Fundamentals of the Java Programming Language, Java SE 6

Student Guide

SL-110-SE6-ES Rev E.1

D61796CS10
Edition 1.0
D64159

This page intentionally left blank.

This page intentionally left blank.

Contenido

Acerca de este curso

Finalidad del curso

Este curso le proporcionará los conocimientos necesarios para:

- Manejar con soltura los conceptos relativos a la tecnología Java, el lenguaje de programación Java y el ciclo de vida de los productos.
- Utilizar diferentes construcciones del lenguaje Java para crear varias aplicaciones Java.
- Utilizar métodos, construcciones de toma de decisión y bucles para determinar el flujo de los programas.
- Implementar conceptos de programación Java de nivel intermedio y conceptos de programación orientada a objetos (OO) en programas Java.

Esquema del curso

Este esquema le permitirá ver los progresos que va realizando con respecto a los objetivos del curso.

Introducción a la programación en Java

- Descripción de la tecnología Java™
- Análisis de un problema y diseño de la solución
- Desarrollo y comprobación de un programa Java

Conceptos fundamentales sobre programación en Java

- Declaración, inicialización y uso de variables
- Creación y uso de objetos

Determinación del flujo de los programas

- Uso de operadores construcciones de toma de decisión
- Uso de bucles
- Desarrollo y uso de métodos

Conceptos intermedios sobre tecnología Java y programación OO

- Implementación de la encapsulación y los constructores
- Creación y uso de arrays
- Implementación de la herencia

Temas no incluidos

En este curso no se tratan los temas indicados a continuación. Muchos de ellos forman parte de otros cursos ofrecidos por los Servicios de Formación Sun.

- Conceptos avanzados sobre programación Java, que se cubren en el curso SL-275: *Programación Java™*
- Conceptos avanzados de análisis y diseño OO, que se cubren en el curso OO-226: *Análisis y Diseño OO con UML*
- Programación de applets (miniaplicaciones) o diseño de páginas web.

Consulte el catálogo de los Servicios de Formación Sun para obtener información detallada sobre los cursos y la inscripción.

¿Está preparado?

Si responde afirmativamente a las siguientes preguntas, se puede considerar preparado para realizar el curso.

- ¿Sabe crear y editar archivos de texto con un editor de texto?
- ¿Sabe utilizar un navegador web (WWW)?
- ¿Sabe resolver problemas de lógica?

Presentaciones

Después de esta introducción sobre el curso, ha llegado el momento de presentarse al profesor y los demás alumnos. Para ello, indique lo siguiente:

- Nombre
- Empresa
- Cargo, función y responsabilidad
- Experiencia relacionada con los temas de este curso
- Razones para inscribirse en el curso
- Expectativas con respecto al curso

Cómo utilizar el material del curso

Para que pueda sacar el máximo provecho a este curso, el temario incluye un módulo didáctico que consta de lo siguiente:

- **Finalidad**: al final del curso debería ser capaz de alcanzar las metas fijadas y cumplir todos los objetivos.
- **Objetivos**: después de cubrir una parte del contenido del curso, debería estar en disposición de cumplir los objetivos. Los objetivos permiten alcanzar la meta y lograr nuevos objetivos.
- **Clases teóricas**: el profesor ofrece información específica para cumplir los objetivos del módulo. Esta información ayuda a adquirir los conocimientos necesarios para realizar correctamente las actividades.
- **Actividades**: las actividades son de diversa índole y pueden incluir ejercicios, autoevaluaciones, discusiones y demostraciones. Las actividades ayudan a dominar las materias.
- **Material visual de apoyo**: el profesor puede utilizar este tipo de ayuda para explicar de forma gráfica conceptos como el de un determinado proceso. La ayuda visual suele constar de gráficos, imágenes animadas y vídeo.

Convenciones

En este curso se usan determinadas convenciones para representar varios elementos y recursos de formación alternativos.

Iconos

Demostración – Indica que en ese momento se recomienda ver la demostración del tema en cuestión.

Discusión – Indica que es aconsejable entablar un debate general o en grupos reducidos sobre el tema tratado.

¿Lo sabía? – Indica la existencia de información adicional que puede ayudar al alumno, pero que no es crucial para entender el concepto que se está explicando.

Nota – Indica la existencia de información complementaria que puede ayudar al alumno, pero que no es crucial para entender el concepto que se está explicando. El alumno debería ser capaz de entender el concepto o de realizar la tarea sin esta información. En las notas se incluyen los métodos abreviados del teclado y los ajustes menores del sistema, entre otra información.

Atención – Indica que existe el riesgo de causar lesiones personales por causas no eléctricas, o daños irreversibles en los datos, el software o el sistema operativo. Advierte de un posible riesgo (del que no se tiene certeza), que depende de la acción que realice el usuario.

Atención – Indica que se ocasionarán lesiones personales o daños irreversibles en los datos, el software o el sistema operativo si se realiza la acción. No advierte de posibles daños, sino de los resultados catastróficos de la acción.

Autoevaluación – Señala las actividades de autoevaluación, como preguntas de respuesta única y múltiple.

Caso de estudio – Indica la existencia de un ejemplo o caso de estudio que se utiliza a lo largo de todo el curso.

Convenciones tipográficas

Se utiliza `Courier` para indicar nombres de comandos, archivos, código de programación y mensajes del sistema que aparecen en la pantalla, por ejemplo:

> Utilice `ls-a` para ver la lista de todos los archivos.
> `system% Tiene correo`.

También se utiliza `Courier` con constructores de programación tales como los nombres de clases, métodos y palabras clave, por ejemplo:

> El método `getServletInfo` se utiliza para obtener información sobre el autor.
> La clase `java.awt.Dialog` contiene el constructor `Dialog`.

Se aplica **`Courier en negrita`** a los números y caracteres que debe introducir el usuario, por ejemplo:

> Para ver la lista de archivos de este directorio, escriba:
> `# ls`

También se usa **`Courier en negrita`** en las líneas de código de programación que se citan en un descripción textual, por ejemplo:

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
```

La interfaz `javax.servlet` se importa para poder acceder a los métodos de su ciclo de vida (línea 2).

Se aplica *Courier en cursiva* a las variables y los nombres de ejemplo de la línea de comandos que se sustituyen por un nombre o valor real, por ejemplo:

Para eliminar un archivo, utilice el comando `rm` *nombre de archivo*.

Se usa ***Courier en cursiva y negrita*** para representar variables cuyos valores debe introducir el alumno durante un ejercicio, por ejemplo:

Escriba **`chmod a+rwx`** ***nombre de archivo*** para conceder derechos de lectura, escritura y ejecución sobre ese archivo a todos, grupos y usuarios.

La fuente *Palatino en cursiva* se utiliza con títulos de libros, palabras o términos nuevos o que se desea enfatizar, por ejemplo:

Consulte el capítulo 6 del *Manual del usuario*.
Se denominan opciones de *clase*.

Otras convenciones

En los ejemplos de programación en Java™ se emplean además las siguientes convenciones:

- Los nombres de método no van seguidos de paréntesis, salvo cuando se incluye una lista de parámetros oficiales o reales, por ejemplo:

  "El método `doIt`..." designa cualquier método denominado `doIt`.

  "El método `doIt()`..." indica un método `doIt` que no admite argumentos.

- Sólo se producen saltos de línea cuando hay separaciones (comas), conjunciones (operadores) o espacios en blanco en el código.
  El código interrumpido se sangra cuatro espacios bajo el código inicial.

- Cuando en el sistema operativo Solaris™ se utiliza un comando distinto al empleado en la plataforma Microsoft Windows, se muestran ambos comandos, por ejemplo:

  En Solaris:

  ```
  $CD SERVER_ROOT/BIN
  ```

  En Microsoft Windows:

  ```
  C:\>CD SERVER_ROOT\BIN
  ```

Módulo 1

Descripción de la tecnología Java™

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Describir los conceptos fundamentales del lenguaje de programación Java.
- Enumerar los tres grupos de productos que componen la tecnología Java.
- Resumir cada una de las siete fases que forman el ciclo de vida de los productos.

Este módulo contiene una introducción a la tecnología Java, el lenguaje de programación Java y el ciclo de vida habitual en el desarrollo de las aplicaciones.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Describir los conceptos fundamentales del lenguaje de programación Java.			
Enumerar los tres grupos de productos que componen la tecnología Java.			
Resumir cada una de las siete fases que forman el ciclo de vida de los productos.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Notas

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender en qué consiste la tecnología Java:

- Qué significan para usted las siguientes palabras:
  - Seguro
  - Orientado a objetos
  - Independiente
  - Dependiente
  - Distribuido
- ¿Qué fases intervienen en la construcción de algo, por ejemplo, una casa o un mueble?

Otros recursos

Otros recursos – Los documentos siguientes proporcionan información complementaria sobre los temas tratados en este módulo:

- Feature Stories About Java Technology. [En la web]. Disponible en: `http://java.sun.com/features/index.html`

  Una historia sobre las personas que hicieron posible el desarrollo de la tecnología Java, escrita con motivo del quinto aniversario de esta tecnología.

- Tutorial de Java. [En la web]. Disponible en: `http://java.sun.com/docs/books/tutorial/`

  Una guía práctica para programadores que incluye cientos de ejemplos y ejercicios completos.

Conceptos fundamentales sobre el lenguaje Java

El lenguaje de programación Java nació en 1991 como parte de un proyecto de investigación destinado a desarrollar un lenguaje de programación llamado “Oak” (en español, roble) que resolvería la falta de comunicación entre numerosos dispositivos de consumo tales como las videograbadoras y los televisores. En concreto, un equipo de expertos en desarrollo de software (el equipo Green) quería crear un lenguaje que permitiese a diferentes dispositivos de consumo dotados de diferentes unidades de procesamiento central (CPU) compartir las mismas mejoras del software.

¿Lo sabía? – James Gosling, uno de los miembros del equipo, denominó Oak al nuevo lenguaje como referencia al roble que veía a través de su ventana.

Este intento inicial fracasó después de varios acuerdos fallidos con varias compañías de dispositivos de consumo. El equipo Green se vio obligado a buscar otro mercado para su nuevo lenguaje.

Afortunadamente, empezaba a popularizarse el uso de la World Wide Web y el equipo reconoció que el lenguaje Oak era perfecto para desarrollar componentes multimedia que mejorasen el contenido de las páginas web. Los primeros desarrollos del lenguaje Oak se aplicaron a estas pequeñas aplicaciones, llamadas applets, y los programadores que usaban Internet adoptaron lo que se convertiría en el lenguaje de programación Java.

Nota – Para conocer con más detalle la historia del lenguaje Java, visite `http://www.java.sun.com` y realice una búsqueda de la siguiente frase “History of Java”.

El lenguaje Java se diseñó para ser:

- Orientado a objetos
- Distribuido
- Simple
- Multihilo
- Seguro
- Independiente de la plataforma

Orientado a objetos

Java es un lenguaje de programación orientada a objetos (OO) porque uno de los principales objetivos del programador de aplicaciones Java es crear objetos, fragmentos de código autónomos que puedan interaccionar con otros objetos para resolver un problema. La programación OO empezó con el lenguaje SIMULA-67 en 1967 y ha dado lugar a lenguajes tan populares como C++, en el que se ha inspirado ligeramente el lenguaje Java.

La programación OO se diferencia de la programación por procedimientos en que ésta última se centra en la secuencia de pasos del código necesarios para resolver un problema, mientras que los lenguajes orientados a objetos se centran en la creación e interacción de los objetos.

En la figura siguiente se muestra la importancia que tiene la secuencia de operaciones en un programa basado en procedimientos.

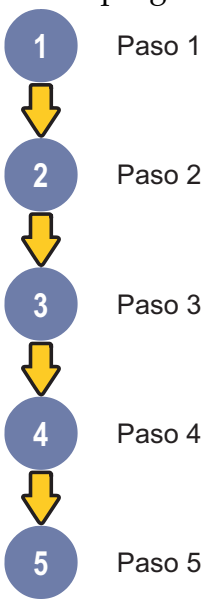


Figura 1-1 Importancia de la secuencia en un programa basado en procedimientos

En la figura siguiente se ilustra la importancia de los objetos y sus interacciones en un programa orientado a objetos.

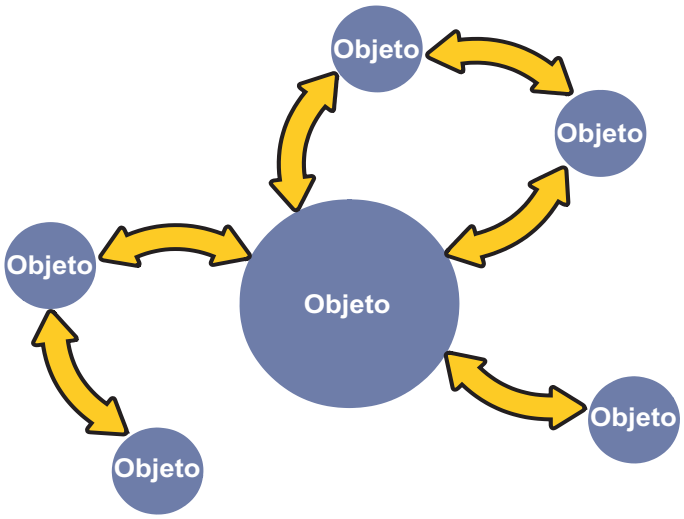


Figura 1-2 Importancia de los objetos y sus interacciones en un programa OO

Distribuido

Java es un lenguaje distribuido porque da cabida a tecnologías de redes distribuidas tales como RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) y URL (Universal Resource Locator).

Asimismo, la capacidad de carga dinámica de clases de Java permite descargar fragmentos de código a través de Internet y ejecutarlas en un equipo informático de tipo PC.

Nota – Los términos tecnología Java y lenguaje de programación Java no se refieren a lo mismo. La tecnología Java hace referencia a una familia de productos Java de la que el lenguaje de programación es sólo una parte.

En la figura siguiente se ilustran las propiedades distribuidas de la tecnología Java.

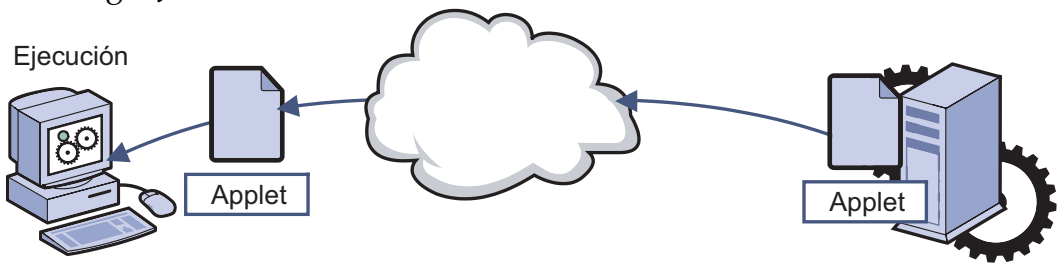


Figura 1-3 El lenguaje de programación Java es distribuido

Simple

El lenguaje Java es simple porque sus diseñadores han suprimido algunas de las construcciones de programación más complejas o difíciles de otros lenguajes populares. Por ejemplo, Java no permite a los programadores manejar directamente los punteros a las direcciones de memoria, una función compleja y, a menudo, mal utilizada de los lenguajes C y C++. En su lugar, sólo les permite manejar objetos utilizando referencias a objetos. Además utiliza una función denominada reciclaje de memoria dinámica (garbage collector) para hacer el seguimiento de los objetos a los que ya no se hace referencia y eliminarlos. Otro aspecto que hace de Java un lenguaje simple es que los tipos de datos booleanos (`boolean`) sólo pueden tener los valores `true` o `false`, frente a otros lenguajes donde también se admiten los valores 1 y 0.

Multihilo

El lenguaje Java admite procesamiento multihilo (multithread), es decir, la ejecución simultánea de tareas diferentes tales como consultar una base de datos y reproducir en la pantalla una interfaz de usuario. Gracias a la ejecución multihilo, los programas Java pueden hacer un uso muy eficiente de los recursos del sistema.

En la figura siguiente se ilustra el procesamiento multihilo del lenguaje Java.

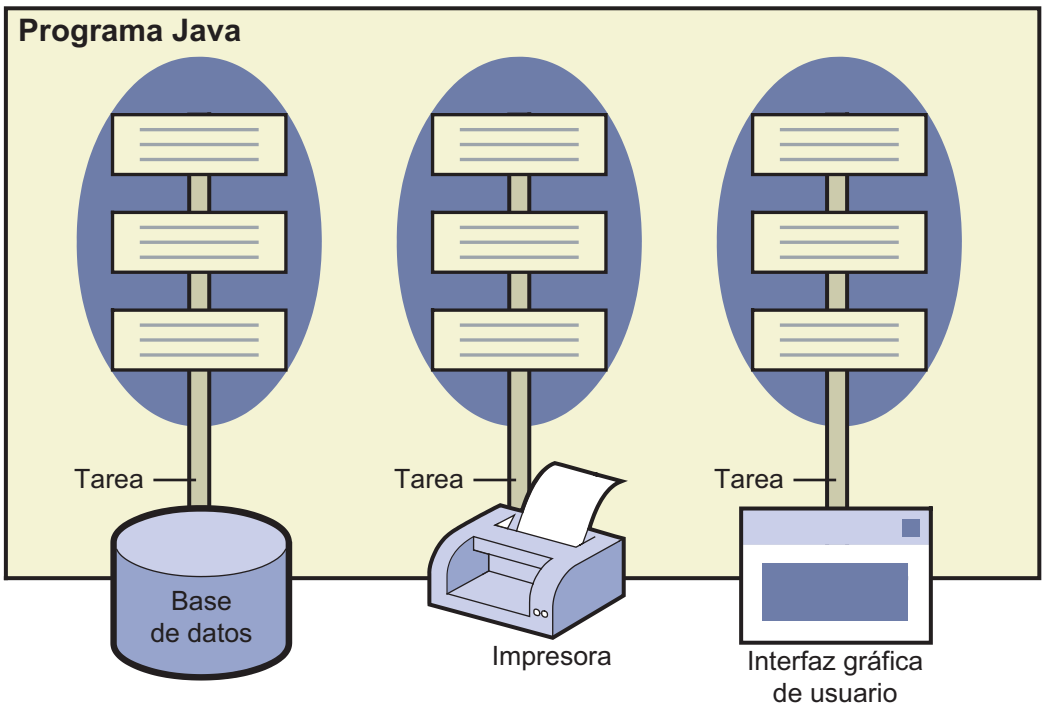


Figura 1-4 El lenguaje de programación Java es multihilo

Seguro

Los programas Java son seguros porque el lenguaje Java, con el entorno en el que se ejecutan los programas con esta tecnología, aplica medidas de seguridad para proteger el código de posibles ataques. Estas medidas incluyen:

- Prohibir la manipulación de la memoria mediante el uso de punteros.
- Impedir que los programas distribuidos, por ejemplo los applets, puedan hacer operaciones de lectura y escritura en el disco duro de un equipo.

- Verificar que todos los programas Java contengan código válido.
- Facilitar el uso de firmas digitales. Una empresa o una persona puede "firmar" el código de Java de forma que quien reciba el código pueda verificar la legitimidad del mismo.

En la figura siguiente puede verse cómo la seguridad integrada en los programas Java impide ejecutar código no válido en un equipo.

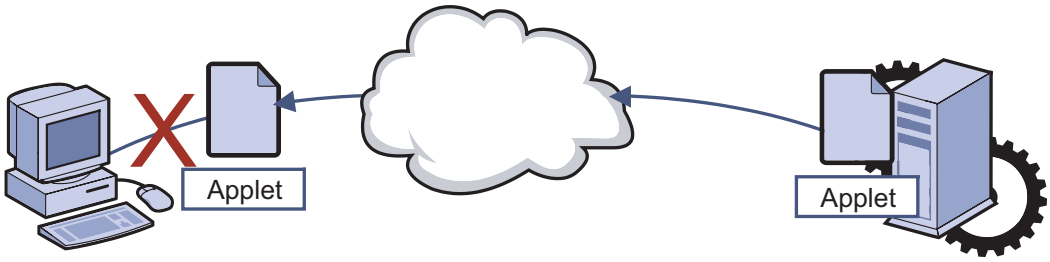


Figura 1-5 No es posible ejecutar código que no sea válido

Independiente de la plataforma

Los programas escritos en la mayoría de los lenguajes normalmente necesitan numerosas modificaciones para poder ejecutarse en diferentes tipos de plataformas informáticas (la combinación de una CPU y un sistema operativo). Esta dependencia de la plataforma se produce porque la mayoría de los lenguajes exigen que se escriba el código específico para la plataforma subyacente. Lenguajes tan populares como C y C++ obligan al programador a compilar y vincular sus programas, lo que da lugar a un programa ejecutable exclusivo de una plataforma. Al revés que C y C++, el lenguaje Java es independiente de la plataforma.

Programas dependientes de la plataforma

Un compilador es una aplicación que convierte un programa escrito por un programador en un código específico para la CPU denominado código máquina. Estos archivos específicos de la plataforma (archivos binarios) a menudo se combinan con otros archivos, tales como bibliotecas de código preelaborado, mediante un vinculador para crear un programa dependiente de la plataforma, llamado ejecutable, que el usuario final puede ejecutar.

En la figura siguiente puede verse la forma en que un compilador crea un archivo binario.

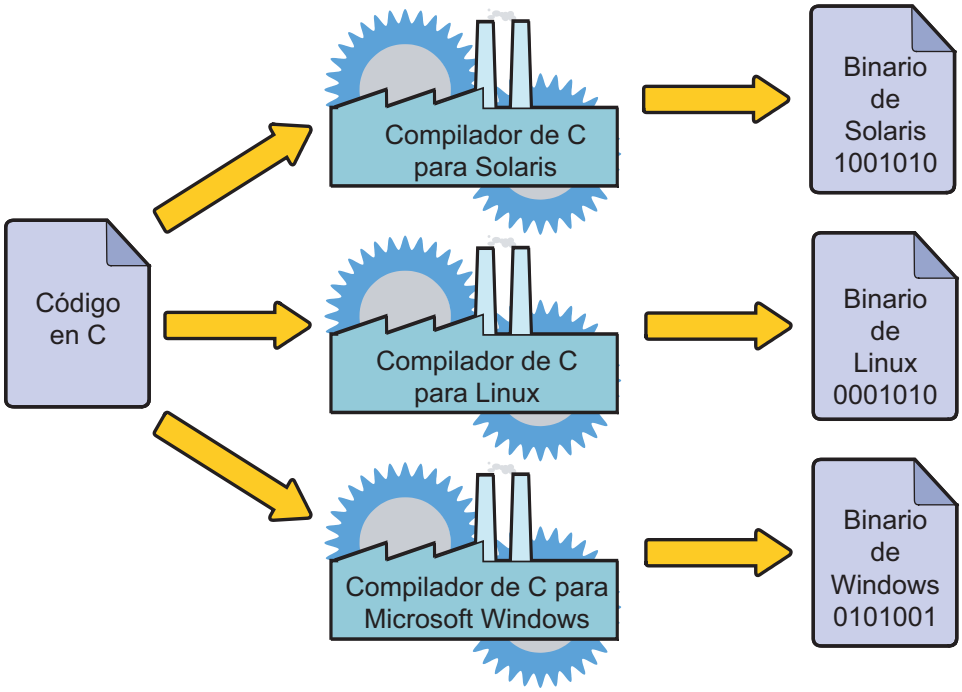


Figura 1-6 Creación de un archivo binario

La figura siguiente muestra cómo se vincula un archivo binario a las bibliotecas para crear un ejecutable dependiente de la plataforma.

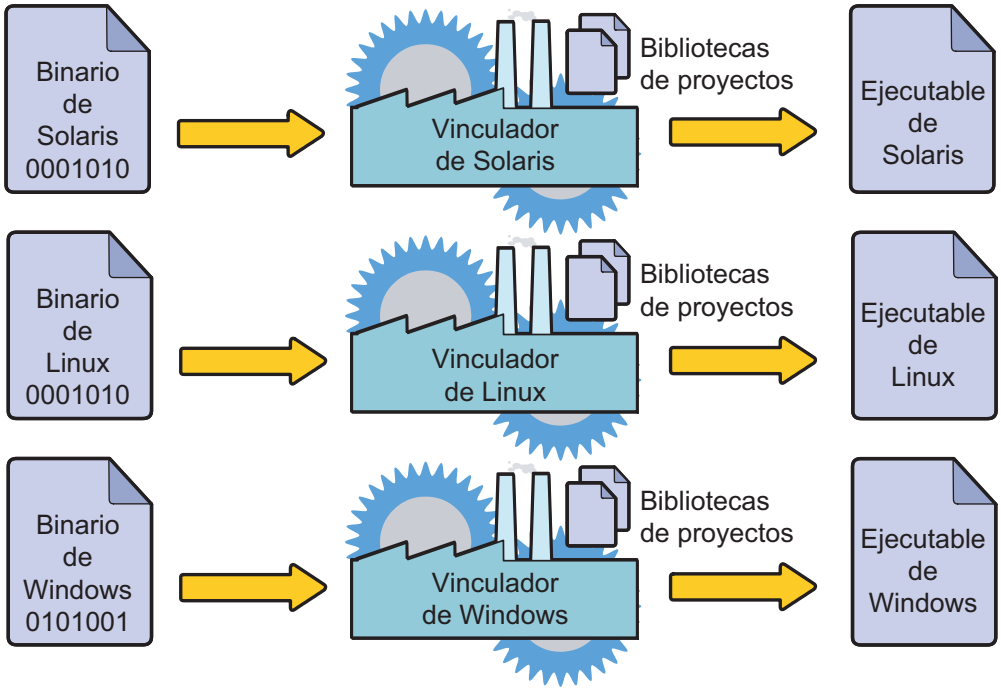


Figura 1-7 Creación de un ejecutable

En la figura siguiente se ilustra cómo los ejecutables dependientes de la plataforma pueden ejecutarse solamente en una plataforma.

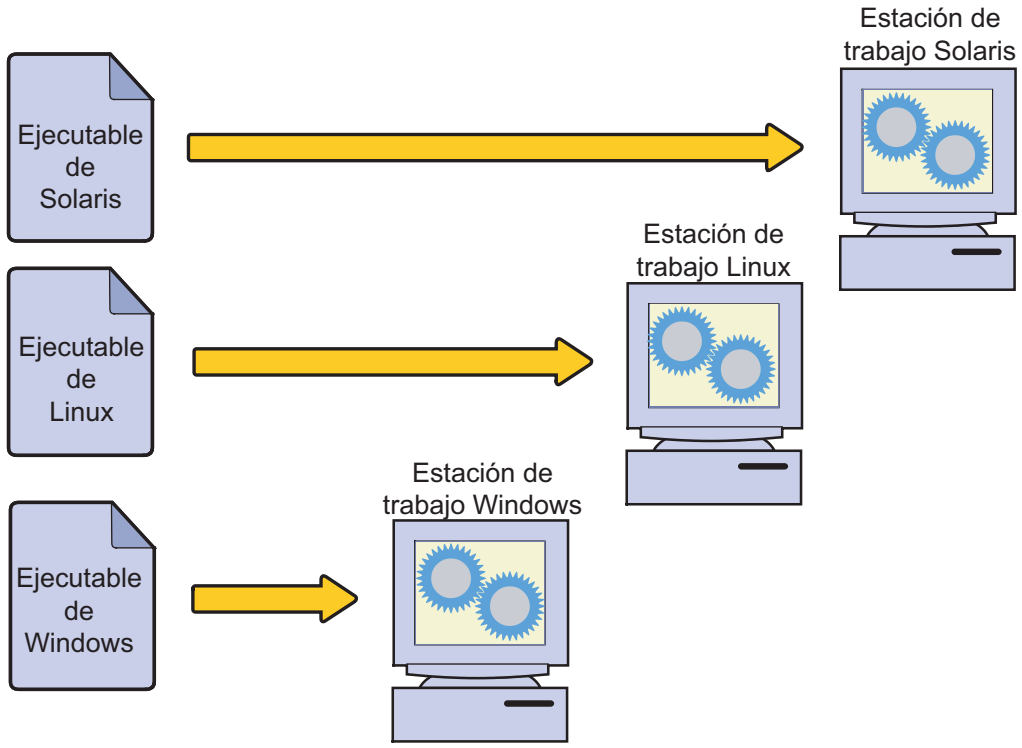


Figura 1-8 Ejecución de un archivo ejecutable

Programas independientes de la plataforma

Un programa Java puede ejecutarse, con escasas o ninguna modificación, en varias combinaciones de CPU y sistemas operativos como, por ejemplo, el sistema operativo Solaris con un procesador SPARC®, MacOS en un procesador Motorola y Microsoft Windows en un procesador Intel.

Nota – Posiblemente sea preciso realizar algunas modificaciones para hacer que un programa Java sea independiente de la plataforma. Por ejemplo, puede ser necesario cambiar los nombres de los directorios a fin de que utilicen los delimitadores adecuados (barras inclinadas e invertidas) para el sistema operativo de base.

Al igual que ocurre con los programas C y C++, los programas Java también se compilan utilizando un compilador específico para la tecnología Java. No obstante, el formato resultante de un programa Java compilado es código de byte Java independiente de la plataforma en lugar de código máquina específico de la CPU. Una vez generado el código de byte, se interpreta (ejecuta) a través de un intérprete llamado máquina virtual o VM. Una máquina virtual es un programa específico de la plataforma que lee el código de byte (independiente de la plataforma) y lo ejecuta en una plataforma concreta. Por este motivo, el lenguaje Java a menudo se define como un lenguaje interpretado y los programas Java se consideran como transportables o ejecutables en cualquier plataforma. Entre los lenguajes interpretados se incluye también Perl.

¿Lo sabía? – El término *máquina virtual* se debe a que es un componente de software que ejecuta código, una tarea normalmente realizada por la CPU o la *máquina de hardware*.

Para que los programas Java sean independientes de la plataforma, es preciso disponer de una máquina virtual llamada *máquina virtual de Java* (JVM™) en cada plataforma donde se vayan a ejecutar. La máquina virtual de Java es la encargada de interpretar el código Java, cargar las clases correspondientes y ejecutar los programas Java.

Sin embargo, un programa Java necesita algo más que una máquina virtual de Java para ejecutarse. También precisa una serie de bibliotecas de clases estándar específicas para la plataforma. Las bibliotecas de clases de Java son bibliotecas de código preescrito que puede combinarse con el código elaborado por el programador para crear aplicaciones sólidas.

La combinación del software JVM y estas bibliotecas de clases se conocen como el entorno de tiempo de ejecución de Java (JRE). Sun Microsystems proporciona entornos de ejecución de Java para numerosas plataformas de uso común.

¿Lo sabía? – A menudo, la tecnología Java se considera una plataforma porque puede realizar todas las tareas de una CPU y un sistema operativo. Sun Microsystems desarrolló un prototipo de sistema informático con una CPU llamado JavaStation™, que sólo entiende código de byte de Java.

En la figura siguiente puede verse cómo el compilador de Java crea código de byte.

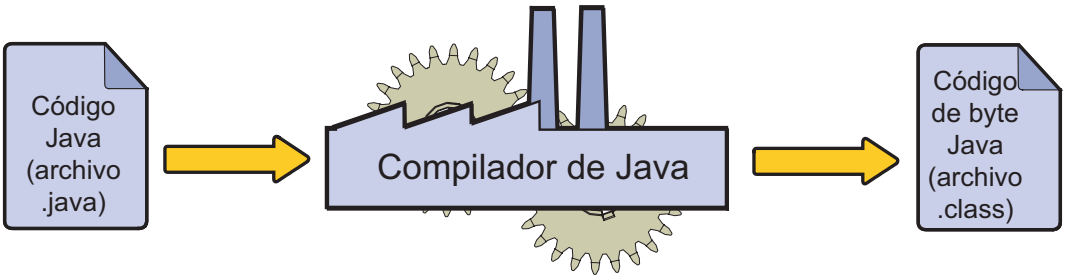


Figura 1-9 Creación de código de byte en Java

En la figura siguiente se muestra cómo un archivo de código de byte de Java se ejecuta en varias plataformas donde existe un entorno de ejecución de Java.

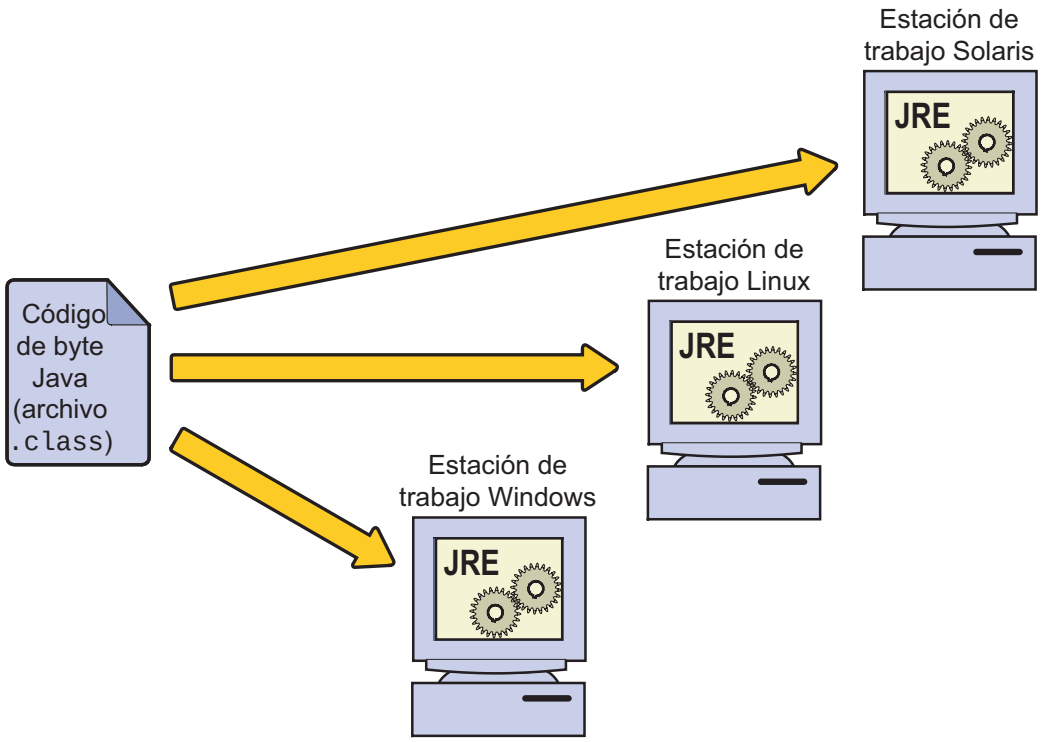


Figura 1-10 Ejecución de código de byte en un entorno de ejecución de Java

La capacidad de los programas Java para ejecutarse en todas las plataformas es fundamental para explicar el concepto del eslogan "Write Once, Run Anywhere™" de Sun Microsystems.

Nota – Los términos API (interfaz de programación de aplicaciones) y biblioteca de clases pueden usarse indistintamente. Asimismo, un API puede hacer referencia a una sola rutina dentro de una biblioteca de código.

Grupos de productos de la tecnología Java

Sun Microsystems proporciona una línea completa de productos de tecnología Java que abarcan desde kits de desarrollo de programas Java hasta entornos de emulación (pruebas) para dispositivos de consumo tales como los teléfonos celulares.

Identificación de los grupos de productos de la tecnología Java

Las tecnologías Java, tales como la máquina virtual de Java, se incluyen (de formas distintas) en tres grupos diferentes de productos, cada uno de ellos diseñado para atender a las necesidades de un determinado segmento de mercado:

- Plataforma Java™, Standard Edition (Java SE): permite desarrollar applets y aplicaciones que se ejecutan, respectivamente, en navegadores web y equipos informáticos de sobremesa.
  Por ejemplo, es posible usar el kit de desarrollo de software (SDK) de Java SE para crear un procesador de texto para PC.

Nota – Los applets y las aplicaciones difieren en varios aspectos. En primer lugar, los applets se ejecutan dentro de un navegador web, mientras que las aplicaciones se ejecutan en un sistema operativo. Aunque este curso se centra fundamentalmente en el desarrollo de aplicaciones, la mayoría de la información que contiene puede aplicarse también al desarrollo de applets.

- Plataforma Java™, Enterprise Edition (Java EE): permite crear grandes aplicaciones empresariales distribuidas para los lados cliente y servidor. Por ejemplo, el SDK de Java EE puede utilizarse para crear una aplicación de comercio electrónico que pueda utilizarse en el sitio web de una compañía comercial.
- Plataforma Java™, Micro Edition (Java ME): permite crear aplicaciones para dispositivos de consumo de recursos limitados. Por ejemplo, el SDK de Java ME puede utilizarse para crear un juego que se ejecute en un teléfono móvil.

La figura siguiente muestra los tres grupos de productos de la tecnología Java y los tipos de dispositivos a los que van dirigidos.

Figura 1-11 Grupos de productos de la tecnología Java

Entre otras tecnologías, cada edición de la plataforma Java incluye un kit de desarrollo de software (SDK) con el que es posible crear, compilar y ejecutar programas Java en una plataforma determinada.

Elección del grupo de productos de tecnología Java correcto

Aunque puede que muchos programadores de Java se especialicen en el desarrollo de aplicaciones para un determinado mercado, normalmente todos ellos empiezan sus carreras creando aplicaciones o applets para PC. Por tanto, el SDK de Java SE es el grupo de productos que utilizan la mayoría de los programadores cuando aprenden a manejar el lenguaje Java.

Uso de los componentes del SDK de la plataforma Java, Standard Edition

Sun Microsystems ha desarrollado una versión del SDK de la plataforma Java, Standard Edition para Solaris™ y el procesador SPARC® (32 y 64 bits), así como para los sistemas operativos Solaris, Linux y Microsoft Windows ejecutados en procesadores (32 y 64 bits) de las compañías Intel Corporation y Advanced Micro Devices, Incorporated. El SDK de la plataforma Java, Standard Edition incluye lo siguiente:

- El entorno de ejecución de Java:
  - Una máquina virtual de Java para la plataforma que elija.
  - Bibliotecas de clases de Java para la plataforma que elija.
- Un compilador de Java
- La documentación de la biblioteca de clases (API) de Java (descarga por separado)
- Herramientas suplementarias, como las utilizadas para crear archivos de almacenamiento Java (JAR) y depurar programas Java
- Ejemplos de programas Java

¿Lo sabía? – Para garantizar que los programas Java se ejecuten en todos los equipos donde haya una máquina virtual de Java y una biblioteca de clases de Java, tanto el software de la JVM como las bibliotecas de clases deben cumplir las especificaciones del lenguaje Java y la máquina virtual de Java. Estas especificaciones están disponibles para las empresas que adquieran la licencia de la tecnología Java y quieran crear su propio software de JVM y sus propios compiladores. De hecho, Sun Microsystems tuvo que seguir sus propias especificaciones para crear un entorno de ejecución de Java para Solaris en el procesador SPARC y para Windows en el procesador Intel.

Demostración – Su profesor le mostrará cuatro tipo de aplicaciones basadas en la tecnología Java. Son las siguientes:

- Aplicación del SDK de Java SE
- Applet del SDK de Java SE
- Aplicación del SDK de Java EE
- Aplicación del SDK de Java ME

Al ver la demostración, debería prestar especial atención a:

- La forma en que se ejecuta el applet o la aplicación (¿mediante comandos, iconos...?)
- El lugar donde se ejecuta el applet o la aplicación (¿en un navegador web, un dispositivo de consumo...?)

Autoevaluación – Establezca las correspondencias adecuadas entre los términos y sus definiciones.

Definición	Término
Se compone de la máquina virtual de Java y bibliotecas de clases de Java	Máquina virtual de Java
Manipula objetos en lugar de punteros	Applet
Ejecuta código de byte independiente de la plataforma	Compilador
Elimina de la memoria objetos a los que no se hace referencia	Referencia
Se ejecuta en un navegador web	Código de byte
Crea código de byte de Java	Entorno de ejecución de Java
Lo crea el compilador y lo ejecuta la máquina virtual de Java	Reciclaje de memoria dinámica

Fases del ciclo de vida de los productos

El ciclo de vida de los productos (Product Life Cycle o PLC) representa un conjunto de fases aceptadas[1] por la industria que un programador debería seguir al desarrollar cualquier producto nuevo. El ciclo se compone de siete fases. Son las siguientes:

1. Análisis
2. Diseño
3. Desarrollo
4. Comprobación
5. Implementación
6. Mantenimiento
7. Fin del ciclo de vida (EOL)

1. El ciclo de vida de los productos sólo es una filosofía para sistematizar las fases de desarrollo de nuevos productos. Existen otras tales como: Rational Unified Process, XP (Extreme Programming) y TDD (Test Driven Design).

Fase de análisis

El análisis es el proceso de investigar un problema que se pretende resolver con un producto. Entre otras tareas, consiste en:

- Definir claramente el problema que se quiere resolver, el nicho de mercado al que se dirige o el sistema que se quiere crear. Los límites del problema también se conocen como el ámbito del proyecto.
- Identificar los subcomponentes fundamentales que forman parte de la totalidad del producto.

En la figura siguiente se ilustra la fase de análisis.

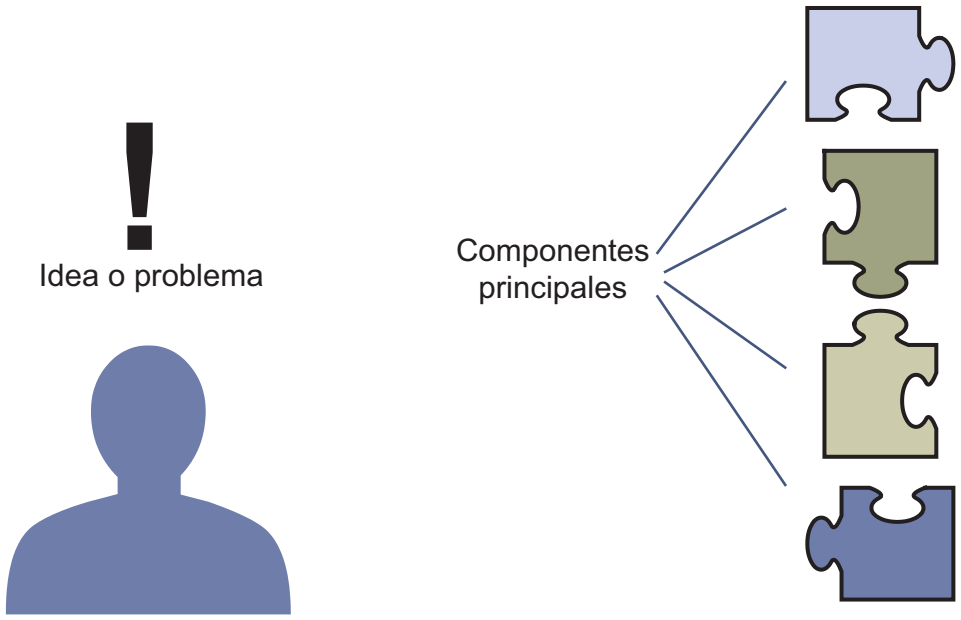


Figura 1-12 Fase de análisis del ciclo de vida de los productos

Fase de diseño

El diseño es el proceso de aplicar las conclusiones extraídas en la fase de análisis al proyecto del producto. La tarea principal en la fase de diseño es desarrollar planos o especificaciones de los productos o los componentes del sistema.

En la figura siguiente se ilustra la fase de diseño.

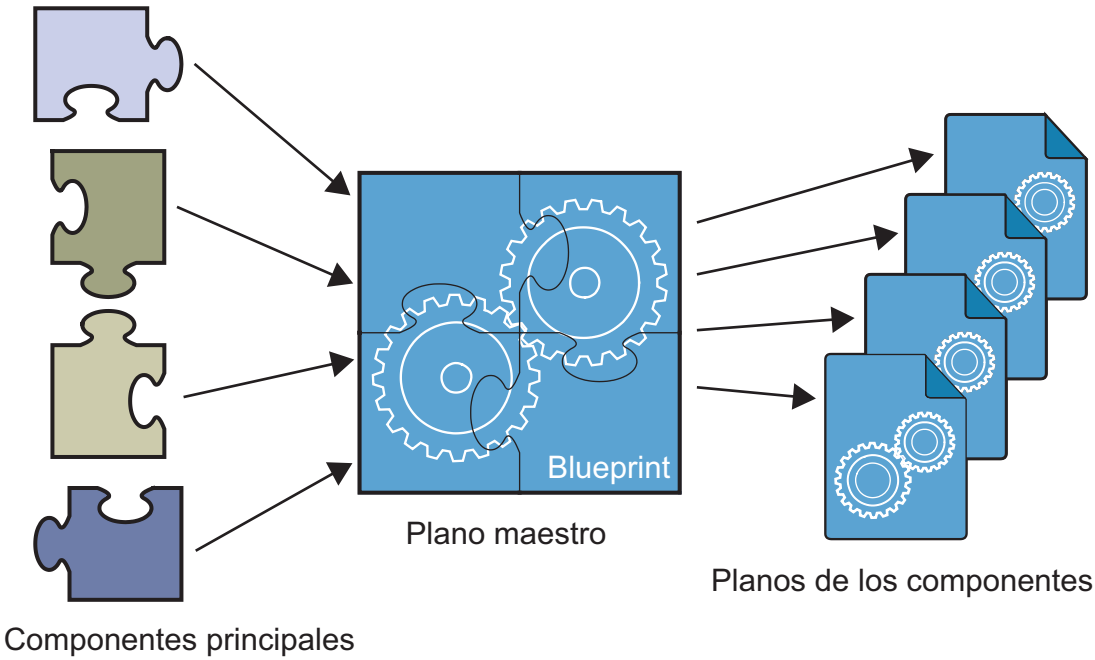


Figura 1-13 Fase de diseño del ciclo de vida de los productos

Fase de desarrollo

El desarrollo consiste en utilizar los planos creados durante la fase de diseño para crear los componentes reales.

En la figura siguiente se ilustra la fase de desarrollo.

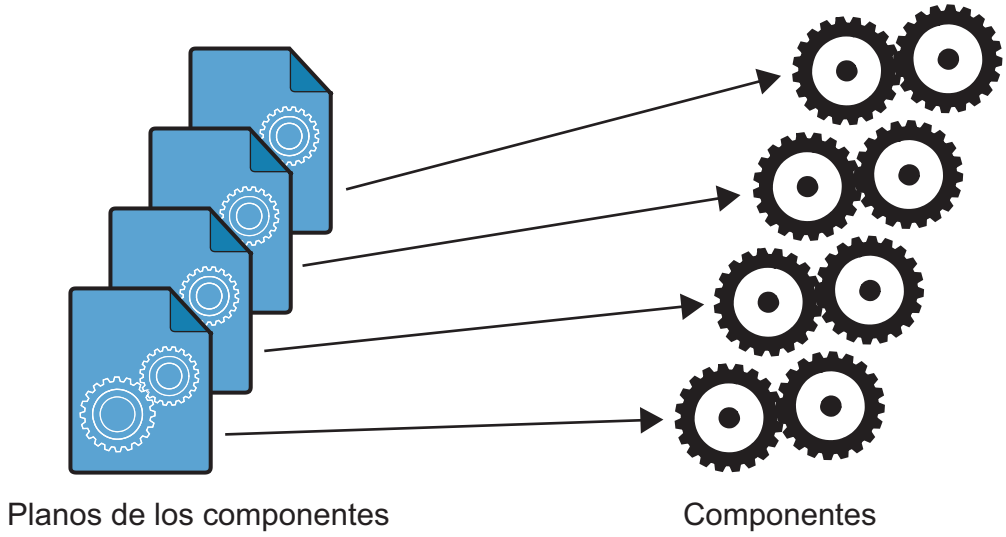


Figura 1-14 Fase de desarrollo del ciclo de vida de los productos

Fase de comprobación

La comprobación consiste en asegurarse de que cada uno de los componentes o la totalidad del producto cumplan los requisitos de la especificación creada durante la fase de diseño.

Nota – Un buen análisis del problema conduce a un buen diseño de la solución y a una reducción del periodo de comprobación y desarrollo.

Normalmente la comprobación la realiza un equipo de personas distinto de aquel que ha desarrollado el producto. Esto garantiza que se probará de forma objetiva, eliminando la subjetividad del programador.

En la figura siguiente se ilustra la fase de comprobación.

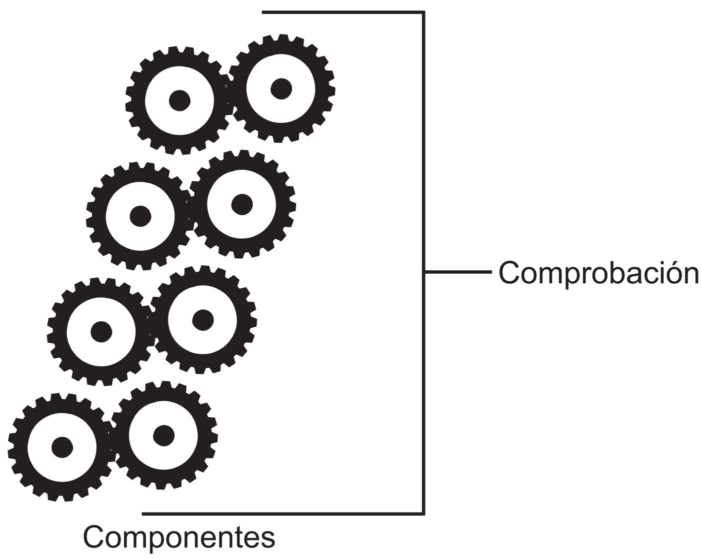


Figura 1-15 Fase de comprobación del ciclo de vida de los productos

---

Nota – Las cuatro primeras fases del ciclo pueden aplicarse a cada uno de los ejercicios de este curso. Para garantizar la realización correcta de los ejercicios, debería (1) analizar el problema, (2) desarrollar un esquema rápido o una especificación para la solución o el programa, (3) desarrollar la solución y (4) comprobar el programa.

Fase de implementación

La implementación consiste en poner el producto a disposición de los clientes.

En la figura siguiente se ilustra la fase de implementación.

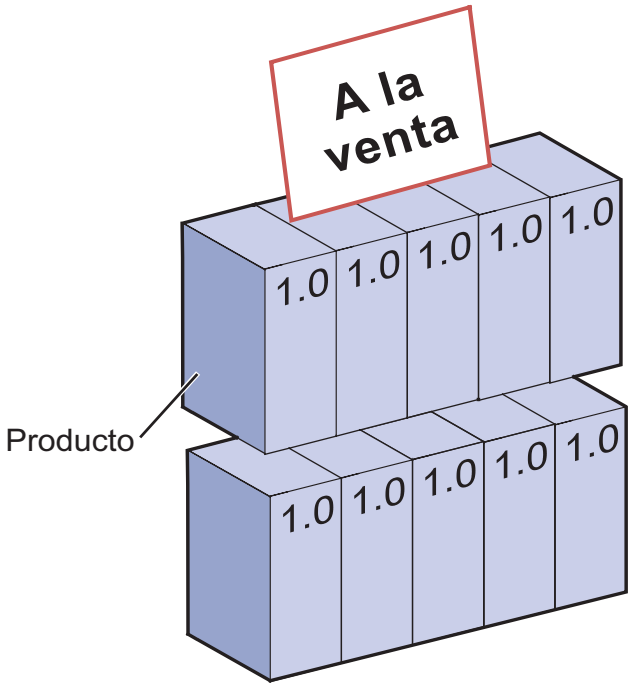


Figura 1-16 Fase de implementación del ciclo de vida de los productos

¿Lo sabía? – A menudo, la fase de implementación se conoce en la industria informática como *primera entrega al cliente* o FCS.

Fase de mantenimiento

El mantenimiento consiste en resolver los problemas del producto y publicar nuevas versiones o revisiones del mismo.

En la figura siguiente se ilustra la fase de mantenimiento.

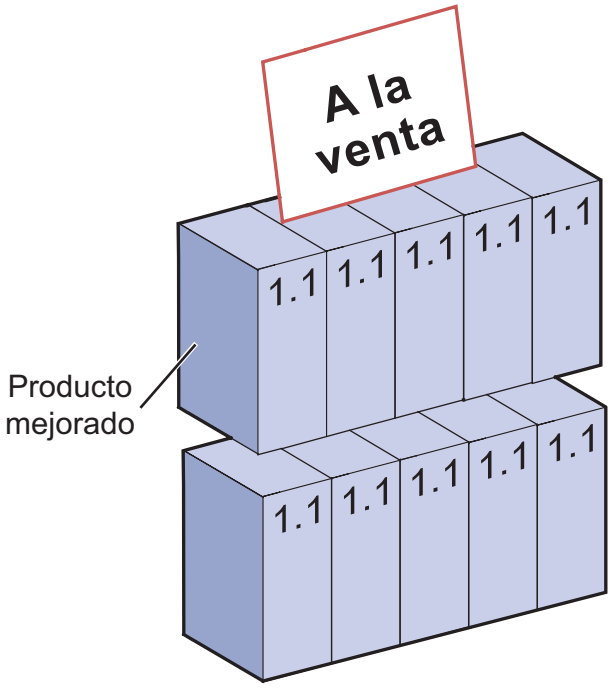
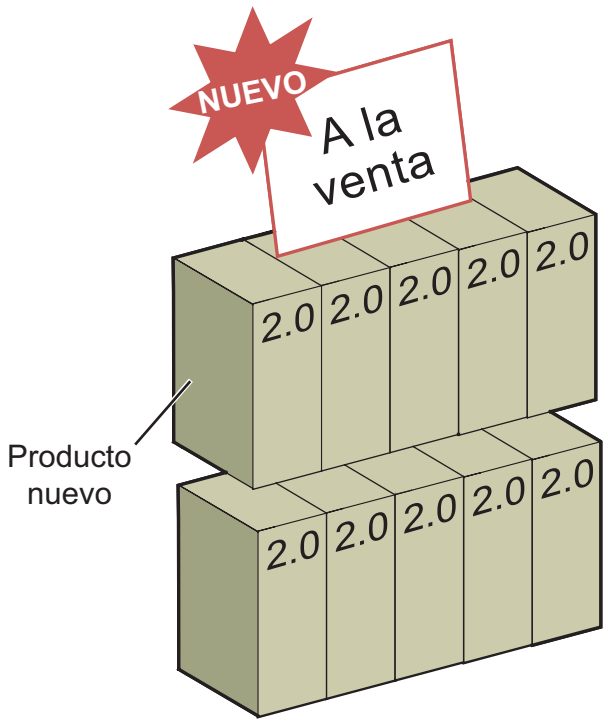


Figura 1-17 Fase de mantenimiento del ciclo de vida de los productos

Fin del ciclo de vida (EOL)

Aunque el ciclo de vida de los productos no incluye una fase específica para iniciar un concepto o un proyecto, sí la tiene para finalizarlo. El fin del ciclo de vida o EOL consiste en realizar todas las tareas necesarias para garantizar que clientes y empleados sean conscientes de que un producto ha dejado de venderse y recibir asistencia técnica, y de que hay un nuevo producto disponible.

En la figura siguiente se ilustra la fase final del ciclo de vida de los productos.



El fin del ciclo de vida implica sustituir el producto por otro recién desarrollado.

Figura 1-18 Fase final del ciclo de vida de los productos

¿Por qué debe seguir las fases del ciclo de vida de los productos?

El PLC es una parte importante del desarrollo de los productos porque ayuda a garantizar que éstos se crearán y suministrarán de forma que se reduzca el tiempo de salida al mercado, la calidad del producto sea alta y se maximice el retorno de la inversión. Los desarrolladores que no siguen sus directrices a menudo encuentran problemas que cuesta corregir y que podrían haberse evitado.

Nota – Este curso no presenta las fases del ciclo en profundidad. No obstante, todo lo que aprenderá a lo largo del mismo se enmarca en alguna de ellas.

Autoevaluación – Haga corresponder cada tarea a la fase correspondiente del ciclo de vida de los productos.

Tarea	Fase
Crear un plano del producto	Comprobación
Determinar el ámbito del problema	Fin del ciclo de vida
Corregir problemas de los clientes y agregar mejoras	Análisis
Comunicar que el producto ya no estará disponible	Implementación
Hacer que el producto funcione como se indica en una especificación	Desarrollo
Crear el producto	Diseño
Realizar la primera entrega del producto	Mantenimiento

Módulo 2

Análisis de un problema y diseño de la solución

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Analizar problemas utilizando el análisis orientado a objetos (OOA).
- Diseñar clases a partir de las cuales crear objetos.

En este módulo se explica cómo analizar un problema propuesto y desarrollar el diseño de una aplicación.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Analizar problemas utilizando el análisis orientado a objetos.			
Diseñar clases a partir de las cuales crear objetos.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender en qué consiste *el análisis y el diseño en programación orientada a objetos (OO)*:

- ¿Cómo decide qué componentes se necesitan para construir algo, como una casa o un mueble?
- ¿Qué es una clasificación taxonómica?
- ¿Qué relación tienen los organismos en una taxonomía?
- ¿Cuál es la diferencia entre atributos y valores?

Otros recursos

Otros recursos – Los documentos siguientes proporcionan información complementaria sobre los temas tratados en este módulo:

- Larman, Craig. *Applying UML and Patterns – An Introduction to Object Oriented Analysis and Design*. Prentice-Hall, 2004.

  Este libro presenta todo el proceso de análisis y diseño OO en profundidad pero sin excesiva complejidad.
- Eckel, Bruce. *Thinking in Java*. Prentice-Hall, 2006.
- Fowler, Martin and Kendall Scott. *UML Distilled*. Addison Wesley Longman, Inc., 2003.

Estudio de un problema utilizando el análisis orientado a objetos

Caso de estudio – Utilizaremos el siguiente caso para ilustrar los conceptos del análisis y el diseño OO.

DirectClothing, Inc. vende camisas por catálogo. El negocio crece a un ritmo del 30 por ciento anual y necesitan un nuevo sistema de introducción de pedidos. La empresa le ha contratado para diseñar ese nuevo sistema.

DirectClothing produce un catálogo de prendas de vestir cada seis meses y se lo envía a sus suscriptores. Cada prenda del catálogo tiene un identificador de artículo (ID), uno o varios colores (cada uno con un código de color), uno o varios tamaños, una descripción y un precio.

La empresa acepta cheques y las principales tarjetas de crédito.

Para hacer un pedido, los clientes pueden llamar directamente al servicio de atención al cliente de la compañía o enviar a DirectClothing un formulario de pedido por fax o correo.

En la figura siguiente se muestra el formulario de pedido que aparece en el catálogo de DirectClothing.

Figura 2-1 Formulario de pedido del catálogo de DirectClothing

Los pedidos que llegan por correo o fax son introducidos por un miembro del servicio de atención al cliente.

DirectClothing quiere dar al cliente la opción de introducir sus pedidos directamente a través de Internet. Los artículos que aparecen en la web tienen el mismo precio que se marca en el último catálogo.

Cuando se introduce un pedido en el sistema, se verifica la disponibilidad de cada artículo (cantidad en stock). Si uno o varios de los artículos no están disponibles en ese momento (en el almacén de DirectClothing), el pedido se pone en espera hasta que llegan al almacén. Cuando todos los artículos están disponibles, se verifica el pago y el pedido se envía al almacén para su preparación y envío a la dirección del cliente.

Si el pedido se recibe por teléfono, el servicio de atención al cliente da a éste un ID de pedido que se utiliza para hacer el seguimiento de la orden de compra a lo largo del proceso. También le proporciona la extensión de teléfono del representante del servicio de atención al cliente.

Nota – En un verdadero análisis, el programador trabajaría en estrecha colaboración con la compañía para obtener información sobre cada aspecto de su negocio. Este caso sólo representa una pequeña parte de los datos que se necesitarían para crear el sistema adecuado para DirectClothing, Inc.

Identificación del dominio de un problema

Dado que el lenguaje Java es un lenguaje orientado a objetos, uno de los primeros objetivos del programador en esta tecnología es crear objetos para componer un sistema o, más concretamente, para resolver un problema.

El ámbito del problema que se va a resolver se denomina dominio del problema.

La mayoría de los proyectos empiezan por definir el dominio del problema mediante la recopilación de los requisitos del cliente y la escritura de un enunciado donde se especifica el alcance del proyecto, es decir, lo que el programador, usted, quiere conseguir. Por ejemplo, un enunciado del alcance del proyecto de DirectClothing podría ser: “Crear un sistema de introducción de pedidos que permita a los operadores introducir y aceptar el pago de los pedidos”.

Una vez determinado el alcance del proyecto, puede empezar a identificar los objetos que interaccionarán para resolver el problema.

Nota – Esta lección contiene una versión condensada de las nociones sobre análisis y diseño. Para los alumnos que deseen profundizar en esta materia, el siguiente curso de Sun Microsystems se concentra en estas fases del ciclo de vida de los productos: OO-226: *Análisis -y Diseño OO con UML*

Identificación de los objetos

Para validar los objetos del dominio de un problema, es preciso identificar primero las propiedades de todos los objetos:

- Los objetos pueden ser físicos o conceptuales: una cuenta de un cliente es un ejemplo de objeto conceptual, porque no es algo que se pueda tocar físicamente. Un cajero automático es algo que muchas personas tocan al cabo del día y es un ejemplo de objeto físico.
- Los objetos poseen atributos (características) tales como el tamaño, el nombre o la forma, entre otros. Por ejemplo, un objeto puede tener el color como atributo.

  El valor de todos los atributos de un objeto a menudo se conoce como el estado actual del objeto. Por ejemplo, un objeto puede tener un atributo de color con el valor "rojo" y un atributo de tamaño con el valor "grande".
- Los objetos pueden realizar operaciones (cosas que pueden hacer), tales como configurar un valor, abrir una pantalla o incrementar la velocidad. Las operaciones suelen afectar a los atributos del objeto y, a menudo, nos referimos a ellas como el comportamiento de los objetos. Así, un objeto puede llevar asociada una operación que permita a otros objetos cambiar su atributo de color de un estado a otro, por ejemplo, de rojo a azul.

¿Lo sabía? – Los objetos se designan mediante nombres como, por ejemplo, "cuenta" o "camisa". Los atributos de los objetos también suelen ser nombres, como "color" o "tamaño". Sin embargo, las operaciones de los objetos suelen designarse mediante verbos o combinaciones de nombre y verbo, como es el caso de "mostrar" o "enviar pedido".

La capacidad para reconocer los objetos en el mundo que le rodea ayudará al programador a definir mejor sus objetos cuando plantee un problema utilizando el análisis OO.

La figura siguiente ilustra las características de una ballena que la convierten en un objeto.

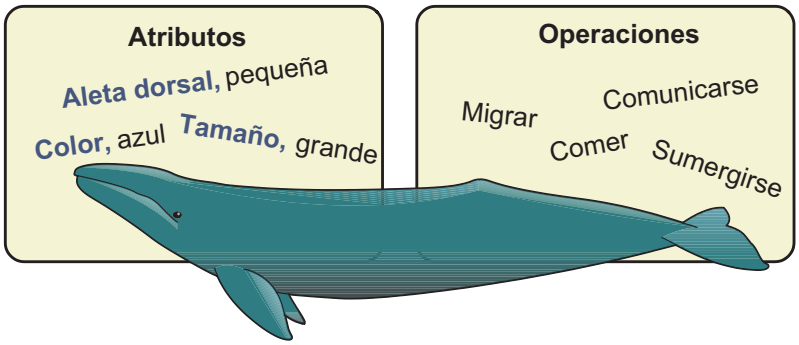


Figura 2-2 Objeto físico con atributos y operaciones

Discusión – Eche un vistazo a la sala. ¿Qué objetos está viendo en este momento?

Por ejemplo, una puerta puede ser un objeto en el dominio del problema de "construir una casa". Una puerta incluye al menos un atributo, que puede tener valores (abierta o cerrada), y una operación, por ejemplo "cerrar puerta" o "abrir puerta", lo que permite cambiar el estado de la puerta.

¿Lo sabía? – Un atributo que sólo puede tener dos estados se conoce como booleano (*Boolean*).

Otros criterios para reconocer objetos

Utilice los criterios siguientes para determinar si algo debería ser considerado como un objeto en el dominio de un problema:

- Relevancia para el dominio del problema
- Existencia independiente

Relevancia para el dominio del problema

Para averiguar si el objeto es relevante para el dominio del problema, pregúntese lo siguiente:

- ¿El objeto existe dentro de los límites del dominio?
- ¿El objeto es necesario para que la solución sea completa?
- ¿Es necesario como parte de una interacción entre un usuario y la solución?

Nota – Algunos elementos del dominio de un problema pueden ser atributos de objetos u objetos en sí. Por ejemplo, la temperatura puede ser un atributo de un objeto en un sistema médico o un objeto en un sistema científico que lleve el control de patrones atmosféricos.

Existencia independiente

Para que un elemento sea un objeto y no un atributo de otro objeto, debe ser independiente en el contexto del dominio del problema. Varios objetos pueden estar relacionados y, sin embargo, tener una existencia independiente. En el caso de DirectClothing, el cliente y el pedido están conectados, pero son independientes entre sí, por lo que ambos serían objetos.

Al evaluar la utilización de objetos potenciales, pregúntese si el objeto debe existir de forma independiente o ser atributo de otro objeto.

Nota – Identificar los objetos de un dominio es un arte, no una ciencia. Cualquier objeto puede ser *válido* si es relevante para el dominio de un problema y posee las características de un objeto, pero eso no significa que sea un *buen* objeto. En cualquier caso, la persona que modele el sistema o la solución debe tener una comprensión global de la totalidad del sistema.

Discusión – ¿Cuáles son los posibles objetos de DirectClothing, Inc.?

¿Los objetos del caso de DirectClothing cumplen los criterios explicados en esta sección?

En la figura siguiente se ilustran tres objetos pertenecientes al dominio del sistema de introducción de pedidos de DirectClothing.

Figura 2-3 Posibles objetos del caso de DirectClothing

Identificación de los atributos y operaciones de los objetos

Después de identificar los objetos, debe especificar sus atributos y operaciones.

Como indicábamos anteriormente, los atributos definen el estado de un objeto. Los atributos pueden ser datos, como el ID de pedido y el ID de cliente del objeto `Order` (Pedido), o bien pueden ser otro objeto, como sería el caso de un cliente que, en lugar de tener únicamente el ID de pedido como atributo, tuviese todo el objeto `Order`.

Como decíamos, las operaciones son comportamientos que normalmente modifican el estado de un atributo. Por ejemplo, es posible imprimir un pedido, agregarle o suprimirle un artículo, etc. (El cliente o el operador del centro de atención al cliente iniciaría esas acciones en la vida real, pero pertenecen al objeto `Order`.)

Atributos como referencias a otros objetos

Un atributo puede hacer referencia a otro objeto. Por ejemplo, el objeto `Customer` (Cliente) puede tener un atributo que sea un objeto `Order`. Esta asociación podría ser necesaria o no, en función del problema que se esté tratando de resolver.

Nota – Utilice nombres de atributos y operaciones que describan con claridad el propósito del atributo o la operación.

En la figura siguiente puede verse cómo el objeto `Customer` contiene un atributo `Order`.

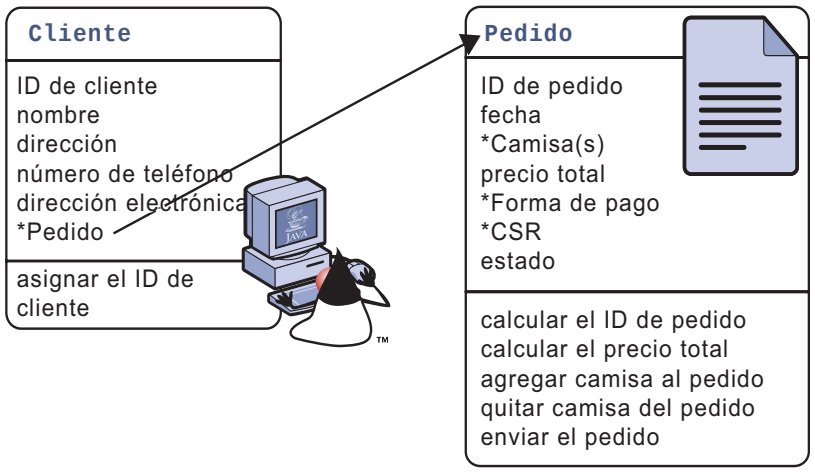


Figura 2-4 Objeto con otro objeto como atributo

Nota – Los asteriscos (*) de los gráficos anteriores indican atributos que son otros objetos.

Discusión – ¿Cuáles son los posibles atributos y operaciones de los objetos en el caso de DirectClothing?

¿Alguno de los atributos que ha elegido hace referencia a otro objeto?

En la figura siguiente se ilustran algunos posibles atributos y operaciones de los objetos `Order`, `Shirt` y `Customer`.

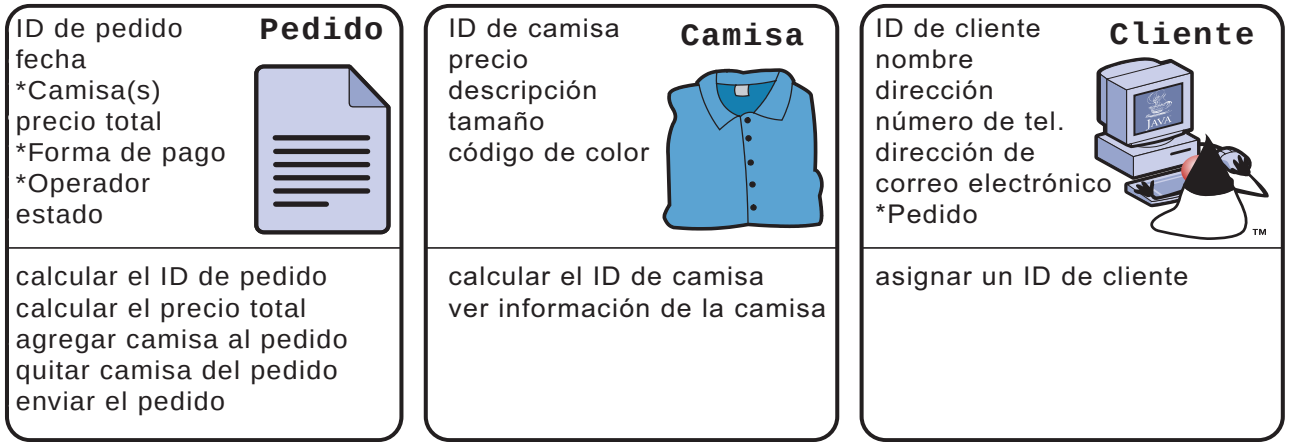


Figura 2-5 Posibles atributos y operaciones para los objetos del caso de DirectClothing, Inc.

Solución del caso de estudio

A continuación se incluyen una serie de cuadros que contienen una lista parcial de los objetos (incluidos la mayoría de sus atributos y operaciones) identificados para el ejemplo de DirectClothing.

`Pedido`	`Camisa`
ID de pedido fecha *Camisa(s) precio total *Forma de pago *Operador estado	ID de camisa precio descripción tamaño código de color
calcular ID de pedido calcular el precio total agregar camisa al pedido quitar camisa del pedido enviar el pedido	calcular el ID de camisa ver información de la camisa

Figura 2-6 Objetos `Pedido` y `Camisa`

`Cliente`	`Forma de pago`
ID de cliente nombre dirección número de teléfono dirección de correo electrónico *Pedido	número de cheque número de tarjeta de crédito fecha de vencimiento
asignar un ID de cliente	verificar el número de tarjeta de crédito verificar el pago del cheque

Figura 2-7 Objetos `Cliente` y `Pago`

Catálogo	Operador
*Camisa(s)	nombre extensión
agregar una camisa quitar una camisa	

Figura 2-8 Objetos Catálogo y Operador

Nota – Aunque están implícitos, no todos los atributos y operaciones de la solución se han definido explícitamente en el caso de estudio. Por ejemplo, el objeto Pago tiene un atributo de fecha de vencimiento para indicar cuándo caduca la tarjeta de crédito.

Diseño de clases

La identificación de objetos ayuda a diseñar la clase o el prototipo de cada uno de los objetos de un sistema. Por ejemplo, los fabricantes de ventanas suelen crear un modelo de cada uno de los estilos de ventana que producen. En estos modelos se definen una serie de colores y estilos que pueden elegirse al adquirir la ventana.

Asimismo, estos modelos sirven de base para generar cualquier cantidad de ventanas con cualquier combinación de color y estilo. En términos del diseño orientado a objetos, cada objeto (ventana) creado utilizando la clase (prototipo genérico) se denomina instancia de una clase. En concreto, cada objeto creado a partir de una clase puede tener un estado específico (valores) para cada uno de sus atributos, pero tendrá los mismos atributos y operaciones.

Nota – El diccionario *American Heritage* define el término *clase* como "un grupo cuyos miembros tienen ciertos atributos en común".

Las clases y objetos se utilizan con frecuencia en el campo de la biología. Por ejemplo, a los biólogos marinos que estudian las criaturas de los mares y océanos a menudo se les pide que clasifiquen estas criaturas por familias o clases. En términos del análisis y el diseño OO, cada animal (por ejemplo, una ballena azul) de una familia como puede ser la de las ballenas se considera una instancia (objeto) de la clase ballena.

En la figura siguiente se ilustran dos instancias de la clase ballena.

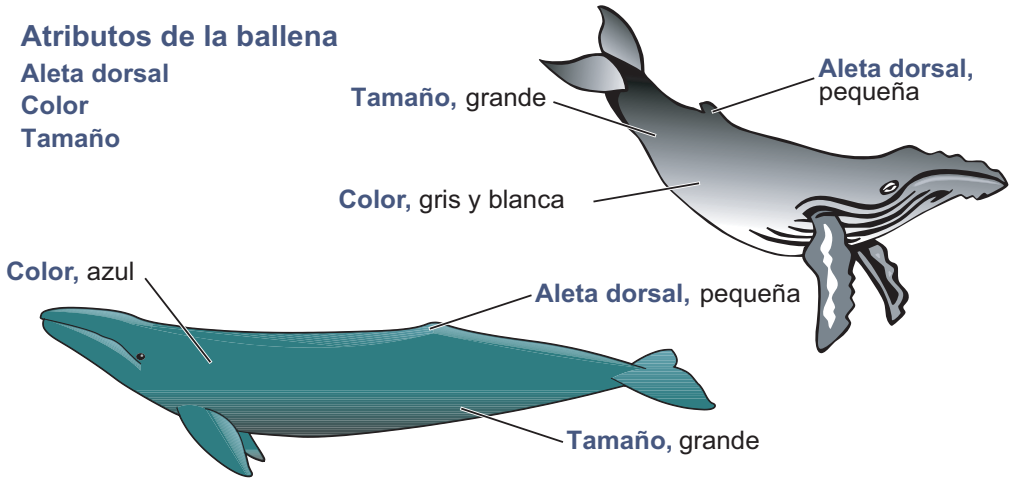


Figura 2-9 Diferentes tipos de ballenas

En lo que se refiere a DirectClothing:

- Una clase es la definición de un objeto. Las clases son categorías descriptivas, plantillas o prototipos. Shirt (Camisa) podría ser una clase que define a todas las camisas como objetos poseedores de un ID de camisa, un tamaño, un código de color, una descripción y un precio
- Los objetos son instancias únicas de las clases. La camisa azul grande que cuesta 29,99 dólares y tiene el ID de camisa 62467-B es una *instancia* de la clase `Shirt`, al igual que la camisa verde pequeña etiquetada con el mismo precio y con el ID 66889-C, o la camisa de dibujos de 39,99 dólares con ID 09988-A. Es posible tener en la memoria dos objetos `Shirt` con atributos que tengan exactamente los mismos valores.

En la figura siguiente puede verse una clase y varios objetos basados en ella.

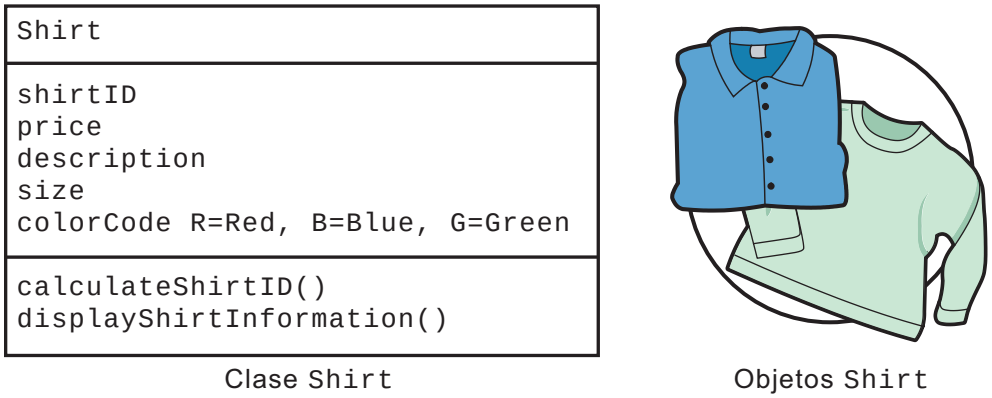


Figura 2-10 Una clase y los objetos resultantes

Discusión – Eche un vistazo a la sala. ¿Qué clases identifica al mirar los objetos que le rodean en este momento?

Pista: algunos nombres que ha utilizado para identifica objetos, como “puerta”, podrían servir para designar una clase de objetos. En otros casos, puede haber identificado objetos como, por ejemplo, “profesor”, que serían una instancia de una clase, en este caso la clase “persona”.

¿Lo sabía? – Cada clase tiene otra clase de la que procede. En la terminología del análisis y diseño OO, esa clase de procedencia de denomina *superclase*. Por ejemplo, la ballena puede proceder de la superclase "mamíferos". Las superclases se explican más adelante en este curso.

Discusión – ¿Cuáles son las posibles clases de los objetos identificados para el caso de DirectClothing?

Nota – A lo largo de todo el curso seguiremos volviendo al ejemplo de la clase `Shirt`.

En programación Java, los atributos se representan mediante variables, y las operaciones se representan mediante métodos. Las variables son el mecanismo que utiliza el lenguaje Java para contener los datos. Los métodos son el mecanismo que utiliza para realizar una operación.

Nota – Además de los datos de atributos, las variables pueden contener otros datos tales como los valores utilizados exclusivamente en el interior de algún método.

Modelado de las clases

La primera parte de la fase de diseño consiste en organizar o modelar visualmente un programa y sus clases. Cada clase de un diseño debería modelarse de manera que quede incluida en un recuadro con el nombre de la clase en la parte superior seguido de una lista de variables de atributos (incluido el rango de valores posibles) y una lista de métodos. La sintaxis para modelar una clase se muestra en la figura siguiente.

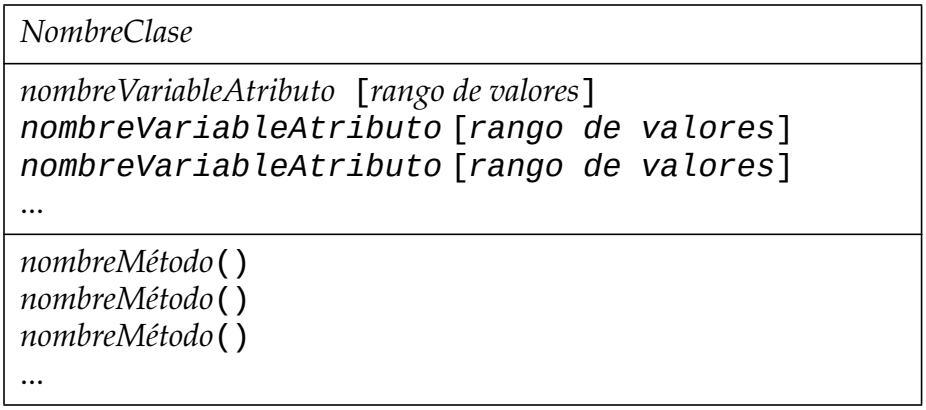


Figura 2-11 Sintaxis para modelar una clase

Donde:

- `NombreClase` es el nombre de la clase.
- `nombreVariableAtributo` es el nombre de la variable correspondiente a un atributo.
- *`rango de valores`* es una serie opcional de valores que el atributo puede contener.
- *`nombreMétodo`* es el nombre de un método.

Por ejemplo, la figura siguiente contiene el modelado de un objeto `Shirt`.

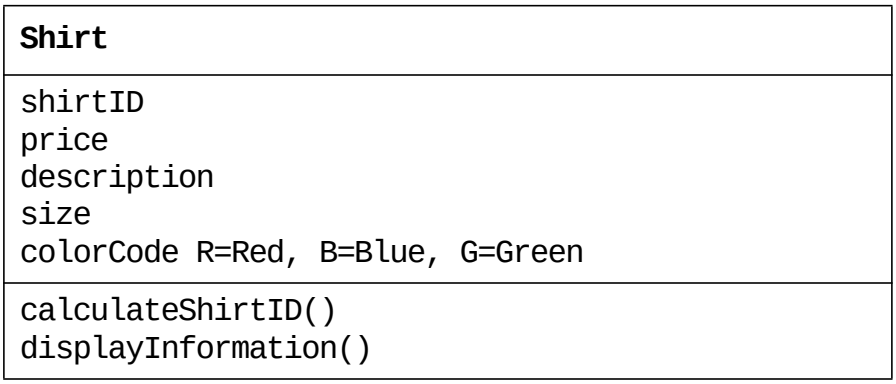


Figura 2-12 Objeto `Shirt` modelado

Nota – Esta técnica de modelado se basa ligeramente en una versión simplificada del sistema de notación UML (Unified Modeling Language), una herramienta utilizada en procesos de modelado (algunos detalles se han suprimido para los programadores con menos experiencia).

Los nombres de métodos y variables se escriben utilizando una forma especial de escritura abreviada de tal manera que, si hacen referencia a un atributo u operación formados por varias palabras, la primera palabra se escribe en minúscula y la inicial de las sucesivas palabras se escribe en mayúscula. Por ejemplo, una operación como “calcular el precio total” se escribiría calcPrecioTotal() o, en inglés, `calcTotalPrice()`. Asimismo, la existencia de paréntesis de apertura y cierre indica que se trata de un método.

Nota – El modelado de clases es similar al modelado de estructuras de bases de datos. De hecho, los objetos creados pueden almacenarse en una base de datos utilizando el API JDBC™ (Java Database Connectivity™). Este API permite leer y escribir registros utilizando sentencias SQL (Structured Query Language) dentro de los programas basados en la tecnología Java.

Módulo 3

Desarrollo y comprobación de un programa Java

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Identificar los cuatro componentes de una clase en el lenguaje de programación Java.
- Usar el método `main` en una clase de prueba (test) para ejecutar un programa Java desde la línea de comandos.
- Compilar y ejecutar un programa Java.

Este módulo proporciona una descripción general de los componentes de una clase. También incluye una explicación sobre la forma de compilar y ejecutar un programa Java compuesto de varias clases.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Identificar los cuatro componentes de una clase en el lenguaje de programación Java.			
Usar el método `main` en una clase de prueba (test) para ejecutar un programa Java desde la línea de comandos.			
Compilar y ejecutar un programa Java.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Notas

Aspectos relevantes

Discusión – La pregunta siguiente es relevante para comprender los conceptos de desarrollo y comprobación de clases:

¿Cómo puede comprobar si algo que ha construido, como una casa, un mueble o un programa, está bien construido?

Otros recursos

Otros recursos – Los documentos siguientes proporcionan información complementaria sobre los temas tratados en este módulo:

- Farrell, Joyce. *Java Programming: Comprehensive*. 1999.

  Es un libro excelente para no programadores. Explica conceptos que no se tratan con tanta profundidad en otros libros más avanzados.

- Tutorial de Java. [En la web]. Disponible en:
  `http://java.sun.com/docs/books/tutorial/index.html`

Identificación de los componentes de una clase

Las clases son los prototipos o patrones que se crean para definir los objetos de un programa. Por ejemplo, en la figura siguiente se ilustran algunos objetos que podrían utilizarse en el programa de introducción de pedidos desarrollado para DirectClothing, Inc.

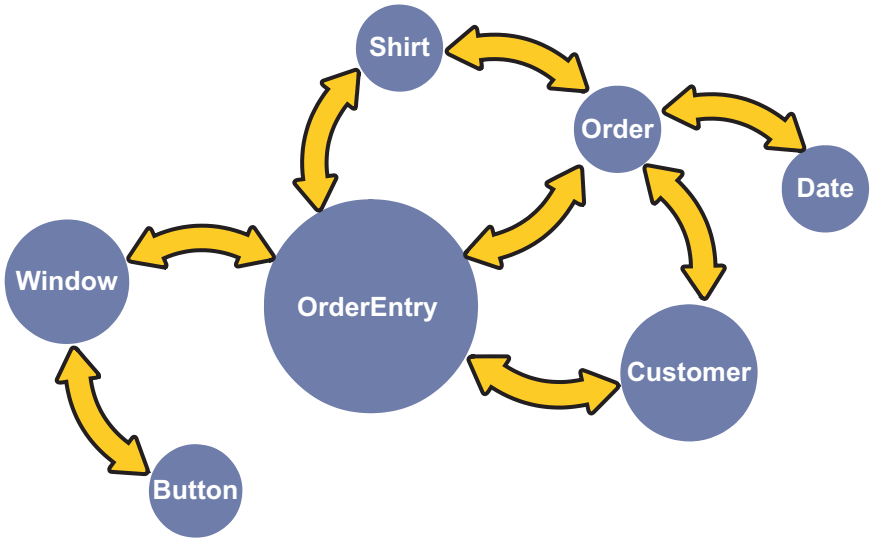


Figura 3-1 Interacción de objetos en el caso de estudio de DirectClothing, Inc.

Una aplicación de escritorio normalmente se compone de un objeto, a menudo llamado objeto de control, objeto principal u objeto de prueba, que es el punto de partida para iniciar el programa. En la figura anterior, un objeto `OrderEntry` (EntradaPedido) podría interaccionar con uno o varios objetos `Window`, `Customer`, `Order` u otros objetos mientras se ejecuta el programa.

Cada objeto de esta figura es una instancia de un prototipo o clase. Por ejemplo, todos los objetos `Window` (Ventana) son instancias de la clase `Window`. Algunas clases, como `Window` (usada para crear ventanas en interfaces gráficas de usuario), son clases de uso general y se suministran como parte del API de Java. Otras clases, como `Shirt`, pueden ser exclusivas de su programa, así que debe crearlas personalmente. En este curso se explica cómo usar las clases existentes y cómo crear y usar sus propias clases.

Estructuración de las clases

Las clases se componen del código Java necesario para instanciar objetos tales como `Shirt`. Este curso divide el código de cada archivo de clase de Java en cuatro secciones distintas:

- La declaración de la clase
- La declaración e inicialización de las variables de atributo (opcional)
- Los métodos (opcional)
- Los comentarios (opcional)

¿Lo sabía? – Una clase no tiene por qué contener métodos y atributos de forma simultánea.

El código de programación de cada clase está contenido en un archivo de texto que debe mantener una cierta estructura. El ejemplo siguiente muestra una clase `Shirt` para todas las camisas que aparecerán en el catálogo de DirectClothing. La clase `Shirt` contiene varias variables de atributo y un método para mostrar en la pantalla los valores de las variables.

Código 3-1 Archivo `Shirt.java` situado en el directorio `getstarted`

```
1   public class Shirt {
2
3     public int shirtID = 0; // ID predeterminado para la camisa
4     public String description = “-description required-”; //
predeterminada
5     // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
6     public char colorCode = ‘U’;
7     public double price = 0.0; // Precio predeterminado para todas las
camisas
8     public int quantityInStock = 0; // Cantidad predeterminada para
todas las camisas
9
10    // Este método muestra los valores de un artículo
11    public void displayInformation() {
12      System.out.println(“ID de camisa: “ + shirtID);
13      System.out.println(“Descripción de la camisa :” + description);
14      System.out.println(“Código de color: “ + colorCode);
15      System.out.println(“Precio de la camisa: “ + price);
16      System.out.println(“Cantidad en stock: “ + quantityInStock);
17
18    } // fin del método display
19  } // fin de la clase
```

Nota – El atributo `quantityInStock` (cantidadEnStock) no aparecería en un catálogo, pero se utilizaría para llevar el control de las unidades que quedan de una determinada camisa en el almacén.

Nota – La palabra clave `public` de Java se utiliza abundantemente en este ejemplo de código. En la página siguiente se ofrece una explicación sobre el uso de esta palabra. De momento, puede olvidarse de ella y centrarse en la estructura de la definición de la clase.

Declaración de clases

Es preciso declarar una clase por cada clase diseñada en el dominio del problema. Cada clase necesita su propia declaración. La sintaxis para declarar una clase es:

```
[modificadores] class identificador_clase
```

Donde:

- Los `[modificadores]` determinan el nivel de acceso que otras clases tienen a esta clase. Los modificadores se explican con mayor profundidad más adelante en este curso. La variable `[modificadores]` es opcional (como indican los corchetes) y puede ser `public`, `abstract` o `final`. Por ahora, utilice el modificador `public`.
- La palabra clave `class` indica al compilador que el bloque de código contiene una declaración de clase. Las palabras clave son palabras reservadas por el lenguaje Java para determinadas construcciones.
- *`identificador_clase`* es el nombre asignado a la clase. La nomenclatura utilizada para las clases sigue las normas siguientes:
  - Los nombres de clases deben ser sustantivos escritos en letras mayúsculas y minúsculas, con la primera letra de cada palabra en mayúscula, por ejemplo `MiClase`.
  - Los nombres de clases deben contener palabras completas. Debe evitarse el uso de siglas y abreviaturas, a menos que la sigla sea más conocida y utilizada que el nombre completo, como es el caso de JVM o UML.

La declaración de clase (línea 1) del código de la clase `Shirt` anterior incluye un modificador de tipo `public` seguido de la palabra clave `class` y del nombre de la clase, `Shirt`.

Requisitos del archivo fuente

En este curso, desarrollará sus clases de forma que el código de Java que escriba para cada clase esté contenido en su propio archivo de texto o archivo de código fuente. En lenguaje Java, el nombre de cada archivo de código fuente *debe* ser idéntico al nombre de la clase public que contiene el archivo y debe ir seguido de la extensión `.java`. Por ejemplo, la clase `Shirt` debe guardarse en un archivo llamado `Shirt.java`.

Autoevaluación – Entre las declaraciones siguientes, seleccione aquellas que se ajusten a la nomenclatura propia de las clases.

a. ___ `class Shirt`

b. ___ `public Class 501Pants`

c. ___ `public Shirt`

d. ___ `public Class Pants`

e. ___ `class ShirtAndPants`

La definición de la clase va seguida de una llave de apertura ({) que indica el comienzo del *`cuerpo_de_clase`*, las variables de atributo y los métodos que componen la clase. Las llaves `{ }` que abren y cierran el *`cuerpo_de_clase`* indican el principio y el final de la clase.

Declaraciones y asignaciones de variables

La primera llave de apertura ({) va seguida del bloque de declaración y asignación de variables de atributo. En general, todas las variables de atributo de la clase se definen después de esta llave.

El ejemplo de código de la clase `Shirt` contiene cinco declaraciones de variables de atributo, una para el ID de camisa (línea 3), otra para la descripción (línea 4), otra para el código de color (línea 6), otra para el precio (línea 7) y otra para la cantidad en stock (línea 8).

Nota – Las líneas del programa que realizan alguna acción, como imprimir o declarar variables, incluyen un signo de punto y coma al final. Las líneas del programa que representan el comienzo de un bloque de código (una o más líneas de código combinadas para realizar una determinada tarea, como las de una clase o un método) se inician con una llave de apertura ({) y se terminan con una llave de cierre (}). En los ejemplos de sintaxis y código de este curso se muestra cuándo usar cada una de ellas.

Comentarios

Debería poner comentarios en cada clase que cree para facilitar la comprensión de aquello que hace el programa. Los comentarios son particularmente importantes en programas de gran tamaño desarrollados por muchos programadores diferentes que deben leer el código. Contribuyen a realizar un buen mantenimiento del programa cuando los nuevos programadores necesitan saber qué está haciendo el código.

Estructuras de los comentarios

Es posible utilizar dos estilos de comentarios:

- Comentarios en una línea**:** un marcador formado por dos barras inclinadas `//` indica al compilador que debe hacer caso omiso de todo lo que encuentre hasta el final de la línea actual. Por ejemplo, en las líneas 3, 4 y 5 de la clase `Shirt` contiene comentarios de una sola línea:

```
public int shirtID = 0; // ID predeterminado para la camisa
public double price = 0.0; // Precio predeterminado para todas las
camisas

// Los códigos de color son R=Rojo, B=Azul, G=Verde
```

  Muchos programadores tratan de facilitar la comprensión de sus códigos utilizando comentarios de una línea para explicar la primera y última líneas de cada clase y método. Por ejemplo, la clase `Shirt` contiene un comentario en la última línea para indicar el fin del método display (línea 18):

```
} // fin del método display
```

  En programas largos, puede ser muy complicado encontrar las llaves de cierre de la clase. Comentar la estructura a la que pertenece cada llave de cierre facilita considerablemente la lectura y corrección de errores del código.

- Comentarios tradicionales**:** la combinación de los caracteres `/*` indica al compilador que haga caso omiso de *todo* el código que encuentre hasta el siguiente marcador de cierre (`*/`). Los programadores suelen utilizar los comentarios tradicionales para proporcionar información sobre un código de bloque largo.

```
/***************************************************
* Sección de declaración de variables de atributo  *
***************************************************/
```

¿Lo sabía? – Hay un tercer tipo de comentario denominado comentario de documentación. Es posible utilizar la herramienta Javadoc™ de la tecnología Java para crear documentación relativa a cualquiera de las clases que vayan a utilizar otros programadores. De hecho, toda la documentación de la biblioteca de clases que se entrega con el SDK de Java SE se ha creado con la herramienta `javadoc`. Los comentarios de documentación deben empezar con una barra inclinada y dos asteriscos (`/**`), y terminar con un asterisco y una barra inclinada (`*/`). El ejemplo anterior de comentario tradicional también sería válido como comentario de documentación.

Métodos

Los métodos van a continuación de las declaraciones de variables en la clase. Su sintaxis es la siguiente:

```
[modificadores] tipo_retorno identificador_método
([argumentos]) {
    bloque_código_método
}
```

Donde:

- `[modificadores]` representa determinadas palabras clave de Java que modifican la forma en que se accede a los métodos. Los modificadores son opcionales (como indican los corchetes).
- *`tipo_retorno`* indica el tipo de valor que devuelve el método (si devuelve alguno). Si devuelve algún valor, es preciso declarar el tipo de valor del que se trata. Los valores de retorno pueden ser utilizados por el método de llamada. Un método puede devolver un valor como máximo. Si no devuelve ningún valor, es preciso usar la palabra clave `void` como `tipo_retorno`.
- *`identificador_método`* es el nombre del método.
- `([argumentos])` representa una lista de variables cuyos valores se pasan al método para que éste los utilice. Los argumentos son opcionales (como indican los corchetes) porque los métodos no están obligados a aceptarlos. Recuerde también que los paréntesis no son opcionales. Un método que no acepta argumentos se declara con paréntesis de apertura y cierre sin información dentro.

- *bloque_código_método* es una secuencia de sentencias ejecutadas por el método. En el bloque o cuerpo del código de un método pueden tener lugar una amplia variedad de tareas.

La clase `Shirt` contiene un método, `displayInformation` (líneas 11-18), que muestra en la pantalla los valores de los atributos de una camisa.

```
public void displayInformation() {

    System.out.println("ID de camisa: " + shirtID);
    System.out.println("Descripción de la camisa :" + description);
    System.out.println("Código de color: " + colorCode);
    System.out.println("Precio de la camisa: " + price);
    System.out.println("Cantidad en stock: " + quantityInStock);

  } // fin del método display
```

Autoevaluación – Establezca las correspondencias adecuadas entre cada bloque de código y su ubicación en una clase.

Ejemplo de código	Definición
`double precio = 0;` `int ID = 0;`	Cuerpo del método
`void showMe()`	Comentario tradicional
`public class Camisa`	Declaración e inicialización de una variable de atributo
`/* Las siguientes líneas de código presentan el resultado del método display */`	Declaración de un método
	Comentario de una línea
	Argumento
	Declaración de una clase

Creación y uso de una clase de prueba

La mayoría de las clases que creará en este curso no pueden utilizarse (ejecutarse y probarse) por sí solas. En realidad, necesitará ejecutar otra clase para crear una instancia de objeto de su clase y poder así verificarla.

A lo largo del curso, utilizará una clase de prueba (test) o main para verificar cada una de sus clases. El código siguiente contiene un ejemplo de una clase de prueba de la clase `Shirt`.

Código 3-2 Archivo `ShirtTest.java` situado en el directorio `getstarted`

```
1   public class ShirtTest {
2
3      public static void main (String args[]) {
4
5      Shirt myShirt;
6      myShirt = new Shirt();
7
8      myShirt.displayInformation();
9
10
11     }
12  }
13
```

Cada clase de prueba del curso debería designarse con un nombre que permita reconocerla como tal. En concreto, su nombre debería incluir el nombre de la clase que va a probar seguido de la palabra "Test". Por ejemplo, la clase diseñada para verificar la clase `Shirt` se llama `ShirtTest`.

Las clases de prueba deben realizar dos tareas diferentes. Son las siguientes:

- Proporcionar un punto de partida, a través del método `main`, para el programa.
- Crear una instancia de su clase y probar sus métodos.

Método `main`

`main` es un método especial que la máquina virtual de Java reconoce como punto de partida para cualquier programa Java que se ejecute desde un indicador o una línea de comandos. Cualquier programa que se quiera ejecutar desde una línea de comandos o un indicador debe tener un método `main`.

¿Lo sabía? – Muchas de las clases de Java que crean los ingenieros no se ejecutan en un sistema operativo. ¿Recuerda los applets? Estas miniaplicaciones se ejecutan en los navegadores y tienen su propio método de inicio.

La sintaxis del método `main` es:

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

El método `main` utiliza la misma sintaxis que todos los métodos descritos con anterioridad. En concreto:

- El método `main` contiene dos modificadores obligatorios, `public` y `static`.
- El método `main` no devuelve ningún valor, con lo que su tipo de retorno es `void`.
- El método `main` tiene como identificador (nombre) "`main`".
- El método `main` acepta cero o más objetos de tipo `String` (`String args[]`). Esta sintaxis permite introducir en la línea de comandos los valores que el programa utilizará durante su ejecución.

Nota – El nombre del array (`args`) aceptado por el método `main` puede cambiarse. La sintaxis `public static void main(String[] args)` también es aceptable. Cada clase puede contener un número cualquiera de métodos main sobrecargados, incluido `public static void main(String [] args)`. El comando `java nombreclase` sólo llamará directamente a un método.

Notas

Compilación y ejecución (comprobación) de un programa

Hemos visto una clase básica de Java llamada `Shirt` y una clase de prueba llamada `ShirtTest`. Juntas, estas clases forman su primer programa de tecnología Java. A continuación, compilará y ejecutará (comprobará) el programa.

Nota – Su sistema debe estar adecuadamente configurado para poder compilar y ejecutar programas Java. El Apéndice A, "Siguientes pasos", contiene información para ayudarle a configurar el sistema.

Compilación de un programa

La compilación convierte los archivos de clase del programador en código de byte que puede ejecutarse mediante una máquina virtual de Java. Recuerde las reglas para asignar nombres a los archivos fuente de Java. Si un archivo fuente contiene una clase pública, dicho archivo debe tener el mismo nombre que la clase seguido de la extensión `.java`. Por ejemplo, la clase `Shirt` debe guardarse en un archivo llamado `Shirt.java`.

Para compilar los archivos de código fuente de `Shirt` y `ShirtTest`, deberá:

1. Ir al directorio donde estén almacenados los archivos de código fuente.
2. Introducir el siguiente comando por cada archivo `.java` que quiera compilar:

   `javac` *`nombrearchivo`*

   Por ejemplo:

   ```
   javac Shirt.java
   ```

   Una vez finalizada la compilación, y suponiendo que no se haya producido ningún error de compilación, debería tener en el directorio un nuevo archivo llamado *`nombreclase`*`.class` por cada archivo fuente que haya compilado. Si compila una clase que hace referencia a otros objetos, las clases correspondientes a esos objetos también se compilan (si no se han compilado ya). Por ejemplo, si ha compilado el archivo `ShirtTest.java` (que hace referencia al objeto `Shirt`), tendrá los archivos `Shirt.class` y `ShirtTest.class`.

Nota – En equipos que utilicen Solaris, para cualquier archivo de texto con extensión `.java` se creará un icono en el Gestor de archivos. Puede utilizar el Gestor de archivos para seleccionar y compilar un archivo de código fuente haciendo clic en el archivo fuente, presionando el botón derecho del ratón y seleccionando Compile en el menú emergente.

Ejecución (comprobación) de un programa

Una vez compilados adecuadamente sus archivos de código fuente, puede ejecutarlos y probarlos utilizando la máquina virtual de Java. Para ejecutar y probar su programa:

1. Vaya al directorio donde estén almacenados los archivos de clase.
2. Introduzca el comando siguiente para el archivo de clase que contenga el método `main`:

   `java` *`nombreclase`*

   Por ejemplo:

   `java ShirtTest`

   Este comando ejecuta la clase `ShirtTest`. Como hemos indicado anteriormente, la clase `ShirtTest` crea una instancia del objeto `Shirt` utilizando la clase `Shirt`. Todos los objetos `Shirt` tienen un método, `display`, que imprime en la pantalla los valores de sus variables de atributo, por ejemplo:

```
ID de camisa: 0
Descripción de la camisa:-description required-
Código de color: U
Precio de la camisa: 0.0
Cantidad en stock: 0
```

Nota – Ejecución y comprobación a menudo son sinónimos y representan la cuarta fase del ciclo de vida de los productos.

Sugerencias para corregir errores de sintaxis en Java

Casi siempre tendrá algún error de sintaxis cuando escriba código nuevo en Java. La sintaxis se refiere a las reglas que determinan la estructura correcta de un lenguaje. Utilice los consejos siguientes cuando vaya a corregir la sintaxis de sus programas Java:

- Los mensajes de error indican el número de la línea donde se ha producido el error. Sin embargo, la línea indicada no siempre es el verdadero origen del error, por lo que quizá tenga que comprobar las líneas de código que preceden y siguen al número de línea suministrado.
- Compruebe si ha escrito el punto y coma de cierre al final de cada sentencia.
- Asegúrese de tener un número par de llaves: debe haber una llave de cierre por cada llave de apertura.
- Asegúrese de haber usado las sangrías de forma coherente y uniforme en el programa, tal y como se muestra en los ejemplos del curso. Aunque los programas se ejecutan sin tener en cuenta las sangrías en diferentes tipos de código, su uso facilita la lectura y depuración de los programas, lo que, a su vez, ayuda a evitar errores durante su escritura.

Módulo 4

Declaración, inicialización y uso de variables

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Identificar la sintaxis y los usos de las variables.
- Enumerar los ocho tipos de datos primitivos del lenguaje Java.
- Declarar, inicializar y usar variables y constantes siguiendo las directrices y normas de codificación del lenguaje Java.
- Modificar los valores de las variables utilizando operadores.
- Usar la promoción y conversión de tipos.

En este módulo se explica cómo crear *variables* utilizando tipos de datos primitivos y la forma de modificar sus valores utilizando operadores, la promoción y la conversión de tipos.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Identificar la sintaxis y los usos de las variables.			
Enumerar los ocho tipos de datos primitivos del lenguaje Java.			
Declarar, inicializar y usar variables y constantes siguiendo las directrices y normas de codificación del lenguaje Java.			
Modificar los valores de las variables utilizando operadores.			
Usar la promoción y conversión de tipos.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Notas

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender los conceptos sobre las variables:

- Una variable se refiere a algo que puede cambiar. Las variables pueden contener un valor o un conjunto de valores. ¿Dónde ha visto variables con anterioridad?
- ¿Qué tipos de datos cree que pueden contener las variables?

Otros recursos

Otros recursos – Los documentos siguientes pueden proporcionar información complementaria sobre los temas tratados en este módulo:

- Farrell, Joyce. *Java Programming: Comprehensive*. 1999.

  Es un libro excelente para no programadores. Explica conceptos que no se tratan con tanta profundidad en otros libros más avanzados.

- Tutorial de Java. [En la web]. Disponible en: `http://java.sun.com/docs/books/tutorial/index.html`

Identificación de la sintaxis y el uso de las variables

Las variables se utilizan para almacenar y recuperar datos utilizados por los programas.

El siguiente ejemplo de código contiene una clase `Shirt` que declara varias variables de atributo.

Código 4-1 Archivo `Shirt.java` situado en el directorio `get_started`

```
1   public class Shirt {
2
3     public int shirtID = 0; // ID predeterminado para la camisa
4
5     public String description = “-description required-”; //
predeterminada
6
7     // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
8     public char colorCode = ‘U’;
9
10    public double price = 0.0; // Precio predeterminado para todas las
camisas
11
12    public int quantityInStock = 0; // Cantidad predeterminada para
todas las camisas
13
14    // Este método muestra los valores de un artículo
15    public void displayInformation() {
16
17      System.out.println(“ID de camisa: “ + shirtID);
18      System.out.println(“Descripción de la camisa :” + description);
19      System.out.println(“Código de color: “ + colorCode);
20      System.out.println(“Precio de la camisa: “ + price);
21      System.out.println(“Cantidad en stock: “ + quantityInStock);
22
23     } // fin del método display
24
25  } // fin de la clase
```

Las variables de atributo (variables declaradas fuera de un método y sin la palabra clave `static`) también se denominan variables miembro o variables de instancia (por ejemplo, `price`, `shirtID` y `colorCode` en la clase `Shirt`). Cuando se instancia un objeto a partir de una clase, estas variables contienen datos específicos de la instancia. Cuando se instancia un objeto de una clase, estas variables se denominan variables de instancia porque pueden contener datos específicos de una determinada instancia de la clase. Por ejemplo, una instancia de la clase `Shirt` puede tener el valor 7 asignado a la variable de atributo `quantityInStock`, mientras que otra instancia de la clase `Shirt` puede tener el valor 100 asignado a la variable de atributo `quantityInStock`.

También es posible definir variables dentro de los métodos. Estas variables se denominan locales porque sólo están disponibles en el interior del método en el que se han declarado. No se han declarado variables locales en la clase `Shirt`. No obstante, si se utilizasen variables locales, se declararían en el método `displayInformation`.

Usos de las variables

El lenguaje Java hace amplio uso de las variables para tareas tales como:

- Almacenar datos de atributo exclusivos de una instancia de objeto (como hemos visto en el caso de las variables `price` e `ID`).
- Asignar el valor de una variable a otra.
- Representar valores dentro de una expresión matemática.
- Imprimir los valores en la pantalla. Por ejemplo, la clase `Shirt` utiliza las variables `price` e `ID` para mostrar en la pantalla el precio y el ID de la camisa:

```
System.out.println("Precio de la camisa: " + price);
System.out.println(“ID de camisa: " + shirtID);
```

- Almacenar referencias a otros objetos.

En este curso se explican numerosas formas de usar variables en las clases de Java.

Declaración e inicialización de variables

La declaración e inicialización de variables de atributo se ajusta a las mismas reglas generales. La declaración e inicialización de estas variables sigue esta sintaxis:

```
[modificadores] identificador de tipo [= valor];
```

Las variables locales pueden declararse e inicializarse por separado (en líneas de código distintas) o en una misma línea de código. La sintaxis para declarar una variable en el interior de un método es:

```
identificador tipo;
```

La sintaxis para inicializar una variable en el interior de un método es:

```
identificador = valor;
```

La sintaxis para declarar e inicializar una variable en el interior de un método es:

```
identificador tipo [= valor];
```

Donde:

- `[modificadores]` representa diferentes palabras clave de Java, como `public` y `private`, que determinan el modo en que otro código accede a una variable de atributo. Los modificadores son opcionales (como indican los corchetes). Por ahora, todas las variables que cree deberían tener el modificador `public` (público).
- `tipo` representa la clase de información o datos que contiene la variable. Algunas variables contienen caracteres, otras contienen números y algunas son booleanas (*boolean*), con lo que sólo pueden admitir uno de dos valores posibles. Es preciso asignar un tipo a cada variable para indicar la clase de información que pueden almacenar.

Nota – No utilice modificadores con las variables locales (variables declaradas dentro de los métodos).

- `identificador` es el nombre asignado a la variable que contiene ese `tipo`.
- `valor` es el valor que se quiere asignar a la variable. Es un elemento opcional porque no es necesario asignar ningún valor a una variable en el momento de declararla.

A continuación figuran las declaraciones de las variables de atributo en la clase `Shirt`:

```
public int shirtID = 0;
public String description = "-descripción-";
public char colorCode = 'U';
public double price = 0.0;
public int quantityInStock = 0;
```

Autoevaluación – Seleccione las declaraciones e inicializaciones de variables de atributo que utilizan la sintaxis correcta para este tipo de operaciones.

a. ___ `public int myInteger=10;`

b. ___ `long myLong;`

c. ___ `long=10;`

d. ___ `private int = 100`

e. ___ `private int myInteger=10;`

Autoevaluación – Defina el término “variable de atributo” o “variable de instancia”.

Respuesta: ____

Autoevaluación – Defina el término “variable local”.

Respuesta: ____

Descripción de los tipos de datos primitivos

Muchos valores de los programas Java se almacenan como tipos de datos primitivos.

Éstos son los ocho tipos de datos primitivos incorporados al lenguaje de programación Java:

- Tipos enteros: `byte`, `short`, `int` y `long`
- Tipos en coma flotante: `float` y `double`
- Tipo textual: `char`
- Tipo lógico: `boolean`

Tipos primitivos enteros

Hay cuatro tipos primitivos enteros en el lenguaje Java y se identifican mediante las palabras clave `byte`, `short`, `int` y `long`. Las variables de este tipo almacenan números que no tienen separador decimal.

Por ejemplo, si necesita guardar edades de personas, una variable de tipo `byte` sería apropiada porque este tipo acepta valores dentro del rango adecuado.

En la tabla siguiente figuran todos los tipos enteros, sus tamaños y el rango de valores que pueden admitir.

Tabla 4-1 Tipos enteros

Tipo	Longitud	Rango	Ejemplos de valores literales permitidos
`byte`	8 bits	-2^7 a 2^7 -1 (de -128 a 127, o 256 valores posibles)	2 -114
`short`	16 bits	-2^{15} a 2^{15} -1 (de -32,768 a 32,767, o 65.535 valores posibles)	2 -32699
`int`	32 bits	-2^{31} a 2^{31} -1 (de -2,147,483,648 a 2,147,483,647 o 4.294.967.296 valores posibles)	2 147334778
`long`	64 bits	-2^{63} a 2^{63} -1 (de -9,223,372,036854,775,808 a 9,223,372,036,854,775,807, o 18.446.744.073.709.551.616 valores posibles)	2 -2036854775808L 1L

Cuando especifique un valor literal para un tipo `long`, escriba una L en mayúscula a la derecha del valor para indicar explícitamente que se trata de un tipo `long`. El compilador considera los enteros como valores de tipo `int`, a menos que se especifique una L para indicar que se trata de un tipo `long`.

La clase `Shirt` contiene dos atributos de tipo `int` para almacenar los valores del ID de camisa y la cantidad disponible en el almacén, y se utilizan valores literales para proporcionar un valor inicial de cero (0) a cada atributo.

```
public int shirtID = 0; // ID predeterminado para la camisa
public int quantityInStock = 0; // Cantidad predeterminada para todas las
camisas
```

Nota – La única razón para utilizar tipos `byte` y `short` en los programas es ahorrar memoria. Dado que la mayoría de los equipos de sobremesa modernos contienen gran cantidad de memoria, hay muchos programadores de aplicaciones de escritorio que no utilizan ni tipos `byte` ni `short`. En los ejemplos de este curso se utilizan fundamentalmente tipos `int` y `long`.

Tipos primitivos en coma flotante

Hay dos tipos para expresar números en coma flotante, `float` y `double`. Estos tipos se utilizan para almacenar números que contienen valores a la derecha del separador decimal (en este caso, el punto), como 12.24 o 3.14159.

La tabla siguiente contiene información sobre los dos tipos en coma flotante.

Tabla 4-2 Tipos en coma flotante

Tipo	Longitud	Ejemplos de valores literales permitidos
`float`	32 bits	99F -32745699.01F 4.2E6F (notación científica de $4.2 * 10^6$)
`double`	64 bits	-1111 2.1E12 99970132745699.999

Cuando especifique un valor literal para un tipo `float`, agregue una F mayúscula (float) a la derecha del número para indicar que se trata de un tipo `float` y no `double`. Los valores literales de los tipos en coma flotante se consideran automáticamente de tipo `double`, a menos que se especifique lo contrario mediante el uso de la F para indicar el tipo `float`.

La clase `Shirt` incluye el uso de un valor literal de tipo `double` para representar el valor predeterminado de precio (`price`):

```
public double price = 0.0; // Precio predeterminado para todas las camisas
```

Nota – Utilice el tipo `double` cuando necesite un rango de valores mayor o un grado de precisión más elevado.

Tipo primitivo textual

Otro tipo de datos que se utiliza para almacenar y manipular la información es el que se expresa a través de un solo carácter. El tipo primitivo utilizado para almacenar caracteres aislados como, por ejemplo, `y`, se llama `char` y tiene un tamaño de 16 bits.

La clase `Shirt` incluye el uso de un valor literal de tipo textual para representar el valor predeterminado del color (`colorCode`):

```
public char colorCode = 'U';
```

Nota – Sólo se puede almacenar un carácter en las variables de tipo `char`. Para almacenar palabras o frases completas, es preciso usar un tipo de objeto (no un tipo primitivo) llamado `String`. El tipo `String` se explica más adelante.

Cuando se asigna un valor literal a una variable `char`, por ejemplo una `t`, es preciso escribir el carácter entre apóstrofos: `'t'`.

Esto permite al compilador reconocer esa `t` como un valor literal y no como una variable `t` que representa otro valor.

El tipo `char` no almacena el carácter escrito como tal, por ejemplo la `t` antes mencionada. La representación almacenada en `char` se transforma en una serie de bits que corresponden a un carácter. Las correspondencias entre cada carácter y el número que representa están establecidas en el juego de caracteres que utilice el lenguaje de programación.

¿Lo sabía? – Muchos lenguajes informáticos utilizan el código ASCII (American Standard Code for Information Interchange), un juego de caracteres de 8 bits que contiene un elemento por cada signo de puntuación, número, letra del alfabeto o carácter utilizados en inglés.

El lenguaje Java utiliza un juego de caracteres de 16 bits llamado Unicode que es capaz de almacenar todos los caracteres que sea necesario reproducir en la gran mayoría de los idiomas utilizados en el mundo moderno. Por tanto, puede escribir sus programas de forma que funcionen y presenten sus datos correctamente en el idioma de la mayoría de los países.

Unicode contiene un subconjunto del juego ASCII (los primeros 128 caracteres).

La clase `Shirt` contiene un tipo primitivo textual, `char`, para almacenar el valor predeterminado del código de color.

```
public char colorCode = 'U';
```

Tipo primitivo lógico

Los programas informáticos a menudo deben tomar decisiones.
El resultado de una decisión, por ejemplo, la evaluación de una sentencia de un programa como verdadera o falsa, puede guardarse en variables `boolean` (booleanas). Las variables de tipo `boolean` sólo pueden almacenar:

- Los literales `true` o `false` del lenguaje Java.
- El resultado de una expresión que sólo se evalúa como *true* o *false*. Por ejemplo, si la variable `respuesta` es igual a 42, la expresión "`if respuesta < 42`" dará como resultado el valor *false*.

Elección del tipo de dato

Una práctica habitual entre los programadores con experiencia es usar los tipos `int`, `long` o `double` para variables numéricas a menos que, por la clase de negocio, exista una razón para ahorrar memoria (por ejemplo, al programar para dispositivos de consumo tales como los teléfonos celulares) o para garantizar tiempos rápidos de ejecución.

Autoevaluación – Establezca las correspondencias entre cada tipo de dato y su tamaño en bits.

Tipo de dato	Tamaño en bits
`double`	8
`char`	16
`int`	32
`byte`	64

Declaración de variables y asignación de sus valores

Las variables deben declararse antes de utilizarlas. En las secciones siguientes se explica cómo declarar las variables y asignarles valor.

Asignación de nombre a las variables

Al igual que ocurre con las clases o los métodos, es preciso asignar a cada variable de un programa un nombre o identificador.

Recuerde que el propósito de las variables es actuar como mecanismo para guardar y recuperar valores. Por tanto, es conveniente que sus identificadores sean sencillos pero descriptivos.

Por ejemplo, si almacena el valor de un ID de artículo, puede asignar a la variable el nombre `miID`, `idArticulo`, `numeroArticulo`, o alguna otra designación que deje claro el uso de la variable tanto para usted como para otras personas que lean el programa.

¿Lo sabía? – Muchos programadores adoptan la convención de usar la primera letra del tipo como identificador. `int i`, `float f`, etc. Este uso es aceptable para programas pequeños que sean fáciles de descifrar, pero, en general, deberían utilizarse identificadores más descriptivos.

Reglas y directrices de asignación de nombre a los identificadores de las variables

Utilice las reglas siguientes como ayuda a la hora de asignar identificadores a las variables:

- Los identificadores de las variables deben empezar por una letra en mayúscula o minúscula, un signo de subrayado (_) o un signo de dólar ($). Después del primer carácter, puede usar dígitos.
- Los identificadores de las variables no pueden contener signos de puntuación, espacios ni guiones.
- Las palabras clave de Java, que aparecen en la tabla siguiente, no pueden utilizarse como identificadores.

Tabla 4-3 Palabras reservadas para la tecnología Java

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

Debería utilizar las directrices siguientes como referencia a la hora de asignar identificadores a las variables:

- Empiece cada variable con una letra en minúscula y el resto de las palabras con la inicial en mayúscula, como, por ejemplo, `miVariable`.
- Elija nombres nemotécnicos que indiquen a cualquier posible lector el propósito de la variable.

¿Lo sabía? – En el lenguaje Java, la grafía en mayúscula o minúscula es un rasgo pertinente. La *pertinencia de la grafía en mayúscula/minúscula* significa que existe diferencia entre la representación en mayúscula o minúscula de cada carácter alfabético. El lenguaje Java considera que dos caracteres del código son distintos si uno se escribe en mayúscula y el otro en minúscula. Por ejemplo, una variable llamada `pedido` será distinta de otra llamada `Pedido`.

Asignación de valores a las variables

A una variable se le puede asignar su valor en el momento de declararla o más adelante. Para asignarle el valor durante su declaración, agréguele el signo igual (=) después de declararla y escriba el valor que le quiera asignar. Por ejemplo, a la variable del atributo `price` en la clase `Shirt` podría asignarle el valor `12.99` como precio de un determinado objeto `Shirt`.

```
double price = 12.99;
```

Un ejemplo de declaración y asignación de valores de variables `boolean` podría ser:

```
boolean esAbierto = false;
```

El operador = asigna el valor del término derecho de la igualdad al elemento del término izquierdo. El signo = debería leerse como "*está asignado a*". Por ejemplo, el ejemplo anterior podría entenderse como: el valor "`12.99` *está asignado a* `price`". Los operadores, como el de asignación (=), se explican más adelante en el curso.

Nota – Las variables de atributo se inicializan automáticamente: los tipos enteros se definen como `0`, los tipos en coma flotante se definen como `0.0`, el tipo `char` recibe el valor de `\u0000` y el tipo `boolean` se establece como `false`. No obstante, las variables de atributos deberían inicializarse de forma explícita para que otras personas puedan leer el código.
Las variables locales (las que se declaran dentro de un método) deben inicializarse explícitamente antes de su uso.

Declaración e inicialización de varias variables en una línea de código

Es posible declarar una o más variables en la misma línea de código, pero sólo si son del mismo tipo. Las sintaxis para declarar varias variables en una misma línea de código es:

```
tipo identificador = valor [, identificador = valor];
```

Por lo tanto, si hubiese un precio de venta al público y otro de venta al por mayor en la clase `Shirt`, se declararían de la forma siguiente:

```
double price = 0.0, wholesalePrice = 0.0;
```

Otras formas de declarar variables y asignarles valor

Los valores de las variables se pueden asignar de diversas formas:

- Asignándoles directamente valores literales (como se ha explicado a lo largo del módulo):

```
int ID = 0;
float pi = 3.14F;
char myChar = 'G';
boolean isOpen = false;
```

- Asignando el valor de una variable a otra:

```
int ID = 0;
int saleID = ID;
```

  La primera línea de código crea un entero llamado `ID` y lo utiliza para guardar el número `0`. La segunda línea crea otro entero llamado `saleID` y lo utiliza para guardar el mismo valor como `ID` (`0`). Si el contenido de `ID` cambia más adelante, el contenido de `saleID` *no* cambia automáticamente. Aunque los dos enteros tienen el mismo valor en ese momento, pueden cambiarse de forma independiente en otra parte posterior del programa.

- Asignando el resultado de una expresión a las variables de tipo entero, en coma flotante o booleanas.

  En las líneas de código siguientes, el resultado de todos los términos situados a la derecha del operador = se asignan a la variable situada a la izquierda del operador.

```
float numberOrdered = 908.5F;
float casePrice = 19.99F;
float price = (casePrice * numberOrdered);

int hour = 12;
boolean isOpen = (hour > 8);
```

- Asignando a una variable el valor de retorno de una llamada a un método. Esta forma de asignación se trata más adelante en el curso.

Constantes

En este módulo hemos hablado de variables cuyos valores puede cambiar el programador. En esta sección, aprenderá a usar constantes para representar valores que no pueden cambiar.

Imagine que está escribiendo parte del programa del catálogo de prendas de la empresa y necesita hacer referencia a la tasa del impuesto sobre la venta. Podría crear la siguiente variable en su clase:

```
double salesTax = 6.25;
```

Potencialmente, este valor podría cambiar dentro del programa, aunque el impuesto debería definirse una vez y permanecer constante a lo largo de la aplicación. Por tanto, es preferible almacenar el valor en un lugar donde no pueda cambiar.

Utilice la palabra clave `final` para convertir el valor en una constante, es decir, decirle al compilador que no quiere que el valor de la variable cambie una vez que haya sido inicializada. Asimismo, por convención, denomine los identificadores de las constantes con palabras en mayúsculas separadas por signos de subrayado. De esta forma, es más fácil saber que se trata de una constante.

```
final double SALES_TAX = 6.25;
```

Cualquier otra persona que necesite acceder al impuesto sobre la venta utilizará la constante SALES_TAX y no necesitará conocer la tasa del impuesto ni preocuparse ante la posibilidad de cambiarlo por error.

Si alguien intenta cambiar el valor de una constante una vez que se le ha asignado un valor, el compilador presentará un error. Si se modifica el código a fin de proporcionar otro valor para la constante, será preciso recompilar el programa.

En caso de que se incrementase el impuesto sobre la venta, debería cambiar la constante en la ubicación del programa donde se haya definido. Todos los usos de la constante adoptarían el cambio automáticamente.

Las constantes también resultan prácticas para valores que son extremadamente largos, como es el caso de la pi (3.14159...).

Directrices de asignación de nombres a las constantes

El nombre que se asigne a las constantes debería ser fácil de identificar. En general, deberían escribirse enteramente en mayúsculas y separar las distintas palabras mediante un signo de subrayado (_).

Discusión – ¿Qué otros tipos de constantes prevé durante el uso de sus programas?

Almacenamiento de tipos primitivos y constantes en la memoria

Cuando se utiliza un valor literal o se crea una constante o una variable y se le asigna un valor, éste se almacena en la memoria del equipo.

En la figura siguiente puede observarse cómo, al almacenarse, las variables locales se separan (en la pila) de las variables de atributo situadas en el espacio de memoria dinámica (heap).

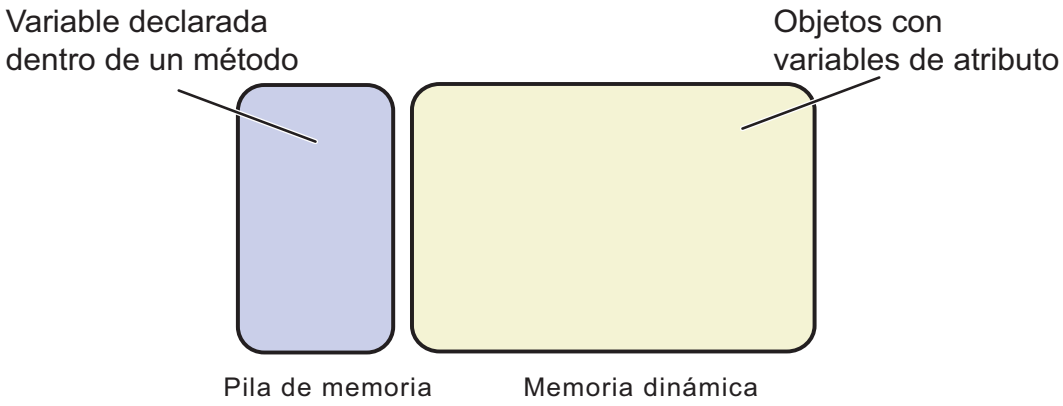


Figura 4-1 Almacenamiento de variables de tipos primitivos en la memoria

Los objetos, sus variables de atributo y sus métodos normalmente se almacenan en el espacio de memoria dinámica. A este espacio se le asignan de forma dinámica fragmentos de memoria que contienen la información utilizada para contener los objetos (incluidas sus variables de atributo y métodos) mientras el programa los necesita. Otras variables suelen almacenarse en la pila de memoria. Esta memoria guarda elementos que sólo se utilizan durante un breve periodo de tiempo (más breve que la vida de un objeto), como es el caso de las variables declaradas en el interior de un método.

Uso de operadores aritméticos para modificar valores

Los programas realizan gran cantidad de cálculos matemáticos, desde los más simples hasta algunos muy complejos. Los operadores aritméticos permiten especificar la forma en que deben evaluarse o combinarse los valores numéricos contenidos en las variables.

Operadores matemáticos estándar

En la tabla siguiente se muestran los operadores matemáticos estándar (a menudo denominados operadores binarios) utilizados en el lenguaje Java.

Tabla 4-4 Operadores binarios

Finalidad	Operador	Ejemplo	Comentarios
Suma	+	`sum = num1 + num2;` Si `num1` es 10 y `num2` es 2, `sum` es 12.	
Resta	–	`diff = num1 – num2;` Si `num1` es 10 y `num2` es 2, `diff` es 8.	
Multiplicación	*	`prod = num1 * num2;` Si `num1` es 10 y `num2` es 2, `prod` es 20.	
División	/	`quot = num1 / num2;` Si `num1` es 31 y `num2` es 6, `quot` es 5.	La división devuelve un valor entero (sin resto).
Resto	%	`mod = num1 % num2;` Si `num1` es 31 y `num2` es 6, `mod` es 1.	El resto halla el resto del cociente entre el primer número dividido por el segundo número. 5 R1 6 | 31 30 1 El resultado del resto siempre tiene el mismo signo que el primer operando.

Operadores de incremento y decremento (++ y --)

Un requisito común en los programas es sumar o restar 1 al valor de una variable. Esto se hace utilizando el operador + de la manera siguiente:

```
age = age + 1;
```

No obstante, aumentar o reducir un valor en incrementos o decrementos de 1 es una operación tan común que existen operadores unarios para realizarla: los operadores de incremento (++) y decremento (--). Estos operadores pueden aplicarse antes (preincremento y predecremento) o después (posincremento y posdecremento) de la variable.

La línea de código anterior, en la que la edad iba aumentando en incrementos de uno también puede escribirse de la forma siguiente:

age++; o ++age;

Utilice estos operadores con precaución en las expresiones. Si se utilizan como prefijo, la operación (incremento o decremento) se aplica antes de cualquier cálculo o asignación posterior. En su forma de sufijo, la operación se aplica después de los sucesivos cálculos u operaciones, de forma que en los cálculos o asignaciones posteriores se utiliza el valor original, no el valor actualizado. En la tabla siguiente se muestra el uso de los operadores de incremento y decremento.

Tabla 4-5 Operadores de incremento y decremento

Operador	Finalidad	Ejemplo	Notas

++	Preincremento (++*variable*)	`int i = 6;`
		`int j = ++i;`
		`i es 7, j es 7`	

	Posincremento (*variable*++)	`int i = 6;`
		`int j = i++;`
		`i es 7, j es 6`
			El valor de `i` se asigna a `j` antes de incrementar `i`. Por tanto, `j` recibe el valor 6.

--	Predecremento (--*variable*)	`int i = 6;`
		`int j = --i;`
		`i es 5, j es 5`	

	Posdecremento (*variable*--)	`int i = 6;`
		`int j = i--;`
		`i es 5, j es 6`
			El valor de `i` se asigna a `j` antes de decrementar `i`. Por tanto, `j` recibe el valor 6.

En el ejemplo siguiente se muestra un uso básico de los operadores de incremento y decremento:

```
int count=15;
int a, b, c, d;
a = count++;
b = count;
c = ++count;
d = count;
System.out.println(a + “, “ + b + “, “ + c + “, “ + d);
```

El resultado de este fragmento de código sería:

```
15, 16, 17, 17
```

Discusión – ¿Cuál es el resultado del código siguiente?


```
int i = 16;
System.out.println(++i + “ “ + i++ + “ “ + i);
```

Orden de precedencia de los operadores

En una sentencia matemática compleja que contenga varios operadores en la misma línea, ¿cómo determina el procesador qué operador debe utilizar primero?

Reglas de precedencia

Para que las operaciones matemáticas mantengan un comportamiento homogéneo, el lenguaje Java sigue las reglas matemáticas habituales para la precedencia de los operadores. Éstos se procesan en el orden siguiente:

1. Operadores incluidos entre paréntesis de apertura y cierre.
2. Operadores de incremento y decremento.
3. Operadores de multiplicación y división evaluados de izquierda a derecha.
4. Operadores de suma y resta evaluados de izquierda a derecha.

Si en una sentencia aparecen sucesivamente varios operadores matemáticos estándar con el mismo orden precedencia, los operadores se evalúan de izquierda a derecha.

Ejemplo de la necesidad de utilizar reglas de precedencia

En el ejemplo siguiente se demuestra la necesidad de utilizar un orden de precedencia de los operadores:

```
c = 25 - 5 * 4 / 2 - 10 + 4;
```

En este ejemplo, no está clara la intención del autor. El resultado puede evaluarse de las formas siguientes:

- Expresión resultante al evaluar estrictamente de izquierda a derecha: 34

  ```
  c = 25 - 5 * 4 / 2 - 10 + 4;
  ```

- Auténtica expresión resultante, evaluada según las reglas de precedencia que se indican mediante los paréntesis: 9

  ```
  c = 25 - ((5 * 4) / 2)) - 10 + 4;
  ```

Uso de los paréntesis

Las expresiones se evalúan automáticamente según las reglas de precedencia, no obstante debería utilizar paréntesis para dejar clara la estructura que pretende utilizar:

```
c = (((25 - 5) * 4) / (2 - 10)) + 4;
c = ((20 * 4) / (2 - 10)) + 4;
c = (80 / (2 - 10)) + 4;
c = (80 / -8) + 4;
c = -10 + 4;
c = -6;
```

Uso de la promoción y conversión de tipos

Nota – Esta sección puede resultar difícil para programadores sin experiencia. Puede que le interese tratar de escribir programas pequeños para practicar con estos conceptos.

Asignar una variable o una expresión a otra variable puede provocar discrepancias entre los tipos de datos de los cálculos y el lugar de almacenamiento utilizado para guardar el resultado. En concreto, el compilador detectará que se va a perder precisión y no permitirá la compilación del programa o bien el resultado será incorrecto. Para resolver este problema, es preciso promover los tipos de las variables a un tipo de tamaño superior o realizar la conversión a un tipo de tamaño inferior.

Por ejemplo, considere la siguiente asignación:

```
int num1 = 53; // 32 bits de memoria para almacenar el
valor
int num2 = 47; // 32 bits de memoria para almacenar el
valor
byte num3; // 8 bits de memoria reservada
num3 = (num1 + num2); // provoca un error de compilación
```

Este código debería funcionar porque un tipo `byte`, aunque es menor que un tipo `int`, es lo suficientemente grande como para almacenar un valor de 100. Sin embargo, el compilador no realizará la asignación y, en su lugar, generará un error de "posible pérdida de precisión" porque un valor `byte` es menor en tamaño que un valor `int`. Para corregir este problema, puede convertir los tipos de datos de la derecha en tipos más pequeños para que coincidan con los tipos de la izquierda, o bien declarar la variable de la izquierda (`num3`) de forma que contenga un tipo de datos más grande, por ejemplo `int`.

El problema se corrige cambiando `num3` por el tipo `int`:

```
int num1 = 53;
int num2 = 47;
int num3;
num3 = (num1 + num2);
```

Promoción

En algunas circunstancias, el compilador cambia el tipo de una variable por otro tipo que admita un valor de mayor tamaño. Esta acción se conoce como promoción. El compilador realiza automáticamente algunas promociones si, al hacerlo, no se pierden datos. Esto ocurre en las siguientes situaciones:

- Si asigna un tipo más pequeño (a la derecha del signo =) a un tipo más grande (a la izquierda del signo =).
- Si asigna un tipo entero a un tipo en coma flotante (aunque podría perderse algo de precisión en los bits menos significativos del valor cuando asigne un valor int o long a un tipo float, o bien un valor long a un tipo double).

El ejemplo siguiente contiene un literal (un valor `int`) que se promoverá automáticamente a otro tipo (`long`) antes de asignar el valor (`6`) a la variable (`big` de tipo `long`). En los ejemplos siguientes se muestra qué tipos promoverá automáticamente el compilador y cuáles no.

```
long big = 6;
```

Dado que el `6` es un tipo `int`, la promoción funciona porque el valor `int` se convierte en un valor `long`.

Atención – Antes de asignarse a una variable, el resultado de una ecuación se coloca en una ubicación temporal de la memoria. El tamaño de esta ubicación siempre es igual que el tamaño de un tipo `int` o es el tamaño del tipo de datos más grande utilizado en la expresión o la sentencia. Por ejemplo, si la ecuación multiplica dos tipos `int`, el tamaño del contenedor tendrá el tamaño de un tipo `int`, o 32 bits.

Si los dos valores multiplicados dan como resultado un valor que supera las dimensiones de un tipo `int` (por ejemplo, 55555*66666=3,703,629,630, que es demasiado grande para caber en un tipo entero), el valor `int` debe truncarse para que el resultado quepa en la ubicación temporal asignada en la memoria. Este cálculo al final produce una respuesta incorrecta porque la variable correspondiente recibe un valor truncado (con independencia del tipo usado para la respuesta).

Para resolver este problema, defina al menos una de las variables de la ecuación con el tipo `long` a fin de garantizar el mayor tamaño posible para el espacio asignado en la memoria temporal.

Conversión de tipos

La conversión de tipos reduce el rango de un valor literalmente "cortándolo" para adaptarlo a un tipo de tamaño más pequeño. Se produce, por ejemplo, cuando se convierte un valor `long` en un valor `int`. Esto se hace para poder usar métodos que aceptan sólo ciertos tipos como argumentos, y poder asignar así valores a una variable de un tipo de menor tamaño, o para ahorrar memoria.

Coloque el *`tipo_destino`* (el tipo en el que se va a convertir el `valor`) entre paréntesis antes del elemento que vaya a convertir. La sintaxis para convertir el tipo de un valor es:

```
identificador = (tipo_destino) valor
```

Donde:

- El *`identificador`* es el nombre que se asigna a la variable.
- *`valor`* es el valor que se quiere asignar al *`identificador`*.
- (*`tipo_destino`*) es el tipo en el que se va a convertir el valor. Recuerde que *`tipo_destino`* debe escribirse entre paréntesis.

Por ejemplo, considere la siguiente asignación:

```
int num1 = 53; // 32 bits de memoria para almacenar el
valor
int num2 = 47; // 32 bits de memoria para almacenar el
valor
byte num3; // 8 bits de memoria reservada
num3 = (num1 + num2); // provoca un error de compilación
```

El error de compilación se corrige convirtiendo el tipo del resultado en un tipo `byte`.

```
int num1 = 53; // 32 bits de memoria para almacenar el
valor
int num2 = 47; // 32 bits de memoria para almacenar el
valor
byte num3; // 8 bits de memoria reservada
num3 = (byte)(num1 + num2); // no hay pérdida de datos
```

Atención – Utilice la conversión de tipos con precaución. Por ejemplo, si se utilizasen números de mayor tamaño para `num1` y `num2`, la conversión en un tipo `byte` truncaría parte de esos datos, con lo que el resultado sería incorrecto.

Otros posibles problemas se indican a continuación:

```
int myInt;
long myLong = 99L;
myInt = (int) (myLong); // No hay pérdida de datos
                        // sólo ceros.
                        // Un número mayor provocaría
                        // la pérdida de datos.

int myInt;
long myLong = 123987654321L;
myInt = (int) (myLong); // El número queda cortado
```

Si convierte un valor con decimales de tipo `float` o `double` en un tipo entero como `int`, se perderán todos los decimales. No obstante, este método de conversión de tipos puede resultar útil a veces si se quiere redondear el número por defecto, por ejemplo 51.9 se convierte en 51.

Suposiciones del compilador con respecto a los tipos enteros y en coma flotante

El compilador de Java parte de algunos supuestos cuando evalúa expresiones. Debe comprender estos supuestos para realizar las conversiones de tipos apropiadas u otro tipo de adaptaciones.

Tipos de datos enteros y operaciones

Cuando se utilizan valores con tipos de datos primitivos en una expresión con determinados operadores (`*`, `/`, `-`, `+`, `%`), los valores se convierten automáticamente en un valor `int` (o superior si es necesario) y luego se ejecuta la operación. Esta conversión puede provocar desbordamiento de datos o falta de precisión.

En el ejemplo siguiente, se produce un error porque dos de los tres operandos (a y b) se promueven automáticamente del tipo short a un tipo int antes de sumarse:

```
short a, b, c;
a = 1 ;
b = 2 ;
c = a + b ;  //error de compilación
```

En la última línea, los valores de a y b se convierten en tipos int y se suman para proporcionar un resultado de tipo int. A continuación, el operador de asignación (=) trata de asignar el resultado int a la variable de tipo short (c). Pero esta asignación no es válida y provoca un error de compilación.

El código funcionará si realiza una de estas operaciones:

- Declarar c como un tipo int en la declaración original:
  ```
  int c;
  ```

- Convertir el tipo del resultado de (a+b) en la línea de asignación:
  ```
  c = (short)(a+b);
  ```

Tipos de datos en coma flotante y forma de asignarlos

Al igual que los tipos enteros adoptan automáticamente la forma int en algunas circunstancias, los valores asignados a los tipos en coma flotante siempre adoptan la forma double, a menos que se especifique expresamente que el valor es de tipo float.

Por ejemplo, la línea siguiente provocaría un error de compilación. Dado que 27.9 se considera automáticamente del tipo double, se produce un error de compilación porque un valor double no cabe en una variable de tipo float.

```
float float1 = 27.9;//error de compilación
```

Las dos alternativas siguientes funcionarían correctamente:

- La F indica al compilador que 27.9 es un valor float:
  ```
  float float1 = 27.9F;
  ```

- 27.9 se convierte en un tipo float:
  ```
  float float1 = (float) 27.9;
  ```

Ejemplo

En el ejemplo del código siguiente se utilizan los principios explicados en esta sección para calcular la edad de una persona expresada en días y segundos.

Código 4-2 Archivo `Person.java` situado en el directorio `data_types`

```
1   public class Person {
2
3     public int ageYears = 32;
4
5     public void calculateAge() {
6
7       int ageDays = ageYears * 365;
8       long ageSeconds = ageYears * 365 * 24L * 60 * 60;
9
10      System.out.println(“Tienes una edad de “ + ageDays + “ días.”);
11      System.out.println(“Tienes una edad de “ + ageSeconds + “
segundos.”);
12
13    } // fin del método calculateAge
14  } // fin de la clase
```

Módulo 5

Creación y uso de objetos

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Declarar, instanciar e inicializar variables de referencia a objetos.
- Explicar cómo se guardan las variables de referencia a objetos en comparación con las variables de tipos primitivos.
- Utilizar una clase (la clase `String`) incluida en el SDK de Java.
- Utilizar la especificación de la biblioteca de clases de Java SE para conocer mejor otras clases de este API.

En este módulo se explica cómo usar objetos en los programas y cómo organizar los programas utilizando sintaxis específica de las clases y los métodos.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Declarar, instanciar e inicializar variables de referencia a objetos.			
Explicar cómo se guardan las variables de referencia a objetos en comparación con las variables de tipos primitivos.			
Utilizar una clase (la clase `String`) incluida en el SDK de Java.			
Utilizar la especificación de la biblioteca de clases de Java SE para conocer mejor otras clases de este API.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Notas

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender en qué consiste la creación y el uso de objetos:

- ¿Qué significa crear una instancia del plano de una casa?
- ¿Cómo hace referencia a diferentes casas de una misma calle?
- Cuando un constructor levanta una casa, ¿construye cada componente de la casa, incluidas las ventanas, las puertas y los armarios?

Otros recursos

Otros recursos – El documento siguiente proporciona información complementaria sobre los temas tratados en este módulo:

- Tutorial de Java. [En la web]. Disponible en:
  `http://java.sun.com/docs/books/tutorial/java/index.html`

  Guía práctica para programadores con cientos de ejemplos y ejercicios completos.

Declaración de referencias a objetos, instanciación de objetos e inicialización de referencias a objetos

Las variables de referencia a objetos son variables que contienen la dirección de un objeto en la memoria. Una carta es como una variable de referencia porque contiene una dirección que remite a un determinado objeto “edificio”. En la ilustración siguiente se muestran varias direcciones que señalan a diferentes casas.

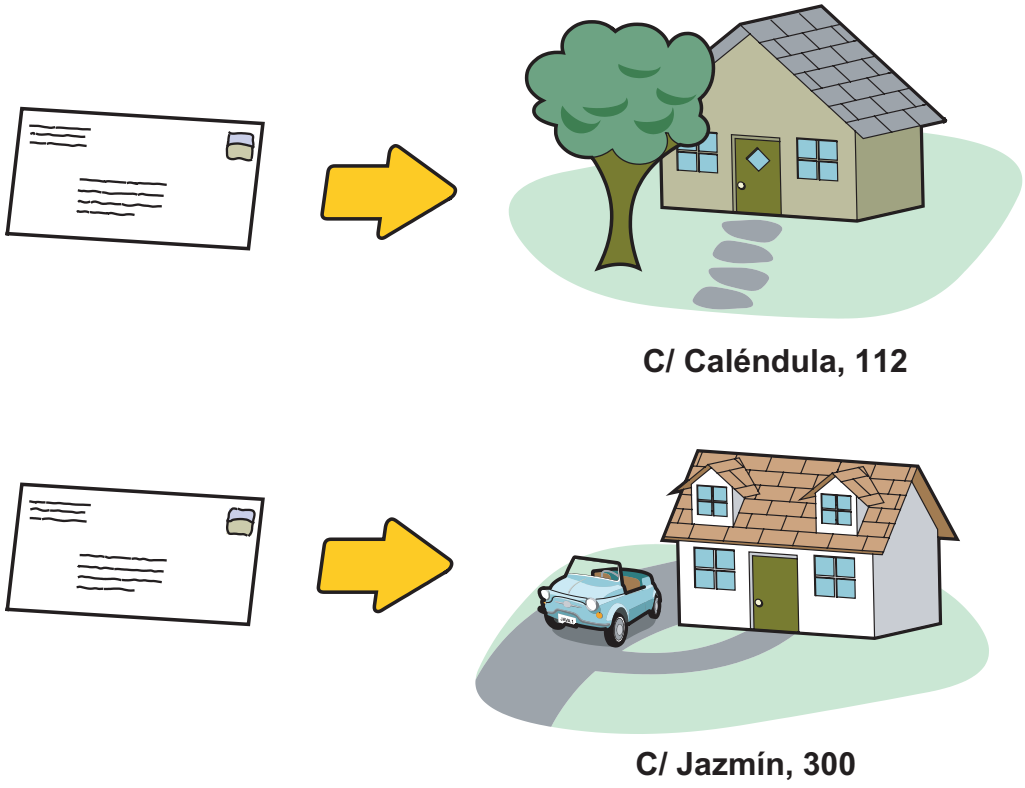


Figura 5-1 Varias cartas que se dirigen a diferentes casas

En esta sección, creará clases e instanciará objetos dentro de esas clases. En el ejemplo siguiente se muestra una clase `ShirtTest` que crea una variable, una variable de referencia a un objeto (`myShirt`), e inicializa la variable para que señale a una instancia de la clase `Shirt` (línea 5). Una vez creada e inicializada la variable de referencia, puede utilizarse para llamar al método `displayInformation` (línea 7) del objeto `Shirt`.

Código 5-1 Archivo `ShirtTest.java` situado en el directorio `getstarted`

```
1   public class ShirtTest {
2
3      public static void main (String args[]) {
4
5      Shirt myShirt = new Shirt();
6
7      myShirt.displayInformation();
8
9      }
10  }
```

Declarar e inicializar una variable de referencia es muy parecido a declarar e inicializar una variable de un tipo primitivo. La única diferencia reside en que es preciso crear una instancia de un objeto (a partir de una clase) al que debe hacer referencia la variable antes de inicializar la instancia del objeto.

Para declarar, instanciar e inicializar variables de referencia a objetos:

1. Declare una referencia al objeto especificando su identificador y el tipo de objeto al que señala la referencia (la clase del objeto).
2. Cree la instancia del objeto utilizando la palabra clave `new`.
3. Inicialice la variable de referencia asignando el objeto a dicha variable.

Nota – Puede realizar estos tres pasos en una sola línea de código (como se ha mostrado anteriormente en el ejemplo de `ShirtTest.java`).

Declaración de variables de referencia a objetos

Para declarar una variable de referencia a un objeto, especifique la clase a partir de la que quiere crear el objeto y seleccione el nombre que desee utilizar para hacer referencia a ese objeto. La sintaxis para declarar variables de referencia a objetos es:

```
Nombreclase identificador;
```

Donde:

- *Nombreclase* es la clase o tipo de objeto al que hace referencia la variable.
- El *identificador* es el nombre asignado a la variable de tipo *Nombreclase*.

Por ejemplo, en la línea 5, la clase `ShirtTest` crea una variable de referencia a un objeto llamada `myShirt`. En principio, las referencias de objeto que son variables miembro se inicializan con `null`. Dado que la variable `myShirt` es una variable local declarada en la línea 5, no se inicializa.

Como ocurre con todas las variables, es importante que el identificador refleje la finalidad de la variable y que siga las reglas de nomenclatura propias de los identificadores.

Instanciación de un objeto

Una vez declarada la referencia a un objeto, puede crearse el objeto al que hace referencia. La sintaxis para instanciar un objeto es:

```
new Nombreclase();
```

Donde:

- La palabra clave `new` crea una instancia de un objeto a partir de una clase.
- *Nombreclase* es la clase o el tipo de objeto que se va a crear.

Por ejemplo, la clase `ShirtTest` crea un objeto del tipo `Shirt` en la línea 5 utilizando la palabra clave `new`.

Nota – La instanciación del objeto y su asignación a la variable de referencia deben hacerse en la misma línea de código. En la sección siguiente se muestra cómo escribir código que instancie un objeto y efectúe la asignación (inicialice la variable de referencia al objeto).

Inicialización de variables de referencia a objetos

El último paso para crear una variable de referencia a un objeto es inicializar la variable asignándole el objeto recién creado. Al igual que ocurre con la asignación o inicialización de las demás variables, esto se hace mediante el signo igual (=). La sintaxis para inicializar un objeto en una variable de referencia es:

```
identificador = new Nombreclase();
```

Por ejemplo, la clase `ShirtTest` asigna el nuevo objeto `Shirt` a la variable de referencia `myShirt`. Se puede decir que “La variable de referencia `myShirt` señala a un objeto `Shirt`”.

```
myShirt = new Shirt();
```

Una vez creada una instancia válida de un objeto, ésta puede utilizarse para manejar los datos del objeto o llamar a los métodos del objeto.

Creación de variables de referencia a objetos utilizando una o dos líneas de código

En el ejemplo de `ShirtTest`, la creación de la referencia de objeto, la creación del nuevo objeto `Shirt` y la inicialización de la variable se efectúan en la misma línea de código. Por ejemplo:

```
Shirt myShirt;
myShirt = new Shirt();
```

No obstante, también está permitido crear la variable de referencia a un objeto en una línea y efectuar la creación del nuevo objeto y la asignación del objeto a la referencia en otra línea. La línea de código siguiente es equivalente a las dos líneas mostradas anteriormente:

```
Shirt myShirt = new Shirt();
```

Uso de una variable de referencia a objetos para manejar datos

El operador punto (.) se utiliza con las referencias a objetos para manipular los valores o hacer llamadas a los métodos de un objeto concreto. Por ejemplo, para cambiar la variable `colorCode` del nuevo objeto por "`G`" (correspondiente a verde), debería agregar la siguiente línea de código a su clase `ShirtTest`:

```
myShirt.colorCode = 'G';
```

A continuación se muestra una clase `ShirtTestTwo` nueva que crea dos variables de referencia, `myShirt` y `yourShirt`, que hacen referencia a diferentes objetos `Shirt` (líneas 5 y 6).

Código 5-2 Archivo `ShirtTestTwo.java` situado en el directorio `object_structure`

```
1   public class ShirtTestTwo {
2
3     public static void main (String args[]) {
4
5       Shirt myShirt = new Shirt();
6       Shirt yourShirt = new Shirt();
7
8       myShirt.displayInformation();
9       yourShirt.displayInformation();
10
11      myShirt.colorCode='R';
12      yourShirt.colorCode='G';
13
14      myShirt.displayInformation();
15      yourShirt.displayInformation();
16
17    }
18  }
```

En este ejemplo, la declaración, instanciación e inicialización de cada variable de referencia se realizan en la misma línea de código. La declaración se sitúa a la izquierda del signo igual y la instanciación del objeto se sitúa a la derecha de este signo. El signo igual asigna cada nueva instancia del objeto a la referencia recién declarada (líneas 5 y 6).

Una vez creadas las referencias a los objetos, cada objeto puede manejarse de forma independiente. En concreto, el objeto al que señala la referencia `myShirt` recibe un código de color (`colorCode`) "`R`" de rojo (línea 11), mientras que el objeto al que señala la referencia `yourShirt` recibe el valor "`G`" de verde (línea 12) en `colorCode`.

Puede escribir líneas de código para crear tantas referencias a objetos `Shirt` como desee, en función de la cantidad de memoria que tenga el equipo informático.

Nota – Una forma de interacción de los objetos tiene como finalidad hacer que un objeto manipule los datos de otro objeto. Asimismo, un objeto puede llamar a los métodos de otro objeto. Por ejemplo, la referencia al objeto `myShirt` llama al método `displayInformation` (línea 14). La llamada a métodos mediante el operador punto (.) se describe más adelante en este curso.

Almacenamiento de variables de referencia a objetos en la memoria

Si las variables de tipos primitivos almacenan valores, las variables de referencia a objetos almacenan la ubicación (dirección de memoria) de los objetos en la memoria.

Nota – Las direcciones de memoria normalmente se escriben en notación decimal, empezando desde 0x (por ejemplo, 0x334009). Cada dirección es exclusiva de un objeto y se asigna mientras se ejecuta un programa.

En la figura siguiente se muestra cómo se almacenan de manera diferente las variables de tipos primitivos y las de referencia a objetos.

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

  int counter;
  counter = 10;
  Shirt myShirt = new Shirt ( );
}
```

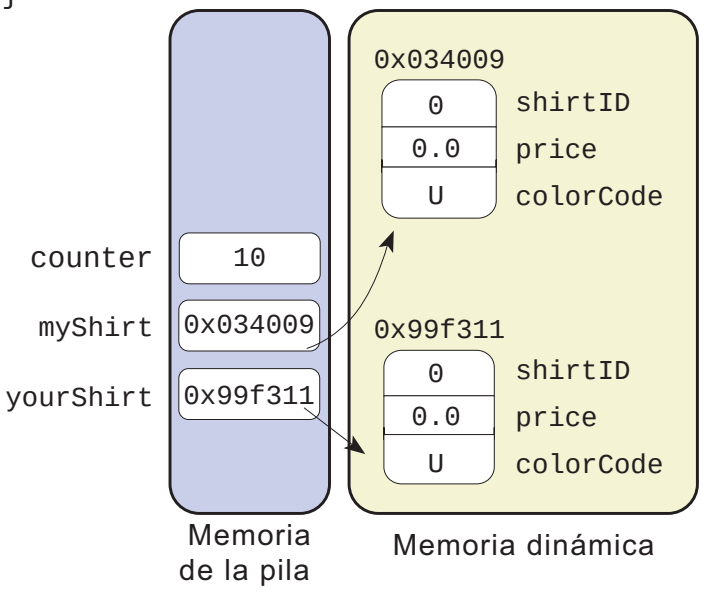


Figura 5-2 Forma en que se almacenan las variables de referencia en la memoria

Nota – Las variables que hacen referencia a objetos locales y sus valores se guardan en la memoria de pila mientras que los objetos a los que señalan se almacenan en el espacio de memoria dinámica (heap). Las variables de referencia a los objetos `myShirt` y `yourShirt` contienen direcciones que señalan a diferentes objetos `Shirt`.

Asignación de una referencia de una variable a otra

¿Qué cree que ocurre cuando se asigna una variable de referencia a otra variable de referencia? Por ejemplo, cuando se ejecuta el siguiente código:

```
Shirt myShirt = new Shirt();
Shirt yourShirt = new Shirt();
myShirt = yourShirt;
```

Las variables de referencia a `myShirt` y `yourShirt` contienen la misma dirección de acceso al mismo objeto.

Como en el caso de las variables de tipos primitivos, el valor del término situado a la derecha se asigna al término de la izquierda. En el ejemplo anterior, la dirección de la variable `yourShirt`, por ejemplo 0x99f311, se asigna a la variable `myShirt`. Ambas variables ahora señalan al mismo objeto, aunque el otro objeto, aquel al que antes señalaba la variable `myShirt`, sigue existiendo.

En la figura siguiente se ilustra esta asignación.

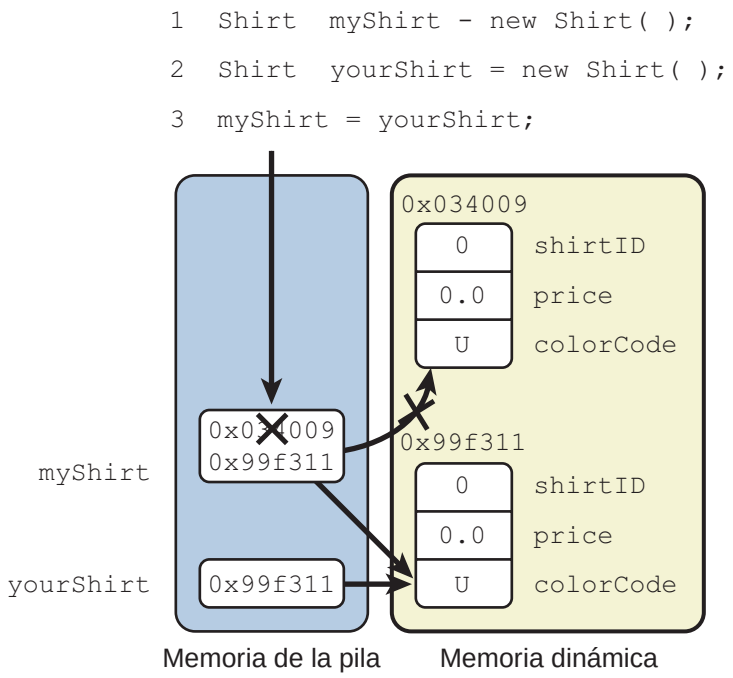


Figura 5-3 Asignación de la referencia de una variable a otra

Nota – Las líneas de código de la figura aparecerían en un método como, por ejemplo, el método `main`.

A menos que haya otra variable de referencia señalando al segundo objeto `Shirt`, el espacio de memoria utilizado por esa dirección se recicla.

Uso de la clase String

String es una de las muchas clases incluidas en las bibliotecas de clases de Java. La clase String permite guardar una secuencia de caracteres. Es una clase muy utilizada a lo largo de los programas. Por tanto, es importante comprender algunas de las características especiales de las secuencias de caracteres en el lenguaje de programación Java.

Creación de un objeto String con la palabra clave new

Hay diferentes maneras de crear e inicializar un objeto String. Una de ellas es usar la palabra clave new para crear un objeto String y, al mismo tiempo, definir la secuencia de caracteres con la que se va a inicializar ese objeto:

```
String myName = new String("Alfredo Lopez");
```

¿Lo sabía? – Cuando se crea un objeto, se llama a un constructor especial denominado *constructor*. Los constructores se explican más adelante en este curso.

Creación de un objeto String con la palabra new origina dos objetos String en la memoria y coloca la representación de caracteres del literal String en un espacio de la memoria dinámica reservado para literales que se denomina grupo de almacenamiento de literales.

Creación de un objeto String sin la palabra clave new

Los tipos String constituyen una excepción porque son la única clase que permite crear objetos sin usar la palabra clave new. Por ejemplo:

```
String myName = "Alfredo Lopez";
```

Esta sintaxis es muy similar a la utilizada para declarar e inicializar variables de tipos de datos primitivos.

La creación de un objeto String sin usar la palabra clave new da lugar a un objeto String y coloca la representación de la secuencia de caracteres en el grupo de almacenamiento de literales (si el literal String no existe ya en ese grupo de almacenamiento). Para evitar la duplicación innecesaria de objetos String en la memoria, es mejor crear objetos String *sin* la palabra clave new.

Almacenamiento de objetos String en la memoria

El almacenamiento de las variables de referencia a objetos `String` en la memoria funciona de la misma forma que otras referencias a objetos: tiene una variable con un nombre, como `myName`, que contiene una referencia en la memoria en la que está situado el objeto `String`. El objeto `String` tiene un valor, una secuencia de caracteres que se ha suministrado durante la inicialización. En la figura siguiente se ilustra cómo aparece en la memoria una variable de referencia a un objeto `String`.

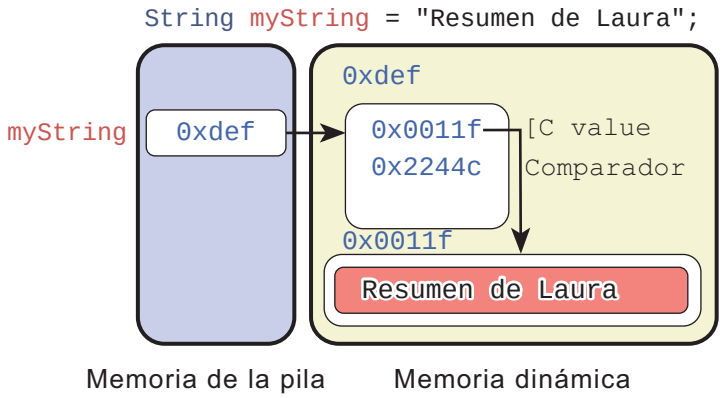


Figura 5-4 Forma en que los objetos `String` se almacenan en la memoria

Nota – La línea de código de la figura aparecería en un método como, por ejemplo, `main`.

La dirección contenida en la variable `myString` hace referencia a un objeto `String` de la memoria. El objeto `String` tiene una variable de atributo llamada `[C value` que contiene una dirección referida a la secuencia de caracteres.

Uso de variables de referencia para objetos String

Las variables de objetos String se utilizan como las variables de tipos primitivos. En el ejemplo siguiente se muestra una clase que indica el nombre y el trabajo de una persona:

Código 5-3 Archivo PersonTwo.java situado en el directorio object_structure

```
public class PersonTwo {

  public String name = “Juan”;
  public String job = “Probador de helados”;

  public void display(){
    System.out.println(“Me llamo “ + name + “, soy “ + job);
  }
} // fin de la clase
```

Nota – En el ejemplo anterior se muestra una clase con los valores predeterminados (“Juan” y “Probador de helados”), que serían el nombre y el trabajo predeterminados para cada objeto PersonTwo. Esta sintaxis permite conocer la clase String sin tener que hacer llamadas especiales a los métodos. En general, no le interesará que sus clases contengan datos específicos de la instancia. Un valor más adecuado para la variable name es “-name required-”. Un valor más adecuado para la variable job es “-job required-”.

Estudio de las bibliotecas de clases de Java

Todos los SDK de tecnología de Java contienen una serie de clases precodificadas que se utilizan en los programas. Estas bibliotecas están documentadas en la especificación de las bibliotecas de clases correspondientes a la versión del SDK que se esté utilizando. La especificación se compone de una serie de páginas web en formato HTML (Hypertext Markup Language) que pueden cargarse en los navegadores web.

Especificación de la biblioteca de clases de Java

Una especificación de biblioteca de Java es un documento muy detallado donde se explican las clases del API. Cada API incluye documentación donde se describe el uso de las clases, las variables y los métodos de los programas.

Cuando busque una forma de realizar ciertos conjuntos de tareas, esta especificación es la mejor fuente de información sobre las clases precodificadas de las bibliotecas de Java.

Para ver la especificación del API de Java SE en Internet, vaya al siguiente sitio web:

```
http://java.sun.com/javase/reference/api.jsp
```

Y seleccione el vínculo de los documentos fundamentales del API (Core API Docs) correspondiente al número de versión del JDK que esté utilizando. Por ejemplo, para elegir el JDK versión 6:

Donde *versión* es el número de versión del SDK de Java SE que esté utilizando. Por ejemplo:

```
http://java.sun.com/javase/6/docs/api/
```

Para descargar la documentación, consulte las instrucciones del Apéndice A, “Siguientes pasos”.

En la figura siguiente puede verse un ejemplo de la documentación correspondiente a una clase del API.

Figura 5-5 Clase `String` de la especificación del API de Java SE

Nota – Es posible que, en estos momentos, la especificación parezca algo técnica y complicada. Sin embargo, al final de la lección utilizará este documento para buscar información sobre varias clases.

Uso de la especificación de la biblioteca de clases de Java para obtener información sobre un método

A lo largo de todo el curso ha estado utilizando la clase `System`.
En concreto, ha utilizado el método `println` para mostrar en la pantalla la salida de los programas:

```
System.out.println(datos_para_imprimir_en_la_pantalla);
```

El método `System.out.println` es un método que viene precodificado en el lenguaje de programación Java. Utilice los pasos siguientes para obtener más información sobre este método a través del API de Java SE:

1. Utilice el navegador web para buscar la primera página de la documentación del API de Java SE.
2. Haga clic en la clase `System` de la lista.

   Observe que la clase `System` posee una variable llamada `out` que es una variable de referencia a un objeto `PrintStream`. De hecho, `println` es un método perteneciente a la clase `PrintStream`.
3. Haga clic en el vínculo de la variable `out` para ver más datos sobre esta variable.
4. Haga clic en el vínculo de la clase `PrintStream` al comienzo de la explicación relativa a la variable `out`.
5. Desplace hacia abajo el texto del método `println`.
   Hay varios métodos `println`. La clase `PrintStream` tiene otros métodos para imprimir la salida en una pantalla.

   Por ejemplo, dispone del método `print`. La documentación aclara que la diferencia entre los métodos `println` y `print` es que `println` aplica un salto de línea, de forma que el texto adicional se imprime en una línea nueva. Por ejemplo:

   ```
   System.out.print("Carpe diem");
   System.out.println("Disfruta de la vida");
   ```

   Presenta lo siguiente en la pantalla:

   ```
   Carpe diem Disfruta de la vida
   ```

   Por tanto, en función de lo que deba hacer el programa, preferirá usar el método `print` en lugar de `println`.

Comprender el modo de usar la documentación de la biblioteca de clases de Java SE es fundamental para explotar al máximo todo el potencial de la tecnología Java.

Nota – A medida que avancen sus conocimientos de programación, se familiarizará con una multitud de clases de la plataforma Java SE, con sus correspondientes variables y métodos.

Módulo 6

Uso de operadores y construcciones de toma de decisiones

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Identificar los operadores relacionales y condicionales.
- Crear construcciones con `if` e `if/else`.
- Usar la construcción `switch`.

En este módulo se explica cómo usar operadores en las expresiones del código y la forma de utilizar esas expresiones en construcciones de toma de decisiones.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivo del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = He podido cumplir el objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Identificar los operadores relacionales y condicionales.			
Crear construcciones con `if` e `if/else`.			
Usar la construcción `switch`.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender en qué consiste el uso de operadores y construcciones de toma de decisiones:

- Cuando necesita tomar una decisión que presenta varias alternativas, ¿cuál es la razón que le lleva a elegir una alternativa frente a las demás?
- Por ejemplo, ¿qué consideraciones le vienen a la mente cuando va a adquirir un artículo?

Otros recursos

Otros recursos – El documento siguiente proporciona información complementaria sobre los temas tratados en este módulo:

- Tutorial de Java. [En la web]. Disponible en: `http://java.sun.com/docs/books/tutorial/`

  Una guía práctica para programadores con cientos de ejemplos y ejercicios completos.

Uso de operadores relacionales y condicionales

Una de las tareas que suelen realizar los programas es evaluar una condición y, en función del resultado, ejecutar diferentes bloques o "ramales" de código. Por ejemplo, su programa podría comprobar si el valor de una variable es *igual* que el valor de otra variable y, en caso afirmativo, realizar *una acción*. En la figura siguiente se ilustra un tipo de decisión que las personas toman cada día.

Figura 6-1 Construcciones de toma de decisiones en conducción

Además de los operadores aritméticos, como los de suma (+) e incremento (++), el lenguaje Java proporciona diferentes operadores relacionales y condicionales que incluyen < y > para expresar los conceptos menor que y mayor que, y && para expresar unión (AND). Estos operadores se utilizan cuando se quiere que los programas ejecuten diferentes boques o partes del código en función de distintas condiciones como, por ejemplo, comprobar si el valor de dos variables es el mismo.

Nota – Cada uno de estos operadores se utiliza en el contexto de una construcción de toma de decisiones del tipo `if` o `if/else`. Estas construcciones se explican más adelante en el módulo.

Ejemplo de un ascensor

Además de los ejemplos pertenecientes al caso de DirectClothing, Inc., en este curso se utilizan diferentes variantes de la clase `Elevator` (Ascensor) para explicar algunos conceptos del lenguaje Java. El ejemplo de código siguiente contiene una de esas variantes de la clase `Elevator`.

Código 6-1 Archivo `Elevator.java` situado en el directorio `opsdec`

```
1   public class Elevator {
2
3     public boolean doorOpen=false; // Puertas cerradas de forma
predeterminada
4     public int currentFloor = 1; // Todos los ascensores empiezan en el
primer piso
5     public final int TOP_FLOOR = 10;
6     public final int MIN_FLOORS = 1;
7
8     public void openDoor() {
9       System.out.println(“Se está abriendo la puerta.”);
10      doorOpen = true;
11      System.out.println(“La puerta está abierta.”);
12    }
13
14    public void closeDoor() {
15      System.out.println(“Se está cerrando la puerta.”);
16      doorOpen = false;
17      System.out.println(“La puerta está cerrada.”);
18    }
19
20    public void goUp() {
21      System.out.println(“Subiendo una planta.”);
22      currentFloor++;
23      System.out.println(“Piso: “ + currentFloor);
24    }
25
26    public void goDown() {
27      System.out.println(“Bajando una planta.”);
28      currentFloor--;
29      System.out.println(“Piso: “ + currentFloor);
30    }
31
32
33  }
```

Un ascensor tiene muchas funciones. Las funciones que se citan en este curso son:

- Las puertas del ascensor pueden abrirse (método `openDoor` desde la línea 8 a la 12).
- Las puertas del ascensor pueden cerrarse (método `closeDoor` desde la línea 14 a la 18).
- El ascensor puede ascender un piso (método `goUp` desde la línea 20 a la 24).
- El ascensor puede descender un piso (método `goDown` desde la línea 26 a la 30).

A lo largo de este módulo y los módulos sucesivos tendrá ocasión de ver diferentes variaciones de esta clase `Elevator`, en algunas de las cuales se ilustra el uso de construcciones de toma de decisiones.

Una clase de prueba similar a la siguiente somete el código del ascensor a diferentes comprobaciones.

Código 6-2 Archivo `ElevatorTest.java` situado en el directorio `opsdec`

```
1   public class ElevatorTest {

2      public static void main(String args[]) {
3
4        Elevator myElevator = new Elevator();
5
6        myElevator.openDoor();
7        myElevator.closeDoor();
8        myElevator.goDown();
9        myElevator.goUp();
10       myElevator.goUp();
11       myElevator.goUp();
12       myElevator.openDoor();
13       myElevator.closeDoor();
14       myElevator.goDown();
15       myElevator.openDoor();
16       myElevator.goDown();
17       myElevator.openDoor();
18     }
19  }
```

Operadores relacionales

Los operadores relacionales comparan dos valores para determinar su relación.

En la tabla siguiente figuran las distintas condiciones que pueden comprobarse utilizando este tipo de operadores.

Tabla 6-1 Operadores relacionales

Condición	Operador	Ejemplo
Es igual que (o "es lo mismo que")	`==`	`int i=1;` `(i == 1)`
No es igual que (o "no es lo mismo que")	`!=`	`int i=2;` `(i != 1)`
Es menor que	`<`	`int i=0;` `(i < 1)`
Es menor o igual que	`<=`	`int i=1;` `(i <= 1)`
Es mayor que	`>`	`int i=2;` `(i > 1)`
Es mayor o igual que	`>=`	`int i=1;` `(i >= 1)`

El resultado de todos los operadores relacionales es un valor `boolean`. Los valores booleanos pueden ser `true` o `false`. Por ejemplo, todos los ejemplos de la tabla anterior dan como resultado el valor `true`.

Nota – El signo igual (=) se utiliza para realizar asignaciones.

Comprobación de la igualdad entre dos secuencias de caracteres

Si se utiliza el operador == para comparar diferentes referencias a objetos `String`, el operador comprueba si la dirección contenida en dichas referencias es la misma, no su contenido.

Discusión – ¿Son iguales todas las referencias a objetos `String` siguientes?

```
String helloString1 = (“hola”);
String helloString2 = “hola”;
String helloString3 = new String(“hola”);
```

Si quiere verificar la equivalencia entre secuencias de caracteres, por ejemplo, para averiguar si el nombre “Fred Smith” es igual que “Joseph Smith” utilice el método `equals` de la clase `String`. La clase siguiente contiene dos nombres de empleados y un método para compararlos.

Código 6-3 Archivo `Employees.java` situado en el directorio `opsdec`

```
1   public class Employees {
2
3     public String name1 = “Fred Smith”;
4     public String name2 = “Joseph Smith”;
5
6     public void areNamesEqual() {
7
8       if (name1.equals(name2)) {
9         System.out.println(“Mismo nombre.”);
10      }
11      else {
12        System.out.println(“Nombre distinto.”);
13      }
14    }
15  }
16
```

Operadores condicionales

También necesitará tener la posibilidad de tomar una sola decisión en función de varias condiciones. En tales circunstancias, puede utilizar los operadores condicionales para evaluar condiciones *complejas* como un todo.

En la tabla siguiente figuran los operadores condicionales comunes del lenguaje Java.

Tabla 6-2 Operadores condicionales comunes

Operación	Operador	Ejemplo

Si una condición Y otra condición	`&&`	`int i = 2;`
		`int j = 8;`
		`((i < 1) && (j > 6))`

Si una condición U otra condición	`||`	`int i = 2;`
		`int j = 8;`
		`((i < 1) || (j > 10))`

NO	`!`	`int i = 2;` `(!(i < 3))`

Por ejemplo todos los ejemplos de la tabla anterior dan como resultado el valor booleano `false`.

Discusión – ¿Qué operadores relacionales y condicionales se reflejan en el párrafo siguiente?

Si el juguete es rojo, lo compraré. Sin embargo, si el juguete es amarillo y cuesta menos que uno rojo, *también* lo adquiriré. Si el juguete es amarillo y cuesta lo mismo o más que otro artículo rojo, no lo compraré. Por último, si el juguete es verde, no lo compraré.

Creación de construcciones `if` e `if/else`

Una sentencia o construcción `if` es aquella que ejecuta un bloque de código si una expresión es *true*.

Construcción `if`

Existen diferentes variantes de la construcción `if` básica. No obstante, la más sencilla es:

```
if (expresión_booleana) {
    bloque_código;
} // fin de la construcción if
// el programa continúa aquí
```

Donde:

- La *`expresión_booleana`* es una combinación de operadores relacionales, operadores condicionales y valores que da como resultado el valor `true` o `false`.
- *`bloque_código`* representa las líneas de código que se ejecutan si (*if*) la *`expresión`* es `true`.

En primer lugar se evalúa la *`expresión_booleana`*. Si la expresión es `true`, se ejecuta *`bloque_código`*. Si la *`expresión_booleana`* no es `true`, el programa salta directamente a la llave que marca el fin del bloque de código de la construcción `if`.

La clase `ElevatorTest` comprueba un objeto `Elevator` efectuando llamadas a sus métodos. Uno de los primeros métodos a los que llama la clase `ElevatorTest` es `goDown`.

Llamar al método goDown antes de que el ascensor haya comenzado a subir genera un problema porque el ascensor empieza en el primer piso y no hay otros pisos por debajo (la constante MIN_FLOORS indica que el primer piso es la planta baja). Cuando se ejecuta, la clase ElevatorTest muestra lo siguiente (en la versión inglesa):

```
Opening door.
Door is open.
Closing door.
Door is closed.
Going down one floor.
Floor: 0 <--- esto es un error en la lógica
Going up one floor.
Floor: 1
Going up one floor.
Floor: 2
...
```

Dos sentencias if pueden solucionar este problema. El método goDown siguiente contiene dos construcciones if que determinan si el ascensor debería descender o mostrar un error.

Código 6-4 Archivo IfElevator.java situado en el directorio opsdec

```
1
2   public class IfElevator {
3
4     public boolean doorOpen=false; // Puertas cerradas de forma
predeterminada
5     public int currentFloor = 1; // Todos los ascensores empiezan en el
primer piso
6     public final int TOP_FLOOR = 10;
7     public final int MIN_FLOORS = 1;
8
9     public void openDoor() {
10      System.out.println(“Se está abriendo la puerta.”);
11      doorOpen = true;
12      System.out.println(“La puerta está abierta.”);
13    }

14    public void closeDoor() {
15      System.out.println(“Se está cerrando la puerta.”);
16      doorOpen = false;
17      System.out.println(“La puerta está cerrada.”);
18    }
```

```
19    public void goUp() {
20      System.out.println(“Subiendo una planta.”);
21      currentFloor++;
22      System.out.println(“Piso: “ + currentFloor);
23    }

24    public void goDown() {
25
26      if (currentFloor == MIN_FLOORS) {
27        System.out.println(“Imposible bajar”);
28      }

29      if (currentFloor > MIN_FLOORS) {
30        System.out.println(“Bajando una planta.”);
31        currentFloor--;
32        System.out.println(“Piso: “ + currentFloor);
33      }
34    }


35  }
```

Si el valor de la variable `currentFloor` es igual que el de la constante `MIN_FLOORS`, aparecerá en la pantalla un mensaje de error y el ascensor no descenderá (líneas de la 26 a la 28). Si el valor de la variable `currentFloor` es mayor que el de la constante `MIN_FLOORS`, el ascensor baja (líneas de la 29 a la 32).

Cuando se ejecuta, la clase `IfElevatorTest` muestra lo siguiente (en la versión inglesa):

```
Opening door.
Door is open.
Closing door.
Door is closed.
Cannot Go down <--- la lógica del ascensor evita el
                    problema
Going up one floor.
Floor: 2
Going up one floor.
Floor: 3
...
```

Sentencias if anidadas

En ocasiones, puede ser necesario ejecutar una sentencia `if` como parte de otra sentencia `if`. En el ejemplo siguiente se muestra cómo usar sentencias `if` anidadas para comprobar los valores de dos variables.

Código 6-5 Archivo `NestedIfElevator.java` situado en el directorio `opsdec`

```
1
2   public class NestedIfElevator {
3
4     public boolean doorOpen=false; // Puertas cerradas de forma
predeterminada
5     public int currentFloor = 1; // Todos los ascensores empiezan en el
primer piso
6     public final int TOP_FLOOR = 10;
7     public final int MIN_FLOORS = 1;
8
9     public void openDoor() {
10      System.out.println(“Se está abriendo la puerta.”);
11      doorOpen = true;
12      System.out.println(“La puerta está abierta.”);
13    }
14
15    public void closeDoor() {
16      System.out.println(“Se está cerrando la puerta.”);
17      doorOpen = false;
18      System.out.println(“La puerta está cerrada.”);
19    }
20
21    public void goUp() {
22      System.out.println(“Subiendo una planta.”);
23      currentFloor++;
24      System.out.println(“Piso: “ + currentFloor);
25    }
26
27    public void goDown() {
28
29      if (currentFloor == MIN_FLOORS) {
30        System.out.println(“Imposible bajar”);
31      }
32
33      if (currentFloor > MIN_FLOORS) {
34
35        if (!doorOpen) {
36
```

```
37            System.out.println("Bajando una planta.");
38            currentFloor--;
39            System.out.println("Piso: " + currentFloor);
40          }
41        }
42      }
43
44
45  }
46
```

Si el valor de la variable `currentFloor` es igual que el de la constante `MIN_FLOORS`, aparecerá en la pantalla un mensaje de error y el ascensor no descenderá (líneas de la 29 a la 31). Si el valor de la variable `currentFloor` es mayor que el de la constante `MIN_FLOORS` y las puertas están cerradas, el ascensor baja (líneas de la 33 a la 40).

Nota – No utilice las construcciones `if/else` con demasiada frecuencia porque pueden ser complicadas de depurar.

Construcción `if/else`

A menudo sucede que se quiere ejecutar un bloque de código si la expresión es `true` y otro bloque de código cuando la expresión es `false`. Para estos casos, puede utilizar una construcción `if` para ejecutar un bloque de código si la expresión es `true` junto con una construcción `else` que sólo se ejecute en caso de que la expresión sea `false`. La sintaxis de las construcciones `if/else` es como sigue:

```
if (expresión_booleana) {

   bloque_código1;

} // fin de la construcción if

else {

   bloque_código2;

} // fin de la construcción else

// el programa continúa aquí
```

Donde:

- *expresión_booleana* es una combinación de operadores relacionales, operadores condicionales y valores que dan como resultado el valor `true` o `false`.
- *bloque_código1* representa las líneas de código que se ejecutan si la *expresión* es `true` y *bloque_código2* representa las líneas de código que se ejecutan si la *expresión* es `false`.

Puede utilizar una sentencia `if/else` para corregir el problema del ascensor que se dirige a una planta no válida. El método `goDown` siguiente contiene una construcción `if` que determina si el ascensor debería descender o mostrar un error.

Código 6-6 Archivo `IfElseElevator.java` situado en el directorio opsdec

```
1   public class IfElseElevator {
2
3     public boolean doorOpen=false; // Puertas cerradas de forma
predeterminada
4     public int currentFloor = 1; // Todos los ascensores empiezan en el
primer piso
5     public final int TOP_FLOOR = 10;
6     public final int MIN_FLOORS = 1;
7
8     public void openDoor() {
9       System.out.println(“Se está abriendo la puerta.”);
10      doorOpen = true;
11      System.out.println(“La puerta está abierta.”);
12    }

13    public void closeDoor() {
14      System.out.println(“Se está cerrando la puerta.”);
15      doorOpen = false;
16      System.out.println(“Door is closed.”);
17    }
18
19    public void goUp() {
20      System.out.println(“Subiendo una planta.”);
21      currentFloor++;
22      System.out.println(“Piso: “ + currentFloor);
23    }
24
25    public void goDown() {
26
```

```
27      if (currentFloor == MIN_FLOORS) {
28        System.out.println(“Imposible bajar”);
29      }

30      else {
31        System.out.println(“Bajando una planta.”);
32        currentFloor--;
33        System.out.println(“Piso: “ + currentFloor);
34      }
35    }
36  }
```

Si el valor de la variable `currentFloor` es igual que el de la constante `MIN_FLOORS`, aparecerá en la pantalla un mensaje de error y el ascensor no descenderá (líneas de la 27 a la 30). En caso contrario (`else`), se considerará que el valor de `currentFloor` es mayor que el de la constante `MIN_FLOORS` y el ascensor bajará de planta (líneas de la 31 a la 34). Cuando se ejecuta, la clase `IfElseElevatorTest` muestra lo siguiente (en la versión inglesa):

```
Opening door.
Door is open.
Closing door.
Door is closed.
Cannot Go down <--- la lógica del ascensor evita el
                    problema
Going up one floor.
Floor: 2
Going up one floor.
Floor: 3
...
```

Construcciones `if/else` encadenadas

Es posible encadenar varias construcciones `if` y `else` para evaluar sucesivas expresiones que den lugar a diferentes salidas en función del resultado. La sintaxis de las construcciones `if/else` encadenadas es como sigue:

```
if (expresión_booleana) {

   bloque_código1;

} // fin de la construcción if

else if (expresión_booleana) {

   bloque_código2;

} // fin de la construcción else if

else {
   bloque_código3;
}
// el programa continúa aquí
```

Donde:

- *`expresión_booleana`* es una combinación de operadores relacionales, operadores condicionales y valores que da como resultado el valor `true` o `false`.
- *`bloque_código1`* representa las líneas de código que se ejecutan si la *`expresión`* es `true`. *`bloque_código2`* representa las líneas de código que se ejecutan si la *`expresión`* es `false` y la condición del segundo `if` es `true`. *`bloque_código3`* representa las líneas de código que se ejecutan si la *`expresión`* del segundo `if` también se evalúa como `false`.

A continuación se muestra una clase `IfElseDate` con varias construcciones `if/else` encadenadas que determinan cuántos días tiene un mes.

Código 6-7 Archivo `IfElseDate.java` situado en el directorio `opsdec`

```
1
2   public class IfElseDate {
3
4     public int month = 10;
5
6     public void calculateNumDays() {
7
8       if (month == 1 || month == 3 || month == 5 || month == 7 ||
9      month == 8 || month == 10 || month == 12) {
10
11        System.out.println("Ese mes tiene 31 días.");
12      }
13
14      else if (month == 2) {
15        System.out.println("Ese mes tiene 28 días.");
16      }
17
18      else if (month == 4 || month == 6 || month == 9 || month == 11) {
19        System.out.println("Ese mes tiene 30 días.");
20      }
21
22      else {
23        System.out.println("Mes no válido");
24      }
25    }
26  }
27
```

El método `calculateNumDays` encadena tres sentencias `if/else` para determinar el número de días que tiene un mes. Aunque el código es sintácticamente correcto, el encadenamiento de sentencias `if/else` puede dar como resultado un código algo confuso y debería evitarse.

Uso de la construcción switch

Otra palabra clave utilizada para tomar decisiones en el programa es `switch`. La construcción `switch` ayuda a evitar confusiones en la programación porque simplifica la organización de los distintos bloques de código que pueden ejecutarse.

El ejemplo de la clase `IfElseDate` podría escribirse utilizando una construcción `switch`. La sintaxis de la construcción `switch` es:

```
switch (variable) {
    case valor_literal:
        bloque_código;
        [break;]
    case otro_valor_literal:
        bloque_código;
        [break;]
    [default:]
        bloque_código;
}
```

Donde:

- La palabra clave `switch` indica una sentencia `switch`.
- *`variable`* es la variable cuyo valor se quiere verificar. La variable sólo puede ser de los tipos `char`, `byte`, `short` o `int`.
- La palabra clave `case` indica un valor que se está comprobando. La combinación de la palabra `case` y un *`valor_literal`* se conoce como etiqueta case.
- El *`valor_literal`* es cualquier valor válido que pueda contener una variable. Es posible tener una etiqueta case por cada valor que se quiera evaluar. Los valores literales no pueden ser ni variables, ni expresiones ni llamadas a métodos. Sí pueden ser constantes (variables finales como `MAX_NUMBER`, definida anteriormente), literales (como `'A'` o `10`) o ambas cosas.
- La sentencia `[break;]` es una palabra clave opcional que provoca la salida inmediata de la sentencia `switch` durante la ejecución. Sin esta sentencia, se ejecutarán todas las sentencias de `bloque_código` que sigan a la sentencia case aceptada (hasta llegar a una sentencia `break` o al final de la construcción `switch`).

- La palabra clave `default` indica un *`bloque_código`* predeterminado que se ejecuta si no se cumple ninguna de las otras condiciones. La etiqueta `default` es similar a la construcción `else` de las sentencias `if/else`. La palabra clave `default` es opcional.

El ejemplo siguiente contiene una clase `SwitchDate` que utiliza una construcción `switch` para determinar cuántos días tiene un mes.

Código 6-8 Archivo `SwitchDate.java` situado en el directorio `opsdec`

```
1
2   public class SwitchDate {
3
4     public int month = 10;
5
6     public void calculateNumDays() {
7
8       switch(month) {
9       case 1:
10      case 3:
11      case 5:
12      case 7:
13      case 8:
14      case 10:
15      case 12:
16        System.out.println(“Ese mes tiene 31 días.”);
17        break;
18      case 2:
19        System.out.println(“Ese mes tiene 28 días.”);
20        break;
21      case 4:
22      case 6:
23      case 9:
24      case 11:
25        System.out.println(“Ese mes tiene 30 días.”);
26        break;
27      default:
28        System.out.println(“Mes no válido”);
29        break;
30      }
31    }
32  }
33
```

El método `calculateNumDays` de la clase `SwitchDate` utiliza una sentencia `switch` para bifurcar el código en función del valor de la variable `month` (línea 8). Si la variable `month` es igual a 1, 3, 5, 7, 8 o 10, el código salta hasta la etiqueta case adecuada y desciende para ejecutar la línea 16.

Cuándo usar construcciones switch

La construcción `switch` sólo se utiliza para verificar igualdades, no para comprobar si los valores son mayores o menores que otro valor. No es posible usar `switch` para realizar comprobaciones con respecto a varios valores y sólo puede utilizarse con tipos enteros.

La construcción `switch` puede considerarse como una opción válida para:

- Pruebas de igualdad
- Pruebas con respecto a un *solo* valor, como `customerStatus`
- Pruebas con respecto al valor de un tipo `int`, `short`, `byte` o `char`

Nota – Es posible anidar otras construcciones, como `if/else`, dentro de las sentencias `switch`.

Módulo 7

Uso de construcciones en bucle

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Crear bucles `while`.
- Desarrollar bucles `for`.
- Crear bucles `do/while`.

En este módulo se explican los distintos tipos de construcciones en *bucle* que controlan la repetición de un conjunto de sentencias de programación en lenguaje Java.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Crear bucles `while`.			
Desarrollar bucles `for`.			
Crear bucles `do/while`.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Aspectos relevantes

Discusión – La pregunta siguiente es relevante para comprender los conceptos sobre los bucles:

¿Cuáles son aquellas situaciones en que querría seguir realizando una determinada acción mientras se dé una cierta condición?

Creación de bucles while

Un bucle while repite la ejecución (itera) de un bloque de código mientras una determinada expresión dé como resultado el valor true. La sintaxis del bucle while es como sigue:

```
while (expresión_booleana) {

   bloque_código;

} // fin de la construcción while

// el programa continúa aquí
```

Todos los bucles tienen los siguientes componentes:

- *expresión_booleana* es una expresión que sólo puede dar como resultado true o false. Esta expresión se procesa antes de cada iteración del bucle.
- *bloque_código* representa las líneas de código que se ejecutan si la *expresión_booleana* es true.

El bucle while repite la ejecución del bloque de código de cero a muchas veces. La parte de la expresión_booleana se procesa antes que el cuerpo del bucle y, si el primer resultado de la evaluación es false, el cuerpo del bucle no se procesa en absoluto. Por ejemplo:

```
int i=0;
while (i<5) {
   System.out.println("Hola Tomás");
   ++i
}
```

En el ejemplo anterior, la frase "Hola Tomás" aparece cinco veces.

El siguiente ejemplo de código muestra un método `setFloor` con un bucle `while` que hace que el ascensor suba o baje.

Código 7-1 Archivo `WhileElevator.java` situado en el directorio `loops`

```
1
2   public class WhileElevator {
3
4     public boolean doorOpen=false;
5     public int currentFloor = 1;
6
7     public final int TOP_FLOOR = 5;
8     public final int BOTTOM_FLOOR = 1;
9
10    public void openDoor() {
11      System.out.println(“Se está abriendo la puerta.”);
12      doorOpen = true;
13      System.out.println(“La puerta está abierta.”);
14    }
15
16    public void closeDoor() {
17      System.out.println(“Se está cerrando la puerta.”);
18      doorOpen = false;
19      System.out.println(“La puerta está cerrada.”);
20    }
21
22    public void goUp() {
23      System.out.println(“Subiendo una planta.”);
24      currentFloor++;
25      System.out.println(“Piso: “ + currentFloor);
26    }
27
28    public void goDown() {
29      System.out.println(“Bajando una planta.”);
30      currentFloor--;
31      System.out.println(“Piso: “ + currentFloor);
32    }
33
34    public void setFloor() {
35
36      // Normalmente pasaría el valor de desiredFloor como argumento al
37      // método setFloor, pero, como aún no ha aprendido a
38      // hacer esta operación, desiredFloor se define con un número
específico (5)
39      // a continuación.
40
```

```
41      int desiredFloor = 5;
42
43      while (currentFloor != desiredFloor){
44        if (currentFloor < desiredFloor) {
45            goUp();
46            }
47        else {
48            goDown();
49            }
50        }
51
52  }
53
```

El método `setFloor` de la clase `Elevator` utiliza un bucle `while` para determinar si el ascensor se encuentra en el piso elegido (líneas de la 35 a la 50). Si el valor de `currentFloor` no es igual que el de la variable `desiredFloor`, el ascensor continuará su recorrido hasta el piso elegido (línea 43).

Bucles `while` anidados

Piense lo que necesitaría para dibujar un rectángulo como el que se muestra a continuación introduciendo los caracteres de uno en uno:

```
@@@@@@@@@@
@@@@@@@@@@
@@@@@@@@@@
```

Podría utilizar un bucle `while` para dibujar una fila del rectángulo y colocar ese bucle dentro de otro bucle para dibujar tres filas. El segundo bucle es un bucle anidado.

El siguiente ejemplo de código imprime en la pantalla un rectángulo de caracteres "@" compuesto de 10 columnas y 3 filas (3 filas formadas por 10 caracteres "@"). La escritura de cada fila se realiza mediante el bucle interno. El bucle externo imprime el resultado de ese código tres veces.

Código 7-2 Archivo `WhileRectangle.java` situado en el directorio `loops`

```
1   public class WhileRectangle {

2     public int height = 3;
3     public int width = 10;

4     public void displayRectangle() {

5       int colCount = 0;
6       int rowCount = 0;

7       while (rowCount < height) {
8         colCount=0;

9         while (colCount < width) {
10          System.out.print("@");
11          colCount++;
12        }

13        System.out.println();
14        rowCount++;
15      }
16    }
17  }
```

El segundo bucle (interno) escribe en la pantalla una fila de caracteres "@" hasta que se alcanza el valor de la variable que indica la anchura, `width` (líneas de la 9 a la 12). El primer bucle (externo) comprueba si se ha alcanzado el valor de la variable que indica la altura, `height` (línea 7). Si no se ha alcanzado la altura, se crea una fila más utilizando el bucle interno. De lo contrario, el rectángulo estará completo.

Discusión – ¿Cómo modificaría el código del ejemplo anterior para imprimir un rectángulo de 4 por 11?

¿Como convertiría el rectángulo en otro de 3 por 10 sin usar los números 0, 1, 3 y 10?

Desarrollo de un bucle for

El bucle `for` permite al programa repetir una secuencia de sentencias un número predeterminado de veces. Este bucle funciona exactamente de la misma manera que el bucle `while`, incluido el hecho de que repite la ejecución de cero a muchas veces, pero tiene una estructura más pensada para recorrer un rango de valores. La sintaxis del bucle `for` es como sigue:

```
for (inicializar[,inicializar]; expresión_booleana;
   actualizar[,actualizar]) {

   bloque_código;

}
```

Donde:

- La parte *`inicializar`*`[,`*`inicializar`*`]` de la construcción `for` contiene sentencias que inicializan las variables (por ejemplo, los contadores del bucle) utilizadas a lo largo del bucle. Esta parte del código se procesa una vez, antes que ninguna otra parte del bucle. Las diferentes variables deben separarse mediante comas.
- *`expresión_booleana`* es una expresión que sólo puede dar como resultado `true` o `false`. Esta expresión se procesa antes de cada iteración del bucle.
- La sección *`actualizar`*`[,`*`actualizar`*`]` es donde se incrementa o decrementa el valor de las variables (contadores del bucle). Esta sección se procesa después del cuerpo pero antes de cada vez que se repite la evaluación de la *`expresión_booleana`*. Las diferentes variables deben separarse mediante comas.
- *`bloque_código`* representa las líneas de código que se ejecutan si la *`expresión_booleana`* es `true`.

El bucle `for` también repite la ejecución del código de cero a muchas veces. La parte del bucle que contiene la expresión se procesa antes del cuerpo del bucle y, si se evalúa inmediatamente como `false`, el cuerpo del bucle no se procesa. Por ejemplo:

```
for (int i=0; i<5; i++) {
   System.out.println("Hola Tomás");
}
```

En el ejemplo anterior, el saludo "Hola Tomás" aparece cinco veces.

El ejemplo siguiente muestra un método `setFloor` que contiene un bucle `for`.

Código 7-3 Archivo `ForElevator.java` situado en el directorio `loops`

```
1
2   public class ForElevator {
3
4     public boolean doorOpen=false;
5     public int currentFloor = 1;
6
7     public final int TOP_FLOOR = 5;
8     public final int BOTTOM_FLOOR = 1;
9
10    public void openDoor() {
11      System.out.println("Se está abriendo la puerta.");
12      doorOpen = true;
13      System.out.println("La puerta está abierta.");
14    }
15
16    public void closeDoor() {
17      System.out.println("Se está cerrando la puerta.");
18      doorOpen = false;
19      System.out.println("La puerta está cerrada.");
20    }
21
22    public void goUp() {
23      System.out.println("Subiendo una planta.");
24      currentFloor++;
25      System.out.println("Piso: " + currentFloor);
26    }
27
28    public void goDown() {
29      System.out.println("Bajando una planta.");
30      currentFloor--;
31      System.out.println("Piso: " + currentFloor);
32    }
33
34    public void setFloor() {
35
36      // Normalmente pasaría el valor de desiredFloor como argumento al
37      // método setFloor. pero, como aún no ha aprendido a
38      // hacer esta operación, desiredFloor se define con un número
específico (5)
39      // a continuación.
```

```
40      int desiredFloor = 5;
41
42      if (currentFloor > desiredFloor) {
43        for (int down = currentFloor; down != desiredFloor; --down) {
44           goDown();
45        }
46      }

47      else {
48        for (int up = currentFloor; up != desiredFloor; ++up) {
49           goUp();
50        }
51      }
52    }
53  }
54
```

En este ejemplo, el valor de la variable `currentFloor` se compara con el de la variable `desiredFloor` para determinar si el ascensor debería subir o bajar (línea 42). Dos bucles `for` llaman a los métodos `goDown` o `goUp` el número de veces necesarias hasta que el valor de la variable `desiredFloor` sea igual que el de la variable `currentFloor` (líneas de la 43 a la 45 y de la 48 a la 50).

Bucles for anidados

El siguiente ejemplo de código imprime en la pantalla un rectángulo de caracteres "@" compuesto de 10 columnas de anchura y 3 filas de altura (3 filas formadas por 10 caracteres "@"). El bucle interno imprime cada fila y el bucle externo imprime el resultado de ese código tres veces.

Código 7-4 Archivo `ForRectangle.java` situado en el directorio `loops`

```
1
2   public class ForRectangle {
3
4     public int height = 3;
5     public int width = 10;
6
7     public void displayRectangle() {
8
9       for (int rowCount = 0; rowCount < height; rowCount++) {
10        for (int colCount = 0; colCount < width; colCount++) {
11           System.out.print("@");
12        }
13        System.out.println();
14      }
15    }
16  }
17
```

El segundo bucle (interno) escribe en la pantalla una fila de caracteres "@" hasta que se alcanza el valor de la variable `width` (líneas de la 10 a la 12). El primer bucle (externo) comprueba si se ha alcanzado el valor de altura, `height` (línea 9). En caso negativo, se crea una fila más utilizando el bucle interno. De lo contrario, el rectángulo estará completo.

Codificación de un bucle do/while

El bucle `do/while` repite la ejecución del bloque de código de una a muchas veces: la condición se sitúa al final del bucle y se procesa *después* del cuerpo del bloque. Por tanto, el cuerpo del bucle se procesa al menos una vez. Si quiere que la sentencia o sentencias del cuerpo se procesen al menos una vez, utilice un bucle `do/while` en lugar de los bucles `while` o `for`. La sintaxis del bucle `do/while` es como sigue:

```
do {

   bloque_código;
}
while (expresión_booleana);// El signo de punto y coma
es obligatorio.
```

Donde:

- *`bloque_código`* representa las líneas de código que se ejecutan más de una vez si la *`expresión_booleana`* es `true`.
- *`expresión_booleana`* es una expresión que da como resultado `true` o `false`. La *`expresión_booleana`* se procesa después de cada iteración del bucle.

Nota – El signo de punto y coma después de la *`expresión_booleana`* es obligatorio porque dicha expresión se sitúa al final del bucle. Por el contrario, no es preciso usar el punto y coma al final de los bucles `while` porque la *`expresión_booleana`* se sitúa al principio del bucle y va seguida de un *`bloque_código`* que finaliza con una llave de cierre.

El siguiente ejemplo de código muestra un método `setFloor` con un bucle `do/while` que hace que el ascensor suba o baje.

Código 7-5 Archivo `DoWhileElevator.java` situado en el directorio `loops`

```
1
2   public class DoWhileElevator {
3
4     public boolean doorOpen=false;
5     public int currentFloor = 1;
6
7     public final int TOP_FLOOR = 5;
8     public final int BOTTOM_FLOOR = 1;
9
10    public void openDoor() {
11      System.out.println("Se está abriendo la puerta.");
12      doorOpen = true;
13      System.out.println("La puerta está abierta.");
14    }
15
16    public void closeDoor() {
17      System.out.println("Se está cerrando la puerta.");
18      doorOpen = false;
19      System.out.println("La puerta está cerrada.");
20    }
21
22    public void goUp() {
23      System.out.println("Subiendo una planta.");
24      currentFloor++;
25      System.out.println("Piso: " + currentFloor);
26    }
27
28    public void goDown() {
29      System.out.println("Bajando una planta.");
30      currentFloor--;
31      System.out.println("Piso: " + currentFloor);
32    }
33
34    public void setFloor() {
35
36      // Normalmente pasaría el valor de desiredFloor como argumento al
37      // método setFloor. pero, como aún no ha aprendido a
38      // hacer esta operación, desiredFloor se define con un número
específico (5)
39      // a continuación.
```

```
40
41      int desiredFloor = 5;
42
43      do {
44        if (currentFloor < desiredFloor) {
45           goUp();
46        }
47       else if (currentFloor > desiredFloor) {
48           goDown();
49        }
50      }
51      while (currentFloor != desiredFloor);
52    }
53
54  }
55
```

El método `setFloor` de la clase `Elevator` utiliza un bucle `do/while` para determinar si el ascensor se encuentra en el piso elegido (líneas de la 43 a la 52). Si el valor de la variable `currentFloor` no es igual al de la variable `desiredFloor`, el ascensor continúa subiendo (línea 41) o bajando (línea 48).

Bucles do/while anidados

El siguiente código imprime en la pantalla un rectángulo de caracteres "@" compuesto de 10 columnas de anchura y 3 filas de altura (3 filas formadas por 10 caracteres "@"). La escritura de cada fila se realiza mediante el bucle interno. El bucle externo imprime el resultado del código tres veces.

Código 7-6 Archivo `DoWhileRectangle.java` situado en el directorio `loops`

```
1
2   public class DoWhileRectangle {
3
4     public int height = 3;
5     public int width = 10;
6
7     public void displayRectangle() {
8
9       int rowCount = 0;
10      int colCount = 0;
11
12      do {
13        colCount = 0;
14
15        do {
16           System.out.print("@");
17           colCount++;
18        }
19        while (colCount < width);
20
21        System.out.println();
22        rowCount++;
23      }
24      while (rowCount < height);
25    }
26  }
27
```

El segundo bucle (interno) escribe en la pantalla una fila de caracteres "@" hasta que se alcanza el valor de la variable `width` (líneas de la 15 a la 19). Al menos se imprime un carácter "@" antes de comprobar el valor de la variable `width`.

El primer bucle (externo) comprueba si se ha alcanzado el valor de la variable de altura, `height` (línea 24). Si no se ha alcanzado el valor de `height`, se crea una fila más utilizando el bucle interno. De lo contrario, el rectángulo estará completo. Al menos se crea una fila antes de comprobar el valor de la variable `height`.

Comparación de las construcciones en bucle

Los bucles `for`, `while` y `do/while` funcionan de forma muy similar. No obstante, en determinadas situaciones, una construcción será probablemente mejor que la otra. Utilice las directrices siguientes para determinar cuál de ellas utilizar:

- Utilice el bucle `while` para recorrer las sentencias indefinidamente y ejecutarlas cero o más veces.
- Utilice el bucle `do/while` para recorrer las sentencias indefinidamente y ejecutarlas *una* o más veces.
- Utilice el bucle `for` para recorrer las sentencias un número predefinido de veces. Este tipo de bucle es más compacto y fácil de leer que `while` porque está pensado para contabilizar un rango finito de valores.

Módulo 8

Desarrollo y uso de métodos

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Describir las ventajas de los métodos y definir métodos worker y de llamada.
- Declarar y llamar a un método.
- Comparar objetos y métodos estáticos.
- Usar métodos sobrecargados.

En este módulo se explica cómo crear y usar métodos para combinar la lógica del programa y llevar a cabo un determinada tarea.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Describir las ventajas de los métodos y definir métodos worker y de llamada.			
Declarar y llamar a un método.			
Comparar objetos y métodos estáticos.			
Usar métodos sobrecargados.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Notas

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender los conceptos sobre los métodos:

¿Cómo se estructuran o implementan las operaciones realizadas sobre un objeto?

Creación de métodos y llamadas a métodos

La mayor parte del código que se escribe para una clase está contenido dentro de uno o varios métodos. Los métodos permiten dividir el trabajo que realiza el programa en diferentes tareas o comportamientos lógicos.

Por ejemplo, puede que quiera desarrollar un programa que calcule y muestre en la pantalla una cifra total. Aunque todo el código necesario para realizar estas tareas puede incluirse en un solo método, agrupar las distintas tareas puede ayudar a crear programas orientados a objetos más apropiados. En otras palabras, se trata de dividir las tareas en bloques para que cada método pueda utilizarse de forma independiente. Por ejemplo, puede que quiera llamar a un método que calcule el total en una clase, pero no muestre el resultado del cálculo.

La sintaxis de todas las declaraciones de métodos es como sigue:

```
[modificadores] tipo_retorno identificador_método
([argumentos]) {
    bloque_código_método
}
```

Donde:

- `[modificadores]` representan determinadas palabras clave de Java que modifican la forma en que se utilizan los métodos. Los modificadores son opcionales (como indican los corchetes).
- *`tipo_retorno`* es el tipo de valor que devuelve un método y que puede utilizarse en algún otro lugar del programa. Los métodos pueden devolver sólo un elemento (valor literal, variable, referencia a un objeto, etc.). Si el método no debe devolver ningún valor, es preciso especificar la palabra clave `void` como tipo de retorno.
- *`identificador_método`* es el nombre del método.
- Los `([argumentos])` representan una lista de variables cuyos valores se pasan al método para que éste los utilice. Los argumentos son opcionales (como indican los corchetes) y muchos métodos no aceptan argumentos.
- *`bloque_código_método`* es una secuencia de sentencias ejecutadas por el método. En el bloque o cuerpo del código del método pueden tener lugar una amplia variedad de tareas.

¿Lo sabía? – A menudo, un conjunto de declaraciones de métodos dentro de una clase se conoce con el nombre de *interfaz* porque define la forma en que los objetos deben interaccionar o relacionarse con otros objetos. Las declaraciones de métodos también se denominan *prototipos de métodos* porque proporcionan al programador suficiente información como para determinar la forma en que se utiliza el método.

Forma básica de un método

El método en su forma básica no acepta argumentos ni devuelve ningún valor. El siguiente método `display` de la clase `Shirt` es un método básico:

```
public void displayInformation() {
    System.out.println(“ID de camisa: " + shirtID);
    System.out.println(“Descripción de la camisa :” + description);
    System.out.println("Código de color: " + colorCode);
    System.out.println("Precio de la camisa: " + price);
    System.out.println("Cantidad en stock: " + quantityInStock);
} // fin del método display
```

Este método muestra en la pantalla varias líneas de información tal como el valor de las variables `itemID` y `price`.

Llamada a un método desde una clase diferente

Para llamar o ejecutar un método situado en otra clase, puede utilizar el operador punto (.) con una variable de referencia a objeto tal y como lo haría para acceder a las variables `public` de un objeto. En el siguiente ejemplo de código se muestra la clase `ShirtTest`.

Código 8-1 Archivo `ShirtTest.java` situado en el directorio `getstarted`

```
1
2   public class ShirtTest {
3
4      public static void main (String args[]) {
5
6      Shirt myShirt;
7      myShirt = new Shirt();
8
9      myShirt.displayInformation();
10
11
12     }
13  }
14
```

En este ejemplo, se declara e inicializa una variable de referencia a un objeto llamada `myShirt` con un objeto `Shirt` en las líneas 6 y 7. A continuación, esa variable hace una llamada al método `display` dentro del objeto `Shirt` (línea 9).

Métodos worker y de llamada

En el ejemplo anterior, la clase `ShirtTest` llama al método `display` desde el interior de otro método (el método `main`). Por tanto, el método `main` se considera como el método de llamada porque está invocando o "llamando" a otro método para que realice algún trabajo. Por su parte, el método `display` se considera como el método de ejecución o "worker" porque ejecuta algún trabajo para el método `main`.

Nota – Muchos métodos pueden ser de llamada o workers porque no sólo realizan algún trabajo, sino que además llaman a otros métodos.

Cuando un método de llamada ejecuta un método worker, el primero detiene su propia ejecución hasta que finaliza el método worker. Cuando éste finaliza, se indica al flujo del programa que regrese al punto del método de llamada situado justo después de la llamada al método worker (vuelve a la línea 9 de la clase `ShirtTest.java`).

En la figura siguiente se ilustra el concepto de métodos worker y de llamada.

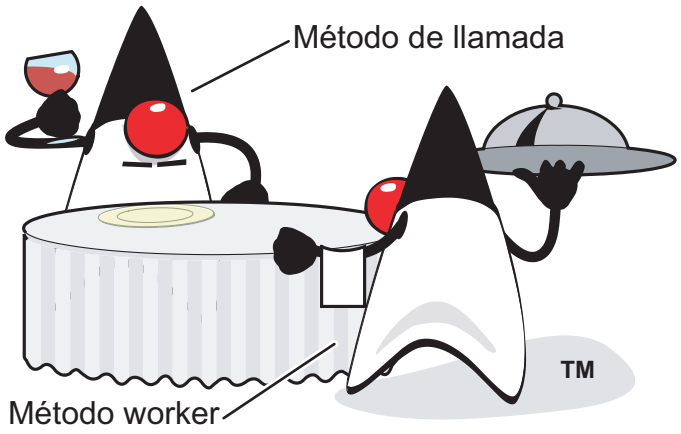


Figura 8-1 Métodos worker y de llamada

Llamada a un método en la misma clase

Llamar a un método dentro de la misma clase es fácil. Sólo tiene que incluir el nombre del método worker y sus argumentos, si los tiene.

El ejemplo de código siguiente contiene una clase con un método que llama a otro método en la misma clase.

Código 8-2 Archivo `Elevator.java` situado en el directorio `methods`

```
1
2   public class Elevator {
3
4     public boolean doorOpen=false;
5     public int currentFloor = 1;
6
7     public final int TOP_FLOOR = 5;
8     public final int BOTTOM_FLOOR = 1;
9
10    public void openDoor() {
11      System.out.println("Se está abriendo la puerta.");
12      doorOpen = true;
13      System.out.println("La puerta está abierta.");
14    }
15
16    public void closeDoor() {
17      System.out.println("Se está cerrando la puerta.");
18      doorOpen = false;
19      System.out.println("La puerta está cerrada.");
20    }
21
22    public void goUp() {
23      System.out.println("Subiendo una planta.");
24      currentFloor++;
25      System.out.println("Piso: " + currentFloor);
26    }
27
28    public void goDown() {
29      System.out.println("Bajando una planta.");
30      currentFloor--;
31      System.out.println("Piso: " + currentFloor);
32    }
33
34    public void setFloor(int desiredFloor) {
35      while (currentFloor != desiredFloor){
```

```
36        if (currentFloor < desiredFloor){
37           goUp();
38           }
39        else {
40           goDown();
41           }
42     }
43    }
44
45    public int getFloor() {
46      return currentFloor;
47    }
48
49    public boolean checkDoorStatus() {
50      return doorOpen;
51    }
52  }
53
```

El método `setFloor` llama a los métodos `goUp` y `goDown` (líneas 37 y 40). Los tres métodos están incluidos en la clase `Elevator`.

Directrices para realizar llamadas a métodos

Éstas son las normas para crear y realizar llamadas a métodos:

- No existen límites en cuanto al número de llamadas a métodos que puede realizar un método de llamada.
- El método de llamada y el worker pueden estar en la misma clase o en clases diferentes.
- La forma de llamar al método worker varía según se encuentre en la misma clase o en una clase distinta de la del método de llamada.
- Las llamadas a métodos pueden hacerse en cualquier orden. Su ejecución no necesariamente debe efectuarse en el orden en el que figuran dentro de la clase en la que se han declarado (la clase que contiene los métodos worker).

Autoevaluación – Seleccione las declaraciones de métodos que se ajusten a la sintaxis de declaración de métodos adecuada.

a. ___ `public displayInfo()`

b. ___ `public void displayInfo`

c. ___ `public String getColor()`

d. ___ `public void displayInfo()`

e. ___ `public setColor(int Color)`

Autoevaluación – ¿Cómo llamaría al método `setColor` desde la clase `Circle` (es decir, desde la misma clase en la que se ha declarado)?

```
public class Circle {
   private int radius;
   private String color;

   public void setColor(String myColor){
      color = myColor;
   }
}
```

Respuesta: _____

Paso de argumentos y devolución de valores

Como hemos mencionado, es posible hacer llamadas a métodos mediante un método de llamada con una lista de argumentos (variables o valores que el método worker puede utilizar). Asimismo, los métodos pueden devolver un valor al método de llamada para que éste lo utilice.

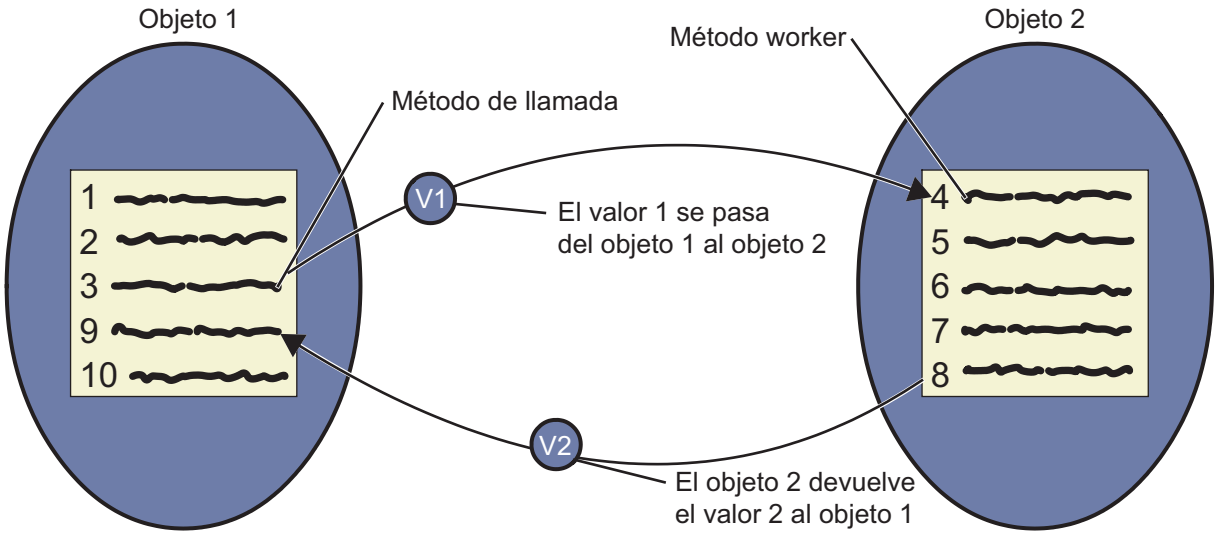


Figura 8-2 Paso y devolución de valores

Declaración de métodos con argumentos

Muchos métodos de los que se declaran o se usan a través de las bibliotecas de clases de Java aceptan argumentos. Por ejemplo, el siguiente método `setFloor` de la clase `Elevator` acepta el valor `int desiredFloor`, que se utiliza en el cuerpo del método.

```
public void setFloor(int desiredFloor) {
    while (currentFloor != desiredFloor) {
      if (currentFloor < desiredFloor) {
        goUp();
      }
      else {
        goDown();
      }
   }
}
public void multiply(int numberOne, int numberTwo)
```

Método `main`

Recuerde que el método `main` que ha estado utilizado en las clases de prueba también debe escribirse de forma que acepte argumentos: una o varias (un array) referencias a objetos `String`.

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

¿Por qué el método `main` acepta estos argumentos? El lenguaje Java obliga a escribir el método `main` de forma que acepte valores cuando el programa se ejecute desde la línea de comandos. Por ejemplo, para pasar un precio de `12.99` y el color `R` a la clase `ShirtTest`, ejecute la clase utilizando la máquina virtual de Java con el argumento adicional de precio:

```
java ShirtTest 12.99 R
```

Para que los valores numéricos pasados al método `main` sean útiles para el programa, deben convertirse las secuencias de caracteres en un tipo de datos primitivo.

Nota – Pasar argumentos utilizando el método `main` es una función de utilidad para probar los programas. No necesita pasar argumentos al método `main` para cumplir ninguno de los objetivos de este curso. Asimismo, para manejar los argumentos pasados al método `main`, es preciso tener conocimientos de utilización de arrays, que es un tema de nivel intermedio explicado más adelante en este curso.

Llamadas a métodos con argumentos

Para pasar argumentos de un método a otro, incluya los argumentos en el paréntesis de la llamada del método. Como argumentos, puede pasar valores literales o variables. Como en el caso de la asignación de valores literales a las variables, debe escribir las cadenas literales entre comillas, usar la letra F después de los tipos `float`, etc.

Escriba la lista de argumentos en el mismo orden que tengan en la declaración del método worker y pase todos los argumentos necesarios. El compilador verifica si el tipo, orden y número de los parámetros pasados coincide con el tipo, orden y número de los parámetros aceptados por el método.

Nota – No es necesario usar los mismos nombres de variables como argumentos al llamar a un método. No obstante, los nombres de variables deben ser de un tipo de datos compatible (puede pasar un tipo byte a un tipo `int`, etc.).

En el ejemplo siguiente se muestra cómo se pasa un valor numérico a un método que realiza una operación utilizando el valor.

Código 8-3 Archivo `ElevatorTest.java` situado en el directorio `methods`

```
1   public class ElevatorTest {
2
3      public static void main(String args[]) {
4
5       Elevator myElevator = new Elevator();
6
7        myElevator.openDoor();
8        myElevator.closeDoor();
9        myElevator.goUp();
10       myElevator.goUp();
11       myElevator.goUp();
12       myElevator.openDoor();
13       myElevator.closeDoor();
14       myElevator.goDown();
15       myElevator.openDoor();
16       myElevator.closeDoor();
17       myElevator.goDown();
18
19       myElevator.setFloor(myElevator.TOP_FLOOR);
20
21       myElevator.openDoor();
22     }
23  }
24
```

Este ejemplo llama al método `setFloor` con un valor entero para establecer el piso de destino (línea 19).

Declaración de métodos con valores de retorno

La mayoría de las declaraciones de métodos que ha visto no devuelven ningún valor (`void`). Sin embargo, muchos de los métodos que creará tendrán valores de retorno y muchos métodos de las bibliotecas de clases de Java también los tienen.

Para declarar un método que devuelva un valor, coloque el tipo de valor de retorno deseado antes del identificador del método. Por ejemplo, la siguiente declaración de método acepta dos tipos `int` y devuelve un tipo `int`:

```
public int sum(int numberOne, int numberTwo)
```

Nota – Un método sólo puede devolver un valor, pero puede aceptar múltiples argumentos.

Devolución de un valor

Para devolver un valor de un método, utilice la palabra clave `return`. Por ejemplo, el código siguiente devuelve el valor contenido en la variable `sum`.

```
public int sum(int numberOne, int numberTwo) {
   int result= numberOne + numberTwo;

   return result;
}
```

El siguiente método `getFloor` de la clase `Elevator` devuelve el valor contenido en la variable `currentFloor`.

```
public int getFloor() {
     return currentFloor;
  }
```

Recepción de valores de retorno

Si llama a un método que devuelve un valor, como en el caso del método `getFloor` anterior, puede usar el valor de retorno en el método de llamada.

El ejemplo de código siguiente contiene una clase de prueba que crea una instancia de la clase `Elevator`, llama a su método `getFloor` y espera de vuelta un valor `int`.

Código 8-4 Archivo `ElevatorTestTwo.java` situado en el directorio `methods`

```
1
2   public class ElevatorTestTwo {
3
4      public static void main(String args[]) {
5
6       Elevator myElevator = new Elevator();
7
8        myElevator.openDoor();
9        myElevator.closeDoor();
10       myElevator.goUp();
11       myElevator.goUp();
12       myElevator.goUp();
13       myElevator.openDoor();
14       myElevator.closeDoor();
15       myElevator.goDown();
16       myElevator.openDoor();
17       myElevator.closeDoor();
18       myElevator.goDown();
19
20       int curFloor = myElevator.getFloor();
21       System.out.println(“Piso actual: “ + curFloor);
22
23       myElevator.setFloor(curFloor+1);
24
25       myElevator.openDoor();
26     }
27  }
28
```

En la línea 20 se declara la variable `curFloor` y se utiliza para recibir el valor de retorno entero del método `getFloor` de la clase Elevator.

Ventajas de usar métodos

Los métodos son los mecanismos mediante los cuales interaccionan los objetos. En los programas Java, es habitual tener varios objetos que interaccionan a través de llamadas recíprocas a sus métodos. He aquí algunas ventajas de usar métodos en los programas:

- Los métodos facilitan la lectura y el mantenimiento de los programas.

  Por ejemplo, es mucho más fácil imaginar lo que hace un programa si el código se divide en diferentes métodos con nombres que explican el comportamiento de cada uno de ellos.

- Los métodos agilizan el desarrollo y el mantenimiento del código.

  Por ejemplo, es posible optar por crear y probar un programa método a método para garantizar que funcionará como un todo una vez que se haya finalizado.

- Los métodos son un componente esencial del software reutilizable.

  Por ejemplo, las bibliotecas de clases de Java contienen numerosas clases con métodos que pueden utilizarse una y otra vez en los programas. Igualmente, un programador puede escribir métodos para que otros programadores también los utilicen.

- Los métodos permiten separar objetos para comunicar y distribuir el trabajo realizado por el programa.

  Un método de un objeto puede llamar a un método de otro objeto. El objeto puede pasar la información del método y recibir un valor de retorno.

Creación de métodos y variables static

Hasta ahora hemos explicado cómo acceder a los métodos y las variables creando un objeto de la clase a la que pertenecen el método o la variable y llamando al método o accediendo a la variable (si ésta es de tipo public). Los métodos y variables que son exclusivos de una instancia se denominan métodos y variables de instancia.

También hemos explicado el uso de métodos que no requieren instanciación de objetos, como es caso del método main. Éstos se denominan métodos de clase o estáticos y se les puede llamar sin tener que crear primero un objeto.

Igualmente, el lenguaje Java permite crear variables estáticas o variables de clase, que pueden utilizarse sin crear ningún objeto.

Nota – main es un método static que ya ha utilizado. No necesita crear ninguna instancia de objeto para usarlo.

Declaración de métodos static

Los métodos estáticos se declaran con la palabra clave static.
A continuación figura la declaración del método getProperties en la clase System del API de Java:

```
static Properties getProperties()
```

Llamada a los métodos static

Dado que los métodos static o de clase no forman parte de ninguna instancia de objeto (sólo de la clase) no deberían utilizarse variables de referencia a objetos para llamarlos. En su lugar, debe utilizarse el nombre de la clase. La sintaxis para llamar a un método static es como sigue:

```
Nombreclase.método();
```

¿Lo sabía? – La máquina virtual de Java utiliza el nombre de clase suministrado en la línea de comandos para llamar al método `main`. Por ejemplo, si se escribe `java Shirt`, la máquina virtual llama al método `Shirt.main`.

A continuación se muestra un ejemplo de un método que podría agregarse a la clase `Shirt` para convertir los números de tamaño de las camisas en tamaños expresados en forma de caracteres, como pequeña, mediana o grande. Este método es `static` porque:

- No utiliza directamente ningún atributo de la clase `Shirt`.
- Puede que sea conveniente llamar al método aunque no se disponga de ningún objeto `Shirt`.

```
public static char convertShirtSize(int numericalSize) {

    if (numericalSize < 10) {
      return 'S';
    }

    else if (numericalSize < 14) {
      return 'M';
    }

    else if (numericalSize < 18) {
      return 'L';
    }

    else {
      return 'X';
    }
}
```

El método `convertShirtSize` acepta un tamaño numérico, determina a qué letra corresponde ese tamaño (S, M, L o X) y devuelve esa letra.

Por ejemplo, para acceder al método `static` `convertShirtSize` de la clase `Shirt`:

char size = Shirt.convertShirtSize(16);

Declaración de variables static

También puede usar la palabra clave `static` para indicar que sólo puede haber en la memoria una copia de la variable asociada a una clase, no una copia por cada instancia de un objeto. Por ejemplo:

```
static double salesTax = 8.25;
```

Acceso a las variables static

Para acceder a una variable `static`, debe utilizarse el nombre de la clase. La sintaxis para llamar a una variable `static` es como sigue:

```
Nombreclase.variable;
```

Por ejemplo, para acceder al valor de la variable `static` `PI` de la clase `Math`:

```
double myPI;
myPI = Math.PI;
```

¿Lo sabía? – Las variables pueden tener tanto el modificador `static` como el modificador `final` para indicar que sólo hay una copia de la variable y que su contenido no puede cambiarse. La variable `PI` de la clase `Math` es una variable `static` `final`.

Métodos y variables estáticos en el API de Java

Determinadas bibliotecas de clases de Java, como la clase `System`, sólo contienen métodos y variables `static`. La clase `System` contiene métodos utilitarios para manejar tareas específicas del sistema operativo (no actúan sobre instancias de objetos). Por ejemplo, el método `getProperties` de la clase `System` obtiene información sobre el equipo informático que se está utilizando.

Existen varias clases del API Java que son clases utilitarias. Estas clases contienen métodos `static` que son útiles para todo tipo de objetos. Algunos ejemplos de clases y métodos utilitarios son:

- Clase `Math`

  Esta clase contiene métodos y variables para realizar operaciones numéricas básicas tales como las funciones exponenciales, de raíz cuadrada, logarítmicas y trigonométricas elementales.

- Clase `System`

  Esta clase contiene métodos y variables para realizar funciones del nivel del sistema tales como recuperar información de las variables de entorno del sistema operativo.

Cuándo declarar un método o una variable `static`

He aquí algunas pistas para saber cuándo declarar un método o una variable como `static`. Considere la posibilidad de declarar un método o una variable como `static` si:

- No es importante realizar la operación sobre un objeto independiente o asociar la variable a un tipo de objeto concreto.
- Es importante acceder a la variable o el método antes de instanciar un objeto.
- El método o la variable no pertenece lógicamente a ningún objeto pero posiblemente pertenece a una clase utilitaria, como `Math`, incluida en el API de Java. La clase `Math` contiene varios métodos utilitarios `static` que pueden emplearse en ecuaciones matemáticas.

Autoevaluación – De las siguientes afirmaciones sobre los métodos y las variables de instancia, seleccione aquellas que sean ciertas.

a. ___ Las variables de instancia son exclusivas de objetos individuales.

b. ___ Los métodos y las variables de instancia necesitan instanciación de objetos.

c. ___ Los métodos y las variables de instancia no necesitan instanciación de objetos.

d. ___ Los métodos y las variables de instancia se declaran con la palabra clave `instance`.

e. ___ El acceso a los métodos y las variables de instancia suele realizarse mediante una variable de referencia a un objeto.

Autoevaluación – De las siguientes afirmaciones sobre las variables y los métodos estáticos, seleccione aquellas que sean ciertas.

a. ___ Los métodos y variables estáticos son exclusivos de objetos individuales.

b. ___ Los métodos y variables estáticos necesitan instanciación de objetos.

c. ___ Los métodos y variables estáticos no necesitan instanciación de objetos.

d. ___ Los métodos y variables estáticos se declaran con la palabra clave `static`.

e. ___ A menudo, la llamada a los métodos estáticos se realiza utilizando el nombre de la clase.

Uso de la sobrecarga de métodos

En programación Java, puede haber varios métodos en una clase que tengan el mismo nombre pero diferentes argumentos (diferentes firmas de método). Este concepto se denomina sobrecarga de métodos. De la misma forma que puede distinguir entre dos alumnos de la misma clase que se llamen Juan llamándolos, por ejemplo, “Juan el de la camisa verde” y “Juan el moreno”, también puede diferenciar dos métodos por su nombre y sus argumentos.

Por ejemplo, puede que quiera crear varios métodos para agregar dos números como, por ejemplo, dos tipos `int` o dos tipos `float`. Con la sobrecarga de métodos, puede crear varios métodos con el mismo nombre pero diferentes firmas. Por ejemplo, el siguiente ejemplo de código contiene varios métodos sum cada uno de los cuales acepta un conjunto distinto de argumentos.

Código 8-5 Archivo `Calculator.java` situado en el directorio `methods`

```
1
2   public class Calculator {
3
4      public int sum(int numberOne, int numberTwo){
5
6        System.out.println(“Método uno”);
7
8        return numberOne + numberTwo;
9      }
10
11    public float sum(float numberOne, float numberTwo) {
12
13       System.out.println(“Método dos”);
14
15       return numberOne + numberTwo;
16    }
17
18    public float sum(int numberOne, float numberTwo) {
19
20      System.out.println(“Método tres”);
21
22      return numberOne + numberTwo;
23    }
24  }
25
```

El primer método `sum` (líneas de la 4 a la 9) acepta dos argumentos `int` y devuelve un valor `int`. El segundo método `sum` (líneas de la 11 a la 16) acepta dos argumentos `float` y devuelve un valor `float`. El tercer método `sum` (líneas de la 18 a la 23) acepta un tipo `int` y un tipo `float` como argumentos y devuelve un tipo `float`.

Para llamar a cualquiera de los métodos `sum` anteriores, el compilador compara la firma de método de la llamada con las firmas de método de una clase. Por ejemplo, a continuación se muestra un método `main` que llama a cada uno de los métodos `sum` contenidos en un objeto `Calculator`.

Código 8-6 Archivo `CalculatorTest.java` situado en el directorio `methods`

```
1   public class CalculatorTest {
2
3     public static void main(String [] args) {
4
5       Calculator myCalculator = new Calculator();
6
7       int totalOne = myCalculator.sum(2,3);
8       System.out.println(totalOne);
9
10      float totalTwo = myCalculator.sum(15.99F, 12.85F);
11      System.out.println(totalTwo);
12
13      float totalThree = myCalculator.sum(2, 12.85F);
14      System.out.println(totalThree);
15    }
16  }
```

Sobrecarga de métodos y el API de Java

Muchos métodos del API de Java están sobrecargados, incluido el método `System.out.println`. La tabla siguiente contiene todas las variaciones del método `println`.

Tabla 8-1 Variaciones del método `System.out.println`

Método	Uso
`void println()`	Finaliza la línea actual escribiendo la secuencia de separación de líneas.
`void println(boolean x)`	Imprime en la pantalla un valor `boolean` y luego finaliza la línea.
`void println(char x)`	Imprime un carácter y termina la línea.
`void println(char[] x)`	Imprime un array de caracteres y termina la línea.
`void println(double x)`	Imprime un tipo `double` y termina la línea.
`void println(float x)`	Imprime un tipo `float` y termina la línea.
`void println(int x)`	Imprime un tipo `int` y termina la línea.
`void println(long x)`	Imprime un tipo `long` y termina la línea.
`void println(Objeto x)`	Imprime un objeto y termina la línea.
`void println(String x)`	Imprime una secuencia de caracteres y termina la línea.

Es posible imprimir en la pantalla valores de tipo `String`, `int`, `float` u otros tipos utilizando el método `System.out.println` porque es un método sobrecargado que utiliza diferentes argumentos según el tipo de dato.

Autoevaluación – ¿Cuántas variaciones del método `System.out.print` existen en el API de Java?

a. ___ 7

b. ___ 10

c. ___ 9

d. ___ 8

e. ___ 3

Sin la sobrecarga, necesitaría crear varios métodos con diferentes nombres para imprimir diferentes tipos de datos, por ejemplo, `printlnint`, `printlnfloat`, etc. O bien tendría que realizar gran número de conversiones de unos tipos en otros antes de usar un método.

Usos de la sobrecarga de métodos

Cuando escriba código, recuerde que conviene definir métodos sobrecargados si las acciones asociadas al método deben realizarse con diferentes tipos de datos. También puede utilizar la sobrecarga para crear varios métodos con el mismo nombre pero con diferente número de parámetros. Por ejemplo, podría crear tres métodos sum, cada uno con un número de argumentos diferente para sumar:

```
public int sum(int numberOne, int numberTwo)
public int sum(int numberOne, int numberTwo, int numberThree)
public int sum(int numberOne, int numberTwo,int numberThree, int
    numberFour)
```

El siguiente ejemplo de código contiene un método sobrecargado con diferentes argumentos.

Código 8-7 Archivo `ShirtTwo.java` situado en el directorio `methods`

```
1
2   public class ShirtTwo {
3
4     public int shirtID = 0; // ID predeterminado para la camisa
5     public String description = “-description required-”; //
predeterminada
6
7     // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
8     public char colorCode = ‘U’;
9     public double price = 0.0; // Precio predeterminado para todas las
camisas
10    public int quantityInStock = 0; // Cantidad predeterminada para
todos los artículos
11
12    public void setShirtInfo(int ID, String desc, double cost){
13      shirtID = ID;
14      description = desc;
15      price = cost;
16    }
17
18    public void setShirtInfo(int ID, String desc, double cost, char
color){
19      shirtID = ID;
20      description = desc;
21      price = cost;
22      colorCode = color;
23    }
24
25    public void setShirtInfo(int ID,  String desc, double cost, char
color, int quantity){
26      shirtID = ID;
27      description = desc;
28      price = cost;
29      colorCode = color;
30      quantityInStock = quantity;
31    }
32
33    // Este método muestra los valores de un artículo
34    public void display() {
35
36      System.out.println(“ID de artículo: “ + shirtID);
```

```
37      System.out.println("Descripción del artículo:" + description);
38      System.out.println("Código de color: " + colorCode);
39      System.out.println("Precio del artículo: " + price);
40      System.out.println("Cantidad en stock: " + quantityInStock);
41
42    } // fin del método display
43  } // fin de la clase
44
```

La clase `ShirtTwo` contiene un método, `setShirtInformation`, que se ha sobrecargado dos veces: una para aceptar un valor de `colorCode` (línea18) y otra para aceptar los valores de `colorCode` y `quantityInStock` (línea 25).

En el siguiente ejemplo de código se crean tres instancias de objeto de la clase `ShirtTwo` y se utilizan sus métodos sobrecargados.

Código 8-8 Archivo `ShirtTwoTest.java` situado en el directorio `methods`

```
1   class ShirtTwoTest {
2
3     public static void main (String args[]) {
4       ShirtTwo shirtOne = new ShirtTwo();
5       ShirtTwo shirtTwo = new ShirtTwo();
6       ShirtTwo shirtThree = new ShirtTwo();
7
8       shirtOne.setShirtInfo(100, "Button Down", 12.99);
9       shirtTwo.setShirtInfo(101, "Long Sleeve Oxford", 27.99, 'G');
10      shirtThree.setShirtInfo(102, "Shirt Sleeve T-Shirt", 9.99, 'B',
50);
11
12      shirtOne.display();
13      shirtTwo.display();
14      shirtThree.display();
15    }
16  }
17
```

La clase `ShirtTwoTest` llama al método `setShirtInfo` sobrecargado llamándolo tres veces:

1. La llamada al método `setShirtInfo` se realiza con un ID, una descripción y un precio como argumentos (línea 8).
2. La llamada al método `setShirtInfo` se realiza con un ID, una descripción, un precio y un código de color como argumentos (línea 9).
3. La llamada al método `setShirtInfo` se realiza con un ID, una descripción, un precio, un código de color y una cantidad en stock como argumentos (línea 10).

Módulo 9

Implementación de la encapsulación y los constructores

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Usar la encapsulación para proteger los datos.
- Crear constructores para inicializar objetos.

En este módulo se describe la forma de combinar los conceptos de programación OO y el lenguaje de programación Java presentados en este curso para escribir programas orientados a objetos con elementos encapsulados donde se utilice un buen diseño y una buena implementación.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = He podido cumplir el objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Usar la encapsulación para proteger los datos.			
Crear constructores para inicializar objetos.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender los conceptos sobre la encapsulación y los constructores:

- Los primeros ascensores y montacargas necesitaban la intervención de un operador que manejase una o varias poleas, cuerdas y engranajes para hacer funcionar el mecanismo. Los ascensores modernos ocultan al usuario los detalles y sólo necesitan la pulsación de unos cuantos botones para funcionar. ¿Cuáles son las ventajas de los ascensores modernos comparados con los antiguos?
- Muchos ascensores, como los montacargas de algunas fábricas, requieren el uso de llaves para poder poner la maquinaria en funcionamiento. Otros ascensores no pueden subir a determinados pisos (como la planta superior de un hotel) si no se utiliza una llave. ¿Por qué estas llaves son importantes?
- ¿Qué le sugieren las palabras *private* y *public*?

Uso de la encapsulación

En programación OO, el término encapsulación se refiere a la acción de ocultar los datos dentro de una clase (una “cápsula de seguridad”) y hacer que estén disponibles sólo a través de ciertos métodos. La encapsulación es importante porque facilita a los programadores el uso de clases de otros programadores e impide la modificación inadecuada de ciertos datos dentro de una clase.

En la figura siguiente se ilustra el concepto de encapsulación mediante una caja fuerte con una interfaz pública (la combinación de la caja) que, cuando se utiliza de la forma adecuada, permite acceder al contenido privado.

Figura 9-1 Encapsulación

El primer paso para conseguir que las clases estén bien encapsuladas es aplicar los modificadores de visibilidad apropiados a la clase, las variables de atributo y las declaraciones de métodos.

Modificadores de visibilidad

Los atributos y métodos pueden tener modificadores, como `public`, que indican los niveles de acceso que otros objetos tienen para utilizar el atributo o el método. Los modificadores más habituales son `public` y `private`.

Modificador public

En todos los ejemplos vistos en este curso se utiliza el modificador `public`. Este modificador hace que la clase, sus atributos y métodos estén visibles para cualquier objeto de su programa.

En la figura siguiente se ve la imagen del ascensor de un hotel con acceso libre (público) a cualquier planta de un edificio, incluidas aquellas áreas donde visitantes o huéspedes no registrados normalmente no entrarían.

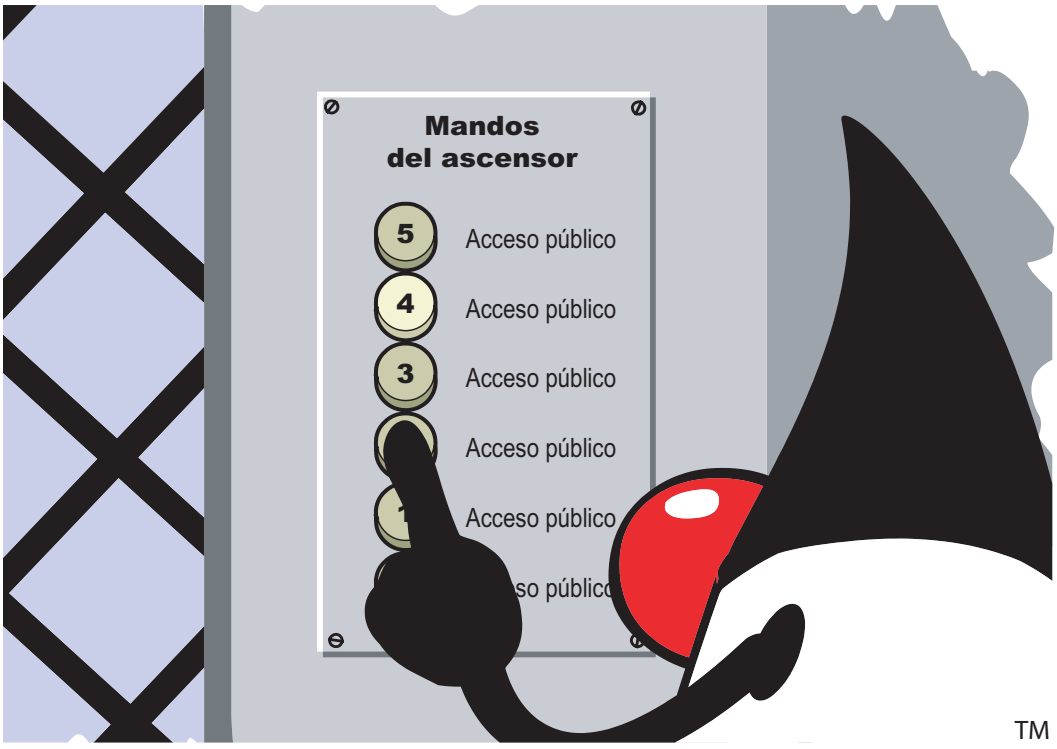


Figura 9-2 Plantas de acceso público con el ascensor

Para hacer que un atributo o método sea de acceso público, ponga el modificador `public` delante del método o de la variable correspondiente al atributo.

```
public int currentFloor=1;

public void setFloor(int desiredFloor) {
   ...
}
```

Problemas potenciales derivados de los atributos Public

El ejemplo de código siguiente ilustra los problemas que surgen cuando todos los atributos de un programa son `public`.

Código 9-1 Archivo `PublicElevator.java` situado en el directorio `object_structure`

```
1
2   public class PublicElevator {
3
4      public boolean doorOpen=false;
5      public int currentFloor = 1;
6      public int weight =0;
7
8      public final int CAPACITY=1000;
9      public final int TOP_FLOOR = 5;
10     public final int BOTTOM_FLOOR = 1;
11  }
12
```

En el código anterior, todos los atributos son `public`, lo que permite cambiar sus valores sin comprobar posibles errores.

El ejemplo siguiente muestra cómo podría escribirse un programa para acceder directamente a los atributos de un objeto `PublicElevator`, lo que provoca varios problemas.

Código 9-2 Archivo `PublicElevatorTest.java` situado en el directorio `object_oriented`

```
1
2   public class PublicElevatorTest {
3
4     public static void main(String args[]) {
5
6       PublicElevator pubElevator = new PublicElevator();
7
8       pubElevator.doorOpen = true; //los usuarios entran en el ascensor
9       pubElevator.doorOpen = false; //las puertas se cierran
10      //descender a la planta 0 (debajo de la planta baja del edificio)
11      pubElevator.currentFloor--;
12      pubElevator.currentFloor++;
13
14      //pasar a la planta 7 (sólo hay 5 plantas en el edificio)
15      pubElevator.currentFloor = 7;
```

```
16      pubElevator.doorOpen = true; //los usuarios entran/salen
17      pubElevator.doorOpen = false;
18      pubElevator.currentFloor = 1; //ir a la primera planta
19      pubElevator.doorOpen = true; //los usuarios entran/salen
20      pubElevator.currentFloor++; //el ascensor se mueve con la puerta
abierta
21      pubElevator.doorOpen = false;
22      pubElevator.currentFloor--;
23      pubElevator.currentFloor--;
24    }
25  }
26
```

Como la clase `PublicElevator` no utiliza encapsulación, la clase `PublicElevatorTest` puede cambiar los valores de sus atributos con libertad y de muchas formas no deseables. Por ejemplo, en la línea 11, el valor del atributo `currentFloor` se reduce a 0, que podría no ser un piso válido. Asimismo, en la línea 15, el atributo `currentFloor` se define con el valor 7, que, de acuerdo con la constante `TOP_FLOOR`, no es un piso válido (sólo hay cinco plantas). Éstas son sólo dos formas en las que la clase `PublicElevatorTest` podría modificar un objeto `PublicElevator` y provocar problemas en un programa.

Nota – En general, debería usar el modificador `public` sólo con métodos y variables de atributos a los que deban acceder directamente otros objetos.

Modificador `private`

El modificador `private` impide que otros objetos puedan acceder a los objetos de una determinada clase, sus atributos y sus métodos.

En la figura siguiente se muestra la imagen de un ascensor donde hay una planta de acceso privado. El acceso privado a determinados pisos protege a sus huéspedes y propiedades del acceso no autorizado por parte de visitantes o personas no registradas.

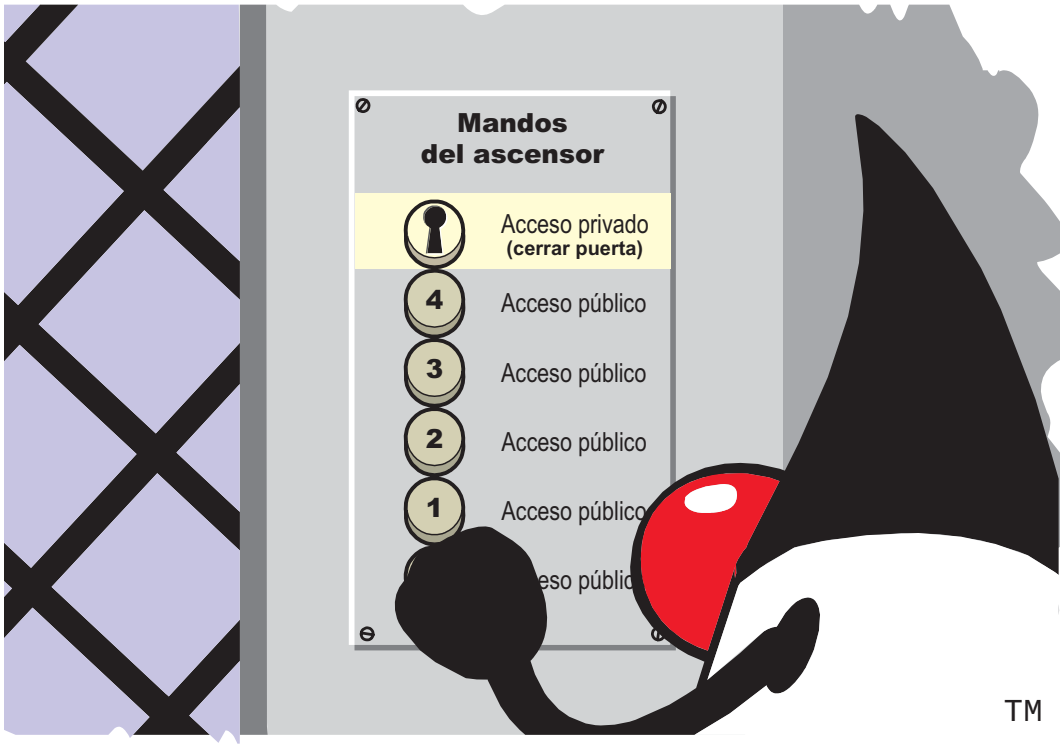


Figura 9-3 Plantas de acceso privado con el ascensor

Para hacer que un atributo o método sea de acceso privado, ponga el modificador `private` delante del método o de la variable correspondiente al atributo.

```
private int currentFloor=1;

private void calculateCapacity() {
    ...
}
```

El siguiente ejemplo de código ilustra la forma de encapsular los datos dentro del ejemplo anterior del ascensor para impedir que los datos se puedan modificar de forma inadecuada.

Código 9-3 Archivo `PrivateElevator1.java` situado en el directorio `object_oriented`

```
1
2   public class PrivateElevator1 {
3
4      private boolean doorOpen=false;
5      private int currentFloor = 1;
6      private int weight =0;
7
8      private final int CAPACITY=1000;
9      private final int TOP_FLOOR = 5;
10     private final int BOTTOM_FLOOR = 1;
11  }
12
```

La clase `PrivateElevator1` sólo contiene variables de atributos `private`.

En el siguiente ejemplo de código, una referencia a un objeto intenta modificar las variables privadas de la clase `PrivateElevator1`.

Código 9-4 Archivo `PrivateElevator1Test.java` situado en el directorio `object_oriented`

```
1
2   public class PrivateElevator1Test {
3
4     public static void main(String args[]) {
5
6       PrivateElevator1 privElevator = new PrivateElevator1();
7
8       /*********************************************
9       * Las siguientes líneas de código no se compilan *
10      * porque intentan acceder a variables de         *
11      * acceso privado.                                *
12      *********************************************/
13
14      privElevator.doorOpen = true; //los usuarios entran en el ascensor
15      privElevator.doorOpen = false; //las puertas se cierran
16      //descender a currentFloor 0 (debajo de la planta baja del
edificio)
17      privElevator.currentFloor--;
18      privElevator.currentFloor++;
```

```
19
20      //pasar a currentFloor 7 (sólo hay 5 plantas en el edificio)
21      privElevator.currentFloor = 7;
22      privElevator.doorOpen = true; //los usuarios entran/salen del
ascensor
23      privElevator.doorOpen = false;
24      privElevator.currentFloor = 1; //ir a la primera planta
25      privElevator.doorOpen = true; //los usuarios entran/salen del
ascensor
26      privElevator.currentFloor++;  //el ascensor se mueve con la
puerta abierta
27      privElevator.doorOpen = false;
28      privElevator.currentFloor--;
29      privElevator.currentFloor--;
30    }
31  }
32
```

El ejemplo de código anterior no se compila porque el método `main` de la clase `PrivateElevator1Test` está intentando cambiar el valor de los atributos `private` de la clase `PrivateElevator1` (líneas de la 14 a la 29).

Sin embargo, la clase `PrivateElevator1` no es de mucha utilidad porque no hay forma de modificar los valores de la clase.

En un programa ideal, la mayoría de los atributos de una clase, o todos ellos, se mantienen como `private`. Ninguna clase externa a su propia clase podrá ver ni modificar los atributos privados. Únicamente podrán verlos y modificarlos los métodos de su clase. Estos métodos deberían contener el código y la lógica de negocio necesarios para garantizar que no se asignarán valores inadecuado a las variables de los atributos.

Interfaz e implementación

Las declaraciones de métodos y variables de atributos `public` a menudo se conocen como la interfaz de una clase porque son los únicos elementos que otra clase puede utilizar. Los detalles sobre cómo una clase realiza una operación dentro de un método se denomina implementación del método. Uno de los objetivos de una buena programación OO es declarar métodos y variables de atributos `public` de forma que la implementación pueda cambiar sin que ello afecte a la interfaz.

En la figura siguiente se ilustra la interfaz de un ascensor y dos implementaciones distintas. Puede haber muchos ascensores en el mundo que tengan la misma interfaz y diferentes implementaciones.

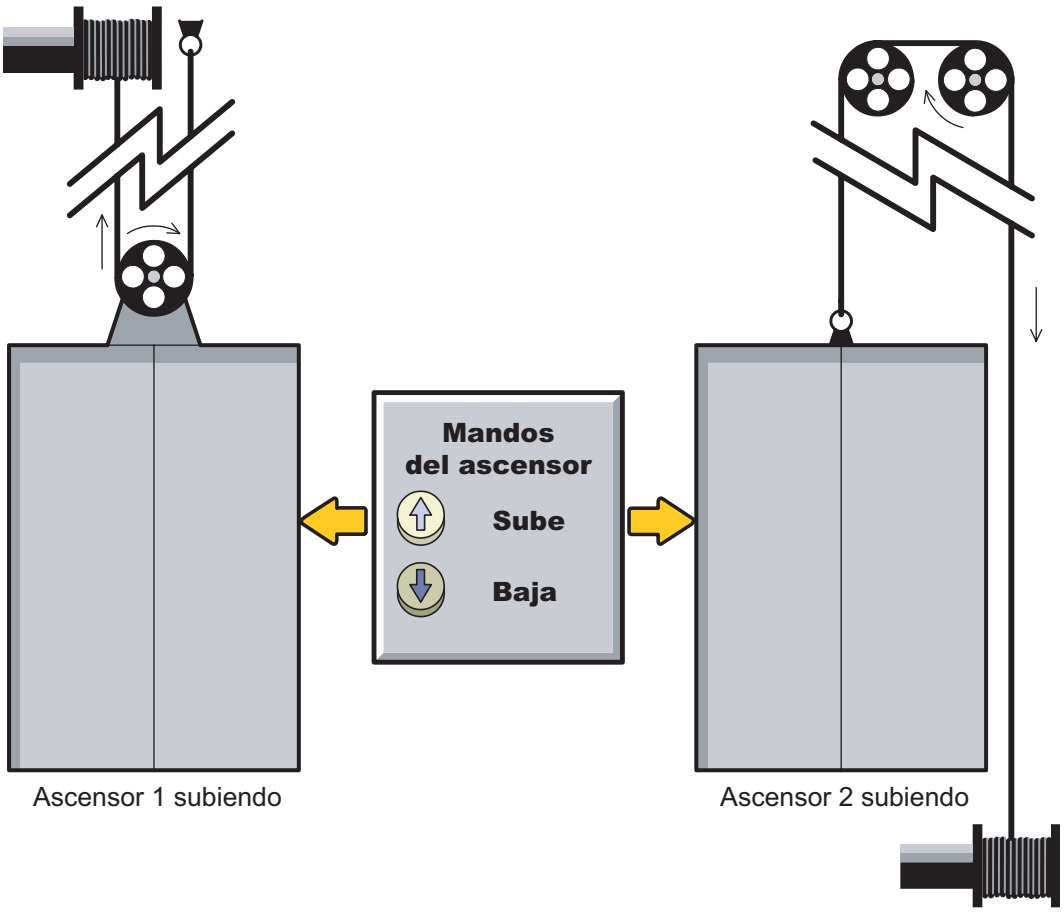


Figura 9-4 Interfaz e implementación

Cuando las clases se encapsulan, otros objetos interaccionan sólo con algunas partes (métodos) de las otras clases. Por ejemplo, un programador puede cambiar el bloque de código de un método `print` tanto como sea necesario, pero, si la declaración de dicho método `print` no cambia, el código que hace referencia a esa declaración tampoco cambiará.

Métodos get y set

Si se asigna el modificador `private` a los atributos, ¿cómo pueden acceder a ellos otros objetos? Un objeto puede acceder a los atributos `private` de otro objeto si éste último proporciona métodos `public` para cada operación que deban efectuar con los valores de los atributos. Por ejemplo, es habitual proporcionar métodos `public` para establecer (set) y obtener (get) el valor contenido en cada atributo `private` de una clase.

El ejemplo siguiente contiene una clase `PrivateShirt1` con atributos `private` y un método `public` para obtener y establecer el valor de la variable `colorCode`.

Código 9-5 Archivo `PrivateShirt1.java` situado en el directorio `object_oriented`

```

public class PrivateShirt1 {

  private int shirtID = 0; // ID predeterminado para la camisa
  private String description = “-description required-”; // predeterminada

  // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
  private char colorCode = ‘U’;
  private double price = 0.0; // Precio predeterminado para todos los artículos
  private int quantityInStock = 0; // Cantidad predeterminada para todos los artículos

  public char getColorCode() {
    return colorCode;
  }

  public void setColorCode(char newCode) {
    colorCode = newCode;
  }

  // Se agregarían otros métodos get y set para shirtID, description,
  // price y quantityInStock

} // fin de la clase

```

Aunque el código es sintácticamente correcto, el método `setcolorCode` no contiene la lógica necesaria para garantizar el establecimiento de los valores correctos.

La clase de prueba (test) del siguiente ejemplo establece un código de color no válido en un objeto `PrivateShirt1`.

Código 9-6 Archivo `PrivateShirt1Test.java` situado en el directorio `object_oriented`

```
1
2   public class PrivateShirt1Test {
3
4     public static void main (String args[]) {
5
6     PrivateShirt1 privShirt = new PrivateShirt1();
7     char colorCode;
8
9     // Establece un código de color (colorCode) válido
10    privShirt.setColorCode('R');
11    colorCode = privShirt.getColorCode();
12
13    // La clase PrivateShirtTest1 puede establecer un valor válido para
colorCode
14    System.out.println("Código de color: " + colorCode);
15
16    // Establece un código de color no válido
17    privShirt.setColorCode('Z');
18    colorCode = privShirt.getColorCode();
19
20    // La clase PrivateShirtTest1 puede establecer un valor no válido
para colorCode
21    System.out.println("Código de color: " + colorCode);
22    }
23  }
24
```

El código de la clase test anterior no sólo establece un color válido (R) en la línea 10, sino también un color incorrecto (Z) en la línea 17. Puede hacerlo porque la clase `PrivateShirt1` no está adecuadamente encapsulada para impedir el establecimiento de códigos de color no válidos.

¿Lo sabía? – Los métodos destinados a obtener y establecer valores a menudo se conocen como métodos *get* y *set*.

Los métodos que utilice deberían contener lógica de negocio adicional para validar la acción que se va a realizar. Por ejemplo, deberían comprobar si los valores se encuentran en el rango adecuado antes de establecerlos.

A continuación figura otra versión de la clase `PrivateShirt1`. No obstante, antes de establecer el valor, esta clase se asegura de que todos los valores que recibe sean válidos.

Código 9-7 Archivo `PrivateShirt2.java` situado en el directorio `object_oriented`

```
1
2   public class PrivateShirt2 {
3
4     private int shirtID = 0; // ID predeterminado para la camisa
5     private String description = “-description required-”; //
predeterminada
6
7     // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
8     private char colorCode = ‘U’;
9     private double price = 0.0; // Precio predeterminado para todos los
artículos
10    private int quantityInStock = 0; // Cantidad predeterminada para
todos los artículos
11
12    public char getColorCode() {
13      return colorCode;
14    }
15
16    public void setColorCode(char newCode) {
17
18      switch (newCode) {
19      case ‘R’:
20      case ‘G’:
21      case ‘B’:
22        colorCode = newCode;
23        break;
24      default:
25        System.out.println(“Código de color no válido. Utilice R, G o
B”);
26      }
27    }
28
29    // Se agregarían otros métodos get y set para shirtID, description,
30    // price y quantityInStock
31
32  } // fin de la clase
33
```

El método `setColorCode` (línea 16) comprueba la validez del código de color utilizando una sentencia `switch`. Si se pasa un código de color no válido al método `setColorCode`, aparece un error en la pantalla (línea 25).

Después de escribir una clase encapsulando sus datos con la palabra clave `private` y declarar los métodos get y set, puede escribir una clase que llame a los citados métodos para acceder a los valores del objeto.

El ejemplo de código siguiente contiene una clase de prueba que llama a los métodos get y set de la clase `PrivateShirt2` encapsulada.

Código 9-8 Archivo `PrivateShirt2Test.java` situado en el directorio `object_oriented`

```
1
2   public class PrivateShirt2Test {
3
4     public static void main (String args[]) {
5        PrivateShirt2 privShirt = new PrivateShirt2();
6        char colorCode;
7
8        // Establece un código de color (colorCode) válido
9        privShirt.setColorCode('R');
10       colorCode = privShirt.getColorCode();
11
12       // La clase PrivateShirtTest2 puede establecer un valor válido
para colorCode
13       System.out.println("Código de color: " + colorCode);
14
15       // Establece un código de color no válido
16       privShirt.setColorCode('Z');
17       colorCode = privShirt.getColorCode();
18
19       // La clase PrivateShirtTest2 puede establecer un valor no
válido para colorCode
20       // El código de color sigue siendo R
21       System.out.println("Código de color: " + colorCode);
22    }
23  }
24
```

La clase de prueba del ejemplo anterior no puede asignar el color incorrecto (Z) en la línea 16. No puede hacerlo porque `PrivateShirt2Test` está adecuadamente encapsulada para impedir el establecimiento de códigos de color no válidos.

Programa de ascensor encapsulado

En el ejemplo de código siguiente se ilustra la forma de encapsular el programa del ascensor para impedir problemas durante su uso.

Código 9-9 Archivo `PrivateElevator2.java` situado en el directorio `object_oriented`

```
1
2   public class PrivateElevator2 {
3
4     private boolean doorOpen=false;
5     private int currentFloor = 1;
6     private int weight = 0;
7
8     private final int CAPACITY = 1000;
9     private final int TOP_FLOOR = 5;
10    private final int BOTTOM_FLOOR = 1;
11
12    public void openDoor() {
13      doorOpen = true;
14    }
15
16    public void closeDoor() {
17      calculateCapacity();
18
19      if (weight <= CAPACITY) {
20        doorOpen = false;
21      }
22      else {
23        System.out.println("El ascensor está sobrecargado.");
24        System.out.println("¡Las puertas permanecerán abiertas hasta
          que salga alguien!");
25      }
26    }
27
28    // En realidad, el ascensor debería tener sensores de peso para
29    // comprobar el peso, pero, para mayor
30    // simplicidad, hemos utilizado un número aleatorio para
31    // representar la carga que lleva
32
33     private void calculateCapacity() {
```

```
34      weight = (int) (Math.random() * 1500);
35      System.out.println("El peso es " + weight);
36    }
37
38    public void goUp() {
39      if (!doorOpen) {
40        if (currentFloor < TOP_FLOOR) {
41          currentFloor++;
42          System.out.println(currentFloor);
43          }
44          else {
45             System.out.println("Ya está en la planta superior.");
46             }
47       }
48       else {
49          System.out.println("¡Las puertas siguen abiertas!");
50          }
51       }
52
53    public void goDown() {
54      if (!doorOpen) {
55        if (currentFloor > BOTTOM_FLOOR) {
56          currentFloor--;
57          System.out.println(currentFloor);
58          }
59        else {
60          System.out.println("Ya está en la planta baja.");
61          }
62       }
63       else {
64          System.out.println("¡Las puertas siguen abiertas!");
65          }
66     }
67
68    public void setFloor(int desiredFloor) {
69      if ((desiredFloor >= BOTTOM_FLOOR) && (desiredFloor<=TOP_FLOOR))
{
70
71        while (currentFloor != desiredFloor) {
72     if (currentFloor < desiredFloor) {
73       goUp();
74     }
75
76     else {
77       goDown();
78     }
```

```
79        }
80      }
81      else {
82        System.out.println(“Piso no válido”);
83      }
84    }
85
86    public int getFloor() {
87      return currentFloor;
88    }
89
90    public boolean getDoorStatus() {
91      return doorOpen;
92    }
93  }
```

El ejemplo anterior contiene atributos `private` y métodos `public` que se utilizan para acceder a las variables de los atributos. Los métodos garantizan que los valores que se establezcan para las variables de atributo serán apropiados.

En el ejemplo de código siguiente se muestra cómo acceder a las variables de atributo de la clase `PrivateElevator2` utilizando métodos public.

Código 9-10 Archivo `PrivateElevator2Test.java` situado en el directorio `object_oriented`

```
1
2   public class PrivateElevator2Test {
3
4      public static void main(String args[]) {
5
6        PrivateElevator2 privElevator = new PrivateElevator2();
7
8        privElevator.openDoor();
9        privElevator.closeDoor();
10       privElevator.goDown();
11       privElevator.goUp();
12       privElevator.goUp();
13       privElevator.openDoor();
14       privElevator.closeDoor();
15       privElevator.goDown();
16       privElevator.openDoor();
17       privElevator.goDown();
18       privElevator.closeDoor();
19       privElevator.goDown();
20       privElevator.goDown();
21
22       int curFloor = privElevator.getFloor();
23
24       if (curFloor != 5 && ! privElevator.getDoorStatus()) {
25           privElevator.setFloor(5);
26       }
27
28       privElevator.setFloor(10);
29       privElevator.openDoor();
30     }
31  }
32
```

Dado que la clase `PrivateElevator2` no permite la manipulación directa de los atributos de la clase, `PrivateElevator2Test` sólo puede hacer llamadas a métodos que actúen sobre las variables de atributos de la clase. Estos métodos realizan comprobaciones para conseguir que se utilicen los valores correctos antes de realizar una tarea y que el ascensor no haga nada inesperado.

Toda la lógica compleja de este programa está encapsulada en los métodos `public` de la clase `PrivateElevator2`. Por tanto, el código de la clase de prueba es fácil de leer y mantener. Este concepto es una de las muchas ventajas de la encapsulación.

Ejemplo de resultado del código

La salida de la clase `PrivateElevator2` es distinta cada vez que se ejecuta. A continuación se muestra un ejemplo del resultado de la clase `Elevator2Test` (en la versión inglesa).

Código 9-11 Archivo `sample.txt` situado en el directorio `object_oriented`

```
The weight is 453
Already on bottom floor.
2
3
The weight is 899
2
Doors still open!
The weight is 974
1
Already on bottom floor.
2
3
4
5
```

La clase `Elevator2` no admite instrucciones para efectuar tareas no válidas porque está adecuadamente encapsulada. Por ejemplo, el ascensor no puede desplazarse a un piso no válido ni cargar más de 500 kilos de peso o moverse cuando las puertas están abiertas.

Descripción del ámbito de las variables

Las variables declaradas dentro de un método, un constructor u otro bloque de código no pueden utilizarse a lo largo de una clase. El ámbito de una variable se refiere a la medida en que esa variable puede utilizarse dentro de un programa. Por ejemplo, las variables de atributos se declaran al comienzo de una clase, lo que permite utilizarlas a lo largo de todo el objeto. No obstante, si una variable se define dentro de un método, sólo puede utilizarse dentro de ese método. Este tipo de variables se denominan variables locales.

En el ejemplo siguiente se muestra el ámbito de la variable de atributo age y la variable name.

Código 9-12 Archivo `Person2.java` situado en el directorio `object_oriented`

```
1   public class Person2 {
2
3     // principio del ámbito de int age
4     private int age = 34;
5
6     public void displayName() {

7       // principio del ámbito de String name
8       String name = “Peter Simmons”;
9       System.out.println(“Me llamo “ + name + “ y tengo “ + age );
10
11    }   // fin del ámbito de String name
12
13    public String getName () {
14
15      return name; // esto provoca un error
16    }
17  }   // fin del ámbito de int age
```

Dado que la variable de atributo age está definida fuera de los métodos (línea 4), dicha variable existirá mientras dure un objeto basado en la clase (el ámbito de age es la clase completa).

Sin embargo, la variable `name` se ha declarado en el método `displayName` (línea 6), lo que hace que sólo exista mientras dure el método. Si se hace referencia a la variable `name` desde otro método como, por ejemplo, `getName` (líneas de la 13 a la 16), provoca un error del compilador.

Dado que las variables declaradas en bucles o sentencias `if` sólo son válidas en el interior del bucle o la sentencia, debería identificar los problemas relativos al ámbito de las variables en todo momento, especialmente al utilizar bucles y sentencias `if`.

Forma en que las variables de instancia y locales aparecen en la memoria

Las variables de instancia o atributo se almacenan en una parte de la memoria distinta de aquella utilizada para almacenar las variables locales.

En la figura siguiente se ilustra cómo se almacenan las variables de instancia y las locales.

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

  int counter = 100;
  Shirt myShirt = new Shirt ( );
  myShirt.shirtID = 425566 ;
}
```

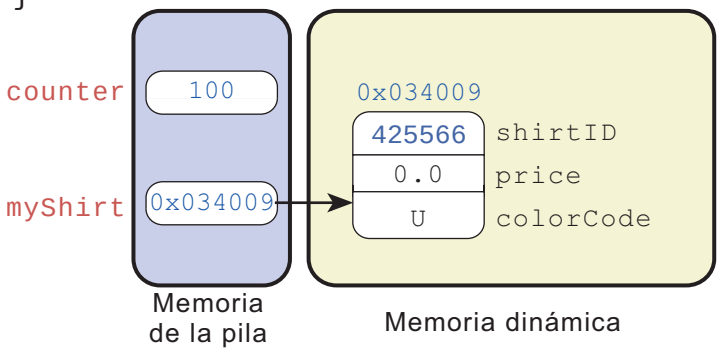


Figura 9-5 Variables de instancia y locales en la memoria

La variable `shirtID` es una variable de instancia o atributo que está contenida en un objeto `Shirt`. Como hemos dicho anteriormente, todos los atributos de objetos y métodos se almacenan en el espacio de memoria dinámica.

`counter` es una variable local declarada dentro de un método, un constructor u otro bloque de código. La variable `counter` se almacena en la pila de memoria porque sólo es necesaria hasta que finaliza el método, el constructor el bloque de código que la contiene.

Autoevaluación – De las afirmaciones siguientes sobre las variables locales, seleccione aquellas que sean ciertas.

a. ___ Las variables locales también se denominan variables de atributo.

b. ___ Las variables locales se almacenan en la pila.

c. ___ Las variables locales se almacenan en el espacio de memoria dinámica.

d. ___ Las variables locales pueden utilizarse a lo largo de toda la clase.

e. ___ Las variables locales sólo están disponibles durante el tiempo que se ejecute un método, un constructor u otro bloque de código.

Autoevaluación – ¿Cuál es el ámbito de la variable `x`?

```
1   public void counter(int startingNumber) {
2
3         for (int x = startingNumber; x<100; ++x) {
4            System.out.println("Total:" + x);
5         }
6   }
```

a. ___ Líneas de la 1 a la 6

b. ___ Desde la línea 3 hasta el final de la 4

c. ___ Desde la línea 3 hasta la llave de cierre de la línea 5

d. ___ Desde la línea 3 hasta la llave de cierre de la línea 6

e. ___ Desde la línea 3 hasta el final de la clase

Creación de constructores

Los constructores son estructuras similares a los métodos a las que se llama automáticamente cuando se instancia un objeto. Normalmente se utilizan para inicializar valores en un objeto.

La sintaxis de los constructores es similar a la de las declaraciones de métodos:

```
[modificadores] class NombreClase {

   [modificadores] NombreConstructor([argumentos]) {
      bloque_código
   }
}
```

Donde:

- `[modificadores]` representa determinadas palabras clave de Java que modifican la forma en que se accede a los constructores. Los modificadores son opcionales (como indican los corchetes).
- `NombreConstructor` es el nombre de un método constructor. El nombre de constructor debe ser igual al valor de `NombreClase` en la declaración de clase.
- `[argumentos]` representa uno o varios valores opcionales pasados al constructor.
- `bloque_código` representa una o varias líneas de código opcionales del constructor.

El ejemplo de código siguiente muestra una clase con un constructor que establece el valor de una variable de atributo.

Código 9-13 Archivo `ConstructorShirt1.java` situado en el directorio `object_oriented`

```
1
2   public class ConstructorShirt1 {
3
4     private int shirtID = 0; // ID predeterminado para la camisa
5     private String description = “-description required-”; //
predeterminada
6
7     // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
8     private char colorCode = ‘U’;
9     private double price = 0.0; // Precio predeterminado para todos los
artículos
10    private int quantityInStock = 0; // Cantidad predeterminada para
todos los artículos
11
12    public ConstructorShirt1(char startingCode) {
13
14      switch (startingCode) {
15      case ‘R’:
16      case ‘G’:
17      case ‘B’:
18        colorCode = startingCode;
19        break;
20      default:
21        System.out.println(“Código de color no válido. Utilice R, G o
B”);
22      }
23    }
24
25    public char getColorCode() {
26       return colorCode;
27    }
28  } // fin de la clase
29
```

La clase mostrada en el ejemplo anterior contiene un constructor llamado `ConstructorShirt1` que acepta un valor `char` para inicializar el código de color de ese objeto (líneas de la 12 a la 23). El constructor `ConstructorShirt1` garantiza el paso de un código válido antes de establecer el código.

El siguiente ejemplo de código ilustra una clase que crea una variable de referencia a un objeto `ConstructorShirt1` e inicializa la variable de atributo `colorCode`.

Código 9-14 Archivo `ConstructorShirt1Test.java` situado en el directorio `object_oriented`

```
1
2   public class ConstructorShirt1Test {
3
4     public static void main (String args[]) {
5
6       ConstructorShirt1 constShirt = new ConstructorShirt1('R');
7       char colorCode;
8
9       colorCode = constShirt.getColorCode();
10
11      System.out.println("Código de color: " + colorCode);
12    }
13  }
14
```

En este ejemplo, se crea una variable de referencia a un objeto llamada `constShirt` y se inicializa con un nuevo objeto `ConstructorShirt1`. Cuando se crea el objeto `ConstructorShirt1`, la llamada al constructor de ese objeto se efectúa con un código de color (línea 6).

Nota – Como en el caso de los métodos, es posible sobrecargar los constructores incluyendo varios constructores en una clase, cada uno de ellos con el mismo nombre y diferentes series de argumentos.

Constructor predeterminado

Si el compilador de Java encuentra una clase que no tiene ningún constructor definido de forma explícita (uno que haya escrito el programador), introduce un constructor predeterminado. El constructor predeterminado sólo se crea para cumplir los requisitos del compilador.

De hecho, a lo largo de todo este curso hemos estado utilizando el constructor predeterminado. Cuando se utiliza el modificador `new` para instanciar un objeto cuya clase no contiene ningún constructor explícito, la palabra clave `new` llama automáticamente al constructor predeterminado de la clase.

```
Shirt shirt1 = new Shirt();
```

`Shirt()` es el constructor predeterminado (introducido por el compilador) de la clase `Shirt`. Sin embargo, el constructor predeterminado nunca aparece en el código en sí.

Si declara su propio constructor, o constructores, evitará que el compilador introduzca el constructor predeterminado en el código. Puesto que la clase `ConstructorShirtOne` ya contiene un constructor definido de forma expresa, la siguiente línea de código provocaría un error:

```
ConstructorShirt1 constShirt = new ConstructorShirt1();
```

No obstante, puede crear su propio constructor sin argumentos agregando a la clase un constructor que no acepte argumentos. En el siguiente ejemplo de código se muestra una clase que contiene un constructor con una declaración similar a la del constructor predeterminado.

Código 9-15 Archivo `DefaultShirt.java` situado en el directorio `object_oriented`

```
1
2   public class DefaultShirt {
3
4     private int shirtID = 0; // ID predeterminado para la camisa
5     private String description = “-description required-”; //
predeterminada
6
7     // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
8     private char colorCode = ‘U’;
9     private double price = 0.0; // Precio predeterminado para todos los
artículos
10    private int quantityInStock = 0; // Cantidad predeterminada para
todos los artículos
11
12    public DefaultShirt() {
13      colorCode = ‘R’;
14    }
15
16    public char getColorCode() {
17      return colorCode;
18    }
19  } // fin de la clase
20
```

En este ejemplo, cuando se efectúa la llamada al constructor predeterminado (línea 12), la variable de atributo `colorCode` se inicializa con “R”.

Sobrecarga de constructores

Al igual que los métodos, los constructores también pueden sobrecargarse. La sobrecarga de constructores ofrece una amplia variedad de formas de crear e inicializar objetos utilizando una sola clase. En el ejemplo de código siguiente, hay tres constructores.

Código 9-16 Archivo `ConstructorShirt2.java` situado en el directorio `object_oriented`

```
1
2   public class ConstructorShirt2 {
3
4     private int shirtID = 0; // ID predeterminado para la camisa
5     private String description = “-description required-”; //
predeterminada
6
7     // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
8     private char colorCode = ‘U’;
9     private double price = 0.0; // Precio predeterminado para todos los
artículos
10    private int quantityInStock = 0; // Cantidad predeterminada para
todos los artículos
11
12    public ConstructorShirt2() {
13      colorCode = ‘R’;
14    }
15
16    public ConstructorShirt2 (char startingCode) {
17
18      switch (startingCode) {
19      case ‘R’:
20      case ‘G’:
21      case ‘B’:
22        colorCode = startingCode;
23        break;
24      default:
25        System.out.println(“Código de color no válido. Utilice R, G o
B”);
26      }
27    }

28    public ConstructorShirt2 (char startingCode, int startingQuantity)
{
29
```

```
30      switch (startingCode) {
31      case ‘R’:
32        colorCode = startingCode;
33        break;
34      case ‘G’:
35        colorCode = startingCode;
36        break;
37      case ‘B’:
38        colorCode = startingCode;
39        break;
40      default:
41        System.out.println(“Código de color no válido. Utilice R, G o
B”);
42      }
43
44      if (startingQuantity > 0 && startingQuantity < 2000) {
45        quantityInStock = startingQuantity;
46      }
47
48      else {
49        System.out.println(“Cantidad no válida. Debe ser > 0 o < 2000”);
50      }
51    }
52
53    public char getColorCode() {
54      return colorCode;
55      }


56    public int getQuantityInStock() {
57        return quantityInStock;
58    }
59
60  } // fin de la clase
61
```

La clase `ConstructorShirt2` contiene tres constructores declarados en las líneas 12, 16 y 28. El constructor `ConstructorShirt2()` no acepta argumentos e inicializa la variable de atributo `colorCode` con el valor “R” (líneas de la 12 a la 14). El constructor `ConstructorShirt2(char startingCode)` acepta un código de color inicial (líneas de la 16 a la 27). Por último, el constructor `ConstructorShirt2(char startingCode, int startingQuantity)` acepta un código de color inicial y una cantidad en stock (líneas de la 28 a la 51).

El siguiente ejemplo de código crea tres objetos utilizando los tres constructores de la clase `ConstructorShirt2`.

Código 9-17 Archivo `ConstructorShirt2Test` situado en el directorio `object_oriented`

```
1
2   public class ConstructorShirt2Test {
3
4     public static void main (String args[]) {
5
6       ConstructorShirt2 constShirtFirst = new ConstructorShirt2();
7       ConstructorShirt2 constShirtSecond = new ConstructorShirt2('G');
8       ConstructorShirt2 constShirtThird = new ConstructorShirt2('B',
1000);
9
10      char colorCode;
11      int quantity;
12
13      colorCode = constShirtFirst.getColorCode();
14      System.out.println("Código de color del objeto 1: " + colorCode);
15
16      colorCode = constShirtSecond.getColorCode();
17      System.out.println("Código de color del objeto 2: " + colorCode);
18
19      colorCode = constShirtThird.getColorCode();
20      quantity = constShirtThird.getQuantityInStock();
21      System.out.println("Código de color del objeto 3: " + colorCode);
22      System.out.println("Cantidad en stock del objeto 3: " + quantity);
23    }
24  }
25
```

La clase `ConstructorShirt2Test` crea tres objetos `ConstructorShirt2` y llama a cada uno de los tres constructores (líneas de la 6 a la 8).

Creación y uso de matrices (arrays)

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Escribir código con arrays unidimensionales.
- Establecer los valores del array utilizando el atributo `length` y un bucle.
- Pasar argumentos al método `main` para su uso en un programa.
- Crear arrays bidimensionales.

En este módulo se describe la forma de usar arrays para manejar múltiples valores en la misma variable.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Codificar arrays unidimensionales.			
Establecer los valores del array utilizando el atributo `length` y un bucle.			
Pasar argumentos al método `main` para su uso en un programa.			
Crear arrays bidimensionales.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Notas

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender los conceptos sobre los arrays:

- Un array es una serie de elementos dispuestos en orden, como una lista. ¿Para qué utiliza la gente los arrays en su vida cotidiana?
- Si un array unidimensional es una lista de elementos, ¿qué es un array bidimensional?
- ¿Cómo se accede a los elementos de un array?

Creación de arrays unidimensionales

Imagine un programa en el que almacene las edades de 10 personas. Podría crear variables diferentes para guardar cada uno de los 10 valores. Por ejemplo:

```
int ageOne = 27;
int ageTwo = 12;
int ageThree = 82;
int ageFour = 70;
int ageFive = 54;
int ageSix = 6;
int ageSeven = 1;
int ageEight = 30;
int ageNine = 34;
int ageTen = 42;
```

¿Pero, qué pasaría si tuviese que almacenar 1000 o 10000 edades? A medida que crece el número de valores, el programa se vuelve más y más difícil de manejar.

El lenguaje Java permite agrupar múltiples valores del mismo tipo (listas) utilizando arrays unidimensionales. Los arrays son útiles cuando se dispone de varios datos relacionados (como las edades de diferentes personas) pero no se quieren crear variables distintas para guardar cada dato.

En la figura siguiente se ilustran varios arrays unidimensionales.

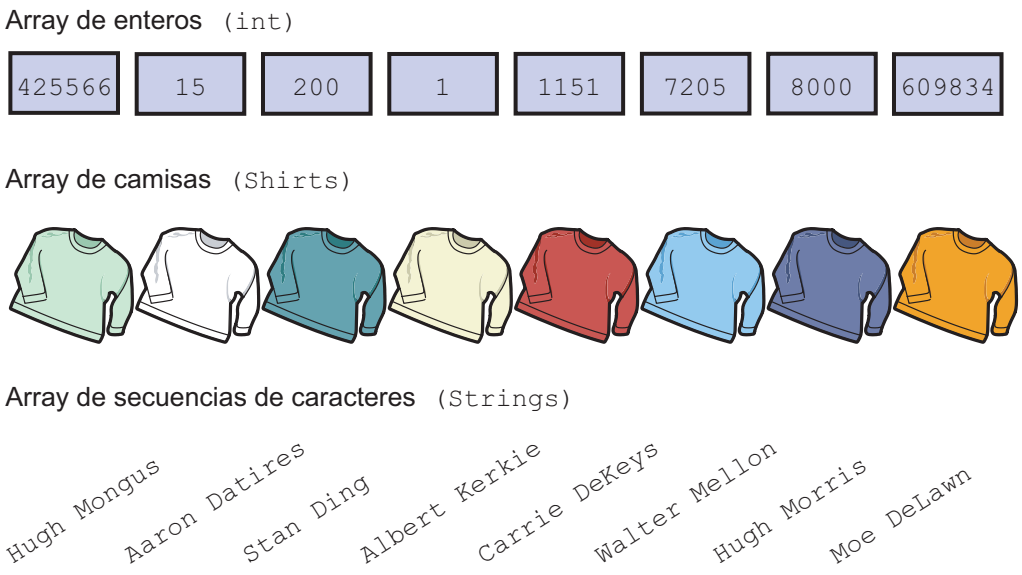


Figura 10-1 Ejemplos de arrays unidimensionales

Es posible crear un array de tipos primitivos, como `int`, o un array de referencias a tipos de objetos, como `Shirt`.

Cada parte del array es un elemento. Si declara un array de 100 tipos `int`, habrá 100 elementos.

Puede acceder a cada elemento concreto utilizando su ubicación o índice dentro del array.

Declaración de un array unidimensional

Los arrays se controlan mediante un objeto `Array` implícito (que no está disponible en el API de Java, pero está disponible en su código). Como en el caso de cualquier otro objeto, debe declarar una referencia al array, instanciar un objeto `Array` e inicializar ese objeto para poder utilizarlo. La sintaxis utilizada para declarar matrices unidimensionales es como sigue:

```
tipo [] identificador_array;
```

Donde:

- *`tipo`* representa el tipo de dato primitivo o el tipo de objeto correspondiente a los valores almacenados en el array.
- Los corchetes, `[]`, indican al compilador que se está declarando un array.
- *`identificador_array`* es el nombre asignado al array.

Utilice el código siguiente para declarar un array de valores `char` llamado `status` y un array de valores `int` llamado `ages`.

```
char [] status;
int [] ages;
```

Utilice el código siguiente para declarar un array de referencias a `Shirt` llamado `shirts` y un array de referencias a `String` llamado `names`.

```
Shirt [] shirts;
String [] names;
```

Nota – Los corchetes de apertura y cierre también pueden aparecer después del identificador. Por ejemplo: *`tipo identificador_array`* `[]`.

Cuando declara un array, el compilador y la máquina virtual de Java (JVM) no saben qué longitud tendrá porque ha declarado variables de referencia que en ese momento no señalan a ningún objeto.

Instanciación de un array unidimensional

Para poder inicializar un array, es preciso instanciar un objeto `Array` lo suficientemente grande como para dar cabida a todos los valores del array. Un array se instancia definiendo el número de elementos que contiene. La sintaxis para instanciar objetos `Array` es como sigue:

```
identificador_array = new tipo [longitud];
```

Donde:

- *`identificador_array`* es el nombre asignado al array.
- *`tipo`* representa el tipo de dato primitivo o el tipo de objeto correspondiente a los valores almacenados en el array.
- *`longitud`* representa el tamaño (número de elementos) del array.

Utilice el código siguiente para instanciar un array de valores `char` llamado `status` y un array de valores `int` llamado `ages`.

```
status = new char [20];
ages = new int [5];
```

Utilice el código siguiente para instanciar un objeto `Array` llamado `names` que contiene referencias a `String` y un array de referencias al objeto `Shirt` llamado `shirts`.

```
names = new String [7];
shirts = new Shirt [3];
```

Cuando se instancia un objeto `Array` cada elemento primitivo se inicializa con el valor cero correspondiente al tipo especificado. En el caso del array de tipo `char` llamado `status`, cada valor se inicializa con `\u0000` (el carácter vacío del juego de caracteres Unicode). En el caso del array `int` llamado `ages`, el valor inicial es el valor entero `0`. En cuanto a los arrays `names` y `shirt`, las referencias al objeto se inicializan con `null`.

Nota – Los arrays de un determinado tipo de objeto pueden contener referencias de objeto.

Inicialización de un array unidimensional

Es posible escribir el contenido un array después de crear el array. La sintaxis para definir los valores de un array es:

```
identificador_array[índice] = valor;
```

Donde:

- *identificador_array* es el nombre asignado al array.
- *índice* representa el lugar del array donde se situará el valor.

Nota – El primer elemento de un array se sitúa en el índice 0 y el último elemento se sitúa en el lugar correspondiente al valor de *longitud* menos 1. Por tanto, el último elemento de un array de seis elementos es el índice 5.

- *valor* es el valor que se asigna al *índice* en el array.

Utilice el código siguiente para establecer los valores del array `ages`:

```
ages[0] = 19;
ages[1] = 42;
ages[2] = 92;
ages[3] = 33;
ages[4] = 46;
```

Utilice el código siguiente para establecer los valores del array `shirts`:

```
shirts[0] = new Shirt();
shirts[1] = new Shirt('G');
shirts[2] = new Shirt('G', 1000);
```

La palabra clave `new` se utiliza para crear los objetos `Shirt` y situar las referencias a dichos objetos en cada posición del array.

Declaración, instanciación e inicialización de arrays unidimensionales

Si conoce los valores que va a incluir en el array en el momento de declararlo, puede declarar, instanciar y definir los valores del objeto `Array` en la misma línea de código. La sintaxis de esta declaración, instanciación e inicialización combinada de valores es como sigue:

```
tipo [] identificador_array =
{lista_de_valores_o_expresiones_separados_por_coma};
```

Donde:

- *`tipo`* representa el tipo de dato primitivo o el tipo de objeto correspondiente a los valores almacenados en el array.
- Los corchetes, `[]`, indican al compilador que se está declarando un array.
- *`identificador_array`* es el nombre asignado al array.
- `{`*`lista_de_valores_o_expresiones_separados_por_coma`*`}` representa una lista de valores que se van a almacenar en el array o una lista de expresiones cuyos resultados se almacenarán en el array.

La sentencia siguiente combina los anteriores ejemplos de declaración, instanciación e inicialización del array `ages`:

```
int [] ages = {19, 42, 92, 33, 46};
```

También se pueden establecer valores para cada objeto (con una referencia que se almacena en un array) utilizando el constructor de cada objeto. La sentencia siguiente combina el anterior ejemplo de declaración, instanciación e inicialización del array `shirt`:

```
Shirt [] shirts = { new Shirt(), new Shirt('G'), new Shirt('G',1000) };
```

No es posible declarar e inicializar un array en líneas diferentes utilizando la lista de valores separados por coma. El código siguiente devolvería un error:

```
int [] ages;
ages = {19, 42, 92, 33, 46};
```

Acceso a un valor dentro de un array

El acceso a cada elemento de un array se realiza a través de su índice. Para acceder a un valor del array, indique el nombre del array y el número de índice del elemento (entre corchetes `[]`) a la derecha de un operador de asignación.

En el siguiente ejemplo de código se muestra cómo establecer el valor de un determinado índice del array:

```
status[0] = ’3’;
names[1] = "Fred Smith";
ages[1] = 19;
prices[2] = 9.99F;
```

En el ejemplo siguiente se muestra cómo recuperar valores de un determinado índice del array:

```
char s = status[0];
String name = names [1];
int age = ages[1];
double price = prices[2];
```

Almacenamiento de arrays unidimensionales en la memoria

Los arrays son objetos a los que hace referencia una variable de referencia a objetos.

En la figura siguiente se ilustra la forma en que se almacena en la memoria un array de tipos de datos primitivos frente a la forma en que se almacena un tipo primitivo.

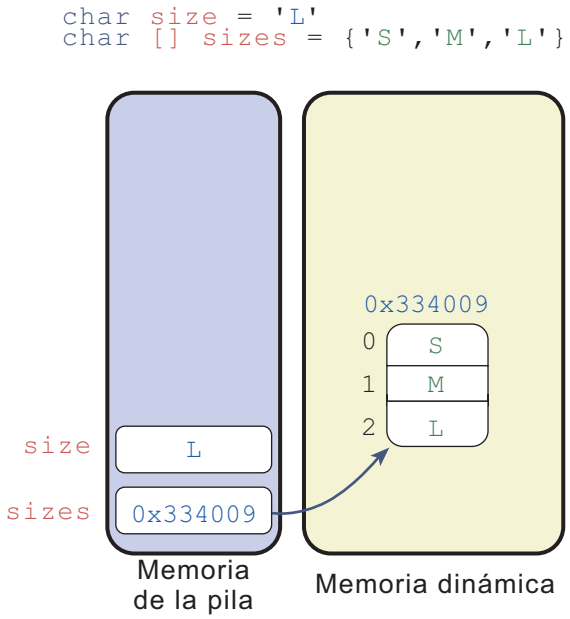


Figura 10-2 Almacenamiento de variables primitivas y arrays de datos primitivos en la memoria

Nota – La línea de código de la figura aparecería en algún método como, por ejemplo, `main`.

El valor de la variable size (un tipo primitivo `char`) es `L`. El valor de `sizes[]` es `0x334009` y señala a un objeto del tipo "array de valores `char`" compuesto de tres valores. El valor de `sizes[0]` es `char S`, el valor de `sizes[1]` es `char M` y el valor de `sizes[2]` es `char L`.

Almacenamiento de variables de referencia y arrays de referencias en la memoria

En la figura siguiente se muestra cómo se guarda un array de referencias en la memoria.

```
1 Shirt myShirt = new Shirt();
2 Shirt [] shirts = {new Shirt(),
                     new Shirt(),
                     new Shirt()};
```

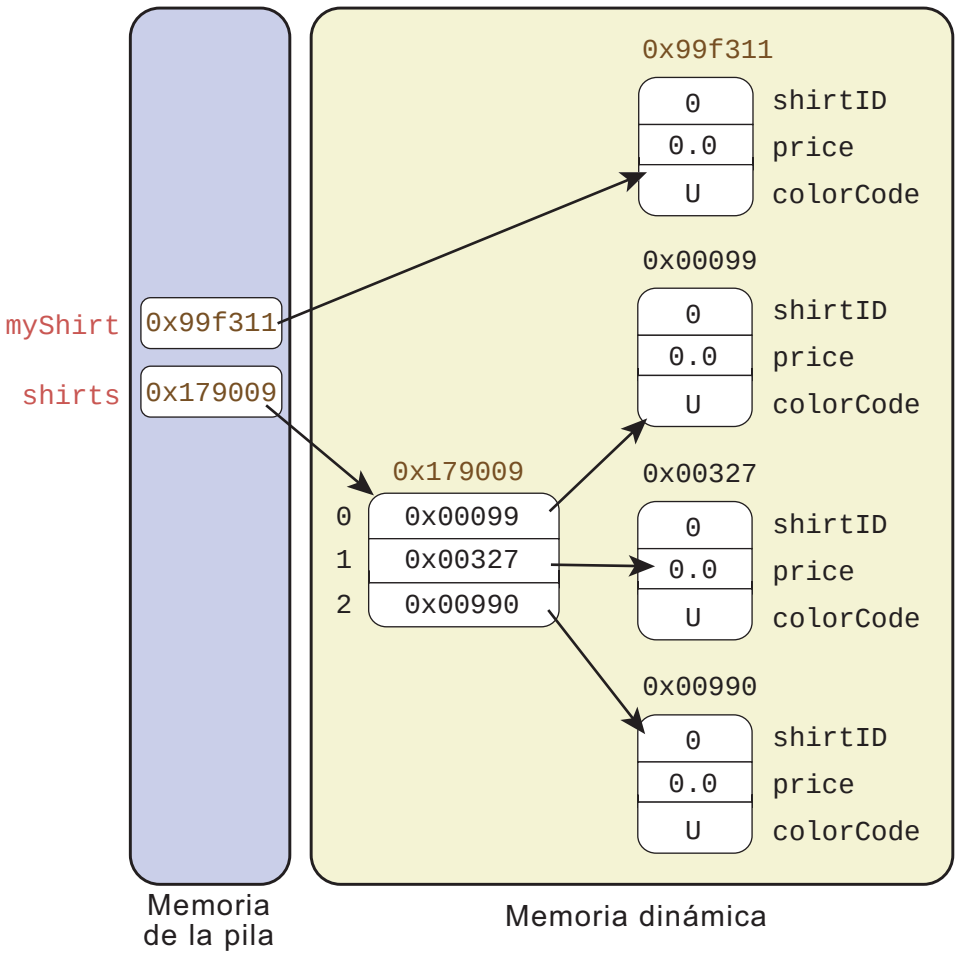


Figura 10-3 Almacenamiento de variables de referencia y arrays de referencias en la memoria

Nota – La línea de código de la figura aparecería en un método como, por ejemplo, `main`.

El valor de la referencia al objeto `myShirt` es `x99f311`, que es una dirección que señala a un objeto de tipo `Shirt` con los valores `0`, `0.0` y `U`. El valor de la referencia al objeto `shirts[]` es `x179009`, una dirección que señala a un objeto de tipo "array de referencias a objetos `Shirt`" formado por tres referencias a objetos:

- El valor del índice `shirts[0]` es `0x00099`, es decir, una referencia que señala a un objeto de tipo `Shirt` con los valores `0`, `0` y `U`.
- El valor del índice `shirts[1]` es `0x00327`, es decir, una referencia que señala a un objeto de tipo `Shirt` con los valores `0`, `0` y `U`.
- El valor del índice `shirts[2]` es `0x00990`, es decir, una referencia que señala a un objeto de tipo `Shirt` con los valores `0`, `0` y `U`.

Establecimiento de valores de arrays utilizando el atributo length y un bucle

Establecer los valores de los arrays es una tarea tediosa porque es preciso escribir una línea de código para definir cada elemento del array. Sin embargo, es posible usar un bucle para ir recorriendo cada elemento del array y establecer los distintos valores.

Atributo length

Todos los objetos Array tienen una variable de atributo llamada length que contiene la longitud del array.

La longitud se conoce también como los límites del array. Los límites de un array como ages[10] van de ages[0] a ages[9] porque todos los arrays empiezan con el elemento cero (0).

El número de elementos de un array se almacenan como parte del objeto array. El software de la máquina virtual de Java utiliza la longitud para asegurarse de que cada acceso al array se corresponda con un elemento actual del array.

Si se produce un intento de acceder a un elemento que no existe, por ejemplo, especificando en el código un valor ages[21] para un array ages que tiene longitud [10], se producirá un error.

Establecimiento de los valores del array utilizando un bucle

Es posible usar el atributo length de la clase Array en la parte condicional de un bucle para iterar en un array. En el ejemplo siguiente se ilustra la forma de iterar a lo largo de un array utilizando un bucle.

```
int [] myArray;
myArray = new int[100];

for (int count = 0; count < myArray.length; count++) {
   myArray[count] = count;
}
```

En este ejemplo, cada índice de `myArray` se llena con un valor equivalente al número de índice (`count`). Por ejemplo, `myArray[10]` es `10`, `myArray[99]` es `99` y así sucesivamente.

Bucle for mejorado

Existe otra forma de la sentencia "for" diseñada para iterar en los arrays. A veces esta forma se conoce como sentencia `for` mejorada y puede utilizarse para hacer los bucles más compactos y fáciles de leer. Observe el array siguiente, que está formado por una serie de números:

```
int [] numbers ={1,3,5,7,9,11,13,15,17,19}
```

El programa siguiente utiliza la sentencia `for` mejorada para ir recorriendo el array incluido en el bucle y mostrar la suma de los números que contiene.

```
class ExampleFor {
     public static void main(String[] args){
          int[] numbers = {1,3,5,7,9,11,13,15,17,19};
           int sum=0;
          for (int item : numbers) {
            sum = sum + item;
           }
           System.out.println("La suma es: " + sum);

       }
}
```

Se recomienda usar esta forma de sentencia for siempre que sea posible.

Uso del array `args` en el método `main`

Ya hemos visto un ejemplo de uso de un array como argumento del método `main`. El array `args` acepta un número indeterminado de objetos `String`:

```
public static void main (String args[]);
```

Cuando se pasan secuencias o cadenas de caracteres a un programa desde la línea de comandos, esas secuencias se escriben en el array `args`. Para utilizarlas, es preciso extraerlas del array `args` y, opcionalmente, convertirlas en el tipo de dato apropiado.

El siguiente ejemplo de código ilustra el paso de secuencias de caracteres al método `main` de un programa.

La clase `ArgsTest` extrae dos argumentos `String` pasados desde la línea de comandos y los muestra en la pantalla.

Código 10-1 Archivo `ArgsTest.java` en el directorio `arrays`

```
public class ArgsTest {

  public static void main (String args[]) {

    System.out.println(“args[0] es “ + args[0]);
    System.out.println(“args[1] es “ + args[1]);
  }
}
```

Nota – Sus programas deberían mostrar una lista de argumentos válidos a cualquier usuario que los utilice. La mayoría de los programas utilizados desde la línea de comandos presenta una lista de argumentos aceptados si: se introduce el nombre del programa sin ningún argumento, se introduce el nombre del programa con un argumento no válido o se introduce explícitamente un argumento que indique al programa que debe mostrar una lista de argumentos aceptados (tal como la `-h` de ayuda).

Conversión de argumentos String en otros tipos

El método main trata todo lo que se escribe como una secuencia literal. Si quiere utilizar la representación literal de un número en una expresión, debe convertir la secuencia en el valor numérico equivalente. Cada tipo de dato tiene una clase asociada que contiene métodos utilitarios static para convertir las secuencias literales en ese tipo de dato (clase Integer para int, clase Byte para byte, clase Long para long y así sucesivamente). Por ejemplo, para convertir el primer argumento pasado al método main en un tipo int:

```
int ID = Integer.parseInt(args[0]);
```

Función varargs

La función varargs (argumentos variables) permite crear un método que pueda aceptar un número variable de argumentos. Imagine el ejemplo siguiente:

```
class VarMessage{
    public static void showMessage(String... names) {
       for (String list: names)
          System.out.println(list);
    }
    public static void main (String args[]){
       showMessage (args)

   }
}
```

En primer lugar, el argumento names está definido como un tipo String.... La declaración del parámetro contiene puntos suspensivos. Esto indica al compilador que el código de llamada puede pasar un número variable de parámetros String. De esta forma, es posible compilar y ejecutar el programa anterior, por ejemplo:

java VarMessage Joe Harry o
java VarMessage Joe Harry Sam Henry Peter

En general, un método puede tener como máximo un parámetro que sea un argumento variable (vararg), debe ser el último parámetro recibido por el método y se expresa mediante el tipo de objeto, los puntos suspensivos (...) y el nombre de la variable.

Descripción de los arrays bidimensionales

Es posible almacenar también matrices de datos utilizando arrays multidimensionales (arrays de arrays). Un array bidimensional (un array de arrays) es parecido a una hoja de cálculo con múltiples columnas (cada columna representa un array o una lista de elementos) y múltiples filas.

En la figura siguiente se ilustra un array bidimensional.

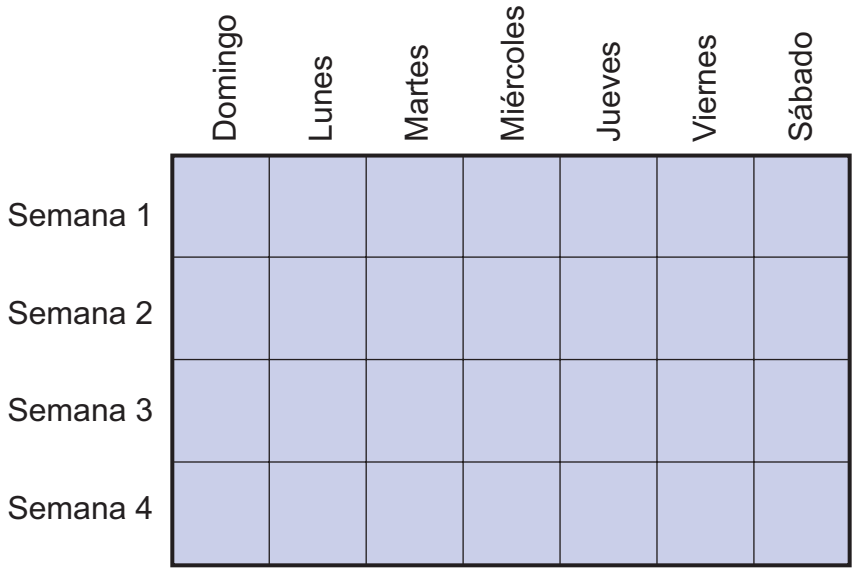


Figura 10-4 Array bidimensional

Declaración de un array bidimensional

Los arrays bidimensionales necesitan un segundo par de corchetes en la declaración. Por lo demás, el proceso de crear y usar arrays bidimensionales es similar al de los arrays unidimensionales. La sintaxis para declarar un array bidimensional es como sigue:

```
tipo [][] identificador_array;
```

Donde:

- *tipo* representa el tipo de dato primitivo o el tipo de objeto correspondiente a los valores almacenados en el array.
- Los corchetes, `[] []`, indican al compilador que se está declarando un array bidimensional.
- *identificador_array* es el nombre asignado al array durante la declaración.

En el ejemplo siguiente se declara un array de arrays para conocer las ventas trimestrales durante cinco años.

```
int [][] yearlySales;
```

Instanciación de un array bidimensional

La sintaxis para instanciar un array bidimensional es como sigue:

identificador_*array* = new *tipo* [*número_de_arrays*] *[longitud]*;

Donde:

- *identificador_array* es el nombre asignado al array durante la declaración.
- *número_de_arrays* es el número de arrays contenidos en el array.
- `longitud` es la longitud de cada array dentro del array.

En el ejemplo siguiente se instancia un array de arrays para obtener las ventas trimestrales durante cinco años.

```
// Instancia un array bidimensional: 5 arrays de 4 elementos cada uno
yearlySales = new int[5][4];
```

Este array bidimensional crea una matriz como ésta:

	Trimestre 1	**Trimestre 2**	**Trimestre 3**	**Trimestre 4**
Año 1				
Año 2				
Año 3				
Año 4				
Año 5				

El array `yearlySales` contiene cinco arrays de tipo `int` (cinco subarrays). Cada subarray tiene un tamaño de cuatro elementos y hace el seguimiento de las ventas durante los cuatro trimestres de un año.

Inicialización de un array bidimensional

Al establecer los valores de un array bidimensional, indique el valor de índice utilizando un número para representar la fila seguido de otro número que represente la columna. He aquí algunos ejemplos de cómo establecer los valores del array `yearlySales` (ventas anuales):

```
yearlySales[0][0] = 1000;
yearlySales[0][1] = 1500;
yearlySales[0][2] = 1800;
yearlySales[1][0] = 1000;
yearlySales[2][0] = 1400;
yearlySales[3][3] = 2000;
```

El ejemplo de código anterior da lugar a la siguiente matriz o array bidimensional:

	Trimestre 1	**Trimestre 2**	**Trimestre 3**	**Trimestre 4**
Año 1	1000	1500	1800	
Año 2	1000			
Año 3	1400			
Año 4				2000
Año 5				

Módulo 11

Implementación de la herencia

Objetivos

El estudio de este módulo le proporcionará los conocimientos necesarios para:

- Definir y verificar el uso de la herencia.
- Explicar el concepto de abstracción.
- Identificar de forma explícita las bibliotecas de clases utilizadas en su código.

En este módulo se describen los conceptos de herencia y abstracción, y la forma de implementar ambos conceptos en el lenguaje Java.

Comprobación de los progresos

Introduzca un número del 1 al 5 en la columna “Principio del módulo” a fin de evaluar su capacidad para cumplir cada uno de los objetivos propuestos. Al finalizar el módulo, vuelva a evaluar sus capacidades y determine la mejora de conocimientos conseguida por cada objetivo.

Objetivos del módulo	**Evaluación (1 = No puedo cumplir este objetivo, 5 = Puedo cumplir este objetivo)**		**Mejora de conocimientos (Final – Principio)**
	Principio del módulo	**Final del módulo**	
Definir y verificar el uso de la herencia.			
Explicar el concepto de abstracción.			
Identificar de forma explícita las bibliotecas de clases utilizadas en su código.			

El resultado de esta evaluación ayudará a los Servicios de Formación Sun (SES) a determinar la efectividad de su formación. Por favor, indique una escasa mejora de conocimientos (un 0 o un 1 en la columna de la derecha) si quiere que el profesor considere la necesidad de presentar más material de apoyo durante las clases. Asimismo, esta información se enviará al grupo de elaboración de cursos de SES para revisar el temario de este curso.

Notas

Aspectos relevantes

Discusión – Las preguntas siguientes son relevantes para comprender los conceptos sobre la herencia:

- La herencia se refiere a la transferencia de propiedades de un organismo a otro. ¿Qué características físicas ha heredado?
- ¿De quién ha heredado esas características?
- ¿De qué jerarquía de clases procede?
- ¿Ha heredado características de múltiples clases?
- ¿A qué nos referimos cuando decimos que algo es “abstracto?”
- ¿Qué cree que es una clase abstracta?

Herencia

Caso de estudio – Utilizaremos el siguiente caso para ilustrar conceptos intermedios del análisis y el diseño OO.

El máximo responsable de DirectClothing, Inc. ha decidido que la compañía amplíe su línea de productos para incluir gorros, calcetines y pantalones. La decisión le lleva a darse cuenta de que necesitará cambiar el sistema de introducción de pedidos previamente desarrollado. Como consecuencia, DirectClothing le ha contratado para poner al día el citado sistema de forma que incorpore estos tipos de productos.

A fin de poder desarrollar las clases necesarias para hacer la actualización, deberá realizar un breve análisis del nuevo sistema y determinar que los gorros, los calcetines y los pantalones son los únicos objetos nuevos que van a introducirse (en el dominio del problema). Además recibe la siguiente información:

- Sólo se venderán gorros y calcetines de talla única.
- Los modelos de pantalones serán distintos para hombres y mujeres.
- Los pantalones se venderán en colores azul y tostado, mientras que los gorros, los calcetines y las camisas se venderán en rojo, azul y verde.

A continuación, deberá diseñar el nuevo sistema modelando las clases que se utilizarán para crear estos objetos. En las dos figuras siguientes se ilustran las nuevas clases y la clase `Shirt` previamente definida.

`Hat`	`Socks`

`ID`
`price`
`description`
`colorCode R=Red, B=Blue, G=Green`
`quantityInStock`
	`ID`
	`price`
	`description`
	`colorCode R=Red, B=Blue, G=Green`
	`quantityInStock`

`calculateID()` `displayInformation()`	`calculateID()` `displayInformation()`

Figura 11-1 Clases `Hat` (Gorro) y `Socks` (Calcetines)

Pants	Shirt
ID price size gender M=Male, F=Female description colorCode B=Blue, T=Tan quantityInStock	ID price description colorCode R=Red, B=Blue, G=Green quantityInStock
calculateID() displayInformation()	calculateID() displayInformation()

Figura 11-2 Clases `Pants` (Pantalones) y `Shirt` (Camisa)

Estas nuevas clases tienen muchas de las características de la clase `Shirt`. Por ejemplo, todas tienen ID, precios y descripciones. Todas tienen operaciones para calcular un ID y mostrar los datos. Sin embargo, el método de salida a la pantalla de cada subclase presenta diferente contenido.

Es posible eliminar la necesidad de duplicar esta información en cada clase implementando un concepto de la programación OO denominado herencia.

Superclases y subclases

La herencia permite a los programadores poner miembros comunes (variables y métodos) en una clase y hacer que otras clases *hereden* esos miembros comunes.

La clase que contiene miembros comunes a otras clases se denomina superclase o clase de nivel superior. Las clases que heredan miembros (amplían) de la superclase se denominan subclases o clases subordinadas.

La herencia da como resultado una jerarquía de clases de la tecnología Java similar a la clasificación taxonómica que se utiliza en biología, por ejemplo, “La ballena azul es una subclase de la ballena”.

En la figura siguiente se ilustra una jerarquía de ballenas.

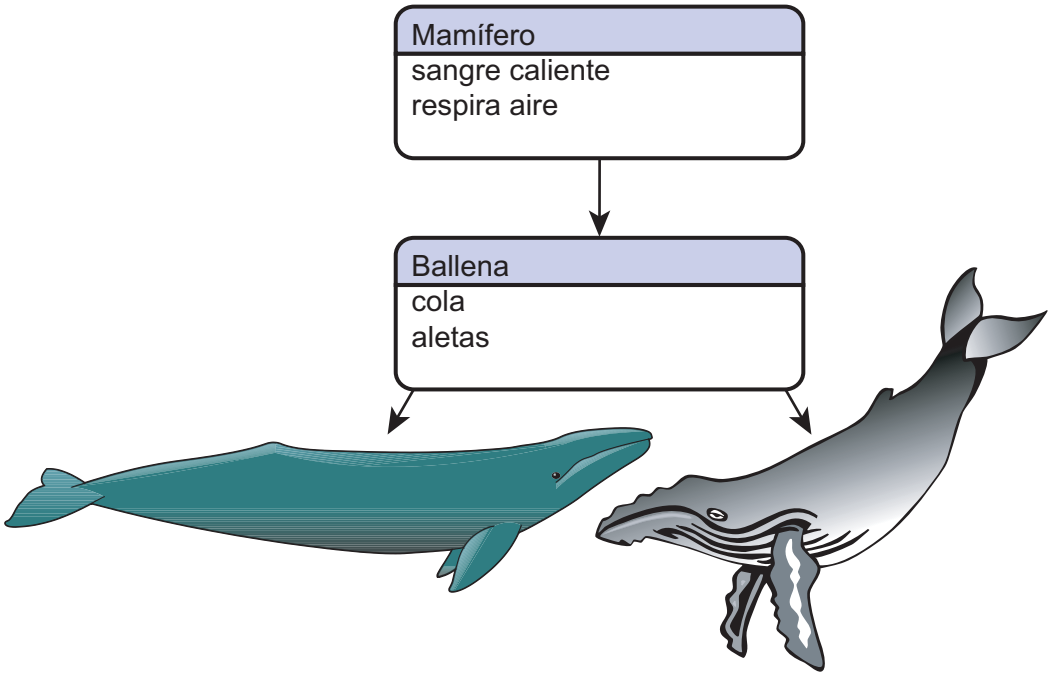


Figura 11-3 Herencia y taxonomía

"Sangre caliente" es un atributo de la superclase mamíferos. La frase "respira aire" representa una actividad que también forma parte de la superclase mamíferos. La cola y las aletas son atributos específicos de las ballenas, que es una subclase de los mamíferos.

Comprobación de las relaciones entre superclases y subclases

En el lenguaje Java, cada clase puede heredar miembros de una sola clase, por lo que es muy importante sopesar la mejor forma de usar la herencia y utilizarla sólo cuando sea completamente válida o inevitable.

¿Lo sabía? – Algunos lenguajes de programación, como C++, permiten a una clase heredar de diferentes superclases. Este concepto se denomina herencia múltiple. El lenguaje Java no admite la herencia múltiple porque su implementación puede resultar confusa para los programadores. En su lugar, el lenguaje Java utiliza el concepto de interfaz.

La forma de comprobar si un vínculo de herencia propuesto es válido es utilizar la expresión "*es un/una*". La frase "una Camisa *es una* Prenda de vestir" expresa un vínculo de herencia válido. La frase "un Gorro *es un* Calcetín" expresa un vínculo de herencia incorrecto.

En el caso estudiado, camisas, pantalones, gorros y calcetines son tipos de prendas de vestir. Por tanto, `Clothing` (Prendas) es un buen nombre para la superclase de estas subclases (tipos) de prendas de vestir.

En la figura siguiente se ilustra esta nueva jerarquía de clases.

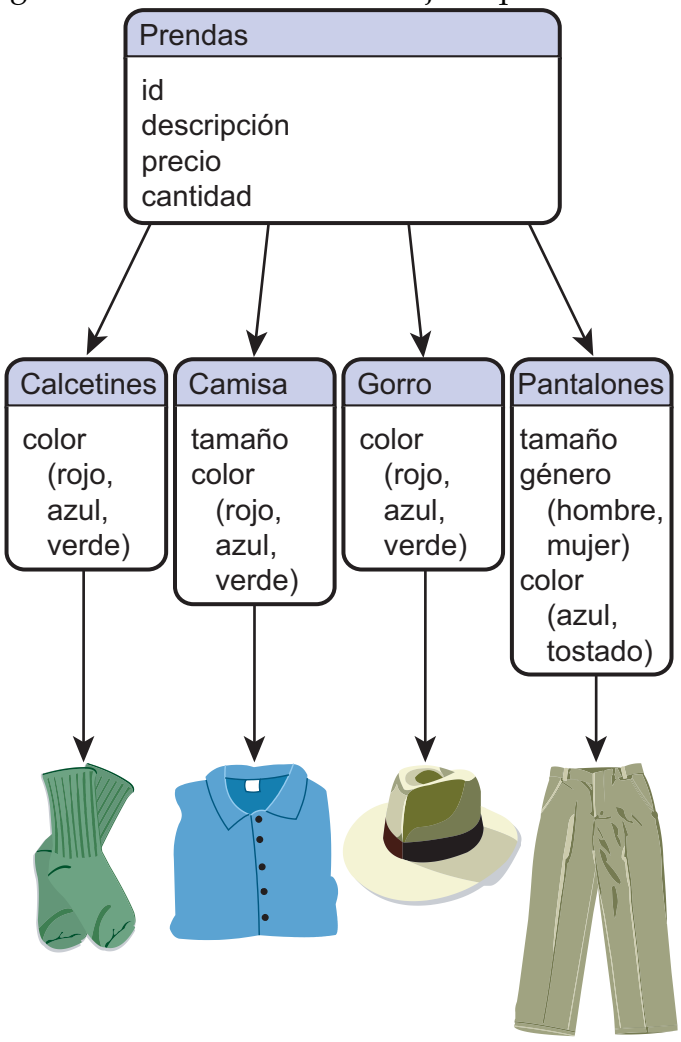


Figura 11-4 Superclase Clothing (Prendas) y subclases

Modelado de las superclases y subclases

Después de identificar la relación entre la superclase y las subclases, puede remodelar todas sus clases. Las subclases sólo contendrán características propias de ellas mismas. En las tres figuras siguientes se muestran las nuevas clases, la clase `Shirt` definida anteriormente y la subclase `Clothing`.

`Hat:Clothing`	`Socks:Clothing`
`colorCode R=Red, B=Blue, G=Green`	`colorCode R=Red, B=Blue, G=Green`
`displayInformation()`	`displaySockInformation()`

Figura 11-5 Clases `Hat` (Gorro) y `Socks` (Calcetines)

`Pants:Clothing`	`Shirt:Clothing`

`size`
`gender M=Male, F=Female`
`colorCode B=Blue, T=Tan`
	`size`
	`colorCode R=Red, B=Blue, G=Green`

`displayClothingInformation()`	`displayInformation()`

Figura 11-6 Clases `Pants` y `Shirt`

`Clothing`

`id`
`price`
`description`
`quantityInStock`

`calculateID()`

Figura 11-7 Clase `Clothing`

Las características comunes a todas las subclases están en la clase `clothing`.

Discusión – ¿Qué otras clases del sistema de prendas de DirectClothing, Inc. utilizarían la herencia? Compruebe cada vínculo de herencia para asegurarse de que sea válido.

Discusión – Dado que la variable `colorCode` aparece en todas las subclases de `Clothing`, ¿cómo rediseñaría la clase `Clothing`? Tenga en cuenta que no todas las subclases admiten las mismas opciones de color.

Declaración de una superclase

Los siguientes ejemplos de código contienen una superclase `Clothing`.

Código 11-1 Archivo `Clothing.java` situado en el directorio `inheritance`

```
1
2   public class Clothing {
3
4     private int ID = 0; // ID predeterminado para todas las prendas
5     private String description = “-description required-”; //
predeterminada
6
7     private double price = 0.0; // Precio predeterminado para todas las
prendas
8     private int quantityInStock = 0; // Cantidad predeterminada para
todas las prendas
9
10    private static int instanceCounter=0; //Miembro estático utilizado
en el constructor para generar un ID exclusivo
11
12    public Clothing() {
13      ID = instanceCounter++;
```

```
14    }
15
16    public int getID() {
17      return ID;
18    }
19
20    public void setDescription(String d) {
21      description = d;
22    }
23
24    public String getDescription() {
25      return description;
26    }
27
28    public void setPrice(double p) {
29      price = p;
30    }
31
32    public double getPrice() {
33      return price;
34    }
35
36    public void setQuantityInStock(int q) {
37      quantityInStock = q;
38    }
39
40    public int getQuantityInStock() {
41      return quantityInStock;
42    }
43
44  } // fin de la clase
45
```

La clase `Clothing` contiene los métodos utilizados por todas las subclases, por ejemplo las clases `Shirt` o `Pants`. Cada subclase heredará estos métodos.

Declaración de una subclase

Utilice la palabra clave `extends` para indicar que una clase hereda miembros de otra clase. Para indicar que una clase es una subclase de otra, utilice la sintaxis siguiente en la declaración de la clase:

```
[modificador_clase] class identificador_clase extends
identificador_superclase
```

Donde:

- La palabra clave *`modificador_clase`* es opcional (como indican los corchetes) y puede ser `public`, `abstract` o `final`. Si no se incluye *`modificador_clase`* en la declaración de clase, ésta adopta automáticamente un valor predeterminado que algunas veces se denomina "nivel de paquete" y permite acceder únicamente a otras clases del mismo paquete.
- La palabra clave `class` indica al compilador que el bloque de código contiene una declaración de clase.
- *`identificador_clase`* es el nombre asignado a la subclase.
- La palabra clave `extends` indica al compilador que se trata de una subclase de otra clase.
- *`identificador_superclase`* es el nombre de la superclase ampliada por esta subclase.

Ejemplo de declaración de una subclase

Por ejemplo, el código siguiente contiene una clase `Shirt` actualizada que amplía la clase `Clothing`.

Código 11-2 Archivo `Shirt.java` situado en el directorio `inheritance`

```
1   public class Shirt extends Clothing {
2
3     // Los códigos de color son R=Rojo, B=Azul, G=Verde, U=Sin definir
4     public char colorCode = 'U';
5
6     // Este método muestra los valores de un artículo
7     public void displayInformation() {
8
9       System.out.println("ID de camisa: " + getID());
10      System.out.println("Descripción de la camisa :" +
getDescription());
11      System.out.println("Código de color: " + colorCode);
12      System.out.println("Precio de la camisa: " + getPrice());
13      System.out.println("Cantidad en stock: " + getQuantityInStock());
14
15    } // fin del método display
16  } // fin de la clase
17
```

La clase `Shirt` amplía la clase `Clothing` en la línea 1. Observe que la clase `Shirt` sólo contiene las variables y los métodos específicos de un objeto `Shirt`.

Abstracción

Una de las características de una buena solución programada en Java es que la jerarquía de las clases que se cree sea muy generalizada en la parte superior a fin de dejar espacio para posteriores añadidos más concretos.

Discusión – Si DirectClothing decidiese seguir diversificando productos y vender juguetes y perfumes, ¿qué otras clases necesitaría en su sistema de introducción de pedidos?

La abstracción en el análisis y el diseño

La abstracción se refiere a la creación de clases que son muy generales y no contienen métodos con una implementación concreta o código del cuerpo del método. Un buen ejemplo de clase abstracta es la clase `Item` (artículo). Un artículo es un concepto abstracto (normalmente no se entra en una tienda para decir "quiero comprar un artículo") que puede referirse a cualquier elemento vendido en una tienda. Sin embargo, todos los artículos podrían tener características similares en un sistema de introducción de pedidos, por ejemplo, un ID o un método para ver la información sobre el artículo. Las clases que son genéricas y no pueden definirse por completo, como es el caso de la clase `Item`, se conocen como clases abstractas. Las clases que amplían una clase abstracta pueden implementar métodos vacíos de la clase abstracta con código específico de la superclase.

Es posible que le interese dedicar más tiempo al análisis y el diseño para asegurarse de que su solución tenga el suficiente nivel de abstracción como para garantizar la flexibilidad.

Discusión – Imagine que posee varios objetos que representan formas como un rectángulo, un triángulo y un círculo. Cada forma contiene métodos para calcular el área y la circunferencia. Además, el rectángulo posee métodos para obtener y definir los valores de altura y anchura, y el círculo tiene un método para obtener y definir el radio. Utilizando el concepto de abstracción, ¿cómo crearía las relaciones de superclase y subclase de estas figuras?

Clases del API de Java

Como programador con escasos conocimientos de Java, una de sus primeras tareas será familiarizarse con las clases del API de Java.

Las clases del lenguaje Java se agrupan en paquetes según su funcionalidad. Por ejemplo, todas las clases relacionadas con el núcleo del lenguaje Java se encuentran en el paquete `java.lang`, que contiene clases fundamentales para la programación en Java tales como `String`, `Math` e `Integer`.

Existen dos categorías de clases en el API de Java: aquellas a las que es posible hacer referencia de forma implícita en cualquier código que escriba (como la clase `System` utilizada para acceder al método `println`) y aquellas que es preciso importar o calificar en su totalidad.

Clases disponibles de forma implícita

En todos los programas es posible hacer referencia de forma implícita a todas las clases del paquete `java.lang`. Este concepto significa que no necesita mencionar el paquete ni la clase cuando utilice la clase.

Importación y calificación de clases

La mayoría de las clases del API de Java no están disponibles de forma implícita para usarlas en los programas. Es preciso utilizar sentencias de importación o hacer referencia a ellas utilizando sus nombres totalmente calificados (nombre del paquete y la clase) para todos los paquetes que quiera utilizar en sus programas. Estos paquetes incluyen los siguientes:

- El paquete `java.awt` contiene clases que componen las herramientas AWT (abstract windowing toolkit). Se utiliza para construir y manejar la interfaz gráfica de usuario (GUI) de la aplicación.
- El paquete `java.applet` contiene clases que proporcionan comportamiento específico de los applets.
- El paquete `java.net` contiene clases para realizar operaciones relacionadas con la red tales como las conexiones de sockets y URL.
- El paquete `java.io` contiene clases que manejan la entrada/salida (E/S) de archivos, por ejemplo, la lectura o escritura en una unidad de disco duro.
- El paquete `java.util` contiene clases utilitarias para tareas tales como la generación de números aleatorios, la definición de propiedades del sistema y el uso de funciones relacionadas con las fechas y el calendario.

Hay dos formas de hacer que estas clases estén disponibles para un programa:

- Puede importar la clase utilizando la sentencia `import`.
- Puede hacer referencia a ella utilizando un nombre de clase con todos sus calificadores.

Sentencia `import`

Puede utilizar sentencias `import` para facilitar la lectura del código ya que estas sentencias acortan el código que debe escribir para hacer referencia explícita a una clase del API de Java. Hay dos formas de sentencias de importación:

```
import nombre_paquete.nombre_clase;
import nombre_paquete.*;
```

Donde:

- La palabra clave `import` permite hacer referencia a las clases del API de Java utilizando un nombre abreviado.
- *`nombre_paquete`* es el nombre del paquete en el que se encuentra la clase.
- *`nombre_clase`* es el nombre de una clase concreta que se va a importar. Si no se especifica ningún nombre de clase, podrán utilizarse todas las clases del paquete en el programa.
- Como alternativa, puede sustituir *`nombre_clase`* por un asterisco (*) a fin de poder hacer referencia a cada clase del paquete por su nombre de clase concreto.

Nota – La sentencia `import` resulta confusa para muchos programadores porque sugiere que en realidad están eligiendo las clases que se combinarán en su programa durante la compilación. Por el contrario, puede utilizar las sentencias `import` para clarificar el código al lector porque estas sentencias acortan el código que debe escribirse para hacer referencia a un método de una clase.

Por ejemplo, la clase siguiente importa el paquete `java.awt` para posibilitar a la clase el acceso a las clases de interfaz gráfica de AWT.

```
import java.awt.*;
public class MyPushButton1 extends Button {
    // sentencias class
}
```

Especificación del nombre totalmente calificado

En lugar de especificar el paquete `java.awt`, puede hacer referencia a la clase `Button` mediante el nombre `java.awt.Button` a lo largo de todo el programa. Por ejemplo, a continuación figura una declaración de clase que utiliza el nombre calificado al especificar su superclase. La sintaxis del nombre calificado es como sigue:

```
nombre_paquete.nombre_clase
```

Donde:

- *`nombre_paquete`* es el nombre del paquete en el que se encuentra la clase.
- *`nombre_clase`* es el nombre de una clase concreta a la que se está haciendo referencia.

Por ejemplo, en la siguiente declaración de clase se utiliza el nombre calificado para ampliar la clase `Button`.

```
public class MyPushButton2 extends java.awt.Button {
    // sentencias class
}
```

Apéndice A

Siguientes pasos

Este apéndice sirve de orientación para saber en qué dirección continuar con la formación teórica y práctica sobre tecnología Java. En concreto, contiene:

- Instrucciones para obtener e instalar el SDK de Java SE.
- Instrucciones para obtener e instalar la especificación del API de Java SE.
- Información sobre los entornos de desarrollo de la tecnología Java.
- Otras referencias de utilidad sobre la tecnología Java.

Cómo prepararse para programar

La información de esta sección tiene como objetivo ayudarle a seguir programando una vez finalizado este curso. Para utilizar la plataforma Java SE fuera de este curso:

1. Descargue el SDK de Java SE.
2. Descargue la especificación del API de Java SE.
3. Configure su PC para que pueda acceder al SDK de Java SE.
4. Opcional: descargue o consiga un depurador o un entorno de desarrollo integrado.

Descarga de la tecnología Java

Debe descargar el SDK de Java SE para poder utilizar el lenguaje Java.

Nota – Puede seguir las instrucciones proporcionadas a continuación pero recuerde que podrían no ser totalmente exactas si los nombres de productos o las direcciones URL han cambiado desde la fecha de publicación de este manual.

Para descargar el kit de desarrollo de Java SE, versión 6.0 (JDK™ 6.0):

1. Utilice el navegador web para ir a:
   http://java.sun.com/javase/downloads/index.jsp
2. Haga clic en el vínculo `Get the JDK Download` (descargar JDK).

Descarga de la especificación del API de Java SE

Cada revisión de la versión del JDK tiene su propia especificación del API. Para descargar la especificación del JDK:

1. Utilice el navegador web para ir a:
   `http://java.sun.com/javase/downloads/index.jsp`
2. Haga clic en el vínculo Get the JDK Download situado en la sección Download J2SE 6.0 Documentation.

Configuración del equipo para desarrollar y ejecutar programas Java

Es preciso establecer como ruta de acceso a las clases la ubicación donde se haya instalado la versión del JDK. Durante el proceso de descarga encontrará un vínculo a un sitio web con instrucciones completas sobre el modo de configurar la ruta de acceso a las clases, la ruta regular y otros aspectos que pueden causarle problemas.

Descarga de un entorno de desarrollo o un depurador

Toda la programación de este curso se realiza con un editor de archivos de texto. Muchos programadores utilizan entornos de desarrollo integrados (IDE) en lugar de editores de texto. Estos entornos son aplicaciones con una interfaz de usuario gráfica que contienen numerosas funciones de utilidad tales como un código de colores especial de los componentes de las clases de Java y herramientas de depuración.

Hay numerosos entornos de desarrollo gratuitos disponibles en el mercado. Puede descargar el entorno NetBeans™ en: `http://www.netbeans.org`.

En la página de descarga de software de la tecnología Java encontrará vínculos con otros entornos.

Nota – Algunos entornos de desarrollo le ayudan en el trabajo de codificación. Sin embargo, tenga presente que el código que generan algunos de ellos no siempre está bien escrito.

Referencias

Utilice los siguientes recursos para obtener información sobre temas concretos. Además de los recursos indicados, puede obtener información sobre otros cursos de los Servicios de Formación Sun en la dirección URL siguiente:

```
http://sun.es/services/training
```

Entre los cursos sugeridos se incluyen los siguientes:

- SL-275: *Programación Java*
- OO-226: *Análisis y Diseño OO con UML*
- SL-285: *Desarrollo de Aplicaciones Avanzadas en Java*

Nociones básicas sobre tecnología Java

La información básica sobre la tecnología Java puede encontrarse en los documentos siguientes:

- Farrell, Joyce. *Java Programming: Comprehensive*. Course Technology. 1999.

  Es un libro excelente para no programadores. Explica conceptos que no se tratan en profundidad en otros libros más avanzados.
- Naughton, Patrick and Herbert Schildt. *Java 2: The Complete Reference*. Osborne McGraw-Hill, 2004.
- Eckel, Bruce. *Thinking in Java*. Prentice Hall Computer Books, 2006.
- Deitel and Deitel. *Java: How to Program*. Prentice Hall, 2004.

  Libro más avanzado para quienes quieren sumergirse de lleno en el lenguaje de programación Java. Incluye numerosos ejercicios y ejemplos de código.
- Documentación sobre la plataforma Java Standard Edition. [En la web]. Disponible en: `http://java.sun.com/docs/index.html`.

Applets

La información sobre los applets puede encontrarse en el documento siguiente:

Hopson, Stephen. *Developing Professional Java Applets*. 1996.

Este libro está dirigido a diseñadores de páginas web y describe cómo utilizar el código de applets Java del libro y el CD-ROM incluido.

Tutorial en Internet

Encontrará un tutorial sobre la tecnología Java en:

`http://java.sun.com/docs/books/tutorial/`

Se trata de un excelente tutorial de autoformación que cubre una amplia variedad de temas e incluye numerosos ejemplos de código.

Artículos, consejos y documentos técnicos

Puede encontrar artículos técnicos y otra información técnica de utilidad en:

- `http://java.sun.com/javase/reference/techart.jsp`
- `http://java.sun.com/javase/whitepapers.jsp`

Apéndice B

Palabras clave del lenguaje de programación Java

Este apéndice contiene la lista de palabras clave del lenguaje Java.

Palabras clave

Las palabras clave son términos especiales del lenguaje de programación reservados para dar instrucciones al compilador. No deben utilizarse como identificadores para clases, métodos, variables u otros componentes.

La tabla siguiente contiene todas las palabras clave de la tecnología Java.

Tabla B-1 Palabras reservadas para la tecnología Java

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

Nota – Las palabras `true`, `false` y `null` son literales en el lenguaje Java.

Apéndice C

Convenciones de asignación de nombres en el lenguaje Java

En este apéndice se resumen las reglas y directrices de uso de la nomenclatura comentadas a lo largo del curso. Una regla es algo impuesto por el compilador o el intérprete de Java. Una directriz es una sugerencia que ofrecemos para mejorar el código.

La información de referencia completa sobre las convenciones de codificación del lenguaje Java está disponible en:
`http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`

Identificadores de clases, métodos y variables

Las reglas siguientes determinan el contenido y la estructura de los identificadores:

- El primer carácter de un identificador debe ser uno de los siguientes:
  - Una letra mayúscula (A–Z)
  - Una letra minúscula (a–z)
  - El signo de subrayado (_)
  - El signo de dólar ($)
  - Cualquier símbolo monetario del juego de caracteres Unicode
- No se pueden utilizar como identificadores las palabras clave del lenguaje Java.

Las directrices siguientes le ayudarán a asignar identificadores adecuados:

- Empiece cada variable o método con una letra en minúscula y el resto de las palabras con la inicial en mayúscula, por ejemplo, `miVariable` o `getValores`.
- Escriba el nombre de las clases con la primera letra en mayúscula y la primera letra de las palabras sucesivas también en mayúscula, como en `CamisaTest`.
- Utilice letras en mayúscula y palabras separadas mediante signos de subrayado para los identificadores de las constantes, como en `IMPUESTO_VENTA`.
- Elija nombres que sean nemotécnicos e indiquen a cualquier posible lector el propósito de la variable.

He aquí algunos ejemplos de identificadores adecuados:

- `idCliente` (identificador de variable)
- `estaCerrado` (identificador de variable `boolean`)
- `interpretar` (identificador de método)
- `interpretarMusica` (identificador de método)
- `Pedido` (identificador de clase)

Nota – En el lenguaje Java, el atributo de mayúscula y minúscula es un rasgo pertinente. La pertinencia de la grafía en mayúscula/minúscula significa que existe diferencia entre la representación en mayúscula o minúscula de cada carácter alfabético. El lenguaje de programación Java considera que dos identificadores son diferentes simplemente con que dicha grafía sea diferente. Por ejemplo, si crea una variable llamada `pedido`, no podrá referirse a ella más adelante con la palabra `Pedido`.

Apéndice D

Desplazamiento por el sistema operativo Solaris™

Este apéndice contiene una lista de los comandos más habituales para desplazarse por el SO Solaris desde una ventana de terminal.

Guía de referencia rápida de Solaris

La tabla siguiente contiene la lista de comandos de terminal básicos para el sistema operativo Solaris y para Windows de Microsoft.

Tabla D-1 Comandos básicos de la ventana de terminal

Comando	Solaris	Microsoft Windows
Ver el nombre del directorio actual	`pwd`	`cd`
Ver el contenido del directorio actual	`ls` `ls -l` para mayor detalle	`dir /w` `dir` para mayor detalle
Cambiar a un directorio situado por encima del directorio actual	`cd ..`	`cd ..`
Cambiar a un directorio situado varios niveles por encima del directorio actual	`cd ../..` (repetir `../` tantas veces como sea necesario)	`cd ..\..`
Cambiar a un directorio situado debajo del directorio actual	`cd` *`directorio`*	`cd` *`directorio`*
Cambiar a un directorio concreto	`cd` *`ruta_completa`*	`cd` *`ruta_completa`*
Copiar un archivo	`cp` *`archivo destino`*	`copy` *`archivo destino`*
Trasladar un archivo	`mv` *`archivo destino`*	`move` *`archivo destino`*
Copiar un directorio	`cp -r` *`directorio destino`*	`xcopy /s /e` *`directorio destino`*
Trasladar un directorio	`mv` *`directorio destino`*	`move` *`directorio destino`*
Crear un nuevo archivo	`touch` *`nombre`*	`N/A (usar un editor de texto)`
Borrar un archivo	`rm` *`archivo`*	`del` *`archivo`*
Crear un directorio nuevo	`mkdir` *`directorio`*	`mkdir` *`directorio`*
Borrar un directorio	`rmdir` *`directorio`* (sólo directorios vacíos) `rm -r` *`directorio`* (directorios con contenido)	`del` *`directorio`* `deltree` *`directorio`* (el directorio y todo su contenido)

Tabla D-1 Comandos básicos de la ventana de terminal (Continuación)

Comando	Solaris	Microsoft Windows
Cambiar el nombre de un archivo	`mv` *`nombreantiguo nombrenuevo`*	`ren` *`nombreantiguo nombrenuevo`*
Cambiar el nombre de un directorio	`mv` *`nombreantiguo nombrenuevo`*	`ren` *`nombreantiguo nombrenuevo`*
Compilar una aplicación Java	`javac` *`nombrearchivo`*`.java`	`javac` *`nombrearchivo`*`.java`
Ejecutar una aplicación Java	`java` *`nombrearchivo`*	`java` *`nombrearchivo`*

También puede utilizar el Gestor de archivos de Solaris para realizar muchas de estas funciones. Esta aplicación proporciona una interfaz gráfica básica.

En caso de necesitar ayuda con los comandos de Solaris o Windows, pregunte a su profesor.