se escribió el código. ¿Pero cómo lo sabe Java?

La sintaxis de las lambdas es delicada porque muchas partes son opcionales. Estas dos líneas hacen exactamente lo mismo:

```
a -> a.canHop()
(Animal a) -> { return a.canHop(); }
```

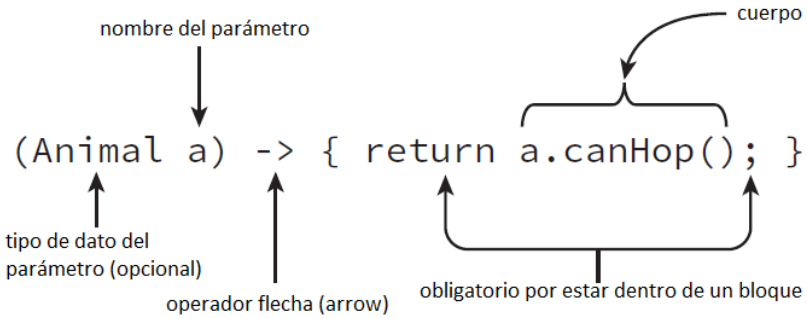
El primer ejemplo, mostrado en la figura siguiente, tiene tres partes:

- Especifica un único parámetro con el nombre a.
- El operador de flecha para separar el parámetro y el cuerpo.
- Un cuerpo que llama a un solo método y devuelve el resultado de ese método.



El segundo ejemplo también tiene tres partes; es más verboso:

- Especifica un único parámetro con el nombre a e indica el tipo Animal.
- El operador de flecha para separar el parámetro y el cuerpo.
- Un cuerpo que tiene una o más líneas de código, incluyendo un punto y coma y una declaración de devolución.



Los paréntesis sólo pueden omitirse si existe un único parámetro.

Se pueden también omitir las llaves cuando sólo se tiene una sentencia en el cuerpo, al igual que con los bucles, por ejemplo. Es necesario tener en cuenta que, cuando se omiten las llaves del cuerpo, no se puede escribir return o el punto y coma final de la sentencia. Este atajo especial no funciona cuando se tienen dos o más sentencias.

Véase un ejemplo de expresiones lambda a continuación:

```
3: print(() -> true); // 0 parameters
4: print(a -> a.startsWith("test")); // 1 parameter
5: print((String a) -> a.startsWith("test")); // 1 parameter
6: print((a, b) -> a.startsWith("test")); // 2 parameters
7: print((String a, String b) -> a.startsWith("test")); // 2 parameters
```

Todos estos ejemplos tienen paréntesis alrededor de la lista de parámetros excepto el que toma sólo un parámetro y no especifica el tipo.

La línea 3 toma 0 parámetros y siempre devuelve la verdad booleana.

La línea 4 toma un parámetro y llama a un método, devolviendo el resultado.

La línea 5 hace lo mismo, excepto que define explícitamente el tipo de variable.

Las líneas 6 y 7 toman dos parámetros e ignoran uno de ellos: no hay una regla que diga que se deben usar todos los parámetros definidos.

Véanse a continuación una serie de ejemplos de sintaxis inválida para expresiones lambda:

```
print(a, b -> a.startsWith("test")); // DOES NOT COMPILE
print(a -> { a.startsWith("test"); }); // DOES NOT COMPILE
print(a -> { return a.startsWith("test") }); // DOES NOT COMPILE
```

La primera línea necesita paréntesis alrededor de la lista de parámetros. Recordar que los paréntesis sólo son opcionales cuando hay un parámetro.

En la segunda línea falta la palabra clave return.

A la última línea le falta el punto y coma.

¿A qué variables puede acceder una expresión lambda?

Las lambdas pueden acceder a variables. Aquí hay un ejemplo:

```
boolean wantWhetherCanHop = true;
print(animals, a -> a.canHop() == wantWhetherCanHop);
```

No pueden acceder a todas las variables. Sí a las variables estáticas y de instancia. También se puede acceder a los parámetros de método y variables locales si no se les asignan nuevos valores.

También se debe tener lo siguiente en cuenta a la hora de trabajar con expresiones lambda, y es que las variables locales no pueden tener el mismo nombre que los parámetros:

```
(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE
```

Por el contrario, la siguiente línea está bien porque usa un nombre de variable diferente:

```
(a, b) -> { int c = 0; return 5;}
```

6.8.3. Predicados

Las Lambdas trabajan con interfaces que tienen un solo método.

Como se verá en el siguiente capítulo las interfaces son clases con una lista de métodos sin implementar. Como las Lambdas utilizan estas interfaces con un solo método, en el caso que nos atañe necesitaríamos crear muchas de estas (una por método: canHop(), canSwim(), etc).

Afortunadamente, Java reconoce que este es un problema común y proporciona una interfaz de este tipo. Se encuentra en el paquete java.util.function y es la siguiente:

```
public interface Predicate<T> {
  boolean test(T t);
}
```

Esta interfaz permite a lambda utilizar predicados de tipo genérico <T>. Estos predicados son expresiones lambda que permitirán realizar una función, serie de funciones sobre un elemento sin necesidad de llamar a otro método.

Esto significa que ya no se necesita una interfaz propia para el método test y se puede poner todo lo relacionado con la búsqueda que se quería realizar con la clase Animal en un solo método print():

```
1: import java.util.*;
2: import java.util.function.*;
3: public class PredicateSearch {
4:   public static void main(String[] args) {
5:     List<Animal> animals = new ArrayList<Animal>();
6:     animals.add(new Animal("fish", false, true));
7:
8:     print(animals, a -> a.canHop());
9:   }
10:  private static void print(List<Animal> animals, Predicate<Animal> checker) {
11:    for (Animal animal : animals) {
```

```
12:      if (checker.test(animal))
13:      System.out.print(animal + " ");
14:    }
15:    System.out.println();
16:  }
17:}
```

Este nuevo método, además de la lista, espera tener un Predicate que hace uso de un tipo Animal. Con esto el método print() realizara lo mismo que el anterior método listCanHop(), pero dándonos la posibilidad de cambiar el criterio de búsqueda sin alterar el método, simplemente cambiando el parámetro de entrada Predicate (expresión lambda):

```
8:    print(animals, a -> a.canHop());
9:    print(animals, a -> a.canSwim());
…
```

Java 8 incluso integró la interfaz Predicate en algunas clases existentes. ArrayList tiene un método removeIf() que toma un predicado. Supóngase una lista de nombres para conejitos. Se decide que se quieren eliminar todos los nombres de conejitos que no empiezan con la letra h porque realmente se quiere que se escoja un nombre con una "H". Se podría resolver este problema escribiendo un bucle. O podría ser resuelto en una línea:

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7: System.out.println(bunnies); // [long ear, floppy, hoppy]
8: bunnies.removeIf(s -> s.charAt(0) != 'h');
9: System.out.println(bunnies); // [hoppy]
```

La línea 8 se encarga de todo. Define un predicado que toma un String y devuelve un boolean. El método removeIf() recorrerá la lista eliminando aquellos elementos que casen con el predicado. Se recuerda que el método en la interfaz llamado test() toma cualquier tipo de parámetro de referencia y devuelve un booleano.

6.9. Resumen

Métodos

- Los métodos Java comienzan con un modificador de acceso público (public), privado (private), protegido (protected) o en blanco (acceso predeterminado). Esto es seguido por un especificador opcional como static, final o abstract. Luego viene el tipo de retorno, que es void o un tipo Java. A continuación, se muestra el nombre del método, utilizando reglas estándar de identificación Java. Cero o más parámetros van entre paréntesis en la lista de parámetros. A continuación, vienen las clases de

excepción opcionales. Finalmente, cero o más enunciados van en llaves para formar el cuerpo del método.

- Usar la palabra clave private significa que el código sólo está disponible dentro de la misma clase. El acceso predeterminado (default - package private) significa que el código sólo está disponible dentro del mismo package. Usar la palabra clave protected significa que el código está disponible en el mismo package o subclases. El uso de la palabra clave public significa que el código está disponible desde cualquier lugar.

- Los métodos estáticos y las variables estáticas son compartidas por la clase. Cuando se hace referencia a ellos desde fuera de la clase, se les llama utilizando el nombre de la clase (por ejemplo, StaticClass.method()). Los métodos y variables de instancia pueden llamar a métodos o variables estáticas, pero los métodos o variables estáticas no pueden llamar a métodos o variables de instancia.

- Los imports estáticos se utilizan para importar variables o métodos estáticos.

- Java utiliza pass-by-value, lo que significa que las llamadas a métodos crean una copia de los parámetros.

- Los métodos sobrecargados son métodos con el mismo nombre, pero con una lista de parámetros diferente. Java llama al método más específico que puede encontrar. Se prefieren coincidencias exactas, seguidas de primitivas de datos de mayor tamaño. Después viene el autoboxing y finalmente varargs.

Constructores

- Los constructores se utilizan para instanciar nuevos objetos.

- El constructor de no-argumento predeterminado se llama cuando no hay ningún constructor creado.

- Están permitidos múltiples constructores y pueden llamarse entre ellos escribiendo this().

- Si this() está presente, debe ser la primera declaración en el constructor.

- Los constructores pueden referirse a las variables de instancia escribiendo this antes de un nombre de variable para indicar que quieren la variable de instancia y no el parámetro de método con ese nombre.

- El orden de inicialización es la superclase (que se cubrirá en el Capítulo 7), las variables estáticas e inicializadores estáticos en el orden en que aparecen, las variables de instancia e inicializadores de instancia en el orden en que aparecen, y finalmente, el constructor.

Encapsulación

- La encapsulación tiene como finalidad evitar que las personas que hacen uso de una clase cambien las variables de instancia directamente. Esto se evita haciendo que las variables de instancia sean privadas y las variables getters/setters/ públicas.

- Inmutabilidad hace referencia a impedir que las personas que hacen uso de una clase modifiquen las variables de instancia. Esto utiliza varias técnicas, incluyendo el borrado de los setters.

- JavaBeans utiliza métodos que comienzan con is and get para tipos de propiedades boolean y no boolean, respectivamente. Los métodos que comienzan con el set se utilizan para los setters.

Expresiones lambda

- Las expresiones Lambda, o lambdas, permiten pasar bloques de código. La sintaxis completa es (String a, String b) -> { return a. igual a. (b); }. Los tipos de parámetros pueden omitirse. Cuando sólo se especifica un parámetro sin un tipo, los paréntesis también pueden omitirse. Las llaves y la declaración de devolución pueden omitirse para una sola declaración, quedando de forma corta (a -> a. igual a (b)).

- Las lambdas se pasan a un método que espera tener una interfaz con un método.

- El predicado es una interfaz común. Tiene un método llamado test que devuelve un boolean y toma cualquier tipo. El método removeIf() en ArrayList toma un predicado.

7. Diseño de clases

7.1. Introducción a la herencia de clases

Cuando una nueva clase de Java se crea, se puede definir que esta herede de otra ya existente. La herencia es el proceso por el cual una nueva subclase hijo, automáticamente incluye cualquier primitiva definida como protected o private, objetos o métodos definidos en la clase padre.

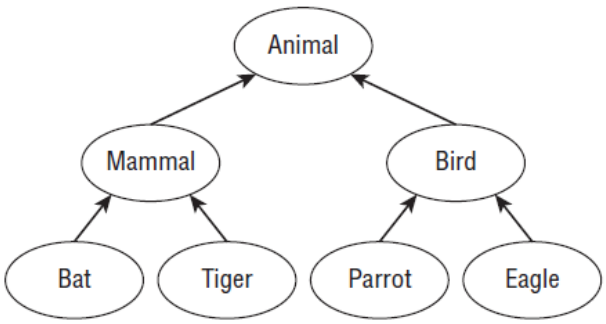
Con el propósito de ilustrar un ejemplo, se refiere a cualquier clase que hereda de otra clase como clase hijo o descendiente de esa clase. De manera similar, se designa como clase padre a aquella de la cual la clase hijo hereda, o clase predecesora. Si el hijo X hereda de la clase Y, que a su vez hereda de la clase Z, la clase hijo X se considerará un hijo indirecto o descendiente de la clase Z.

Java permite la herencia única, por la cual una clase puede solo heredar de un pariente directo. También permite herencia a distintos niveles, con lo cual una clase puede extender a otra clase que, a su vez, extiende a otra distinta. Se puede extender una clase un número indeterminado de veces, permitiendo así que cada descendiente gane acceso a los componentes de su predecesor.

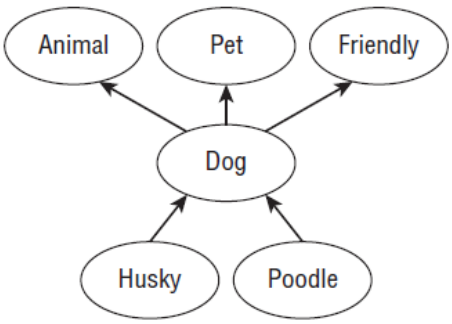
Para realmente entender la herencia única, puede ser de ayuda contrastar este concepto con la herencia múltiple, por la cual, una clase puede tener múltiples predecesores directos. Por cómo está estructurado y diseñado, Java no permite esta multiplicidad hereditaria mediante su sintaxis, puesto que hay estudios que corroboran que la herencia múltiple puede complicar el código volviéndolo difícil de mantener y gestionar. Java permite, sin embargo, una excepción a esta regla: las clases pueden implementar diversas interfaces a la vez, como se podrá ver más tarde en este capítulo.

En la figura de abajo se ilustran varios tipos de modelos de herencia. Los ítems de la izquierda son considerados como herencia única puesto que cada clase hijo tiene exactamente una clase padre. Como se puede observar, esta norma de diseño no impide que un predecesor tenga varios descendientes directos.

La parte derecha muestra ítems que tienen herencia múltiple. Por ejemplo, un objeto de clase perro tiene designadas varias clases padre. Una de las razones principales por las que la herencia múltiple es complicada es la determinación de los valores que el hijo hereda de cada predecesor. Por ejemplo, si hay un objeto o método definido en todos los predecesores, ¿cuál va a heredar la clase hijo? No hay un orden natural para los predecesores en este caso, por lo que Java evita este tipo de problemas impidiendo la herencia múltiple.



Herencia simple

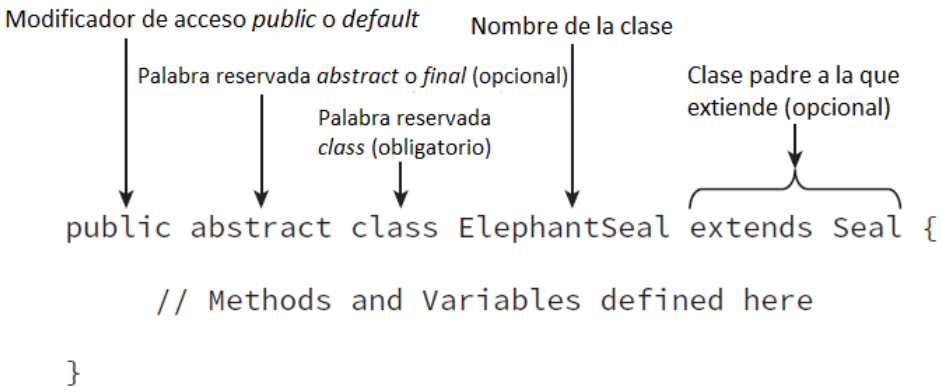


Herencia múltiple

En Java es posible evitar que una clase sea extendida marcándola con el modificador final. Si se trata de definir una clase que herede a esta clase marcada, el compilador arrojará un error y no compilará. Si no se especifica de otra manera, durante este capítulo se va a asumir que las clases con las que se va a trabajar no están marcadas como final.

7.1.1. Extendiendo una clase

En Java se puede extender una clase añadiendo el nombre de la clase padre en la definición, utilizando la palabra clase extends. La sintaxis para definir y extender una clase se muestra en la figura.



En este capítulo se definirá adecuadamente qué significa que una clase sea abstract y final, así como los modificadores de acceso de clases.

Dado que Java solo permite una clase public por fichero, se pueden crear dos de ellos, Animal.java y Leon.java, donde Leon extiende a la clase Animal. Asumiendo que están en el mismo paquete, no hará falta una sentencia import en la clase Leon.java para acceder a la clase Animal.

Este es el contenido de Animal.java:

```
public class Animal{
  private int edad;
    public int getEdad(){
      return edad;
    }
  public void setEdad(int Edad){
    this.edad = edad;
  }
}
```

Y este es el contenido de la clase Leon.java:

```
public class Leon extends Animal{
  private void rugido(){
```

```
    System.out.println("El leon, de  " + getEdad() + " años de edad dice: Roar!" );
  }
}
```

Nótese el uso de extends como palabra clave en Leon.java para indicar que la clase Leon extiende a la clase Animal. En este ejemplo, se puede ver que getEdad() y setEdad() son accesibles desde la clase Leon, puesto que están marcados como public en la clase padre. La primitiva edad está marcada como private y por tanto no es accesible para la subclase Leon. Por esto, el siguiente código no compilaría:

```
private void rugido(){
  System.out.println("El leon, de  " + edad + " años de edad dice: Roar!" );//NO COMPILA
}
```

Sin embargo, aunque este parámetro no es accesible desde la clase hijo se tiene una instancia del objeto Leon, existe un valor de edad dentro de la relación. Este valor simplemente no puede ser referenciado directamente por la clase hijo. De esta manera, el objeto Leon es "mayor" que el objeto Animal en el sentido de que incluye todas las propiedades del objeto Animal (aunque no todas sean accesibles directamente) combinadas con sus propios atributos como objeto Leon.

7.1.2. Aplicación de los modificadores de acceso para clases

Como se presentó en el capítulo 6, se pueden aplicar modificadores de acceso (public, private, protected, default) tanto para métodos como para variables. No será sorprendente pues que dichos modificadores se puedan aplicar también a la definición de clases. De hecho, se ha estado añadiendo el modificador de acceso public a prácticamente todas las clases que se han presentado hasta ahora.

El modificador public aplicado a una clase indica que puede ser referenciada y usada en cualquier otra clase. El paquete modificador privado por defecto, que se aplica cuando no hay ningún modificador de acceso aplicado por el/la programador/a, indica que la clase solo puede ser accedida por una subclase dentro del mismo paquete.

Como sabe el lector, un fichero Java puede tener muchas clases, pero como mucho, puede tener una clase pública. De hecho, puede no tener ninguna clase pública. Una característica de utilizar el modificador por defecto es que se pueden definir tantas clases como se deseen en un mismo fichero de Java. Por ejemplo, la siguiente definición podría aparecer en un único fichero de Java llamado marmota.java, puesto que solo contiene una clase public:

```
class Rodent{}
public class Marmota extends Rodent {}
```

Si se tuviese que actualizar la clase Rodent con el identificador public el fichero no compilaría a no ser que esta clase se trasladara a su propio fichero Rodent.java.

Las reglas para aplicar los modificadores de acceso de las clases son idénticos para las interfaces. Solo puede haber una clase o interfaz public en un documento Java. Como en las clases, las interfaces también se pueden declarar con los modificadores default o public. Más tarde en este capítulo, se presentarán con mayor detalle.

7.1.3. Creando objetos Java

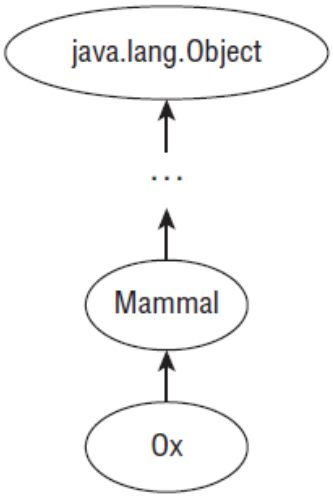
En diversas ocasiones se ha lanzado la palabra object.En Java, todas las clases heredan de una única clase, *java.lang.Object*. Es más, *java.lang.Object* es la única clase que no tiene clases predecesoras.

El lector se debe estar preguntando, "Ninguna de las clases que he escrito extienden a *java.lang.Object*, ¿cómo es posible que todas las clases hereden de esta entonces?" La respuesta es que el compilador ha estado añadiendo automáticamente este código a cualquier clase que se escriba y que no extienda a ninguna otra de manera específica. Por ejemplo, considérese las siguientes dos definiciones equivalentes de una clase:

```
public class Zoo{
}
public class Zoo extends java.lang.Object {
}
```

La clave es que cuando Java observa que el/la desarrollador/a define una clase que no extiende a ninguna otra, inmediatamente añade la sintaxis extends java.lang.Object a la definición de la clase.

Si se define una nueva clase que extiende a una clase existente, Java no añade dicha sintaxis, aunque esta nueva clase sigue heredando de java.lang.Object. Dado que todas las clases heredan de ella, extendiendo una clase ya existente implica que la clase hijo también hereda de java.lang.Object por construcción. Esto significa que si se mira a la estructura de herencias de cualquier clase, siempre se va a llegar a java.lang.Object en la cima del árbol, como se muestra en la figura.



Todos los objetos heredan de java.lang.Object

7.1.4. Definiendo Constructores

Como el lector puede recordar del Capítulo 6, toda clase tiene al menos un constructor. En el caso de que no haya sido declarado, el compilador automáticamente insertará un constructor sin argumentos por defecto. En caso de extender a una clase, las cosas se ponen un poco más interesantes.

En Java, la primera sentencia de un constructor es o bien una llamada a otro constructor dentro de la clase, utilizando this(), o una llamada al constructor del predecesor directo, utilizando super(). Si el

constructor de la clase padre tuviera argumentos, el constructor super, tomaría también estos argumentos. Por simplificar esta sección se referirá al comando super() como cualquier constructor predecesor, incluso aquellos que toman argumentos. Nótese el uso de super() y super(edad) en el siguiente ejemplo:

```
public class Animal{
  private int edad;
  public Animal(int edad){
    super();
    this.edad = edad;

  }
}

public class Zebra extends Animal{
  public Zebra(int edad){
    super(edad);
  }
  public Zebra(){
    this(4);
  }
}
```

En la primera clase, Animal, la primera sentencia del constructor es para llamar al constructor de su predecesor java.lang.Object, que no tiene argumentos. En la segunda clase, Zebra, la primera sentencia del primer constructor es una llamada al constructor de Animal, que toma un único argumento. La clase Zebra también incluye un segundo constructor sin argumentos que no llama a super() sino que llama al otro constructor de la clase Zebra utilizando this(4).

De la misma manera que el comando this() que se vio en el Capítulo 6, el comando super() solo se puede utilizar como primera sentencia del constructor. Por ejemplo, las siguientes definiciones de clases no compilarían:

```
public class Zoo {
  public Zoo() {
    System.out.println("Zoo creado");
    super(); //NO COMPILA
  }
}
public class Zoo {
  public Zoo() {
    super();
    System.out.println("Zoo creado");
    super(); //NO COMPILA
```

```
  }
}
```

La primera clase no compilará puesto que la llamada al constructor predecesor debe hacerse en la primera sentencia del constructor. En el segundo trozo de código, super() aparece tanto como primera sentencia como tercera. Dado que super() solo se puede usar como primera sentencia, el código no compilará tal y como está presentado.

Si la clase padre tiene más de un constructor, la clase hijo puede utilizar cualquier constructor válido de su predecesor en su definición, como se muestra en el siguiente ejemplo:

```
public class Animal {
  private int age;
  private String name;
  public Animal(int age, String name) {
    super();
    this.age = age;
    this.name = name;
  }
  public Animal(int age) {
    super();
    this.age = age;
    this.name = null;
  }
}

public class Gorilla extends Animal {
  public Gorilla(int age) {
    super(age,"Gorilla");
  }
  public Gorilla() {
    super(5);
  }
}
```

En este ejemplo el primer constructor hijo tiene un argumento, edad, y llama al constructor predecesor, que tiene dos argumentos, edad y nombre. El segundo constructor hijo no toma ningún argumento y al llamar al constructor predecesor, toma un único argumento, edad.

En este ejemplo, hay que tener en cuenta que los constructores hijo no necesitan llamar al constructor correspondiente de la clase padre. Cualquier constructor valido del predecesor es válido siempre que los parámetros de entrada sean los apropiados.

a. Comprendiendo las mejoras del compilador

Hasta ahora, se han definido numerosas clases que no llamaban explícitamente a los constructores de las clases predecesoras a través de la palabra clave super(), ¿cómo es posible que el código compilase? La respuesta es que el compilador de Java inserta automáticamente una llamada al constructor sin argumentos super() si la primera sentencia no es dicha llamada al constructor. Por ejemplo, las siguientes tres definiciones de clases y constructores son equivalentes, porque el compilador automáticamente las convierte para que sean como la última del ejemplo:

```
public class Burro {
}
public class Burro {
  public Burro() {
  }
}
public class Burro {
  public Burro() {
    super();
  }
}
```

Antes de seguir, asegurarse de que se entienden las diferencias entre las tres definiciones de la clase Burro y porqué Java automáticamente las convertirá. Revisar el proceso de compilación de Java y tenerlo en cuenta mientras se revisan los próximos ejemplos.

¿Qué pasa si la clase padre no tiene un constructor sin argumentos? Recordar que no es necesario y que solo se inserta si no hay ningún constructor definido en la clase. En ese caso, el compilador de Java no ayudará y se deberá crear al menos un constructor en la clase hijo que llame explícitamente al constructor padre mediante el comando super(). Por ejemplo, el siguiente código no compilará:

```
public class Mamifero {
  public Mamifero(int edad) {
  }
}
public class Elefante extends Mamifero { //NO COMPILA
}
```

En este ejemplo no se ha definido ningún constructor en la clase Elefante por lo que el compilador intenta insertar el constructor sin argumentos con una llamada super(), como hizo en el ejemplo de Burro. El compilador se para cuando se da cuenta que no hay constructor vacío en la clase predecesora. En este ejemplo, se debe explícitamente definir al menos un constructor, como en el siguiente código:

```
public class Mamifero {
  public Mamifero(int edad) {
  }
}
public class Elefante extends Mamifero {
```

```
  public Elefante() { //NO COMPILA
  }
}
```

Este código no compila porque el compilador intenta insertar en super() sin argumentos como la primera sentencia del constructor en la clase Elefante, pero no existe en la clase predecesora. Esto se puede solucionar añadiendo una llamada al constructor padre que toma un argumento fijo:

```
public class Mamifero {
  public Mamifero(int edad) {
  }
}
public class Elefante extends Mamifero {
  public Elefante() {
    super(10);
  }
}
```

Este código compilará porque se ha añadido el constructor con una llamada explicita al constructor padre. La clase Elefante tiene ahora un constructor vacío aunque la clase padre Mamifero no lo tenga. Las subclases pueden definir constructores vacíos aunque sus predecesores no los tengan, dado que el constructor de los hijos se une al de los predecesores a través de una llamada explícita con el comando super().

Se debería ser cauteloso en cualquier situación donde una clase predecesora define un constructor que toma argumentos y no define un constructor vacío.

b. Revisión de los roles de los constructores

Reglas de definición de constructores:

1. La primera sentencia de cualquier constructor debe ser una llamada a otro constructor dentro de la clase utilizando this(), o una llamada al constructor de su predecesor directo con el comando super();
2. La llamada super() no puede ser utilizada después de la primera sentencia del constructor;
3. Si no hay una llamada super() declarada en el constructor, Java inserta un super() vacío como primera sentencia del constructor.
4. Si la clase padre no tienen un constructor vacío y el hijo no define ningún constructor, el compilador arroja un error e intenta insertar un constructor vacío por defecto en la clase hijo.
5. Si la clase padre no tiene un constructor sin argumentos, el compilador requiere de una llamada explícita a un constructor de la clase padre en cada constructor de una clase hijo.

c. Llamando a los constructores

Ahora que ya se ha cubierto como definir un constructor valido, se va a mostrar como Java llama a los constructores. En Java, los constructores padres siempre se ejecutan antes que el constructor hijo. Por ejemplo, determinar cuál es el output del siguiente código:

```
class Primate {
  public Primate() {
    System.out.println("Primate");
  }
}
class Ape extends Primate {
  public Ape() {
    System.out.println("Ape");
  }
}
public class Chimpanzee extends Ape {
  public static void main(String[] args) {
    new Chimpanzee();
  }
}
```

El compilador primero inserta el comando super() como primera sentencia de los constructores en ambas clases Primate y Ape. Después, el compilador inserta un constructor por defecto vacío en la clase Chimpanzee con un comando super() como primera sentencia de su constructor. El código se ejecutará llamando primero a los constructores padres y arroja el siguiente output:

Primate

Ape

7.1.5. Llamando a miembros de clase heredada

Las clases de Java pueden usar cualquier miembro public o protected de sus clases predecesoras, incluyendo métodos, primitivas o referencias a objetos. Si ambas clases son parte del mismo paquete, la clase hijo también podrá acceder cualquier miembro por defecto (friendly) de la clase padre.

Finalmente, una clase hijo nunca podrá acceder a los miembros definidos como private en la clase padre, por lo menos no de manera directa. Como se ha podido ver en el primer ejemplo de este Capítulo, un miembro private edad se puede acceder indirectamente a través de un método public o protected.

Para referenciar un miembro en la clase predecesora, se le puede llamar directamente, como en el siguiente ejemplo:

```
class Pez {
  protected int size;
  private int edad;
  public Pez(int edad) {
```

```
    this.edad= edad;
  }
  public int getEdad() {
    return edad;
  }
}
public class Tiburon extends Pez {
  private int numeroDeDientes = 8;
  public Tiburon(int edad) {
    super(edad);
    this.size = 4;
  }
  public void displayTiburonDetalles() {
    System.out.print("Tiburon con edad: "+getEdad());
    System.out.print(" y "+size+" metros de largo");
    System.out.print(" con "+ numeroDeDientes +" dientes");
  }
}
```

En la clase hijo se ha usado el método público getEdad() y el miembro protected size para acceder a los valores de la clase padre.

Tal vez se recuerde del Capítulo 6 que se puede utilizar la palabra clave this para acceder un miembro de la clase. También se puede usar para acceder a los miembros de la clase padre que son accesibles desde la clase hijo, puesto que esta hereda todos los miembros de la clase predecesora. Considerar la siguiente definición alternativa para el método displayTiburonDetalles() del ejemplo anterior:

```
public void displayTiburonDetalles() {
  System.out.print("Tiburon con edad: "+this.getEdad());
  System.out.print(" y "+this.size+" metros de largo");
  System.out.print(" con "+this. numeroDeDientes +" dientes");
}
```

En Java se puede hacer referencia explícita a un miembro de la clase padre usando la palabra clave super, tal y como se demuestra en la siguiente versión alternativa de la definición del método displayTiburonDetalles():

```
public void displayTiburonDetalles() {
  System.out.print("Tiburon con edad: "+super.getEdad());
  System.out.print(" y "+super.size+" metros de largo");
  System.out.print(" con "+this.numeroDeDientes +" dientes");
}
```

En el ejemplo anterior, se puede usar this o super para acceder al miembro de la clase padre, pero, ¿es esto también aplicable para un miembro de la clase hijo? Considerar este ejemplo:

```
public void displayTiburonDetalles() {
  System.out.print("Tiburon con edad: "+super.getEdad());
  System.out.print(" y "+super.size+" metros de largo ");
  System.out.print(" con "+super.numeroDeDientes +" dientes"); //NO COMPILA
}
```

Este código no compilará porque numeroDeDientes es solo un método de la clase actual, no de la clase predecesora. En otras palabras, se puede ver que this y super se pueden usar para métodos o variables definidos en la clase padre, pero solo this se puede usar para miembros definidos en la clase actual.

Como se podrá ver en la sección siguiente, si una clase hijo sobrescribe un miembro de la clase padre, this y super pueden tener efectos muy distintos cuando se aplican a un miembro de la clase.

super() vs super

Como se ha visto en el Capítulo 6, this() y this no están relacionados en Java. De la misma manera, super() y super son bastante diferentes pero se pueden usar en los mismos métodos. El primero, super(), es una sentencia que llama explícitamente al constructor padre y que solo se puede usar en la primera línea del constructor de una clase hijo. El segundo super, es la palabra clave usada para referenciar un miembro definido como una clase padre y se puede usar a lo largo de la clase hijo.

Por ejemplo, considerar el código siguiente:

```
public Conejo(int edad) {
  super();
  super.setEdad(10);
}
```

La primera sentencia del constructor llama al constructor padre, mientras que la segunda línea llama a la función definida en la clase padre. Contrastar esto con el siguiente código, que no compila:

```
public Conejo(int edad) {
  super; //NO COMPILA
  super().setEdad(10); //NO COMPILA
}
```

Este código parece similar al del ejemplo previos, pero ninguna de las líneas compila porque las palabras clave se usan de manera incorrecta.

7.1.6. Heredando métodos

Heredar una clase garantiza acceso a los miembros public y protecte de la clase padre, pero también puede causar colisión entre los distintos métodos definidos en la clase padre e hijo. En esta sección se va a revisar las reglas para la herencia de métodos y como gestiona Java tales situaciones.

a. Sobrescribir un método

¿Qué pasaría si hay un método definido en ambas clases padre e hijo? Por ejemplo, se puede querer definir una nueva versión de un método existente en la clase hijo que utilice la definición de la clase padre. En este caso se puede sobrescribir el método declarando uno nuevo con la etiqueta @override y un return del mismo tipo que en la clase padre. Como se recordará del Capítulo 6, la cabecera del método incluye en nombre y la lista de los parámetros de entrada.

Cuando se sobrescribe un método, se puede hacer referencia a la versión del predecesor utilizando la palabra clave super. De esta forma, las palabras claves this y super permiten seleccionar entre la versión de la clase hijo y de la clase padre, respectivamente. Esto se ilustra en el siguiente ejemplo:

```
public class Can {
  public double getPesoMedio() {
    return 50;
  }
}
public class Lobo extends Can {
  public double getPesoMedio() {
    return super. getPesoMedio()+20;
  }
  public static void main(String[] args) {
    System.out.println(new Can().getPesoMedio ());
    System.out.println(new Lobo().getPesoMedio ());
  }
}
```

En este ejemplo, en el cual la clase heredera Lobo sobrescribe a la clase heredada Can, el método getPesoMedio() se ejecuta sin ningún problema y los outputs son los siguientes:

50.00

70.00

¿Era necesario utilizar super? Por ejemplo, qué haría el siguiente código si quitásemos la palabra reservada super en getPesoMedio() de la clase Lobo?

```
public class Lobo extends Can {
  public double getPesoMedio() {
    return  getPesoMedio()+20;//BUCLE INFINITO
  }
}
```

En este ejemplo, el compilador no llamaría al método de la clase Can; llamaría al método de la clase Lobo puesto que piensa que estás pidiendo una llamada recursiva. Las llamadas recursivas siempre deben tener una condición de fin. Como en este ejemplo no existe dicha condición, la aplicación se llama a si misma de manera indefinida y producirá un stack overflow error al ejecutar (at runtime).

Sobrescribir un método tiene sus limitaciones. El compilador hace las siguientes comprobaciones cuando se intenta sobrescribir un método que no es privado:

1. El método en la clase hijo debe tener la misma cabecera que el método en la clase predecesora;
2. El método en la clase hijo debe ser tan accesible o más que el método en la clase predecesora;
3. El método en la clase hijo no debe arrojar excepciones obligatorias que sean nuevas o más amplias que las que lance el método de la clase padre;
4. Si el método devuelve un valor, este debe ser del mismo tipo en ambas clases, o una subclase del método en la clase predecesora, conocido como covariant return types.

La primera regla a la hora de sobrescribir un método es auto explicativa. Si dos métodos, tienen el mismo nombre, pero diferente cabecera del método, los métodos están sobrecargados, no sobrescritos. Los métodos sobrecargados, como se explicó en el Capítulo 6, no están relacionados y no comparten sus propiedades.

Sobrecargar vs Sobrescribir

Sobrecargar y sobrescribir un método son similares estrategias dado que ambos se refieren a redefinir un método usando el mismo nombre. La diferencia es que cuando se sobrecarga un método se usa una definición estructural distinta que con un método sobrescrito. La distinción permite mucha más libertad sintáctica para los métodos sobrecargados. Por ejemplo:

```
public class Pajaro {
  public void volar() {
    System.out.println("El pajaro esta volando");
  }
  public void comer(int comida) {
    System.out.println("El pajaro esta comiendo "+comida+" comida ");
  }
}
public class Aguila extends Pajaro {
  public int volar(int altura) {
    System.out.println("El pajaro esta volando a"+ altura +" metros");
    return altura;
  }
  public int comer(int comida) { // DOES NOT COMPILE
    System.out.println("El pajaro esta comiendo "+ comida+ " comida");
    return comida;
  }
}
```

El primer método volar(), está sobrecargado en la subclase Aguila, dado que su definición estructural cambia de un constructor vacío a un constructor con un argumento de tipo int. Dado que el método está sobrecargado, la variable de retorno puede cambiar de void a int sin problema.

El segundo método comer(), está sobrescrito en la subclase Aguila, puesto que su cabecera es la misma que la de su clase predecesor Pajaro – ambos toman un único argumento int. Dado que el método está siendo sobrescrito, el tipo de la variable de retorno del método en Aguila debe ser una subclase del tipo de la variable de retorno en el método de la clase Pajaro. En este ejemplo void no es una subclase de int con lo que el compilador lanzará una excepción sobre la definición de este método.

Se van a revisar algunos ejemplos de las tres reglas para sobrescribir métodos para que se puedan entender los posibles problemas que vayan a surgir:

```
public class Camello {
  protected String getNumeroDeJorobas() {
    return "Indefinido";
  }
}
public class BactrianCamello extends Camello {
  private int getNumeroDeJorobas () { // NO COMPILA
    return 2;
  }
}
```

En este ejemplo, el método de la clase hijo no compila por dos razones. La primera, viola la segunda regla para los métodos sobrescritos: el método hijo debe ser al menos tan accesible como el padre. En el ejemplo, el método padre utiliza el modificador protected pero el método hijo utiliza el modificador private, de manera que es menos accesible. Tampoco respeta la cuarta regla: el tipo de las variables de retorno debe ser co-variante. En este ejemplo, el tipo de retorno del método padre es String, mientras que el retorno del método hijo es int, por lo tanto, no son covariantes.

Siempre que parezca que un método está sobrescrito, es buena idea comprobar que en realidad no está sobrecargado. Una vez que se confirme que se ha sobrescrito, se deben comprobar los modificadores, los tipos de las variables de retorno y las excepciones definidas en los métodos para asegurar que existe compatibilidad entre ellos. Se van a ver unos ejemplos de métodos que usan excepciones:

```
public class InsufficientDataException extends Exception {}
public class Reptile {
  protected boolean hasLegs() throws InsufficientDataException {
    throw new InsufficientDataException();
  }
  protected double getWeight() throws Exception {
    return 2;
  }
}
public class Snake extends Reptile {
  protected boolean hasLegs() {
    return false;
  }
```

```
  protected double getWeight() throws InsufficientDataException{
    return 2;
  }
}
```

En este ejemplo, las clases padre e hijo definen dos métodos, ‘’hasLeg()’’ y ‘’getWeight()”. El primer método ‘’hasLeg()’’, lanza una excepción ‘’InsufficientDataException’’ en la clase padre pero no lanza una excepción en la clase hijo. Esto no viola la tercera regla de métodos generales, sin embargo, no se define ninguna nueva excepción. En otras palabras, el método hijo puede ocultar o eliminar una excepción de un método padre sin problema.

El segundo método, ‘’getWeight()”, lanza una excepción en la clase padre y un ‘’InsufficientDataException’’ en la clase hijo. Esto también está permitido, como ‘’InsufficientDataException’’ es una subclase de excepción por construcción. Ninguno de los métodos en el ejemplo anterior viola la tercera regla de sobrescribir los métodos, por lo que el código se compila y se ejecuta sin ningún problema.

Se repasarán algunos ejemplos que sí que violan la tercera regla:

```
public class InsufficientDataException extends Exception {}
public class Reptile {
  protected double getHeight() throws InsufficientDataException {
    return 2;
  }
  protected int getLength() {
    return 10;
  }
}
public class Snake extends Reptile {
  protected double getHeight() throws Exception { // DOES NOT COMPILE
    return 2;
  }
  protected int getLength() throws InsufficientDataException { // DOES NOT COMPILE
    return 10;
  }
}
```

A diferencia del ejemplo anterior, ninguno de los métodos en la clase hijo de este código se compilará. El método ‘’getHeight()” en la clase padre lanza un ‘’InsufficientDataException’’ mientras que el método en la clase hijo lanza una excepción. Como excepción no es una subclase de ‘’InsufficientDataException’’, la tercera regla es violada y por tanto el código no compilara. Casualmente, excepción es una superclase de ‘’InsufficientDataException’’.

Por otra parte, el método “getLength()’’ no lanza una excepción en la clase padre, pero este lanza una excepción ‘’InsufficientDataException’’, en la clase hijo. De esta forma, la clase hijo define una ‘’new exception’’ pero en la clase padre no, que es una violación de la tercera regla de los métodos de sobrescribir.

Las últimas tres reglas para sobrescribir el método pueden parecer arbitrarias o confusas al principio, pero como veras más adelante en este capítulo cuando se hable de polimorfismo, se necesitan para lograr consistencia del lenguaje. Sin estas reglas, es posible crear contradicciones en el lenguaje Java.

b. Redeclarando métodos privados

En el apartado anterior se define el comportamiento de sobrescribir un método público o protegido en una clase. Ahora se expandirá la discusión a los métodos privados. En Java, no es posible sobrescribir un método privado en una clase padre ya que no se puede acceder al método padre desde la clase hijo. El hecho de que una clase hijo no tenga acceso al método padre significa que la clase hijo no puede definir su propia versión del método. Estrictamente hablando significa que el nuevo método no es una versión sobrescrita del método de la clase padre.

Java le permite re declarar un nuevo método en la clase hijo con la misma firma que tenía el método en la clase padre. Este método en la clase hijo es independiente y no está relacionado de la versión del método padre, por lo que ninguna de las reglas de los métodos sobrescritos se invoca. Por ejemplo, se vuelve al ejemplo de Camello que se usó en la sección anterior y muestra dos clases relacionadas que definen el mismo método.

```
public class Camello {
  private String getNumeroDeJorobas() {
    return "Indefinido";
  }
}
public class BactrianCamel extends Camello {
  private int getNumeroDeJorobas() {
    return 2;
  }
}
```

Este código compila sin problema. Aunque el tipo de devolución difiere en el método hijo de String a int. En este ejemplo el método getNumeroDeJorobas() en la clase padre está oculto, por lo que el método en la clase hijo es un método nuevo y no un método sobrescrito de la clase padre. Como se vio en la sección anterior, si el método en la clase padre era público o protegido, el método en la clase hijo no compilará porque este está violando dos reglas. El método padre en este ejemplo es privado, por tanto, no se encontrarán tales problemas.

c. Ocultando métodos estáticos

Se produce un método oculto cuando una clase hijo define un método estático con el mismo nombre y la definición estructural como un método estático definido en una clase padre. El método de ocultar es similar pero no es exactamente igual que el método de sobrescribir. En primer lugar, las cuatro reglas anteriores para sobrescribir un método se siguen también para un método oculto. Además, se agrega una nueva regla, el uso de la palabra clave estática debe ser la misma para la clase padre e hijo. La siguiente lista reúne las cinco reglas para ocultar un método:

1. El método en la clase hijo debe tener la misma definición estructural que el método en la clase padre.
2. El método en la clase hijo debe ser al menos tan accesible o más accesible que el método en la clase padre.
3. El método en la clase hijo no puede lanzar una excepción que sea nueva o mejor dicho que la clase de cualquier excepción debe estar lanzada en el método de la clase padre.
4. Si el método devuelve un valor, debe ser el mismo o una subclase del método de la clase padre, conocida como tipos de retorno covariante.
5. El método definido en la clase hijo debe marcarse como estático si está marcado como estático en la clase padre (método oculto). Del mismo modo, el método no debe marcarse como estático en la clase hijo si no está marcado como estático en la clase padre (método primordial).

Tener en cuenta que las primeras cuatro reglas son las mismas que para sobrescribir un método. Se repasarán algunos ejemplos de la nueva regla:

```
public class Bear {
  public static void eat() {
    System.out.println("Bear is eating");
  }
}
public class Panda extends Bear {
  public static void eat() {
    System.out.println("Panda bear is chewing");
  }
  public static void main(String[] args) {
    Panda.eat();
  }
}
```

En este ejemplo, el código se compila y se ejecuta sin problemas. El método eat() en la clase hijo oculta el método ‘’eat()’’ en la clase padre. Porque ambos están marcados como estáticos, esto no se considera un método sobrescrito. Se va a contrastar esto con ejemplos que violan la quinta regla :

```
public class Bear {
  public static void sneeze() {
    System.out.println("Bear is sneezing");
  }
  public void hibernate() {
    System.out.println("Bear is hibernating");
  }
}
public class Panda extends Bear {
  public void sneeze() { // DOES NOT COMPILE
    System.out.println("Panda bear sneezes quietly");
```

```
  }
  public static void hibernate() { // DOES NOT COMPILE
    System.out.println("Panda bear is going to sleep");
  }
}
```

En este ejemplo, sneeze() se marca como estático en la clase padre pero no en la clase hijo. El compilador detecta que se intenta sobrescribir un método que debería ocultarse y genera un error de compilador. En el segundo método, hibernate() es un miembro de instancia en la clase padre pero un método estático en la clase hijo. En este caso, el compilador cree que se está tratando de ocultar un método que debe ser sobrescrito y también genera un error de compilación.

Como se vio en el ejemplo anterior, ocultar los métodos estáticos es complicado con trampas y problemas potenciales y se debe evitar. Evitar ocultar métodos estáticos, ya que tienden a generar un código confuso y difícil de leer. No se debe volver a utilizar un método estático en una clase si ya está usado en la clase padre.

d. Métodos de sobrescritura vs. ocultación

En la descripción de la ocultación de los métodos estáticos, se indicó que había una distinción entre métodos de sobrescritura y métodos de ocultación. A diferencia de reescribir un método, en el que un método secundario reemplaza el método padre donde las llamadas están definidas tanto en padre como en hijo, los métodos ocultos solo reemplazan a los métodos principales en las llamadas definidas en la clase hijo. En el tiempo de ejecución, la versión secundaria de un método reemplazado siempre se ejecuta para una instancia independientemente de si la llamada al método está definida en un método clase primaria o secundaria. De esta manera, el método padre nunca se usa a menos que se haga una llamada explicita al método padre referenciado, utilizando la sintaxis ‘’ParentClassName.method()’’. Alternativamente, en la versión de un método oculto siempre se ejecuta si la llamada al método se define en la clase padre. Ejemplo:

```
public class Marsupial {
  public static boolean isBiped() {
    return false;
  }
  public void getMarsupialDescription() {
    System.out.println("Marsupial walks on two legs: "+isBiped());
  }
}
public class Kangaroo extends Marsupial {
  public static boolean isBiped() {
    return true;
  }
  public void getKangarooDescription() {
    System.out.println("Kangaroo hops on two legs: "+isBiped());
  }
  public static void main(String[] args) {
```

```
    Kangaroo joey = new Kangaroo();
    joey.getMarsupialDescription();
    joey.getKangarooDescription();
  }
}
```

En este ejemplo, el código se compila y se ejecuta sin problemas, dando como resultado lo siguiente:

Marsupial walks on two legs: false

Kangaroo hops on two legs: true

Obsérvese que isBiped() devuelve falso en la clase padre y verdadero en la clase hijo. En la primera llamada al método, se ejecuta el método padre getMarsupialDescription(). La clase Marsupial solo conoce isBiped() de su propia clase de definición, por lo que genera falso. En la segunda llamada al método, el hijo ejecuta un método isBiped(), que oculta la versión del método padre y devuelve verdadero. Contrastar este primer ejemplo con el siguiente ejemplo, que utiliza una versión sobrescrita de isBiped() en lugar de una versión oculta:

```
class Marsupial {
  public boolean isBiped() {
    return false;
  }
  public void getMarsupialDescription() {
    System.out.println("Marsupial walks on two legs: "+isBiped());
  }
}
public class Kangaroo extends Marsupial {
  public boolean isBiped() {
    return true;
  }
  public void getKangarooDescription() {
    System.out.println("Kangaroo hops on two legs: "+isBiped());
  }
  public static void main(String[] args) {
    Kangaroo joey = new Kangaroo();
    joey.getMarsupialDescription();
    joey.getKangarooDescription();
  }
}
```

En este ejemplo, el método isBiped() se reemplaza, no se oculta, en la clase hijo. Por tanto, se reemplaza en tiempo de ejecución en la clase padre con la llamada al método de la clase hijo. Asegurarse de comprender estos ejemplos, ya que muestran como oculto y sobrescritos los métodos y son fundamentalmente diferentes. Este ejemplo hace uso del polimorfismo, que se discutirá más adelante en este capítulo.

e. Creando métodos final

Se concluye la discusión sobre herencia de métodos con algo como una regla auto-explicativa: los métodos final no pueden sobrescribirse. Si se vuelve a la explicación sobre modificadores del capítulo 6, se puede crear un método con la palabra final. Haciendo esto, sin embargo, se prohíbe a la clase hijo que sobrescriba este método. Esta regla se aplica cuando se sobrescribe un método y cuando se esconde un método. En otras palabras, no se puede esconder un método estático en una clase padre si está marcado como final.

Ejemplo:

```
public class Bird {
  public final boolean hasFeathers() {
    return true;
  }
}
public class Penguin extends Bird {
  public final boolean hasFeathers() { // DOES NOT COMPILE
    return false;
  }
}
```

En este ejemplo, el método hasFeathers() está definido como final en la clase padre Bird, así pues la clase hijo Penguin no puede sobrescribir el método padre, resultando un error de compilación. Tener en cuenta que si el método hijo utilizó o no la palabra clave es irrelevante- el código no compilará tampoco de esta forma.

¿Por qué definir un método como final?

Aunque definiendo métodos como final se previene de que sean sobrescritos, esto tiene ventajas en la práctica. Por ejemplo, se definirá un método como final cuando se esté definiendo una clase padre y se quiera garantizar cierto comportamiento de un método en la clase padre, independientemente de qué clase hijo esté invocando el método.

Por ejemplo, en el ejemplo previo con Bird, el autor de la clase padre puede querer asegurarse que el método hasFeathers() siempre devuelva true, independientemente de la clase hijo en la que se invoca. El autor está seguro de que no hay ejemplo de un Bird en el cual las propiedades no estén presentes.

Los métodos no son comúnmente definidos en la práctica como final, aunque, es por esto que puede ser difícil para el autor del método de la clase padre considerar todas las posibles maneras de usar su clase hijo. Por ejemplo, aunque todos los pájaros adultos tienen plumas, un polluelo no; por lo tanto, si tienes una instancia de un pájaro que es un polluelo, no tendría plumas. En resumen, los modificadores final son solo usados en métodos cuando el autor de la clase padre quiere garantizar un comportamiento muy preciso.

7.1.7. Heredar variables

Cuando se habla de sobrescribir métodos, hay muchas reglas para los casos en los que los métodos tienen la misma cabecera y se definen tanto en la clase padre como hijo. Por suerte, las reglas para variables con mismo nombre en ambas clases son mucho más simples puesto que Java no permite sobrescribir las variables, sino ocultarlas.

a. Esconder variables

Cuando se esconde una variable, se define una variable con el mismo nombre que la de la clase padre. Esto genera dos copias de la variable dentro de una instancia de la clase hijo: una instancia definida para la referencia de la clase padre y otra definida para la referencia de la clase hijo.

Cuando se esconde un método estático, no se podrá sobrescribir una variable; solo se puede esconder. De un modo similar a la ocultación de métodos estáticos, las reglas para acceder las variables de la clase padre e hijo son similares. Si la variable está siendo referenciada desde la clase padre, la variable definida en la clase padre se usa. Por otra parte, si se está referenciando desde la clase hijo, la variable definida en la clase hijo se usa. De manera muy semejante, se puede referenciar el valor de la variable padre con el uso explícito de la palabra clave super. Considerar ahora el siguiente ejemplo:

```
public class Rodent {
  protected int tailLength = 4;
  public void getRodentDetails() {
    System.out.println("[parentTail="+tailLength+"]");
  }
}
public class Mouse extends Rodent {
  protected int tailLength = 8;
  public void getMouseDetails() {
    System.out.println("[tail="+tailLength +",parentTail="+super.tailLength+"]");
  }
  public static void main(String[] args) {
    Mouse mouse = new Mouse();
    mouse.getRodentDetails();
    mouse.getMouseDetails();
  }
}
```

El código compilará sin ningún problema y las variables de salida serán las siguientes una vez ejecutado el programa:

[parentTail=4]

[tail=8,parentTail=4]

Observar que la instancia de Mouse contiene dos copias de las variables tailLength: una definida en el padre y otra definida en la clase hijo. Estas instancias se mantienen separadas unas de otras, lo que permite a la instancia Mouse referenciar ambos valores tailLength independientemente. En la primera

llamada de método, getRodentDetails(), el método padre genera el valor padre de la variable tailLength. En la segunda llamada de método, getMouseDetails(), el método de la hijo genera ambos valores hijo y padre de las variables tailLength, usando la palabra clave super para acceder al valor de la variable padre.

La idea importante a recordar es que no hay una noción sobre sobrescribir una variable. Por ejemplo, no hay cambios en el código que puedan causar que Java sobrescriba el valor de tailLength, haciendo lo mismo para ambos padre e hijo. Estas reglas son las mismas sin importar si la variable es una variable de instancia o una variable estática.

No esconder variables en la práctica

Aunque Java permite ocultar una variable definida en la clase padre con una variable definida en la clase hijo, esto se considera una muy mala práctica de programación. Por ejemplo, observar el siguiente código, el cual usa la variable oculta length, marcada como public en ambas clases padre e hijo.

```
public class Animal {
  public int length = 2;
}
public class Jellyfish extends Animal {
  public int length = 5;
  public static void main(String[] args) {
    Jellyfish jellyfish = new Jellyfish();
    Animal animal = new Jellyfish();
    System.out.println(jellyfish.length);
    System.out.println(animal.length);
  }
}
```

Este código compila sin errores. La salida que muestra es:

5

2

Se debe tener en cuenta que el mismo tipo de objeto ha sido creado dos veces, pero la referencia al objeto determina qué valor se va a ver a la salida. Si el objeto Jellyfish se le pasase a un método con referencia a Animal, como se podrá ver en la sección "Entender Polimorfismo", más adelante en este capítulo, el valor incorrecto puede ser utilizado.

Esconder variables hace que el código sea muy confuso y difícil de leer, especialmente si se empieza a modificar el valor de las variables tanto en el método padre como en el de hijo, dado que puede que no esté claro qué variable está siendo modificada.

Cuando se define una nueva variable en la clase hijo, se considera una buena práctica si se selecciona un nombre para la variable que aún no sea una variable de tipo public, protected o default, en uso dentro de la clase predecesora. Esconder variables private se considera menos problemático puesto que la clase hijo no tiene acceso a la variable de la clase padre.

7.2. Creando clases abstractas

Se supone que se quiere definir una clase padre sobre la cuál otros desarrolladores van a hacer una subclase. El objetivo es proporcionar variables y métodos reutilizables a los desarrolladores en la clase padre, de modo que a los desarrolladores de la clase hijo se les proporcione implementaciones específicas u "overrides" de otros métodos. Además, se supone que tampoco se quiere una instancia de la clase padre a menos que ésta sea una de clase hijo.

Por ejemplo, se puede definir una clase padre Animal de la cuál extienden un cierto número de clases, pero que no se pueda realizar una instancia de Animal en sí. Se requiere que todas las subclases de la clase Animal, como por ejemplo la subclase Swan, implementen un método getName(), pero no hay una implementación de dicho método en la clase padre de Animal. ¿Cómo asegurar que todas las clases que extiendan de Animal proporcionen una implementación para este método?

En Java, se puede realizar esta tarea usando una clase abstracta y un método abstracto. Una clase abstracta es una clase que está marcada con la palabra clave abstract y no puede ser instanciada. Un método abstracto es un método marcado con la palabra clave abstract y que está definido en una clase abstracta, para el cuál no hay una implementación en la clase que ha sido declarado.

El siguiente código está basado en la descripción anterior:

```
public abstract class Animal {
    protected int age;
    public void eat() {
        System.out.println("Animal is eating");
    }
    public abstract String getName();
}
public class Swan extends Animal {
    public String getName() {
        return "Swan";
    }
}
```

Lo primero a destacar de este código de ejemplo es que la clase Animal está declarada como abstract y la clase Swan no. Lo siguiente a destacar, es que la variable age y el método eat() están declarados como protected y public respectivamente; por lo tanto, se heredan en subclases como Swan. Finalmente, el método abstracto getName() termina con un punto y coma y no tiene una implementación en la clase padre. Este método está implementado con el mismo nombre y cabecera del método de la clase padre en la clase Cisne.

7.2.1. Definiendo una clase abstracta

El código de ejemplo anterior ilustra un número importante de reglas sobre las clases abstractas. Por ejemplo, una clase abstracta puede incluir métodos y variables que no sean abstractos, como la variable age y el método eat(). De hecho, no se requiere que una clase abstracta defina métodos abstractos. Por ejemplo, el siguiente código compila sin ningún problema aunque no defina ningún método abstracto:

```
public abstract class Cow {
}
```

Aunque una clase abstracta no tiene por qué definir ningún método abstracto, un método abstracto solamente puede ser definido en una clase abstracta. Por ejemplo, el siguiente código no compila porque el método abstracto no está definido en una clase abstracta:

```
public class Chicken {
  public abstract void peck();  // DOES NOT COMPILE
}
```

Otro ejemplo, ningún método en el siguiente código compila porque los métodos están marcados como abstract:

```
public abstract class Turtle {
  public abstract void swim() {} // DOES NOT COMPILE
  public abstract int getAge() { // DOES NOT COMPILE
    return 10;
  }
}
```

El primer método, swim(), no compila porque tiene las llaves en vez de un punto y coma, y Java interpreta que se va a implementar código en el método abstracto. El segundo método, getAge(), no compila porque también implementa código en el método abstracto.

Implementación de métodos predeterminados en clases abstractas

Aunque no se puede proporcionar una implementación predeterminada a un método abstracto en una clase abstracta, se puede definir un método con un cuerpo –no se tiene que marcar como abstracto. Siempre que no se marque como final, la subclase tiene la opción de sobrescribirla, como se ha explicado en la sección anterior.

De esta forma, queda claro que una clase abstracta no puede ser marcada como final por razones obvias. Por definición, una clase abstracta es una clase que va a ser extendida por otra clase para ser instanciada, mientras que una clase final no puede ser extendida por otra clase. Marcando una clase abstracta como final, se está diciendo que la clase nunca puede ser instanciada, por lo que el compilador no procesa el código. Por ejemplo, el siguiente código no compilará:

```
public final abstract class Tortoise { // DOES NOT COMPILE
}
```

Del mismo modo, un método abstracto no puede ser marcardo como final por las mismas razones que una clase abstracta no puede ser marcada como final. Una vez marcada como final, el método no puede ser sobrescrito en una subclase, haciendo así que sea imposible crear una instancia concreta de la clase abstracta.

```
public abstract class Goat {
public abstract final void chew(); // DOES NOT COMPILE
}
```

Finalmente, un método no puede marcarse al mismo tiempo como abstract y privado. Esta regla tiene mucho sentido. ¿Cómo se definiría una subclase que implementa un método requerido si el método no está accesible por la propia subclase? La respuesta es que no se puede, por lo que el compilador mostrará si se intenta hacerlo.

```
public abstract class Whale {
  private abstract void sing(); // DOES NOT COMPILE
}
public class HumpbackWhale extends Whale {
  private void sing() {
    System.out.println("Humpback whale is singing");
  }
}
```

En este ejemplo, el método abstracto sing() definido en la clase padre Whale no está visible para la subclase HumpbackWhale. Aunque HumpbackWhale tiene una implementación, esta no está considerada como un "override" del método abstracto ya que el método abstracto no es visible. El compilador reconoce esta situación en la clase padre y lanza una excepción tan pronto como private y abstract son aplicados al mismo método. Si se cambia el modificador de acceso de private a protected en la clase padre, ¿compilará el código? Veamos:

```
public abstract class Whale {
  protected abstract void sing();
}
public class HumpbackWhale extends Whale {
  private void sing() { // DOES NOT COMPILE
    System.out.println("Humpback whale is singing");
  }
}
```

En este ejemplo modificado, el código no compilará pero por un motivo totalmente diferente. Si se recuerdan las reglas descritas en este capítulo sobre la sobrescritura de un método, la subclase sigue sin poder ver el método sing() de la clase padre. Debido a que el método está declarado como protected

en la clase padre, éste debería ser marcado como protected o public en la clase hijo. Incluso con métodos abstractos, se deben seguir las reglas para sobrescritura de métodos.

7.2.2. Creando una clase concreta (Concret class)

Cuando se trabaja con clases abstractas, es importante recordar que no pueden ser instanciadas por sí solas y, por lo tanto, no hacen más que definir variables y métodos estáticos. Por ejemplo, el siguiente código no compilará ya que es un intento de instanciar una clase abstracta:

```
public abstract class Eel {
  public static void main(String[] args) {
    final Eel eel = new Eel(); // DOES NOT COMPILE
  }
}
```

Una clase abstracta se vuelve útil cuando es extendida por una clase hijo. Una "clase concreta" es la primera subclase no abstracta que extiende una clase abstracta y es requerida para implementar todos los métodos abstractos heredados. Cuando se vea una clase concreta extendiendo una clase abstracta, hay que comprobar que implementa todos los métodos abstractos requeridos. Se verá en el siguiente ejemplo:

```
public abstract class Animal {
  public abstract String getName();
}
public class Walrus extends Animal { // DOES NOT COMPILE
}
```

Primero, nótese que la clase Animal está marcada como abtract y la clase Walrus no lo está. En este ejemplo, Walrus está considerado como la primera subclase concreta de Animal. Partiendo de que Walrus es la primera subclase concreta, esta debe heredar todos los métodos abstractos, en este ejemplo únicamente debería heredar getName(). Debido a que no se ha hecho, el compilador devuelve un error.

Nótese que cuando se define una clase concreta como la primera subclase no abstracta, incluimos la posibilidad de que otra clase no abstracta pueda extender una clase no abstracta existente. La idea clave es que la primera clase en extender la clase no abstracta debe implementar todos los métodos heredables. Por ejemplo, la siguiente variación no compila:

```
public abstract class Animal {
  public abstract String getName();
}
public class Bird extends Animal { // DOES NOT COMPILE
}
public class Flamingo extends Bird {
  public String getName() {
    return "Flamingo";
```

```
  }
}
```

Aunque una segunda subclase Flamingo implementa el método abstracto getName(), Bird es la primera subclase concreta; por lo tanto, la clase Bird no compilará.

7.2.3. Extendiendo una clase abstracta

Se va a expandir la discusión sobre las clases abstractas introduciendo el concepto: extendiendo una clase abstracta con otra abstracta. Se repetirá el ejemplo previo con Walrus pero con una pequeña variación:

```
public abstract class Animal {
  public abstract String getName();
}
public class Walrus extends Animal { // DOES NOT COMPILE
}
public abstract class Eagle extends Animal {
}
```

En este ejemplo, de nuevo se tiene una clase abstracta Animal con una clase concreta Walrus que no compila ya que no implementa el método getName(). También se tiene una clase abstracta Eagle, que como Walrus extiende de Animal y no implementa el método getName(). En esta situación, Eagle compila porque está marcada como abstracta.

Como se ha visto en este ejemplo, una clase abstracta puede extender a otra clase abstracta pero no requiere que ésta implemente ninguno de los métodos abstractos. Además, se ha visto que una clase concreta que extiende una clase abstracta debe implementar todos los métodos abstractos heredables. Por ejemplo, la clase concreta Lion del siguiente ejemplo implementará dos métodos, getName() y roar():

```
public abstract class Animal {
  public abstract String getName();
}
public abstract class BigCat extends Animal {
  public abstract void roar();
}
public class Lion extends BigCat {
  public String getName() {
    return "Lion";
  }
  public void roar() {
    System.out.println("The Lion lets out a loud ROAR!");
  }
}
```

En este código de ejemplo, BigCat extiende de Animal pero está marcada como abstract; por lo que, no se requiere una implementación para el método getName(). La clase Lion no está marcada como abstract¸y como es la primera clase concreta, debe implementar todos los métodos abstractos heredables no definidos en la clase padre.

Hay una excepción a la regla para los métodos abstractos y las clases concretas: no se requiere una subclase concreta para proporcionar una implementación de un método abstracto si una clase abstracta intermedia proporciona la implementación. Por ejemplo, observar la siguiente variación del ejemplo anterior:

```
public abstract class Animal {
  public abstract String getName();
}
public abstract class BigCat extends Animal {
  public String getName() {
    return "BigCat";
  }
  public abstract void roar();
}
public class Lion extends BigCat {
  public void roar() {
    System.out.println("The Lion lets out a loud ROAR!");
  }
}
```

En este ejemplo, BigCat proporciona una implementación para el método abstracto getName() definido en la clase abstracta Animal. Por lo tanto, Lion hereda solo un método abstracto, roar(), y no requiere la implementación del método getName().

Una forma de pensar esto es: si una clase intermedia proporciona una implementación para un método abstracto, ese método es heredado por subclases como un método concreto, no como uno abstracto. En otras palabras, las subclases no lo consideran como un método abstracto heredado porque no es abstracto en el momento que llegan a las subclases.

Las siguientes son listas de reglas para clases abstractas y métodos abstractos que se han abordado en esta sección.

Reglas para la definición de clases abstractas:

1. Las clases abstractas no pueden ser instanciadas directamente.
2. Las clases abstractas deben ser definidas con cualquier número, incluido cero, de métodos abstractos o no abstractos.
3. Las clases abstractas no deben ser marcadas como private o final.
4. Una clase abstracta que extiende a otra clase abstracta hereda todos sus métodos abstractos como suyos.
5. La primera clase concreta que extiende una clase abstracta debe proporcionar una implementación para todos los métodos abstractos heredados.

Reglas para la definición de métodos abstractos:

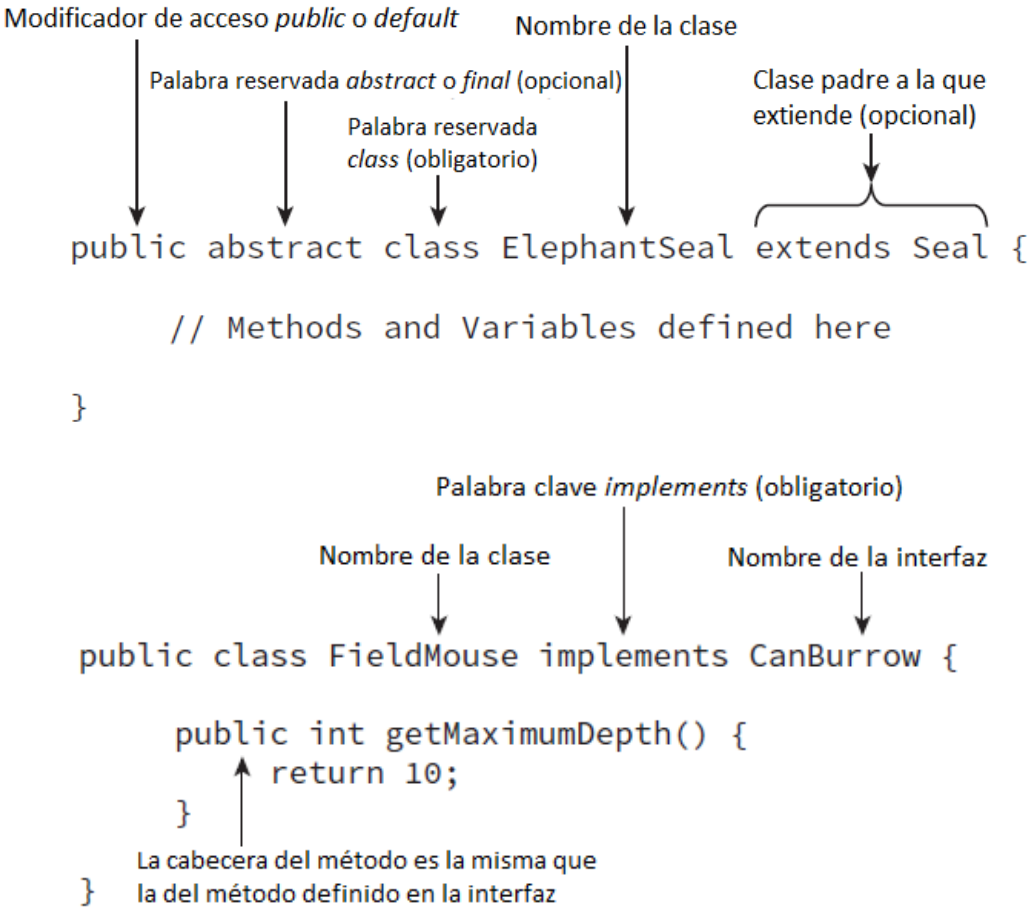
- Los métodos abstractos solo se pueden definir en clases abstractas.
- Los métodos abstractos no pueden ser declarados como private o final.
- Los métodos abstractos no deben implementar el cuerpo en la clase abstracta en la que han sido declarados.
- La definición de un método abstracto en una subclase sigue las mismas reglas para sobrescribir un método. Por ejemplo, el nombre y la cabecera deben ser la misma, y la visibilidad del método en la subclase debe ser al menos tan accesible como en el método de la clase padre.

7.3. Implementando interfaces

Aunque Java no permite herencia múltiple, permite que las clases implementen cualquier número de interfaces. Una interfaz es un tipo de datos abstracto que define una lista de métodos abstractos que cualquier clase que implemente a la interfaz debe proporcionar. Una interfaz también puede incluir una lista de variables constantes y métodos por defecto, los cuales se verán en esta sección.

En Java, una interfaz se define con la palabra clave interface, de la misma manera que se usa class para definir una clase. La clase implementa la interfaz mediante la palabra clave implements cuando se define la clase.

En la siguiente figura se muestra la utilización correcta de sintaxis.



Como se puede ver en el ejemplo anterior, una interfaz no se declara como una clase abstracta, aunque sí tiene muchas propiedades comunes con ésta. Nótese que los métodos de la interfaz en el ejemplo (abstract y public) son asumidos. En otras palabras, si no se ponen, el compilador los inserta automáticamente como parte de la definición del método.

Una clase puede implementar múltiples interfaces, separadas por comas:

```
public class Elephant implements WalksOnFourLegs, HasTrunk, Herbivore {}
```

En el ejemplo, si alguna de las interfaces define algún método abstracto, la clase concreta Elephant requerirá implementarlos.

En java 8 se ha introducido los conceptos de métodos estáticos y por defecto (static, default) en las interfaces, los cuales se verán al final de esta sección.

7.3.1. Definiendo una Interfaz

Puede resultar de ayuda pensar que una interfaz es un tipo de clase abstracta especializada, ya que comparte muchas de las mismas propiedades y reglas que una clase abstracta. A continuación, se detalla una lista de reglas para crear una interfaz, muchas de la cuales son como adaptaciones de las reglas para definir clases abstractas.

- Las interfaces no se pueden instanciar directamente.
- Una interfaz no requiere implementar métodos.
- Una interfaz no puede estar marcada como final.
- Se asume que todas las interfaces de alto nivel tienen acceso público o por defecto (public, default), y deben incluir el modificador abstract en su definición. Por lo tanto, marcar una interface como private, protected o final, lanzará un error de compilación, al ser incompatible con esta definición.
- Se asume que todos los métodos, en una interfaz, que no están marcados como default tienen los modificadores abstract y public en su definición. Por lo tanto, marcar un método como private, protected o final lanzará un error de compilación al ser éstos incompatibles con las palabras reservadas abstract y public.

La cuarta regla no se aplica a interfaces anidadas. Las tres primeras reglas son idénticas a las tres primeras reglas para crear una clase abstracta. Imaginar que se tiene una interfaz "WalksOnTwoLegs", definida así:

```
public interface WalksOnTwoLegs {}
```

Compila sin problemas al no tener que definir ningún método por ser una interfaz. Ahora se tienen los siguientes dos ejemplos que no compilan:

```
public class TestClass {
  public static void main(String[] args) {
      WalksOnTwoLegs example = new WalksOnTwoLegs(); // DOES NOT COMPILE
  }
```

```
  public final interface WalksOnEightLegs { // DOES NOT COMPILE
}
```

El primer ejemplo no compila, al ser "WalksOnTwoLegs", una interfaz y no puede ser instanciada directamente. El segundo ejemplo, "WalksOnEightLegs", no compila al no poder marcar una interfaz como final por la misma razón que la clase abstracta no puede ser marcada como final.

La cuarta y quinta regla sobre "palabras clave asumidas" pueden ser nuevas, pero se debe pensar en ellas de la misma manera en que el compilador inserta por defecto el constructor sin argumentos o la llamada super() en el constructor. Se pueden proporcionar estos modificadores uno mismo, aunque el compilador los inserta automáticamente si no se hace. Por ejemplo, las siguientes dos definiciones de interfaces son equivalentes, ya que el compilador las convertirá a ambas como el segundo ejemplo.

```
public interface CanFly {
  void fly(int speed);
  abstract void takeoff();
  public abstract double dive();
}
public abstract interface CanFly {
  public abstract void fly(int speed);
  public abstract void takeoff();
  public abstract double dive();
}
```

En este ejemplo, la palabra reservada abstract se añade primero, automáticamente, a la definición de la interfaz. Entonces, cada método se supone con las palabras reservadas public y abstract. Si el método ya tiene cualquiera de éstas, no precisa ningún cambio. Ahora, se verá un ejemplo que viola las palabras reservadas asumidas:

```
private final interface CanCrawl { // DOES NOT COMPILE
  private void dig(int depth); // DOES NOT COMPILE
  protected abstract double depth(); // DOES NOT COMPILE
  public final void surface(); // DOES NOT COMPILE
}
```

Ninguna de las líneas de este ejemplo compila. La primera no compila por dos razones. Primero, está marcada como final, cosa que no se puede aplicar a ninguna interfaz por entrar en conflicto con la palabra reservada asumida abstract. La siguiente razón es que está marcada como private, que entra en conflicto con el requisito de acceso public o default de las interfaces. La segunda y tercera línea no compilan porque todos los métodos de la interfaz se asumen que son public y marcarlos como private o protected provoca un error de compilación. Finalmente, la última línea no compila debido a que el método está declarado como final y los métodos de las interfaces se asume que son abstract, entonces el compilador lanza una excepción por utilizar ambas palabras reservadas a la vez.

Añadiendo las palabras clave asumidas a una interfaz es cuestión de preferencia de cada uno, aunque es considerado como buena práctica hacerlo. El código con las palabras claves asumidas que están

escritas, tiende a ser más legible, lo cual reduce posibles conflictos potenciales, como se ha podido ver en los ejemplos anteriores.

7.3.2. Heredando una interfaz

Hay dos reglas para la herencia que se deben seguir cuando se extiende una interfaz:

- Una interfaz que extiende a otra, así como una clase abstracta que implementa una interfaz, hereda todos los métodos abstractos como métodos abstractos propios.
- La primera clase concreta que implementa una interfaz, o extiende una clase abstracta que implementa una interfaz, debe proveer una implementación para todos los métodos abstractos heredados.

Como una clase abstracta, una interfaz puede extenderse utilizando la palabra reservada extends. De este modo, la nueva interfaz hijo hereda todos los métodos abstractos de la interfaz padre. A diferencia de una clase abstracta, una interfaz puede extender múltiples interfaces. Se considera el siguiente ejemplo:

```
public interface HasTail {
  public int getTailLength();
}
public interface HasWhiskers {
  public int getNumberOfWhiskers();
}
public interface Seal extends HasTail, HasWhiskers {
}
```

Cualquier clase que implemente la interfaz "Seal" debe proporcionar una implementación para todos los métodos de las interfaces padre – en este caso, "getTailLength()" y "getNumberOfWhiskers()".

¿Qué sucede cuando una clase abstracta implementa una interfaz? En este caso, la clase abstracta es tratada de igual manera que una interfaz extendiendo a otra. En otras palabras, la clase abstracta hereda los métodos abstractos de la interfaz, pero no se requiere su implementación. Ocurre igual, como en una clase abstracta, que la primera clase que extiende la clase abstracta debe implementar los métodos heredados de la interfaz. Se muestra en el siguiente ejemplo:

```
public interface HasTail {
  public int getTailLength();
}
public interface HasWhiskers {
  public int getNumberOfWhiskers();
}
public abstract class HarborSeal implements HasTail, HasWhiskers {
}
public class LeopardSeal implements HasTail, HasWhiskers { //      DOES NOT COMPILE
}
```

En este ejemplo, se ve que "HarborSeal" es una clase abstracta que compila sin problemas. Cualquier clase que extienda "HarborSeal" tiene que implementar los métodos en las interfaces "HasTail" y "HasWhiskers". Por otra parte, "LeopardSeal" no es una clase abstracta, así que debe implementar todos los métodos definidos en la interfaz. En este ejemplo, "LeopardSeal" no implementa los métodos de la interfaz, por lo que el código no compila.

a. Clases, interfaces y palabras clave

Las clases pueden implementar una interfaz, pero no pueden extender una interfaz. Asimismo, mientras una interfaz puede extender otra, no permite implementarla.

El siguiente código enseña estos principios:

```
public interface CanRun {}
public class Cheetah extends CanRun {} // DOES NOT COMPILE
public class Hyena {}
public interface HasFur extends Hyena {} // DOES NOT COMPILE
```

El primer ejemplo muestra una clase intentando extender a una interfaz por lo que no compila. El segundo ejemplo se trata de una interfaz intentando extender una clase, lo cual tampoco compila.

b. Métodos abstractos y múltiples interfaces

Desde Java se permite herencia múltiple mediante interfaces, pero ¿qué ocurre si se define una clase que hereda de dos interfaces que contienen el mismo método abstracto?

```
public interface Herbivore {
  public void eatPlants();
}
public interface Omnivore {
  public void eatPlants();
  public void eatMeat();
}
```

En este escenario, la definición de los dos métodos de interfaz "eatPlants()" son compatibles, por lo que se puede definir que completa ambas interfaces simultáneamente:

```
public class Bear implements Herbivore, Omnivore {
  public void eatMeat() {
    System.out.println("Eating meat");
  }
  public void eatPlants() {
    System.out.println("Eating plants");
  }
}
```

¿Por qué funciona? En este ejemplo, hay que recordar que los métodos de la interfaz son abstractos y definen el "comportamiento" que debe tener la clase que los implementa. Si dos métodos abstractos de una interfaz tienen comportamientos idénticos, o en este caso la misma definición del método, en el momento que se crea una clase que implementa uno de los dos métodos, automáticamente implementa el segundo. De este modo, los métodos de la interfaz se consideran duplicados, ya que tienen la misma forma.

¿Qué sucede si los dos métodos están definidos de distinta manera? Si el nombre del método es el mismo pero los parámetros son distintos, no hay conflicto ya que se considera que son métodos sobrecargados. Se demuestra este principio en el siguiente ejemplo:

```
public interface Herbivore {
  public int eatPlants(int quantity);
}
public interface Omnivore {
  public void eatPlants();
}
public class Bear implements Herbivore, Omnivore {
  public int eatPlants(int quantity) {
    System.out.println("Eating plants: "+quantity);
    return quantity;
  }
  public void eatPlants() {
    System.out.println("Eating plants");
  }
}
```

En este ejemplo se ve que la clase que implementa ambas interfaces debe proporcionar una implementación de las dos versiones de "eatPlants()", al ser considerados como métodos separados. No importa si el valor de retorno es del mismo tipo o no, ya que el compilador trata a ambos métodos como independientes.

Desafortunadamente, si el nombre del método y los parámetros son iguales, pero el valor de retorno es distinto, la clase o interfaz que intenta heredar ambas interfaces no compilará. La razón por la que el código no compila tiene que ver con el diseño de la clase y no con las interfaces, como se discutió en el capítulo 6. No es posible en Java definir dos métodos en una clase con el mismo nombre y mismos parámetros, pero diferente valor de retorno. Dadas las siguiente dos definiciones de las interfaces "Herbivore" y "Onivore", el siguiente código no compila:

```
public interface Herbivore {
  public int eatPlants();
}
public interface Omnivore {
  public void eatPlants();
}
```

```
public class Bear implements Herbivore, Omnivore {
  public int eatPlants() { // DOES NOT COMPILE
    System.out.println("Eating plants: 10");
    return 10;
  }
  public void eatPlants() { // DOES NOT COMPILE
    System.out.println("Eating plants");
  }
}
```

El código no compila, porque la clase define dos métodos con el mismo nombre y parámetros, pero diferente valor de retorno. Si se eliminase alguna de las dos definiciones de "eatPlants()", el compilador se pararía a causa de que falta uno de los dos métodos requeridos. En otras palabras, no hay ninguna manera de implementar ambas interfaces "Herbivore" y "Omnivore" que pueda aceptar el compilador.

El compilador lanzará una excepción si se define una interfaz o clase abstracta que hereda de dos interfaces conflictivas, como se muestra aquí:

```
public interface Herbivore {
  public int eatPlants();
}
public interface Omnivore {
  public void eatPlants();
}
public interface Supervore extends Herbivore, Omnivore {} // DOES NOT COMPILE
public abstract class AbstractBear implements Herbivore, Omnivore {} // DOES NOT COMPILE
```

Incluso sin implementar detalles, el compilador detecta el problema con la definición abstracta y previene la compilación.

Con esto concluye la discusión de los métodos abstractos de la interfaz y la herencia múltiple. Se volverá a la discusión pronto, después de explicar los métodos por defecto (default) de las interfaces. Se verá que esto funciona diferente con los métodos por defecto (default) de las interfaces.

7.3.3. Variables en las interfaces

Se expande la discusión de la inclusión de variables en las interfaces, las cuales pueden estar definidas dentro de una interfaz. Como los métodos de la interfaz, las variables de la interfaz se asumen que son públicas (public). A diferencia de los métodos de la interfaz, las variables de la interfaz son también asumidas como estáticas (static) y finales (final).

Aquí hay dos reglas de las variables en interfaces:

- Las variables en una interfaz se asumen que son public, static y final. Por lo tanto, marcar una variable como private o protected lanzara un error de compilación, ya que todas las variables se marcarán como abstract.
- El valor de una variable de la interfaz debe declararse al estar declarada como final.

De esta manera, las variables de la interfaz son esencialmente variables constantes definidas a nivel de interfaz. Como se asumen que son estáticas (static), son accesibles incluso sin una instancia de la interfaz. Como en el anterior ejemplo "CanFly", las siguientes dos interfaces son equivalentes, porqué el compilador convertirá ambas como en el segundo ejemplo:

```
public interface CanSwim {
  int MAXIMUM_DEPTH = 100;
  final static boolean UNDERWATER = true;
  public static final String TYPE = "Submersible";
}
```

Como se ve en este ejemplo, el compilador insertará automáticamente las palabras reservadas public, static y final a todas las variables de la interfaz a las que les falta. Es una práctica común definir las constantes en mayúsculas.

Basándose en estas reglas, no es una sorpresa que el siguiente código no compile:

```
public interface CanDig {
  private int MAXIMUM_DEPTH = 100; // DOES NOT COMPILE
  protected abstract boolean UNDERWATER = false; // DOES NOT COMPILE
  public static String TYPE; // DOES NOT COMPILE
}
```

El primer ejemplo, "MAXIMUM_DEPTH", no compila porque está declarada como private, y todas las variables de la interfaz se asume que son public. La segunda línea, "UNDERWATER", no compila por dos razones. Esta declarada como protected lo que entra en conflicto con public, y está también declarada como abstract, lo que entra en conflicto con final. Finalmente, el último ejemplo, "TYPE", no compila debido a que no está inicializada. A diferencia de otros ejemplos, la declaración es correcta, pero como recordareis del capítulo 6, se debe proveer de un valor a una variable de la clase declarada como static final.

7.3.4. Métodos por defecto de las interfaces

Con el lanzamiento de Java 8, sus autores han introducido un nuevo tipo de método a las interfaces, referenciados como los métodos por defecto. Un método por defecto (default) es un método definido dentro de la interface con la palabra clave default en el que se proporciona el cuerpo del método. Por el contrario, los métodos "normales" de la interface se asumen que son abstractos y no pueden estar implementados.

Un método por defecto dentro de una interfaz define un método abstracto con una implementación por defecto. De este modo, las clases tienen una opción para sobrescribir el método por defecto si lo necesitan, pero no están obligadas a hacerlo. Si la clase no sobrescribe el método, la implementación del método por defecto se utilizará. Así, la definición del método no es abstracta.

El propósito de añadir métodos por defecto al lenguaje de Java era en parte para ayudar con el código de desarrollo y la compatibilidad de versiones antiguas. Imaginar que se tiene una interfaz que está compartida entre docenas o incluso cientos de usuarios a la cual se quiere añadir un nuevo método. Si

se actualiza directamente la interfaz con el nuevo método, la implementación se rompería para todos los subscriptores, quienes se verán forzados a actualizar su código. En la práctica, esto puede incluso disuadir a uno de realizar los cambios. Al proporcionar una implementación por defecto del método, la interfaz se vuelve compatible con el código existente a la vez que proporciona la opción de sobrescribirlo, a quienes lo deseen.

El siguiente es un ejemplo de un método por defecto declarado en una interfaz:

```
public interface IsWarmBlooded {
  boolean hasScales();
  public default double getTemperature() {
    return 10.0;
  }
}
```

Este ejemplo define dos métodos de la interfaz, uno es un método abstracto normal, y el otro es un método por defecto. Ambos métodos se asumen que son públicos. El primer método termina con punto y coma, y no tiene implementación, mientras el segundo implementa el cuerpo del método. Cualquier clase que implemente "IsWarmBlooded" puede utilizar la implementación por defecto de "getTemperature()" o sobrescribir el método y crear su propia versión.

Nótese que la declaración de acceso por defecto del capítulo 6, es completamente diferente del método por defecto definido en este capítulo. Se definió la declaración de acceso por defecto en el capítulo 6, al no declarar que tipo de acceso se tiene, lo que indica que una clase puede acceder a otra clase, método o valor dentro de otra clase si ambas se encuentran en el mismo paquete. En este capítulo, se habla especialmente sobre la palabra clave default aplicada a un método de la interfaz. Como todos los métodos declarados en una interfaz se asumen que son públicos, los métodos por defecto también lo son.

A continuación, las reglas de los métodos por defecto de las interfaces:

- Un método por defecto solamente puede estar declarado en una interfaz y no en una clase o clase abstracta.
- Los métodos por defecto deben estar declarados con la palabra clave default, y debe implementar el cuerpo del método.
- Los métodos por defecto no se asumen que son estáticos, finales o abstractos, al poder ser utilizados y sobrescritos por la clase que implemente la interfaz.
- Como todos los métodos de una interfaz, un método por defecto se asume que es público y no compilará si se marca como privado o protegido.

La primera regla debería ser fácil de comprender, ya que las interfaces son las únicas que implementan estos métodos. La segunda regla habla de la sintaxis, estos métodos deben utilizar la palabra reservada default. Por ejemplo, el siguiente trozo de código no compila:

```
public interface Carnivore {
  public default void eatMeat(); // DOES NOT COMPILE
  public int getRequiredFoodAmount() { // DOES NOT COMPILE
```

```
    return 13;
  }
}
```

En este ejemplo, el primer método "eatMeat()" , no compila debido a que está declarado como método por defecto pero no implementa el cuerpo de éste. El segundo método, "getRequiredFoodAmount()", también da error de compilación porque implementa el cuerpo del método sin estar declarado como método por defecto.

A diferencia de las variables de la interfaz, que se asumen que son estáticas, los métodos por defecto no pueden ser estáticos y requieren una instancia de la clase implementando a la interfaz para ser invocados. También pueden no estar marcados como finales o abstractos, porque se les permite ser sobrescritos en subclases, pero no es obligatorio sobrescribirlos.

Cuando una interfaz extiende a otra que contiene un método por defecto, puede elegir ignorarlo, en cuyo caso la implementación por defecto se utilizará. Por otro lado, la interfaz puede sobrescribir la definición del método por defecto utilizando las reglas estándar para sobrescribir los métodos, tal como no limitar la accesibilidad del método y utilizar el mismo valor de retorno. Finalmente, la interfaz puede volver a declarar el método como abstracto, forzando a las clases que implementen la nueva interfaz para proporcionar, explícitamente, un cuerpo del método. Se aplican las mismas opciones para una clase abstracta que implementa una interfaz.

Por ejemplo, la siguiente clase sobrescribe un método por defecto y vuelve a declarar el segundo método como abstracto.

```
public interface HasFins {
  public default int getNumberOfFins() {
    return 4;
  }
  public default double getLongestFinLength() {
    return 20.0;
  }
  public default boolean doFinsHaveScales() {
    return true;
  }
}
public interface SharkFamily extends HasFins {
  public default int getNumberOfFins() {
    return 8;
  }
  public double getLongestFinLength();
  public boolean doFinsHaveScales() { // DOES NOT COMPILE
    return false;
  }
}
```

En este ejemplo, la primera interface, "HasFins", define tres métodos por defecto: "getNumberOfFins()", "getLongestFinLength()", y "doFinsHaveScales()". La segunda interfaz, "SharkFamily", extiende "HasFins" y sobrescribe el método por defecto "getNumberOfFins()" con un nuevo método que devuelve un valor distinto. Después, la interfaz "SharkFamily" reemplaza el método por defecto "getLongestFinLentgh()" con un nuevo método abstracto, forzando a toda clase que implemente "SharkFamily" a implementar dicho método. Finalmente, la interfaz "SharkFamily" sobrescribe el método "doFinsHaveScales()" pero no lo marca como método por defecto. Como las interfaces solamente pueden contener métodos implementados si se declaran como métodos por defecto, el código no compila.

a. Métodos por defecto y herencia múltiple

Permitiendo métodos por defecto en las interfaces, junto con la posibilidad de que una clase implemente múltiples interfaces, Java abre la puerta a problemas de herencia múltiple. Por ejemplo, ¿qué valor de salida tiene el siguiente código?

```
public interface Walk {
  public default int getSpeed() {
    return 5;
  }
}
public interface Run {
  public default int getSpeed() {
    return 10;
  }
}
public class Cat implements Walk, Run { // DOES NOT COMPILE
  public static void main(String[] args) {
    System.out.println(new Cat().getSpeed());
  }
}
```

En este ejemplo, Cat hereda los dos métodos por defecto para getSpeed(), así que, ¿cuál usa? Dado que Walk y Run se consideran hermanos en términos de como son usados en la clase Cat, no está claro si la salida del código debería ser 5 o 10. La respuesta es que el código no compila.

Si una clase implementa dos interfaces que tienen métodos por defecto con el mismo nombre y misma cabecera, el compilador lanza un error. Sin embargo, hay una excepción para esta regla: si la subclase sobrescribe los métodos por defecto duplicados, el código compilara sin problema, puesto que la ambigüedad sobre qué versión del método se llama, desaparece. Por ejemplo, la siguiente modificación del código compilara y tendrá salida 1:

```
public class Cat implements Walk, Run {
  public int getSpeed() {
    return 1;
```

```
  }
  public static void main(String[] args) {
    System.out.println(new Cat().getSpeed());
  }
}
```

Se puede ver que tener una clase que implementa o hereda dos métodos por defecto duplicados fuerza a la clase a implementar una nueva versión del mismo, para poder compilar el código. Esta regla se mantiene para clases abstractas que implementan interfaces múltiples, puesto que los métodos por defecto pueden ser llamados desde un método concreto de una clase abstracta.

7.3.5. Métodos estáticos en interfaces

Java 8 también acepta la definición de métodos estáticos en las interfaces. Estos métodos se definen de manera explícita con la palabra reservada static y funcionan casi idénticamente a los métodos estáticos definidos en las clases, como se explicó en el Capítulo 6. De hecho, hay en realidad solo una distinción entre los métodos estáticos definidos en una clase y en una interfaz. El método definido en una interfaz no es heredado en ninguna de las clases que implementan la interfaz.

Aquí se pueden ver las reglas para los métodos estáticos en interfaces:

- Como todos los métodos de una interfaz, el método estático se asume público y no compilará si se define como protected o private.
- Para referenciar un método estático, se debe utilizar una referencia al nombre de la interfaz.

El siguiente es un ejemplo de un método estático definido en una interfaz:

```
public interface Hop {
  static int getJumpHeight() {
    return 8;
  }
}
```

7.4. Entendiendo el polimorfismo

Java soporta el polimorfismo, la propiedad de un objeto de adquirir diferentes formas. Para mostrar esto de una forma más precisa, un objeto en Java puede ser accedido usando una referencia con el mismo tipo que el objeto, una referencia que es una superclase del objeto, o una referencia que define una interfaz que el objeto implementa, directamente o a través de una superclase. Además, no se requiere un casting si el objeto está siendo asignado a un súper tipo o interfaz del objeto.

Se verá la propiedad del polimorfismo con el siguiente ejemplo:

```
public class Primate {
  public boolean hasHair() {
    return true;
```

```
  }
}
public interface HasTail {
  public boolean isTailStriped();
}
public class Lemur extends Primate implements HasTail {
  public boolean isTailStriped() {
    return false;
  }
  public int age = 10;
  public static void main(String[] args) {
    Lemur lemur = new Lemur();
    System.out.println(lemur.age);
    HasTail hasTail = lemur;
    System.out.println(hasTail.isTailStriped());
    Primate primate = lemur;
    System.out.println(primate.hasHair());
  }
}
```

Este código se compila y se ejecuta sin problema, produciendo el siguiente código:

10

False

True

La cosa más importante de este ejemplo es que únicamente un objeto, Lemur, es creado y referenciado. La habilidad de una instancia de Lemur de ser tomada como instancia de la interfaz que implementa, HasTail, así como una instancia de una de sus superclases, Primate, es la naturaleza del polimorfismo.

Una vez el objeto ha sido asignado a un nuevo tipo de referencia, únicamente los métodos y variables disponibles para ese tipo referencia están disponibles para ser llamadas en el objecto sin un cast explícito. Por ejemplo, el siguiente fragmento de código no compilará:

```
HasTail hasTail = lemur;
System.out.println(hasTail.age); // NO COMPILA
Primate primate = lemur;
System.out.println(primate.isTailStriped()); // NO COMPILA
```

En este ejemplo, la referencia hasTail tiene acceso directo sólo a los métodos definidos con la interfaz HasTail; además, no sabe si la variable age es parte del objeto. Del mismo modo, la referencia primate solo tiene acceso a los métodos definidos en la clase Primate, y no tiene acceso directo al método isTailStriped().

7.4.1. Objecto vs. Referencia

En Java, todos los objetos son accedidos por referencia, así que como desarrollador nunca se tendrá acceso directo al objeto en sí. Conceptualmente, se debería considerar el objeto como la entidad que existe en memoria, almacenada por el entorno de ejecución de Java. Independientemente del tipo de la referencia que se tenga del objeto en memoria, el objeto en sí mismo no cambia. Por ejemplo, como todos los objetos heredan de java.lang.Object, pueden ser reasignados a java.lang.Object, como se muestra en el siguiente ejemplo:

```
Lemur lemur = new Lemur();
Object lemurAsObject = lemur;
```

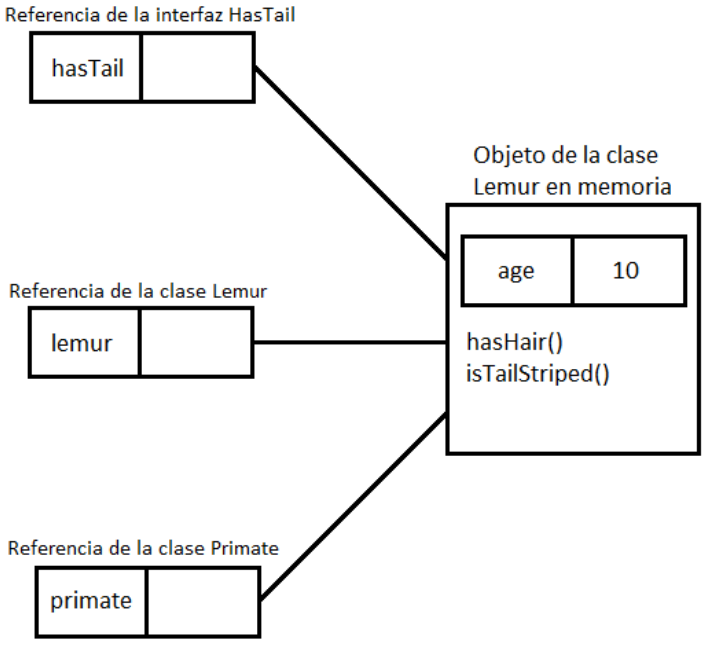
Aunque el objeto Lemur ha sido asignado a la referencia con un tipo diferente, el objeto en sí no ha cambiado y sigue existiendo como un objeto Lemur en memoria. Lo que ha cambiado es la habilidad para acceder métodos dentro la clase Lemur con la referencia lemurAsObject. Sin un cast explícito a Lemur, como se verá en la siguiente sección, no se tendrá acceso a las propiedades del objeto Lemur.

Se puede resumir este principio con las dos reglas siguientes:

- El tipo del objeto determina que propiedades existen dentro del objeto en memoria.
- El tipo de la referencia al objeto determina qué métodos y variables son accesibles por el programa Java.

Por lo tanto, cambiar la referencia de un objeto a un nuevo tipo de referencia puede proporcionar acceso a nuevas propiedades del objeto, pero estas propiedades ya existían antes de que la referencia cambiase.

Se ilustrará esta propiedad usando el ejemplo anterior en la figura que aparece abajo. Como se puede ver en la figura, el mismo objeto existe en memoria independientemente de que referencia está apuntándole. Dependiendo del tipo de referencia, se puede tener acceso solo a ciertos métodos. Por ejemplo, la referencia hasTail tiene acceso al método isTailStriped() pero no tiene acceso a la variable age definida en la clase Lemur. Como se aprenderá en la siguiente sección, es posible reclamar acceso a la variable age haciendo un casting explícito en la referencia hasTail a una referencia de tipo Lemur.



7.4.2. Hacer casting a objetos

En el ejemplo previo, se ha creado una única instancia del objeto Lemur, a la cual se ha accedido a través de una referencia de superclase y una referencia de interfaz. Una vez se ha cambiado el tipo de referencia, se pierde acceso a métodos más específicos definidos en la subclase que todavía existe dentro del objeto. Se puede reclamar estas referencias haciendo un casting al objeto a la subclase específica de la que procede:

```
Primate primate = lemur;
Lemur lemur2 = primate; // NO COMPILA
Lemur lemur3 = (Lemur)primate;
System.out.println(lemur3.age);
```

En este ejemplo, en primer lugar se intenta convertir la referencia primate a la referencia lemur, lemur2, sin un casting explícito. El resultado es que el código no compila. En el segundo ejemplo, sin embargo, se hace el casting explícito a la subclase del objeto Primate y se gana acceso a todos los métodos disponibles en la clase Lemur.

A continuación se tienen unas reglas básicas para tener en mente cuando hacemos casting a variables:

1. Hacer casting a un objeto desde una subclase a una superclase no requiere un casting explícito.
2. Hacer casting a un objeto desde una superclase a una subclase requiere un casting explícito.
3. El compilador no va a permtir casting a tipos no relacionados.
4. Incluso cuando el código compila sin problemas, una excepción puede ser lanzada en tiempo de ejecución si el objeto al que se le está haciendo el casting no es en ese momento una instancia de esa clase.

La tercera regla es importante. Por ejemplo, se puede hacer casting una referencia de Primate a una referencia de Lemur, porque Lemur es una subclase de Primate y por lo tanto relacionada con ella.

Ver este ejemplo:

```
public class Bird {}
public class Fish {
  public static void main(String[] args) {
    Fish fish = new Fish();
    Bird bird = (Bird)fish; // NO COMPILA
  }
}
```

En este ejemplo, las clases Fish y Bird no están relacionadas a través de ninguna jerarquía de clases que el compilador sea consciente; por lo tanto, el código no compila.

Hacer casting tiene sus limitaciones. Aunque dos clases compartan una jerarquía relacionada, esto no significa que una instancia de una clase pueda automáticamente ser objetivo de un cast de otra clase. Aquí se ve un ejemplo:

```
public class Rodent {
}
public class Capybara extends Rodent {
```

```
  public static void main(String[] args) {
    Rodent rodent = new Rodent();
    Capybara capybara = (Capybara)rodent; // Throws ClassCastException at runtime
  }
}
```

Este código crea una instancia de Rodent y entonces intenta hacer casting a una instancia de la clase Capybara. Aunque este código compilará sin problemas, lanzará una ClassCastException en tiempo de ejecución debido a que el objeto referenciado no es una instancia de la clase Capybara. Lo que hay que tener en mente en este ejemplo es que el objeto que ha sido creado no está relacionado con la clase Capybara de ninguna manera.

Hay que tener en mente que el operador instanceof puede ser utilizado para comprobar si un objeto pertenece a una clase en particular y para prevenir ClassCastExceptions en tiempo de ejecución. A diferencia del ejemplo previo, el siguiente fragmento de código no lanza ninguna excepción en tiempo de ejecución y efectúa el casting únicamente si el operador instanceof devuelve true.

```
if(rodent instanceof Capybara) {
  Capybara capybara = (Capybara)rodent;
}
```

7.4.3. Métodos virtuales

La característica más importante del polimorfismo, y una de las razones primarias por las que se tiene una estructura de clases, es soportar métodos virtuales. Un método virtual es un método en el que la implementación específica no está determinada hasta el tiempo de ejecución. De hecho, todos los métodos no finales, no estáticos y no privados de Java son considerados métodos virtuales, por el hecho de que cualquiera de ellos puede ser sobrescrito en tiempo de ejecución. Lo que hace especial a un método virtual en Java es que, si se llama a un método de un objeto que reescribe a otro método, se recibe el método reescrito, incluso si la llamada al método está en una referencia a un objeto padre o clase padre.

Se ilustra este principio en el siguiente ejemplo:

```
public class Bird {
  public String getName() {
    return "Unknown";
  }
  public void displayInformation() {
    System.out.println("The bird name is: "+getName());
  }
}
public class Peacock extends Bird {
```

```
  public String getName() {
    return "Peacock";
  }
  public static void main(String[] args) {
    Bird bird = new Peacock();
    bird.displayInformation();
  }
}
```

Este código compila y se ejecuta sin problemas y produce la siguiente salida:

The bird name is: Peacock

Como se vio en ejemplos similares en secciones anteriores, el método getName(), es reescrito en la clase hijo Peacock. Con mayor importancia, el valor del método getName() en tiempo de ejecución en el método displayInformation() es sustituido por el valor de la implementación de la subclase Peacock.

En otras palabras, si bien la clase padre Bird define su propia versión de getName() y no sabe nada de la clase Peacock durante el tiempo de compilación, en tiempo de ejecución la instancia utiliza la versión reescrita del método, definido en la instancia del objeto.

Se enfatiza este punto usando una referencia a la clase Bird en el método main(), aunque el resultado habría sido el mismo si se hubiera usado una referencia a Peacock.

Ahora se sabe el verdadero propósito de reescribir un método y como se relaciona con el polimorfismo. La naturaleza del polimorfismo es que un objeto puede adquirir diferentes formas. Combinando el conocimiento del polimorfismo con la reescritura de métodos, se puede ver que los objetos pueden ser interpretados de muchas maneras diferentes en tiempo de ejecución, especialmente en métodos definidos en una superclase de los objetos.

7.4.4. Parámetros polimorfos

Una de las aplicaciones más útiles del polimorfismo es la habilidad de pasar instancias de una subclase o interfaz a un método. Por ejemplo, se puede definir un método que tome una instancia de una interfaz como parámetro. De esta forma, cualquier clase que implemente la interfaz, podrá ser pasada al método. Desde que se realiza el casting de un subtipo a un supertipo, no es necesario un cast explícito. Se refiere a esta propiedad como parámetros polimórficos de un método y se demuestra en el siguiente ejemplo:

```
public class Reptile {
  public String getName() {
    return "Reptile";
  }
}
public class Alligator extends Reptile {
  public String getName() {
    return "Alligator";
```

```
  }
}
public class Crocodile extends Reptile {
  public String getName() {
    return "Crocodile";
  }
}
public class ZooWorker {
  public static void feed(Reptile reptile) {
    System.out.println("Feeding: "+reptile.getName());
  }
  public static void main(String[] args) {
    feed(new Alligator());
    feed(new Crocodile());
    feed(new Reptile());
  }
}
```

Este código compila y se ejecuta sin ningún problema. Dará el siguiente resultado:

Feeding: Alligator

Feeding: Crocodile

Feeding: Reptile

Se va a centrar en el método feed(Reptile reptile) de este ejemplo. Como se puede ver, este método acepta instancias de Alligator y Crocodile sin problemas, ya que ambas son subclases de la clase Reptile. También permite aceptar la clase Reptile. Si se hubiera intentado pasar una clase no relacionada, como las clases definidas anteriormente Rodent o Capybara o una superclase como java.lang.Object, al método feed(), el código no hubiera compilado.

Parámetros Polimórficos y reusabilidad del código

Si se define un método que será accesible desde fuera de la clase actual, o bien desde subclases de la clase actual o objectos desde fuera de la clase actual, se considera una buena práctica de codificación el usar un tipo de superclase o de interfaz para los datos de entrada, en la medida de lo posible.

Como se puede recordar del capítulo 5 "API de Java", el tipo java.util.List es una interfaz, no una clase. Aunque hay muchas clases que implementan java.util.List, como java.util.ArrayList y java.util.Vector, cuando se pasa una List existente, normalmente no se está interesado en la subclase particular del List. De este modo, un método que pasa un List debería usar la interfaz java.util.List como tipo de parámetro polimórfico, en lugar de un tipo de una clase que implemente List, ya que el código será reutilizado por otros tipos de listas.

Por ejemplo, es común ver código, como el siguiente, que usa el tipo de referencia de la interfaz en lugar del tipo de la clase para una mayor reutilización.

java.util.List list = new java.util.ArrayList();

7.4.5. Polimorfismo y reescritura de métodos

Se finalizará este capítulo volviendo a las tres reglas para sobrescritura de métodos para demostrar como el polimorfismo requiere que sean incluidas como parte de la especificación de Java. Verá como sin dichas reglas implantadas, es fácil construir un ejemplo con polimorfismo en Java.

La primera regla es que un método reescrito debe ser accesible al menos de la misma manera que el método al que reescribe. Se asume que esta regla no es necesaria y se considera el siguiente ejemplo:

```
public class Animal {
  public String getName() {
    return "Animal";
  }
}
public class Gorilla extends Animal {
  protected String getName() { // DOES NOT COMPILE
    return "Gorilla";
  }
}
public class ZooKeeper {
  public static void main(String[] args) {
    Animal animal = new Gorilla();
    System.out.println(animal.getName());
  }
}
```

A propósito de esta discusión, se ignorará el hecho de que la implementación de getName() en la clase Gorilla no compila porque es menos accesible que la versión que se está reescribiendo en la clase Animal.

Como se puede ver, este ejemplo crea un problema de ambigüedad en la clase Zookeeper. La referencia animal.getName() está permitida porque el método es público en la clase Animal, pero debido al polimorfismo, el objeto Gorilla en si ha sido reescrito por una versión menos accesible, no disponible para la clase ZooKeeper. Esto crea una contradicción en la que el compilador no debería permitir acceder a este método, pero debido a que es referenciado como una instancia de Animal, está permitido. Además, Java elimina esta contradicción, impidiendo que un método sea reescrito por una versión menos accesible del método.

Del mismo modo, una subclase no puede declarar un método reescrito con una nueva excepción o una excepción más genérica que en la superclase, ya que el método puede ser accedido usando una referencia a la superclase. Por ejemplo, si la instancia de la subclase es pasada a un método usando una referencia a superclase, entonces el método en cuestión no sabría nada sobre ninguna nueva excepción "obligatoria" que exista en métodos para este objeto, potencialmente llevando a código compilado con excepciones "obligatorias" no capturadas. Así mismo, el compilador Java no permite métodos reescritos con nuevas excepciones o excepciones genéricas.

Finalmente, los métodos reescritos deben usar tipos de retorno covariantes por las mismas razones que se acaban de discutir. Si un objeto es "casteado" a una referencia de una superclase y el método reescrito es llamado, el tipo de retorno debe ser compatible con el tipo de retorno del método padre. Si el tipo de

retorno en la hijo es muy genérico, resultará en una inherente excepción de cast cuando sea accedido a través de una referencia de la superclase.

Por ejemplo, si el tipo de retorno de un método es Double en la clase padre y es reescrito en una subclase con un método que devuelve Number, una superclase de Double, entonces el método de la subclase podría devolver cualquier Number válido, incluído Integer, otra subclase de Number. Si se está usando el objeto con una referencia a la superclase, esto significa que un Integer podría ser devuelto, cuando se esperaba un Double. Como Integer no es una subclase de Double, esto llevaría a una excepción implícita de cast tan pronto como el valor sea referenciado. Java soluciona este problema permitiendo solamente tipos de retorno covariantes para métodos reescritos.

7.5. Resumen

En este capítulo, se ha visto que:

Introducción a la herencia de clases

- La estructura básica que se presentó en el Capítulo 6 se ha expandido introduciendo la noción de herencia.

- Las clases de Java siguen un nivel múltiple de herencia única en la que cada clase tiene exactamente una clase padre directa, con todas las clases, eventualmente, heredando de java.lang.Object.

- Heredar de una clase da acceso a todos sus métodos public y protected de la clase, pero se tiene que seguir reglas especiales para constructores y métodos sobrescritos o el código no compilará.

Creando clases abstractas

- Una clase abstracta es una clase que está marcada con la palabra clave abstract y no puede ser instanciada.

- Un método abstracto es un método marcado con la palabra clave abstract y que está definido en una clase abstracta, para el cuál no hay una implementación en la clase que ha sido declarado.

- Una clase abstracta puede incluir métodos y variables que no sean abstractos

Implementando interfaces

- Una interfaz es un tipo de datos abstracto que define una lista de métodos abstractos que cualquier clase que implemente a la interfaz debe proporcionar.

- Una interfaz se define con la palabra clave interface y la clase implementa la interfaz mediante la palabra clave implements cuando se define la clase.

Entendiendo el polimorfismo

- La diferencia entre métodos sobrecargados, sobrescritos y ocultos, especialmente en términos de polimorfismo.

- Se ha introducido la noción de ocultar variables, aunque se desaconseja su uso, ya que lleva a confusión y dificulta el mantenimiento del código.

8. Excepciones

8.1. Entendiendo las excepciones

Un programa puede fallar por cualquier razón. Aquí hay algunas posibilidades:

- El código intenta conectarse a una página web, pero la conexión a Internet ha caído.
- Existe un fallo en el código e intento de acceso a un elemento de un array que no es válido.
- Un método que llama a otro con un valor que el método no soporta.

Como se puede observar, algunos son errores en el código. Otros están completamente fuera del control del programador. El programa no puede hacer nada si la conexión a Internet falla. Lo que se puede hacer es analizar y tratar la situación.

Para comenzar se van a analizar la función de las excepciones. A continuación se van a tratar los diferentes tipos de excepciones y se indica una explicación sobre cómo lanzar una excepción en Java.

8.1.1. La función de las excepciones

Una excepción es la forma de Java de decir, "me doy por vencido. No sé qué hacer ahora. Tú te encargas". Cuando se escribe un método, se puede lidiar con una excepción o que llame al código de error.

Como ejemplo, pensar en Java como un chico que visita el zoo. El *camino feliz* es cuando nada va mal. El chico continúa para ver a los animales hasta que el programa termina. Nada ha ido mal y no ha habido excepciones que manejar.

La hermana de este chico no vive la experiencia del *camino feliz*. Con toda la emoción, se tropieza y cae. Por suerte, no es una mala caída. La chica se levanta y continúa para mirar más animales. Ha manejado el problema por sí misma. Desafortunadamente, vuelve a caer más tarde y empieza a llorar. Esta vez, está diciendo que necesita ayuda (llorando). La historia termina bien. Su padre le consuela y le da un abrazo. Vuelven a ver más animales y disfruta del resto del día.

Estas son los dos los enfoques que Java usa para tratar las excepciones. Un método puede manejar una excepción por sí mismo o hacer que el que llama a este método sea el que se encargue de ésta. Se ha visto las dos en el viaje al zoo.

Se ha visto una excepción en el Capítulo 3, "Java Building Blocks", con un ejemplo muy simple del Zoo. Se escribió una clase que imprimía el nombre del zoo:

```
1: public class Zoo {
2:     public static void main(String[] args) {
3:         System.out.println(args[0]);
4:         System.out.println(args[1]);
5: } }
```

Entonces se intentó llamarla sin argumentos suficientes:

$ javac Zoo.java

$ java Zoo Zoo

En la línea 4, Java se dio cuenta que solo hay un elemento en el array y el índice 1 no está permitido. Java dice, "me doy por vencido" y nos muestra la excepción:

```
ZooException in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1 at mainmethod.Zo
o.main(Zoo.java:7)
```

Las excepciones pueden ocurrir y ocurren todo el tiempo, hasta en programas robustos. En elejemplo, que los niños se caigan son cosas de la vida. Cuando se escriben programas más avanzados, se necesitará tratar con fallos que pueden aparecer al acceder a ficheros, redes y otros servicios. Por ejemplo, un programa puede intentar acceder a una posición no válida de un array. La clave está en recordar que las excepciones alteran el curso del programa.

Códigos de retorno vs Excepciones

Las excepciones se usan cuando "algo va mal". Sin embargo, la palabra "mal" es subjetiva. El siguiente código devuelve -1 en lugar de lanzar una excepción si no encuentra una coincidencia:

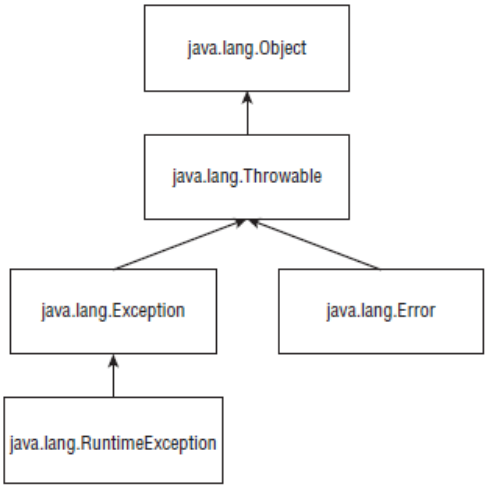
```
public int indexOf(String[] names, String name) {
  for (int i = 0; i < names.length; i++) {
    if (names[i].equals(name)) { return i; }
  }
  return -1;
}
```

Esta aproximación es común cuando se escribe un método que hace una búsqueda. Por ejemplo, se tiene que buscar el nombre Josefina en el array. Es posible que Josefina no esté en el array. Cuando esto pasa, un valor especial es el que se devuelve. Una excepción debería reservarse para condiciones excepcionales como que los nombres sean null.

En general, se intenta evitar códigos que devuelvan un valor. Este tipo de métodos se suelen usar en búsquedas, así que otros programadores los esperan. En otros métodos, se sorprenderá al que llamen al método devolviendo un valor especial. Una excepción fuerza al programa a tratarla o a terminar con una excepción no tratadas, mientras que los códigos de retorno pueden ser ignorados accidentalmente y causar problemas en el programa más tarde. Una excepción es como gritar, "¡Trata conmigo!".

8.1.2. Entendiendo los tipos de excepción

Cómo se ha explicado, una excepción es un evento que altera el curso del programa. Java tiene una superclase Throwable para todos los objetos que representan todos estos eventos. No todos tienen la palabra excepción en su nombre de clase, puede ser confuso. La figura muestra las subclases clave de Throwable.



Error significa que algo ha ido tan mal que el programa no debería intentar recuperarse de esto. Por ejemplo, la unidad de disco ha "desaparecido". Este tipo de condiciones no se deben tratar.

Una excepción runtime se define con la clase RuntimeExcepcion y sus subclases. Las excepciones Runtime no suelen esperarse, pero no son necesariamente fatales. Por ejemplo, acceder a un índice no válido de un array. Las excepciones en tiempo de ejecución también se conocen como *unchecked exceptions* (excepciones no obligatorias).

a. Runtime vs el momento que el programa se ejecuta

Una excepción runtime (unchecked, no obligatoria) es un tipo específico de excepción. Todas las excepciones ocurren en el momento que el programa se ejecuta. (La alternativa es en tiempo de compilación, que sería un error de compilación). La gente no se refiere como excepciones en tiempo de ejecución porque sería muy fácil confundirse con runtime. Cuando se vea runtime, significa unchecked (no obligatoria).

Una *checked exception* incluye Exception y todas las subclases que no extienden de RuntimeException. Excepciones checked tienden a ser más previsibles, por ejemplo, intentar leer un fichero que no existe.

¿Excepciones checked? ¿Qué se está comprobando? Java tiene una regla llamada *la regla de encargarse o declarar*. Para las excepciones checked, Java obliga o a encargarse de esta o declararla en la definición del método.

Por ejemplo, este método declara que se puede lanzar una excepción:

```
void fall() throws Exception {
  throw new Exception();
}
```

Hay que tener en cuenta que se están utilizando dos palabras clave en el método: throw le indica a Java que se quiere lanzar una nueva Exception. throws declara que el método puede lanzar una Exception. También puede no lanzarla. Se verán las palabras clave throw y thrown más adelante en el capítulo.

Debido a que las excepciones checked tienden a ser previsibles, Java fuerza a que el programa haga algo para mostrar la excepción que puede aparecer. Quizás se manejó en el método. O quizás el método declara que no puede manejar la excepción y otro debería encargarse.

Un ejemplo de una excepción runtime es un NullPointerException que ocurre cuando intentas llamar a un miembro cuya referencia es null. Esto puede ocurrir en cualquier método. Si tuvieras que declarar excepciones runtime en todo, cada método tendría esta complicación.

b. Excepciones Checked vs. Unchecked (Runtime)

Anteriormente los desarrolladores usaban excepciones checked más a menudo de lo que lo hacen ahora. Según Oracle, están destinadas para problemas de los que un programador "se espera que pueda recuperarse". Los desarrolladores empezaron a escribir código donde una cadena de métodos seguían lanzando la misma excepción y nadie realmente lo manejaba. Algunas librerías empezaron a usar excepciones runtime para problemas de los que se espera que un programador puede recuperarse. Muchos programadores pueden debatir sobre que aproximación es mejor.

8.1.3. Lanzando una Exception

Cualquier código de java puede lanzar una excepción; incluido el código que se escribe. Se verán dos tipos de código que pueden lanzar una excepción. El primer código es código que está mal. Por ejemplo:

```
String[] animals = new String[0];
System.out.println(animals[0]);
```

El código lanza una excepción de tipo ArrayIndexOutOfBoundsException.

La segunda forma de que el código lance una excepción es pedirle explícitamente a Java que lance una. Java permite escribir declaraciones como estas:

```
throw new Exception();
throw new Exception("Ow! I fell.");
throw new RuntimeException();
throw new RuntimeException("Ow! I fell.");

throw new Exception(e);
```

La palabra clave throw le dice a Java que se quiere que alguna otra parte del código trate con la excepción. De la misma forma que la niña lloraba para que su padre le ayudase. Alguien necesita darse cuenta sobre qué hacer con la excepción.

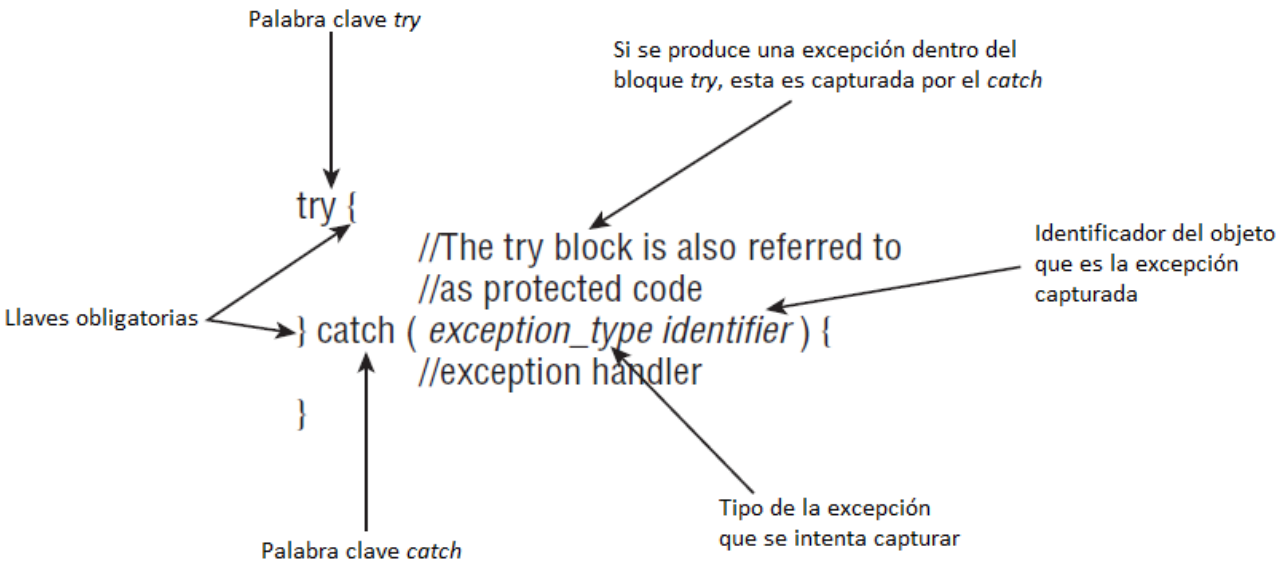
Cuando se crea una excepción normalmente se pasa un parámetro como String con un mensaje o no se le pasa nada para que use los valores por defecto. También es posible pasar como parámetro una excepción (e), lo que permite no perder la información del problema original. Se dice "normalmente" porque es una costumbre. Se puede crear una clase de excepción que no tiene un constructor que acepte un mensaje. Los primeros dos ejemplos crean un nuevo objeto de tipo Exception y lo lanzan. Los últimos dos muestran lo mismo sin importar que tipo de excepción lanzan.

Estas reglas son muy importantes.

Tipo	Como reconocerla	¿El programa puede manejarla?	¿El programa tiene que manejarla o declararla?
Runtime exception	Subclase de RuntimeException	Si	No
Checked exception	Subclase de Exception pero no sucblase de RuntimeException	Si	Si
Error	Subclase de Error	No	No

8.2. Usando la sentencia try

Una vez sabido lo que son las excepciones, se explicará como tratarlas. Java usa la sentencia try para separar la parte del programa que puede lanzar una excepción de la parte del programa que se encarga de la excepción. A continuación, se muestra la sintaxis de la sentencia *try*.



El código en el bloque *try* funciona con normalidad. Si cualquiera de las sentencias lanza una excepción, ésta puede ser capturada por el tipo de excepción especificado en el bloque *catch*, el bloque *try* para de funcionar y la ejecución pasa a la sentencia *catch*. Si ninguna de las sentencias en el bloque *try* lanza una excepción que pueda ser capturada, la cláusula *catch* no se ejecuta.

Se utilizan indistintamente las palabra "bloque" y "cláusula" . "Bloque" es correcto porque hay llaves presentes. "Cláusula" es correcto porque es parte de la declaración *try*.

No hay muchas reglas de sintaxis aquí. Las llaves son requeridas para la sentencia *try* y los bloques *catch*.

En el ejemplo, la niña se levanta por sí misma la primera vez que se cae. Aquí está lo que parece:

```
3: void explore() {
4:   try {
5:     fall();
6:     System.out.println("never get here");
7:   } catch (RuntimeException e) {
8:     getUp();
9:   }
10:  seeAnimals();
11: }
12: void fall() { throw new RuntimeException(); }
```

Primero, en la línea 5 llama al método *fall().* La línea 12 lanza una excepción. Esto significa que Java salta derecho al bloque *catch* omitiendo la línea 6. La niña se levanta (*gets up*) en la línea 8. Ahora, la declaración *try* está terminada y la ejecución continua con normalidad en la línea 10.

A continuación se muestra una declaración *try* inválida. ¿Se distingue lo que está mal aquí?

```
try // DOES NOT COMPILE
  fall();
catch (Exception e)
  System.out.println("get up");
```

El problema está en la falta de llaves. Debería verse así:

```
try {
  fall();
} catch (Exception e) {
  System.out.println("get up");
}
```

Las declaraciones *try* son como métodos en los que las llaves son requeridas, incluso si hay solo una declaración dentro del código del bloque. Las declaraciones *if* y los bucles son especiales respecto a esto, ellos permiten omitir las llaves.

¿Qué hay de esto?

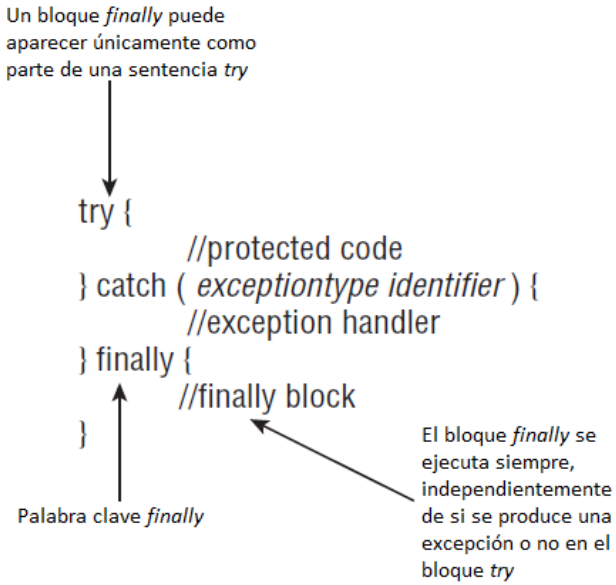
```
try { // DOES NOT COMPILE
  fall();
}
```

Este código no compila porque el bloque *try* no tiene nada después de él. Es importante saber que el punto de la sentencia *try* es para algo que pasa si una excepción es lanzada. Sin otra cláusula, la declaración *try* está sola.

Una vez explicado lo básico, se añaden más características a las excepciones. Las siguientes secciones muestran como añadir una cláusula *finally* a la declaración *try* y capturar diferentes tipos de excepciones y describir que pasa si una excepción es lanzada en *catch* o en *finally.*

8.2.1. Añadiendo un bloque finally

La declaración try también permite ejecutar código al final con la cláusula *finally* a pesar de si se lanza una excepción. La figura siguiente muestra la sintaxis de la declaración *try* con esta funcionalidad extra.



Hay dos rutas a través del código con un *catch* y un *finally*. Si una excepción es lanzada, el bloque *finally* es ejecutado después del bloque *catch*. Si no hay excepción lanzada, el bloque *finally* es ejecutado después de que el bloque *try* se complete.

Se va a volver al ejemplo de la niña, esta vez con *finally*:

```
12: void explore() {
13:   try {
14:     seeAnimals();
15:     fall();
16:   } catch (Exception e) {
17:     getHugFromDaddy();
```

```
18:    } finally {
19:      seeMoreAnimals();
20:    }
21:    goHome();
22: }
```

La niña cae (*falls*) en la línea 15. Si ella se levanta por sí misma, el código continua con el bloque *finally* y ejecuta la línea 19. Entonces la sentencia try ha finalizado y el código procede a la línea 21. Si la niña no se levanta por sí sola, lanza una excepción. El bloque catch se ejecuta y recibe un abrazo *(getHugFromDaddy())* en la línea 17. Entonces la sentencia try finaliza y el código procede a la línea 21. De cualquier manera, el final es el mismo. El bloque *finally* es ejecutado y la declaración *try* terminada.

¿Se ve por qué lo siguiente compila o no?

```
25: try { // DOES NOT COMPILE
26:   fall();
27: } finally {
28:   System.out.println("all better");
29: } catch (Exception e) {
30:   System.out.println("get up");
31: }
32:
33: try { // DOES NOT COMPILE
34:   fall();
35: }
36:
37: try {
38:   fall();
39: } finally {
40:   System.out.println("all better");
41: }
```

El primer ejemplo (líneas 25-31) no compila porque los bloques *catch* y *finally* están en el orden incorrecto. El segundo ejemplo (líneas 33-35) no compila porque debe haber bloque *catch* y *finally*. El tercer ejemplo (líneas 37-41) está bien. La sentencia *catch* no es obligatoria si existe *finally*.

Finally es comúnmente usado para cerrar recursos como archivos o bases de datos-. Hay que preguntarse por qué devuelve este código:

```
String s = "";
try {
  s += "t";
} catch(Exception e) {
  s += "c";
} finally {
```

```
  s += "f";
}
s += "a";
System.out.print(s);
```

La respuesta es "tfa". El bloque *try* es ejecutado. Hasta que no se lanza ninguna excepción, Java va directo al bloque *finally*. Entonces el código después de la sentencia try se ejecuta.

System.exit

Hay una excepción a la regla "el bloque finally siempre se ejecuta después del bloque catch": Java define un método que se le llama System.exit(0);. El parámetro integer es el error de código que es devuelto. System.exit le dice a Java, "Para. Finaliza el programa ahora mismo. No pases. No cobres 200$." Cuando se llama a System.exit en el bloque try o catch, finally no se ejecuta.

8.2.2. Capturando varios tipos de excepciones

Hasta aquí, únicamente se ha capturado un tipo de excepción. Se va a analizar qué ocurre cuando varios tipos de excepciones se lanzan desde el mismo método.

Hay que ser capaz de reconocer si la excepción es *checked* o *unchecked*. A continuación, se necesita determinar si alguna de las excepciones son subclases de las otras.

```
class AnimalsOutForAWalk extends RuntimeException { }
class ExhibitClosed extends RuntimeException { }
class ExhibitClosedForLunch extends ExhibitClosed { }
```

En este ejemplo, hay tres excepciones predefinidas. Todas son excepciones unchecked porque directa o indirectamente extienden a *RuntimeException*. A continuación, se capturan ambos tipos de excepciones y se manjean imprimiendo el mensaje apropiado:

```
public void visitPorcupine() {
  try {
    seeAnimal();
  } catch (AnimalsOutForAWalk e) { // first catch block
    System.out.print("try back later");
  } catch (ExhibitClosed e) { // second catch block
     System.out.print("not today");
  }
}
```

Hay tres posibilidades para cuando ejecute el código. Si *seeAnimal*() no lanza una excepción, no se imprime nada. Si el animal sale a dar un paseo (*AnimalsOutForAWalk*), únicamente se ejecuta el primer catch. Si la exhibición está cerrada (*ExhibitClosed*), sólo se ejecuta el segundo catch.

Existe una regla para el orden de los bloques catch. Java los trata en el orden que aparecen. Si es imposible ejecutarse para uno de los bloques catch, ocurre un error de compilación sobre *unreachable code*. Eso sucede cuando una superclase es capturada antes que una subclase. Recordar, Se debe poner atención a cualquier excepción de una subclase.

En el ejemplo del puercoespín, el orden de los bloques catch puede ser reservado porque las excepciones no heredan de las otras. Es cierto que un puercoespín puede ser cogido para caminar con una correa.

El siguiente ejemplo muestra tipos de excepciones que heredan de otras:

```
public void visitMonkeys() {
  try {
    seeAnimal();
  } catch (ExhibitClosedForLunch e) {// subclass exception
    System.out.print("try back later");
  } catch (ExhibitClosed e) {// superclass exception
    System.out.print("not today");
  }
}
```

Si la excepción más específica ExhibitClosedForLunch es lanzada, el primer bloque catch se ejecuta. Si no, Java comprueba si la excepción de la superclase ExhibitClosed es lanzada y capturada. Esta vez, el orden de los bloques catch importa. Al revés no funciona.

```
public void visitMonkeys() {
  try {
    seeAnimal();
  } catch (ExhibitClosed e) {
    System.out.print("not today");
  } catch (ExhibitClosedForLunch e) {// DOES NOT COMPILE
    System.out.print("try back later");
  }
}
```

Esta vez, si la excepción más específica ExhibitCloseForLunch es lanzada, el bloque catch para ExhibitClosed se ejecuta- lo que significa que no hay forma para el segundo bloque catch para ser ejecutado alguna vez. Java dice que hay un *unreachable catch block.*

Se intenta una vez más. ¿Se observa por qué no compila este código?

```
public void visitSnakes() {
  try {
    seeAnimal();
  } catch (RuntimeException e) {
    System.out.print("runtime exception");
  } catch (ExhibitClosed e) {// DOES NOT COMPILE
```

```
    System.out.print("not today");
  } catch (Exception e) {
    System.out.print("exception");
  }
}
```

Es el mismo problema. ExhibitClosed es un RuntimeException. Si es lanzado, el primer bloque catch se ocupa de ello, asegurándose que no hay manera de obtener el segundo bloque catch. Para revisar la captura multiple de excepciones, recordar que por lo menos un bloque catch se ejecutará y será el primer bloque catch el que se encargará de ello.

8.2.3. Lanzando una segunda Exception

Hasta ahora, se ha limitado a una sentencia try en cada ejemplo. Sin embargo, un bloque catch o finally puede tener cualquier código de Java válido; incluyendo otra sentencia try.

El siguiente código intenta leer un archivo:

```
16: public static void main(String[] args) {
17:   FileReader reader = null;
18:   try {
19:     reader = read();
20:   } catch (IOException e) {
21:     try {
22:       if(reader != null) reader.close();
23:     } catch (IOException inner) {
24:     }
25:   }
26: }
27: private static FileReader read() throws IOException {
28:   // EL CÓDIGO VA AQUÍ
29: }
```

El caso más fácil es si en la línea 28 no lanza una excepción. Entonces el bloque entero del catch en la línea 20-25 se salta. Ahora, se considera que en la línea 28 lanza un NullPointerException. Esto no es una IOException, así que la línea del bloque catch de las líneas 20-25 seguirán siendo saltadas.

Si en la línea 28 lanza una IOException, el *catch* de las líneas 20-25 se ejecuta. La línea 22 intenta cerrar el *reader*. Si funciona bien, el código se completa y el método main() termina correctamente. Si el método close() lanza una excepción, Java busca más bloques *catch*. No hay más en este caso, así que el método main lanza una nueva excepción. De todos modos, la excepción de la línea 28 se encarga de ello. Una excepción diferente puede ser lanzada, pero la de la línea 28 está terminada.

Este ejemplo muestra que sólo la última excepción lanzada es la que importa.

```
26: try {
27:   throw new RuntimeException();
28: } catch (RuntimeException e) {
29:   throw new RuntimeException();
30: } finally {
31:   throw new Exception();
32: }
```

La línea 27 lanza una excepción que se recoge en la línea 28. El bloque catch lanza otra excepción en la línea 29. Si no hubiera un bloque finally, la excepción de la línea 29 sería lanzada. Sin embargo, el bloque finally va después del bloque try. Desde que el bloque finally lanza una excepción por sí mismo en la línea 31, esta se lanza. La excepción del bloque catch es olvida. Esto es porque a menudo se ve otro bloque try/catch dentro de un bloque finally; para asegurarse que no se oculta una excepción del bloque catch.

¿Qué devuelve este método?

```
30: public String exceptions() {
31:   String result = "";
32:   String v = null;
33:   try {
34:     try {
35:       result += "before";
36:       v.length();
37:       result += "after";
38:     } catch (NullPointerException e) {
39:       result += "catch";
40:       throw new RuntimeException();
41:     } finally {
42:       result += "finally";
43:       throw new Exception();
44:     }
45:   } catch (Exception e) {
46:     result += "done";
47:   }
48:   return result;
49: }
```

La respuesta correcta es *before catch finally done*. Todo es normal hasta la línea 35, cuando se añade "before". La línea 36 lanza un NullPointerException. La línea 37 se salta ya que Java va directamente al bloque *catch*. La línea 38 maneja la excepción y "catch" se añade en la línea 39. Ahora la línea 40 lanza un RuntimeException. El bloque *finally* se ejecuta después del *catch* sin que le importe que una excepción sea lanzada; añade "finally" a la variable *result*. En este punto se ha completado el try interno que va desde la línea 34 a la 44. El bloque *catch* de fuera ve que una excepción ha sido lanzada y lo maneja en la línea 45; añade "done" a la variable *result*.

8.3. Reconocimiento de los tipos comunes de excepciones

Se explicarán los ejemplos comunes de cada tipo. Hay que reconocer los tipos de una excepción, qué es y si es lanzado por la JVM o por un programador. Entonces se reconocerán. A continuación se muestran algunos ejemplos de código para esas excepciones:

8.3.1. Runtime Exceptions

Runtime exception extiende de *RuntimeException.* Estas excepciones no tienen que ser tratadas o declaradas. Estas excepciones pueden ser lanzadas por el programador o por la JVM. Las excepciones comunes de *runtime exceptions* incluyen lo siguiente:

ArithmeticException Es lanzada por la JVM cuando el código intenta dividirse entre cero.

ArrayIndexOutOfBoundsException Es lanzada por la JVM cuando el código usa un índice incorrecto para acceder al array.

ClassCastException Es lanzada por la JVM cuando se hace un intento para convertir una excepción a una subclase de que esto no es una instancia.

IllegalArgumentException Es lanzada por el programador para indicar que un método ha sido lanzado como un argumento ilegal o inadecuado.

NullPointerException Es lanzada por la JVM cuando hay una referencia nula donde requieren un objeto.

NumberFormatException Es lanzada por el programador cuando se hace un intento para convertir a String un tipo numérico pero el texto no tiene un formato apropiado.

a. ArithmeticException

El intento de dividir un entero por cero da un resultado indefinido. Cuando esto ocurre la JVM lanzará una *ArithmeticException:*

int answer = 11 / 0;

El resultado de ejecutar este código, devuelve la siguiente salida:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Java no explica detalladamente la palabra "divide". Esto es correcto, sin embargo, se sabe que el operador de división y que Java trata de decir que se ha dividido por cero.

El hilo "main" (principal) dice que el código fue llamado directamente o indirectamente desde el programa con un método principal. Después llega el nombre de la excepción, seguida por la información extra (si alguno) que va con la excepción.

b. ArrayIndexOutOfBoundsException

Como ya se sabe, los índices del array empiezan en 0 y terminan en la longitud del array menos 1. – esto quiere decir que el código lanzará una excepción ArrayIndexOutOfBoundsException:

```
int[] countsOfMoose = new int[3];
System.out.println(countsOfMoose[-1]);
```

Esto es un problema porque no hay ningún array con índice negativo. El resultado de ejecutar este código, devuelve la siguiente salida:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
```

Al menos Java dice que índice era invalido. ¿Se observa el error en el siguiente código?

```
int total = 0;
int[] countsOfMoose = new int[3];
for (int i = 0; i <= countsOfMoose.length; i++)
total += countsOfMoose[i];
```

El problema es que el bucle for, debería tener < en vez de <=. Sobre el final de la iteración del bucle, Java intenta llamar a countsOfMoose[3], que es invalido. El array incluye solo tres elementos, haciendo dos el mayor índice posible. La salida se parece a esto:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
```

c. ClassCastException

Java intenta proteger de las conversiones imposibles. Este código no compila porque el número entero no es una subclase de String:

```
String type = "moose";
Integer number = (Integer) type; // DOES NOT COMPILE
```

El código más complicado impide los intentos de Java por protegerle. Cuando la conversión falla en el tiempo de ejecución, Java lanzara una excepción ClassCastException:

```
String type = "moose";
Object obj = type;
Integer number = (Integer) obj;
```

El compilador ve una conversión de objeto (Object) a número entero (Integer). Esto podría estar bien. El compilador no comprende que hay un texto (String) en un objeto (Object). Cuando se ejecuta el código, se produce la siguiente salida:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Integer
```

Java indica que ambos tipos estuvieron implicados en el problema, es evidente que es incorrecto.

d. IllegalArgumentException

IllegalArgumentException es el camino para que su programa se proteja. Primero, se vio el siguiente método regulador en el que la clase cisne del capítulo 4, "Métodos y encapsulación".

```
6: public void setNumberEggs(int numberEggs) {// setter
7:   if (numberEggs >= 0) // guard condition
8:     this.numberEggs = numberEggs;
9:}
```

Este código funciona, pero realmente no se quiere ignorar la petición de la llamada cuando ellos dicen que un Cisne tiene – 2 huevos. Se le quiere decir al que llama al método que algo es incorrecto – preferiblemente de un modo muy obvio cuando la llamada al método no puede hacer caso, de modo que el programador pueda arreglar el problema. Las excepciones son un eficiente modo para hacer esto. Viendo el código final con una excepción es un gran recordatorio de que algo es erróneo.

```
public static void setNumberEggs(int numberEggs) {
  if (numberEggs < 0)
    throw new  IllegalArgumentException("# eggs must not be negative");
  this.numberEggs = numberEggs;
}
```

El programa lanza una excepción cuando no está satisfecho con el valor del parámetro. La salida se parece a la siguiente:

Exception in thread "main" java.lang.IllegalArgumentException: # eggs must not be negative

Evidentemente este es un problema que debe ser solucionado si el programador quiere el programa para hacer algo útil.

e. NullPointerException

Las variables de instancia y los métodos deben llamarse como una referencia no-nula. Si la referencia es nula, la JVM lanzará un NullPointerException. Normalmente es sutil, como este ejemplo, que comprueba si se recuerda la variable de instancia de referencias por defecto como nula.

```
String name;
public void printLength() throws NullPointerException {
  System.out.println(name.length());
}
```

Cuando se ejecuta el código, se produce la siguiente salida:

```
Exception in thread "main" java.lang.NullPointerException
```

f. NumberFormatException

Java proporciona métodos para convertir textos a números. Cuando estos son pasados como un valor invalido, ellos lanzan una excepción NumberFormatException. La idea es similar a la de la excepción IllegalArgumentException. Puesto que esto es un problema común, Java da una clase separada. De hecho, NumberFormatException es una subclase de IllegalArgumentException. Aquí hay un ejemplo que trata de convertir algo no numérico en un int:

```
Integer.parseInt("abc");
```

La salida se parece a esto:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
```

8.3.2. Excepciones controladas (Checked Exceptions)

Las excepciones controladas tienen *Exception* en su jerarquía, pero no *RuntimeException*. Ellas deben ser tratadas o declaradas. Ellas pueden ser lanzadas por el programador o por la JVM. Las excepciones comunes de tiempo de ejecución incluidas son las siguientes:

FileNotFoundException Lanzada programáticamente cuando el código trata de referirse a un archivo que no existe.

IOException Lanzada programáticamente cuando hay un problema de lectura o de escritura en el fichero.

8.3.3. Errors

Los errores extienden de la clase Error. Ellos son lanzados por la JVM y no deberán ser tratados o declarados. Los errores son escasos, los más comunes son:

ExceptionInInitializerError Lanzado por la JVM cuando un inicializador estático lanza una excepción y no lo trata.

StackOverflowError Lanzado por la JVM cuando un método se llama a si mismo muchas veces (esto es llamado como "*infinite recursion*" porque el método típico se llama así mismo sin acabar)

NoClassDefFoundError Lanzado por la JVM cuando una clase que usa el código está disponible al tiempo de compilar, pero no en el tiempo de ejecución.

a. ExceptionInInitializerError

Las ejecuciones de Java son de inicialización estática cuando se utiliza la clase por primera vez. Si una inicialización estática lanza una excepción, Java no puede empezar a usar la clase. Esto anuncia un rechazo por el lanzamiento de la excepción ExceptionInInitializerError. Este código muestra un ArrayIndexOutOfBounds en un inicializador estático:

```
static {
  int[] countsOfMoose = new int[3];
```

```
  int num = countsOfMoose[-1];
}
public static void main(String[] args) { }
```

Este código da información sobre dos excepciones:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: -1
```

Se consigue el ExceptionInInitializerError porque el error ocurrió en la inicialización estática. Aquella información única no es útil para solucionar el problema. Por lo tanto, Java también indica la causa original del problema, es decir, el ArrayIndexOutOfBoundsException que nosotros tenemos que solucionar.

La ExceptionInInitializerError es un error porque Java falla al cargar la clase entera. Este error le impide a Java continuar.

b. StackOverflowError

Cuando Java llama a los métodos, esto introduce parámetros y variables locales sobre la pila. Después hace esto muchas veces, la pila se queda sin el espacio y se desborda. Esto se llama como StackOverflowError. Casi siempre, este error ocurre cuando un método se llama a sí mismo.

```
public static void doNotCodeThis(int num) {
  doNotCodeThis(1);
}
```

La salida contiene esta línea:

```
Exception in thread "main" java.lang.StackOverflowError
```

Desde las llamadas del método a sí mismo, esto nunca se terminará. Finalmente, Java se queda sin el espacio en la pila y lanza el error. Esto es llamado recursión infinita. Es mejor que un bucle infinito porque al menos java cogerá y lanzará el error. Con un bucle infinito, Java unicamente usa toda su CPU hasta que pueda acabar con él.

c. NoClassDefFoundError

Únicamente hay que saber que esto es un error. NoClassDefFoundError ocurre cuando Java no encuentra la clase en el tiempo de ejecución.

8.4. Llamando a métodos que provocan excepciones

Cuando se llama a un método que produce excepciones, las reglas son las misma que sin un método. ¿Por qué no compila el siguiente?

```
class NoMoreCarrotsException extends Exception {}
public class Bunny {
  public static void main(String[] args) {
    eatCarrot();// DOES NOT COMPILE
  }
  private static void eatCarrot() throws NoMoreCarrotsException {}
}
```

El problema es que NoMoreCarrotsException es comprobada como una excepción. Todas las excepciones deben ser declaradas. El código debería compilar si se cambia la función main() a cualquiera de estas:

```
public static void main(String[] args) throws NoMoreCarrotsException {// declare exception
  eatCarrot();
}
public static void main(String[] args) {
  try {
    eatCarrot();
  } catch (NoMoreCarrotsException e ) {// handle exception
    System.out.print("sad rabbit");
  }
}
```

Se puede observar que eatCarrot() en realidad no inicia una excepción, sino que se declaró como que podía. Esto es suficiente para que el compilador necesite de una persona para manejar o declarar la excepción.

El compilador está todavía en busca de código inalcanzable. Declarar una excepción sin usar no se considera código inalcanzable. El método da la opción de cambiar la implementación para alcanzar la excepción en el futuro:

```
public void bad() {
  try {
    eatCarrot();
  } catch (NoMoreCarrotsException e ) {// DOES NOT COMPILE
     System.out.print("sad rabbit");
  }
}
public void good() throws NoMoreCarrotsException {
  eatCarrot();
}
private static void eatCarrot() { }
```

Java sabe que eatCarrot() { } no puede lanzar una excepción comprobada – que significa que no hay camino para bloquear bad() al llegar. En comparación good() es libre para declarar otras excepciones.

8.4.1. Subclases:

Ahora que se tiene una comprensión más profunda de las excepciones, se van a explicar más métodos con excepciones en la declaración del método. Cuando una clase reemplaza un método de una superclase o implementa un método de una interfaz, no está permitido agregar nuevas excepciones al método. Por ejemplo, en este código no está permitido:

```
class CanNotHopException extends Exception { }
class Hopper {
  public void hop() { }
}
class Bunny extends Hopper {
  public void hop() throws CanNotHopException { } // DOES NOT COMPILE
}
```

Java sabe que hop() no tiene permitido lanzar excepciones porque la superclase Hopper no está declarada. Imaginar lo que sucede si las subclases podrían agregar excepciones – se puede escribir el código que llame al método Hopper´s hop() y que no maneje ninguna excepción. Entonces si Bunny es usado en su lugar, el código no sabe manejar o declarar CanNotHopException.

Una subclase permite declarar menos excepciones que una superclase o interfaz. Esto es legal porque ya está declarado en la superclase.

```
class Hopper {
  public void hop() throws

  CanNotHopException { }
}
class Bunny extends Hopper {
  public void hop() { }
}
```

Una subclase sin declarar una excepción es similar a un método que lanza una excepción que en realidad nunca se lanza. Esto es perfectamente legal. De forma similar, una clase permite declarar a una subclase de un tipo de excepción. La idea es la misma. La superclase o interfaz ya se ha ocupado. Por ejemplo:

```
class Hopper {
  public void hop() throws Exception { }
}
class Bunny extends Hopper {
  public void hop() throws CanNotHopException { }
}
```

Bunny puede declarar lanzar excepción directamente o puede declarar lanzar un tipo concreto de excepción. Incluso podría declarar que no lance nada en absoluto. Esta regla es aplicada solo a comprobar excepciones. En el siguiente código es legal porque tiene un *‘‘runtime exception’’* en la versión subclase:

```
class Hopper {
  public void hop() { }
}
class Bunny extends Hopper {
  public void hop() throws IllegalStateException { }
}
```

La razón por la que está bien declarar nuevas excepciones de *''runtime''* en un método de subclase es que la declaración es redundante. Los métodos son libres de lanzar cualquier excepción de *''runtime''* que deseen sin ser mencionados en la declaración del método.

8.4.2. Imprimir una excepción:

Hay tres formas de imprimir una excepción. Se puede permitir que Java lo imprima, únicamente un mensaje o que imprima de donde proviene. Este ejemplo muestra los tres enfoques o tipos:

```
5: public static void main(String[] args) {
6:   try {
7:      hop();
8:   } catch (Exception e) {
9:      System.out.println(e);
10:     System.out.println(e.getMessage());
11:     e.printStackTrace();
12:  }
13: }
14: private static void hop() {
15:   throw new RuntimeException("cannot hop");
16: }
```

El resultado del codigo anterior es el siguiente:

```
java.lang.RuntimeException: cannot hop
cannot hop
java.lang.RuntimeException: cannot hop
at trycatch.Handling.hop(Handling.java:15)
at trycatch.Handling.main(Handling.java:7)
```

La primera línea muestra lo que Java imprime de manera predeterminada: el tipo de excepción y el mensaje. La segunda línea muestra el mensaje. El resto muestra el *''stack trace''*. El *''stack trace''* es normalmente el más útil porque muestra donde ha ocurrido la excepción en cada método que pasa. El *''stack trace''* muestra todos los métodos. Cada vez que llamas al método, Java lo añade a la pila. Cuando se lanza una excepción, pasa por la pila hasta que se encuentra un método que puede manejarlo o se queda sin pila.

¿Por qué no se debe ignorar una excepción?

Debido a que las excepciones comprobadas requieren que se les maneje o se les declare, hay una tentación de atraparlos para que se 'vayan'. Pero hacerlo puede causar problemas. En el siguiente código hay un problema al leer el archivo.

```
public static void main(String[] args) {
  String textInFile = null;
  try {
    readInFile();
  } catch(IOException e) {
    // ignore exception
  }
  // imagine many lines of code here
  System.out.println(textInFile.replace("", ""));
}
private static void readInFile() throws IOException {
  throw new IOException();
}
```

El código da como resultado un ''NullPointerException''. Java no muestra nada sobre el ''IOException''. Es difícil pero se puede manejar. Cuando se escribe el código, se imprime un seguimiento de la pila o un mensaje cuando captura una excepción. Además, hay que considerar si es bueno continuar. En el ejemplo, el programa no puede hacer nada después de que no pueda leer el archivo. También podría haber lanzado simplemente un ''IOException''.

8.5. Resumen

En este capítulo, se ha visto que:

Entendiendo las excepciones

- Una excepción indica que ha sucedido algo inesperado.

- Un método puede manejar una excepción atrapándolo o declarándolo.

- Se lanzan muchas excepciones por las bibliotecas de Java.

- Se puede lanzar una excepción propia con código o lanzar un *''new Exception()''*.

Usando la sentencia try

- Java requiere excepciones controladas para ser manejadas o declaradas. Si una sentencia *''try''* tiene multiples bloques *''catch'',* se puede ejecutar como máximo un bloque *''catch''*.

- Java busca una excepción que pueda capturarse en cada bloque *''catch''* en el orden en que aparecen y el primero se ejecuta.

- Si ambos *''catch'' y "finally" lanzan una excepción, la de "finally"* se lanza.

Reconocimiento de los tipos comunes de excepciones

- Las subclases de *java.lang.Error* son excepciones que un programador no debería encargarse.

- Las subclases de *java.lang.RuntimeException* son excepciones de tiempo de ejecución.

- Las subclases *de java.lang.Exception*, pero no las de *java.lang.RuntimeException,* son manejadas como excepciones.

- Las excepciones comunes de tiempo de ejecución incluyen:

ArithmeticException

ArrayIndexOutOfBoundsException

ClassCastException

IllegalArgumentException

NullPointerException

NumberFormatException

- *'IllegalArgumentException''* y *''NumberFormatException''* suelen ser lanzados por el programador, mientras que los otros suelen ser lanzados por la JVM.

- Las excepciones comprobadas comunes incluyen:

IOException

FileNotFoundException

Common errors include:

ExceptionInInitializerError

StackOverflowError

NoClassDefFoundError

Llamando a métodos que provocan excepciones

- Cuando un método sobrescribe a otro método en una superclase o interfaz, no está permitido añadir *''exception''*.

- Se permite declarar menos excepciones o declarar una subclase de excepción declarada.

- Los métodos declaran excepciones con la palabra reservada *throw*.