Programacion Orientada a Objetos

Lara

March 18, 2024

1 Objetos y clases

un programa informático en un lenguaje orientado a objetos, estaremos creando en nuestro equipo un modelo de una cierta parte del mundo. Los componentes a partir de los cuales se construye el modelo son los objetos que aparecen en el dominio del problema concreto que analicemosEsos objetos deben representarse en el modelo informático que estemos desarrollando. Los objetos pueden clasificarse, y una clase sirve para describir, de una manera abstracta, todos los objetos de un tipo concreto. nos referimos a cada objeto particular con el nombre de instancia. hablaremos de instancias cuando queramos insistir en el hecho de que se trata de objetos de una clase determinada El área de la parte inferior de la pantalla en la que se muestra el objeto se denomina banco de objetos (object bench).

el nombre de un método y los tipos de parámetros presentes en su cabecera reciben el nombre de signatura del método. lasoperaciones que podemos emplear para manipular el círculo se denominan métodos. los métodos se llaman o invocan. los valores adicionalesrequeridos por algunos métodos se denominan parámetros.
void movehorizontal(int distance) eso se denomina cabezera del método, la pare enre parénesis es el parámetro requerido definido por un nombre y un tipo
La signatura que acabamos de mostrar indica que el método requiere un parámetro de tipo int denominado distance. El nombre proporciona una pista acerca del significado de los datos que se espera que se introduzcan. En conjunto, el nombre de un método y los tipos de parámetros presentes en su cabecera reciben el nombre de signatura del método.

Tipos de datos: Un tipo especifica cuáles son los datos que pueden traspasarse a un parámetro. El tipo int hace referencia a los números enteros. Si un método no tiene ningún parámetro, el nombre del método irá seguido por un par de paréntesis vacíos. Si dispone un parámetro, mostrará el tipo y el nombre de dicho parámetro. Los comentarios se incluyen para proporcionar información para el lector (humano) del programa.
Una vez que disponemos de una clase, podemos crear tantos objetos (o instancias) de dicha clase como queramos. Podemos cambiar un atributo de un objeto (por ejemplo, su tamaño) invocando un método sobre dicho objeto. Esto afectará a dicho objeto concreto, pero no a los restantes objetos de la misma clase.

estado: El conjunto de valores de todos los atributos que definen un objeto se denomina también estado del objeto.
En BlueJ, el estado de un objeto puede inspeccionarse seleccionando la función Inspect en el menú emergente del objeto. Cuando se inspecciona un objeto, se muestra lo que se denomina un inspector de objetos (object inspector). El inspector de objetos es una vista ampliada del objeto, en la que se muestran los atributos almacenados dentro del mismo
Algunos métodos, al ser invocados, modifican el estado de un objeto. Java denomina campos a esos atributos de los objetos. todos los objetos de la misma clase tienen los mismos campos. Es decir, el número, el tipo y los nombres de los campos son idénticos, mientras que los valores concretos de cada campo particular de cada objeto pueden ser diferentes. Por el contrario, los objetos de clases diferentes pueden tener diferentes campos. La razón es que el número, los tipos y los nombres de

los campos se definen dentro de una clase, no en un objeto. Lo mismo cabe decir de los métodos. Los métodos se definen en la clase del objeto. Como resultado, todos los objetos de una misma clase tendrán los mismos métodos. Sin embargo, los métodos se invocan sobre objetos concretos.
código java
Cuando programamos en Java, lo que hacemos esencialmente es escribir instrucciones para invocar métodos sobre los objetos observaciones:
Vemos en qué consiste el proceso de creación y de denominación de un objeto. Técnicamente, lo que estamos haciendo es almacenar el objeto Person en una variable; hablaremos de esto en detalle en el siguiente capítulo.
Apreciamos que, para llamar a un método de un objeto, lo que hacemos es escribir el nombre del objeto seguido de un punto y seguido del nombre del método. El comando termina con una lista de parámetros, o con un par de paréntesis vacíos si no hay parámetros.
Todas las instrucciones Java terminan con un punto y coma.

interacción entre objetos si queremos realizar una secuencia de tareas en Java, normalmente no lo haremos a mano. En lugar de ello, crearemos una clase que lo haga por nosotros.
Lo importante aquí es que los objetos pueden crear otros objetos y pueden invocar también los métodos de otros objetos. En un programa Java normal puede haber perfectamente cientos o miles de objetos. El usuario del programa se limita a iniciar el programa (lo que normalmente hace que se cree un primer objeto) y todos los demás objetos son creados, directa o indirectamente, por dicho objeto. Cada clase tiene un cierto código fuente asociado. El código fuente es el texto que define los detalles de la clase. El código fuente es texto escrito en el lenguaje de programación Java. Define los campos y métodos que tiene una clase y define también qué es exactamente lo que sucede cuando se invoca cada método. los métodos pueden devolver un valor como resultado. De hecho, la cabecera de cada método nos dice si el método devuelve o no un resultado y cuál es el tipo de ese resultado. Los métodos con valores de retorno nos permiten obtener información de un objeto mediante la invocación de un método. Esto significa que podemos emplear métodos para cambiar el estado de un objeto o para averiguar cuál es ese estado.
los objetos pueden pasarse como parámetros a los métodos de otros objetos. Cuando un método espera un objeto como parámetro, la signatura del método especifica como tipo de parámetro el nombre de la clase del objeto esperado.

resumen:
En este capítulo, hemos explorado los fundamentos de las clases y de los objetos. Hemos explicado el hecho de que los objetos se especifican mediante clases. Las clases representan el concepto general de las cosas, mientras que los objetos representan instancias concretas de una clase. Podemos tener múltiples objetos de cualquier clase determinada.

Los objetos disponen de métodos que utilizamos para comunicarnos con ellos. Podemos emplear un método para efectuar un cambio en el objeto o para obtener información del objeto. Los métodos pueden tener parámetros y los parámetros tienen sus correspondientes tipos. Los métodos tienen tipos de retorno, que especifican el tipo de dato que van a devolver. Si el tipo de retorno es void, entonces es que no devuelven nada.

Los objetos almacenan los datos en campos (que también tienen tipos). El conjunto de todos los valores de datos de un objeto se conoce como estado del objeto.

Los objetos se crean a partir de definiciones de clases que han sido escritas en un lenguaje de programación concreto. Buena parte de la tarea de programación en Java está relacionada con cómo escribir esas definiciones de clases. Un programa Java de gran tamaño tendrá muchas clases, cada una de las cuales contará con varios métodos, que pueden llamarse unos a otros de varias formas distintas.

Para desarrollar programas Java, necesitamos aprender a escribir definiciones de clases, entre ellas sus campos y métodos, y a ensamblar estas clases correctamente. El resto de este libro se ocupa precisamente de estas cuestiones. Términos introducidos en el capítulo: objeto, clase, instancia, método, signatura, parámetro, tipo, estado, código fuente, valor de retorno, compilador

2 definicion de clases

los elementos básicos de las definiciones de clase: campos, constructores, parámetros y métodos. Los métodos contienen instrucciones.

El texto de una clase puede dividirse en dos partes principales: un envoltorio exterior que simplemente da nombre a la clase y una parte interna, mucho más larga, que se encarga de realizar todo el trabajo. El envoltorio exterior de las diferentes clases se parece bastante. Ese envoltorio exterior contiene la cabecera de la clase, cuyo propósito principal es proporcionar a la clase un nombre. De acuerdo con un convenio ampliamente aceptado, los nombres de las clases comienzan siempre con una letra mayúscula. Siempre que se emplee de manera constante, este convenio permite distinguir fácilmente los nombres de las clases de otros tipos de nombres, como los nombres de variables y los nombres de métodos, que describiremos más adelante. Encima de la cabecera de la clase se incluye un comentario (en texto de color azul) que indica alguna observación sobre la clase.

palabras clave: Las palabras "public" y "class" forman parte del lenguaje Java A términos como "public" y "class" los denominamos palabras clave o palabras reservadas; ambos términos se utilizan con frecuencia y de manera intercambiable. En Java existen unas 50 palabras de este tipo, y el lector pronto se acostumbrará a reconocer la mayor parte de ellas. Un aspecto que conviene recordar es que las palabras clave Java nunca contienen letras mayúsculas, mientras que las que elegimos como programadores (por ejemplo, "TicketMachine") son a menudo una mezcla de mayúsculas y minúsculas.

parte interna de la clase: campos, consrucores méodos
En la parte interna de la clase definimos los campos, los constructores y los métodos que proporcionan a los objetos de dicha clase sus características y comportamientos propios. Resumimos las características esenciales de estos tres componentes de una clase de la forma siguiente:

- Los campos almacenan datos de manera persistente dentro de un objeto.
- Los constructores son responsables de garantizar que un objeto se configure apropiadamente en el momento de crearlo por primera vez.
- Los métodos implementan el comportamiento de un objeto; proporcionan su funcionalidad.

Los campos también se conocen con el nombre de variables de instancia, porque la palabra variable se utiliza como término general para todos aquellos elementos que permiten almacenar datos en un programa. Los campos son pequeñas cantidades de espacio dentro de un objeto que pueden emplearse para almacenar datos de manera persistente. Todos los objetos tendrán espacio para cada campo declarado en su clase. Cada campo dispone de su propia declaración en el código fuente. Dentro de la definición completa de la clase. los campos siempre se definen como privados (private). Como los campos pueden almacenar valores que varíen con el tiempo, se conocen también con el nombre de variables. En caso necesario, el valor almacenado en un campo puede modificarse con respecto a su valor inicial.
cada vez que definamos una variable de campo dentro de una clase: **private + tipo(int, String...) + nombre + ;**

Constructores: Los constructores tienen un papel especial que cumplir. Son responsables de garantizar que cada objeto se configure adecuadamente en el momento de crearlo por vez primera. En otras palabras, garantizan que cada objeto esté listo para ser utilizado inmediatamente después de su creación. Este proceso de construcción también se denomina inicialización. Una de las características

distintivas de los constructores es que tienen el mismo nombre de la clase en la se encuentran definidos. En general, el nombre del constructor sigue inmediatamente a la palabra public, sin ningún otro elemento entre ellos. Cabe esperar que exista una estrecha conexión entre lo que sucede en el cuerpo de un constructor y en los campos de la clase. Esto se debe a que uno de los papeles principales del constructor es el de inicializar los campos. o pasado como parametros el valor del campo necesario para el nuevo objeto o inicializando los campos a una constante siempre que se cree un objeto de esa clase.
Los constructores y los métodos desempeñan papeles muy distintos en la vida de un objeto, pero la forma en que ambos reciben valores desde el exterior es la misma: a través de parámetros. Los parámetros son otro tipo de variable, igual que los campos, por lo que se utilizan para almacenar datos. Los parámetros son variables que se definen en la cabecera de un constructor o de un método. Los parámetros se emplean como una especie de mensajeros temporales, que transportan datos cuyo origen se sitúa fuera del constructor o método y que hacen que esos datos estén disponibles en el interior del constructor o método.
espacio adicional para el objeto que solo se crea cuando el constructor se ejecuta. Lo denominaremos espacio del constructor del objeto (o espacio del método cuando hablemos acerca de métodos y no de constructores), ya que en aquel caso la situación es exactamente la misma. El espacio del constructor se utiliza para proporcionar espacio en el que almacenar los valores de los parámetros del constructor. En nuestros diagramas, todos las variables se representan mediante recuadros blancos. Distinguiremos entre los nombres de los parámetros dentro de un constructor o método y los valores externos de los parámetros; denominaremos a los nombres parámetros formales y a los valores parámetros reales. Un parámetro formal solo está disponible para un objeto dentro del cuerpo de un constructor o método que lo declare. Decimos que el ámbito de un parámetro está restringido al cuerpo del constructor o método en el que se declara. Por el contrario, el ámbito de un campo es todo el conjunto de la definición de la clase: puede accederse a él desde cualquier punto de la misma clase. Se trata de una diferencia muy importante entre estos dos tipos de variables.
Un concepto relacionado con el ámbito de las variables es el tiempo de vida de las mismas. El tiempo de vida de un parámetro está limitado a una única llamada a un constructor o método. Cuando se invoca un constructor o método, se crea el espacio adicional para las variables de parámetro y los valores externos se copian en dicho espacio. Una vez que la llamada ha completado su tarea, los parámetros formales desaparecen y los valores que contenían se pierden. En otras palabras, cuando el constructor ha terminado de ejecutarse, se elimina todo el espacio del constructor), junto con las variables de parámetro contenidas dentro del mismo.
Por el contrario, el tiempo de vida de un campo coincide con el del objeto al que pertenece. Cuando se crea un objeto, se crean también todos los campos del mismo, y esos campos persisten mientras dure el tiempo de vida del objeto.
Al igual que cabía esperar que existiera una estrecha conexión entre un constructor y los campos de una clase, también es de esperar que exista una estrecha conexión entre los parámetros del constructor y los campos, porque a menudo se necesitarán valores externos para configurar los valores iniciales de uno o más de esos campos. Cuando suceda así, los tipos de los parámetros se asemejarán estrechamente a los tipos de los campos correspondientes.

Una de las cosas que puede que haya observado es que los nombres de variables que utilizamos para los campos y los parámetros tienen una estrecha conexión con el propósito de la variable.

Las instrucciones de asignación funcionan tomando el valor que aparece en el lado derecho del operador y copiando dicho valor en la variable especificada en el lado izquierdo. El lado derecho se denomina expresión. En su forma más general, las expresiones son elementos que calculan un valor, pero en este caso la expresión consiste en una sola variable, cuyo valor se copia en la variable. Una regla relativa a las instrucciones de asignación es que el tipo de la expresión del lado derecho debe corresponderse con el tipo de la variable a la que se asigna.
Esta misma regla también se aplica entre parámetros formales y reales: el tipo de una expresión de parámetro real debe corresponderse con el tipo de la variable que actúa como parámetro formal.

Métodos Los métodos tienen dos partes: una cabecera y un cuerpo. Es importante distinguir entre las cabeceras de los métodos y las declaraciones de los campos, porque pueden parecer bastante similares. las cabeceras de los métodos siempre incluyen una pareja de paréntesis –"(" y ")"– y no incluyen punto y coma al final de la cabecera. El cuerpo del método es el resto del método, es decir, el código situado después de la cabecera. Siempre se encierra entre un par de llaves Los cuerpos de los métodos contienen las declaraciones y las instrucciones que definen lo que hace un objeto cuando se invoca ese método. Las declaraciones se utilizan para crear espacio adicional de variables temporales, mientras que las instrucciones describen las acciones del método
Cualquier conjunto de declaraciones e instrucciones situado entre una pareja de llaves se conoce con el nombre de bloque.
El método tiene un tipo de retorno mientras que el constructor no tiene ningún tipo de retorno. El tipo de retorno se escribe justo delante del nombre del método. Esta es una diferencia que se aplica en todos los casos. En Java, una regla que se aplica de manera general es que los constructores no pueden tener un tipo de retorno. Por otro lado, tanto los constructores como los métodos pueden tener cualquier número de parámetros formales, incluyendo ninguno.
Cuando un método contiene una instrucción de retorno, será siempre la instrucción final de dicho método, porque una vez que se ejecute dicha instrucción de retorno no se podrá ejecutar ninguna instrucción adicional en ese método. Los tipos de retorno y las instrucciones de retorno funcionan conjuntamente.

Métodos selectores y mutadores: se denominan métodos selectores (o simplemente selectores) devuelven al llamante información acerca del estado de un objeto; proporcionan acceso a información sobre el estado del objeto. Un selector suele contener una instrucción de retorno, para poder devolver dicha información.
Existe confusión acerca de lo que realmente significa "devolver un valor". A menudo se tiende a creer que significa que el programa imprime algo, pero no es así , En realidad, devolver un valor significa que se transfiere una cierta información internamente entre dos partes diferentes del programa. Una parte del programa ha solicitado la información de un objeto mediante la invocación de un método y el valor de retorno es la forma que tiene el objeto de devolver dicha información al llamante.

De la misma forma que podemos pensar en una llamada a un selector como si fuera una solicitud de información (una pregunta), veremos una llamada a un mutador como si fuera una solicitud para que un objeto cambie su estado. La forma más básica de mutador es aquella que admite un único parámetro cuyo valor se utiliza para sobrescribir directamente lo que haya almacenado en uno de los campos del objeto. Como complemento directo de los métodos "get", este conjunto de métodos se denominan a menudo métodos "set". Un efecto distintivo de un mutador es que un objeto mostrará a menudo un comportamiento ligeramente distinto antes y después de invocar a ese mutador.
Un tipo de retorno void indica que el método no devuelve ningún valor al llamante. Este tipo de retorno es significativamente distinto a todos los demás tipos de retorno. En BlueJ, la diferencia más destacable es que no se muestra ningún cuadro de diálogo de valor de retorno después de una llamada a un método void. Dentro del cuerpo de un método void, esta diferencia se refleja en el hecho de que no hay instrucción de retorno

impresion desde métodos: una instrucción como: System.out.println("algo");
imprime literalmente la cadena de caracteres que aparece entre la pareja de caracteres de dobles comillas. al método println del objeto System.out que está incorporado en el lenguaje Java, y lo que aparece entre los paréntesis es el parámetro de cada llamada al método, como cabría esperar. En println usamos nombres literales, llamados literal de cadena con compillas y si queremos imprimir el valor de por ejemplo un nombre del campo, este no lo ponemo on comillas pq queremos su valor, no su nombre.
Cuando se utiliza entre una cadena y cualquier otra cosa, "+" es un operador de concatenación de cadenas (es decir, concatena o junta cadenas de caracteres con el fin de crear una nueva cadena) y no

un operador de suma aritmética.

Resumen sobre métodos: Los métodos implementan las acciones fundamentales realizadas por los objetos. Un método con parámetros recibirá los datos que se le transfieran desde la entidad que invoca a ese método y usará dichos datos para poder llevar a cabo una tarea concreta. Sin embargo, no todos los métodos utilizan parámetros; muchos hacen uso simplemente de los datos almacenados en los campos del objeto para llevar a cabo su tarea.
Si un método tiene un tipo de retorno distinto de void, devolverá algún dato al lugar desde el que fue invocado, y dicho dato será utilizado, casi con total seguridad, en el llamante para realizar cálculos adicionales o para controlar la ejecución del programa. Muchos métodos, sin embargo, tienen un tipo de retorno void y no devuelven nada, aunque realizan una tarea útil dentro del contexto de su objeto. Los métodos
selectores tienen tipos de retorno distintos de void y devuelven información acerca del estado de un objeto. Los métodos mutadores modifican el estado de un objeto. Los mutadores suelen tener parámetros, cuyos valores se utilizan en la modificación, aunque es perfectamente posible escribir un método mutador que no admita ningún parámetro.
Hemos visto que la clase tiene una pequeña capa externa que proporciona un nombre a la clase y un cuerpo interno de mayor tamaño que contiene campos, un constructor y varios métodos. Los campos se utilizan para almacenar datos que permiten a los objetos mantener un estado que persiste entre llamadas sucesivas a los métodos. Los constructores se utilizan para configurar un estado inicial cuando se crea el objeto. Disponer de un estado inicial apropiado permitirá a los objetos responder adecuadamente a las llamadas a métodos que se produzcan inmediatamente después de la creación de esos objetos. Los métodos implementan el comportamiento definido para los objetos pertenecientes a esa clase. Los métodos selectores proporcionan información acerca del estado de un objeto y los métodos mutadores modifican el estado de un objeto.
Hemos visto que los constructores se distinguen de los métodos porque tienen el mismo nombre que la clase en la que están definidos. Tanto los constructores como los métodos pueden aceptar parámetros, pero solo los segundos pueden tener un tipo de retorno. Los tipos de retorno distintos de void nos permiten pasar un valor desde el interior de un método hacia el lugar desde el que el método fue invocado. Un método con un tipo de retorno distinto de void debe tener al menos una instrucción de retorno dentro de su cuerpo; a menudo, dicha instrucción será la última del método. Los constructores nunca tienen un tipo de retorno, ni siquiera void.

insrucciones condicionales: Las instrucciones condicionales también se conocen con el nombre de instrucciones if, debido a la palabra clave usada en la mayoría de los lenguajes de programación para implementarlas. Una instrucción condicional nos permite llevar a cabo una de dos acciones posibles con base en el resultado de una prueba o comprobación. Si la comprobación es verdadera entonces hacemos una cosa; en caso contrario, hacemos algo distinto.

BlueJ muestra el código fuente con algunos detalles de formato adicionales: concretamente, sitúa recuadros coloreados alrededor de algunos elementos

Estas indicaciones de color se conocen con el nombre de representación visual del ámbito y pueden ayudarnos a clarificar las unidades lógicas del programa. Un ámbito (también denominado bloque) es una unidad de código que normalmente está encerrada entre llaves. El cuerpo completo de una clase es un ámbito, como también lo son el cuerpo de cada método y las partes if y else de una instrucción condicional. Como puede ver, los ámbitos están a menudo anidados: la instrucción if se encuentra dentro de un método, que a su vez se encuentra dentro de una clase. BlueJ ayuda a diferenciar los distintos ámbitos empleando distintos colores.

Variables locales Hasta ahora, nos hemos encontrado con dos tipos diferentes de variables: campos (variables de instancia) y parámetros. Ahora introducimos un tercer tipo. Lo que tienen en común todos estos tipos de variable es que almacenan datos, pero cada tipo de variable desempeña

un papel diferente.
el cuerpo de un método puede contener tanto declaraciones como instrucciones..
Las declaraciones de variables locales parecen similares a las declaraciones de campos, pero las palabras clave private y public nunca aparecen en la declaración. Los constructores también pueden tener variables locales. Al igual que los parámetros formales, las variables locales tienen un ámbito que está limitado a las instrucciones del método al que pertenecen. Su tiempo de vida coincide con el tiempo durante el cual se está ejecutando el método: se crean cuando se invoca un método y se destruyen cuando el método termina.
Cabe preguntarse para qué hacen faltan las variables locales si ya disponemos de campos. Las variables locales se usan principalmente como almacenamiento temporal, para ayudar a un método a completar su tarea; podemos considerarlas como un almacenamiento de datos para un único método. Por el contrario, los campos se utilizan para almacenar datos que permanecen durante toda la vida de un objeto completo. Los datos almacenados en campos son accesibles para todos los métodos del objeto. Tenemos que intentar evitar declarar como campos aquellas variables que solo tienen un uso local (en el nivel de método), es decir, cuyos valores no necesitan recordarse más allá de una única llamada al método. Por tanto, incluso aunque dos o más métodos de una misma clase utilicen variables locales con un propósito similar, no sería apropiado definirlas como campos si sus valores no necesitan persistir más allá del momento en que termina la ejecución de esos métodos.
Al igual que los parámetros formales, las variables locales tienen un ámbito que está limitado a las instrucciones del método al que pertenecen. Su tiempo de vida coincide con el tiempo durante el cual se está ejecutando el método: se crean cuando se invoca un método y se destruyen cuando el método termina.

Campos parámetros y variables locales: Los tres tipos de variables son capaces de almacenar un valor que se corresponda con su tipo definido. Los campos se definen fuera de los constructores y métodos. Los campos se utilizan para almacenar datos que persisten durante toda la vida de un objeto. Por ello, mantienen el estado actual de un objeto. Tienen un tiempo de vida que coincide con la duración del objeto al que pertenecen. Los campos tienen un ámbito que coincide con la clase: son accesibles desde cualquier punto de la clase a la que pertenecen, de modo que se pueden utilizar dentro de cualquiera de los constructores o métodos de la clase en la que han sido definidos.
Mientras se definan como privados (private) no se podrá acceder a los campos desde ningún punto situado fuera de la clase en la que están definidos.
Los parámetros formales y las variables locales solo persisten mientras que se está ejecutando un constructor o método. Su tiempo de vida coincide con la duración de una única invocación, por lo que sus valores se pierden entre invocaciones sucesivas. Desde ese punto de vista, actúan como ubicaciones de almacenamiento temporal, no permanente.
Los parámetros formales se definen en la cabecera de un constructor o método. Reciben sus valores del exterior, siendo inicializados de acuerdo con los valores de los parámetros reales que forman parte de la llamada al constructor o al método. Los parámetros formales tienen un ámbito que está limitado al constructor o método en los que se los define. Las variables locales se definen dentro del cuerpo de un constructor o método. Solo pueden inicializarse y utilizarse dentro del cuerpo del constructor o método en el que se las define. Las variables locales deben inicializarse antes de poder ser utilizadas en una expresión; no se les proporciona un valor predeterminado.
Las variables locales tienen un ámbito que está limitado al bloque en el que están definidas. No se puede acceder a ellas desde ningún punto situado fuera de dicho bloque.

3 interacción de objetos

Abstracción y modulatización: A medida que la complejidad de un problema aumenta, cada vez se hace más difícil controlar todos los detalles simultáneamente.
La solución que usaremos para tratar con el problema de la complejidad es la abstracción. Dividiremos el problema en una serie de subproblemas, que a su vez dividiremos en sub-subproblemas, y así sucesivamente, hasta que cada problema individual sea suficientemente pequeño para poder resolverlo de

manera sencilla. Una vez resuelto uno de los subproblemas, ya no dedicaremos más tiempo a pensar en los detalles de esa parte, sino que trataremos la solución como si fuera un único bloque componente que podemos emplear para solucionar el siguiente problema. Esta técnica se denomina en ocasiones divide y vencerás.
la modularización y la abstracción se complementan entre sí. La modularización es el proceso de dividir grandes cosas (problemas) en partes más pequeñas, mientras que la abstracción es la capacidad de ignorar los detalles para centrarse en la panorámica general.

Para ayudarnos a mantener una visión panorámica en los problemas complejos, tratamos de identificar subcomponentes que podamos programar como entidades independientes. Después, intentamos usar esos subcomponentes como si fueran partes simples, sin preocuparnos acerca de su complejidad interna. En la programación orientada a objetos, estos componentes y subcomponentes son precisamente objetos.
las clases definen tipos. los nombres de clases pueden utilizarse como tipos. La declaración de un campo u otra variable de tipo de clase no crea automáticamente un objeto de ese tipo; al contrario, el campo inicialmente está vacío. El objeto asociado debe crearse explícitamente, y veremos cómo se hace cuando analicemos el constructor de la clase

3.1 Diagramas de clases y diagramas de objetos

El diagrama de clases muestra la vista estática. En él se refleja lo que tenemos en el momento de escribir el programa. (Tenemos dos clases y la flecha indica que la clase ClockDisplay hace uso de la clase Number- Display (NumberDisplay aparece mencionado en el código fuente de ClockDisplay). También vemos, por eso, que ClockDisplay depende de NumberDisplay.)
el diagrama de objetos muestra la situación en tiempo de ejecución (cuando se está ejecutando la aplicación). Esto se denomina también vista dinámica. El diagrama de objetos también muestra otro detalle importante: cuando una variable almacena un objeto, el objeto no se almacena directamente en la variable, sino que lo que la variable contiene es una referencia a objeto. El objeto al que se hace referencia está almacenado fuera del objeto en el que aparece la referencia, y es precisamente esa referencia a objeto lo que enlaza los dos objetos entre sí. BlueJ proporciona solo la vista estática.
Para planificar y comprender programas Java hemos de ser capaces de construir diagramas de objetos sobre papel o en nuestra cabeza. Cuando pensemos en lo que va a hacer nuestro programa, pensaremos en las estructuras de objetos que creará y en cómo interaccionarán esos objetos. Es esencial saber visualizar las estructuras de objetos.

Tipos primitivos y tipos de objeto: Java trabaja con dos especies muy distintas de tipos: tipos primitivos y tipos de objeto. Los tipos primitivos están todos ellos predefinidos en el lenguaje Java. Los tipos de objeto son aquellos que están definidos mediante clases. Algunas clases se definen mediante el sistema Java estándar (como por ejemplo String); otras son las que escribimos nosotros mismos. Tanto los tipos primitivos como los tipos de objeto pueden emplearse como tipos, pero hay situaciones en las que se comportan de forma diferente. Una de las diferencias afecta al modo en que se almacenan los valores. Como hemos podido ver en nuestros diagramas, los valores primitivos se almacenan directamente en una variable. Por el contrario, los objetos no se almacenan directamente en la variable, sino que lo que se almacena es una referencia al objeto
(Si uno de los operandos de una operación suma es una cadena y el otro no, entonces automáticamente se convierte el otro operando a una cadena, para realizar después una concatenación. Esto funciona para todos los tipos. Con independencia del tipo que se "sume" a una cadena, dicho tipo se convertirá automáticamente a una cadena y luego se concatenará.)

La sintaxis de una operación de creación de un nuevo objeto es: **new NombreClase (lista-parámetros)**
La operación new hace dos cosas: 1 Crea un nuevo objeto de la clase indicada
2 Ejecuta el constructor de dicha clase.
Si el constructor de la clase se ha definido de manera que incluya parámetros, entonces habría que suministrar los parámetros reales en la instrucción new.

Es común que las definiciones de clases contengan versiones alternativas de los constructores o de los métodos que proporcionan diversas formas de llevar a cabo una tarea concreta y que se diferencian entre sí por sus conjuntos de parámetros. Esto se conoce con el nombre de **sobrecarga** de un constructor o de un método.
Si el método se encuentra dentro la misma clase que la propia llamada al método, decimos que es una llamada a un método interno. Las llamadas a métodos internos tienen la sintaxis **nombreMetodo (lista-parámetros)** Una llamada a un método interno no tiene nombre de variable, No se necesita una variable ya que, con una llamada a un método interno, el objeto llama al método de por sí.
Cuando se encuentra una llamada a método, se ejecuta el método correspondiente, y luego la ejecución vuelve a la llamada a método y continúa con la siguiente instrucción situada después de la misma.
Para que la signatura de un método se corresponda con la llamada a método, tanto el nombre como la lista de parámetros deben corresponderse. Esta necesidad de ajustarse tanto al nombre del método como a la lista de parámetros es importante, porque puede haber más de un método con el mismo nombre dentro de una clase, en caso de que ese método esté sobrecargado.
Es común que las definiciones de clases contengan versiones alternativas de los constructores o de los métodos que proporcionan diversas formas de llevar a cabo una tarea concreta y que se diferencian entre sí por sus conjuntos de parámetros. Esto se conoce con el nombre de sobrecarga de un constructor o de un método.

llamadas a métodos externos: La sintaxis de una llamada a método externo es: **objeto . nombreMetodo (lista-parámetros)** Esta sintaxis se conoce como notación con punto. Está compuesta por un nombre de objeto, un punto, el nombre del método y los parámetros para la llamada. Es muy importante darse cuenta de que lo que utilizamos aquí es el nombre de un objeto y no el nombre de una clase.
La diferencia entre llamadas a métodos internos y externos está clara; la presencia de un nombre seguido por un punto nos informa de que el método invocado forma parte de otro objeto.
El conjunto de métodos que un objeto pone a disposición de otros objetos se denomina interfaz. Veremos las interfaces con mucho más detalle más adelante en el libro.

en ocasiones resulta ventajoso utilizar herramientas adicionales para entender mejor cómo se ejecuta un programa. Una de esas herramientas, a la que ahora echaremos un vistazo es el depurador. Un depurador es un programa que permite a los programadores ejecutar una aplicación paso a paso. Normalmente, proporciona funciones para iniciar y detener un programa en puntos seleccionados del código fuente, así como para examinar los valores de las variables. El nombre debugger En inglés, los errores en programas informáticos se los denomina coloquialmente bugs. Por ello, a los programas depuradores que ayudan a eliminar esos errores se les conoce con el nombre de debuggers. No está del todo claro de dónde proviene el término bug, que en inglés significa "insecto". Hay una famosa anécdota de lo que se conoce como "el primer error informático", que fue debido a un insecto real (en realidad, una polilla). Ese insecto fue encontrado, en 1945, dentro de la computadora Mark II por Grace Murray Hopper, una de las primeras personas que trabajó en el campo de la informática. Todavía se conserva en el Museo Nacional de Historia Americana del Instituto Smithsoniano un libro de registro en el que aparece una entrada con esta polilla pegada con cinta adhesiva al libro y con la anotación "Primer caso real de localización de un insecto (bug)". Sin embargo, tal como está redactada esa anotación, se sugiere que el término bug ya había estado utilizándose antes de que este insecto real causara problemas en el Mark II.
Los depuradores varían mucho en complejidad. Los utilizados por desarrolladores profesionales tienen una gran cantidad de funciones que resultan útiles para hacer exámenes sofisticados de múltiples facetas de una aplicación BlueJ tiene un depurador incorporado que es mucho más simple. Podemos utilizarlo para detener nuestro programa, para ejecutar el código línea por línea y para examinar los valores de las variables. Sin embargo, a pesar de la aparente falta de sofisticación de este depurador, es más que suficiente para obtener una gran cantidad de información.

La palabra clave this: tenemos una situación que se conoce con el nombre de sobrecarga de nombres; es la situación en que se utiliza el mismo nombre para dos entidades diferentes.
Es importante entender que los campos y los parámetros son variables diferentes, que existen independientemente unas de otras, aun cuando compartan nombres similares. El hecho de que un parámetro y un campo compartan un nombre no es ningún problema en Java. Lo que sí es un problema es cómo referenciar las variables para poder distinguir entre los dos conjuntos de valores.
Si simplemente usamos el nombre de variable ¿qué variable se utilizaría, el parámetro o el campo? La especificación del lenguaje Java permite responder a esta pregunta. Especifica que se utilice siempre la definición que tenga su origen en el bloque circundante más próximo. todo lo que necesitamos es un mecanismo que nos permita acceder a un campo cuando exista una variable definida más próxima con el mismo nombre. Para esto se utiliza precisamente la palabra clave this. La expresión this hace referencia al objeto actual. Escribir this.from hace referencia al campo from del objeto actual. Por tanto, esta estructura nos da un medio de referirnos al campo, en lugar de al parámetro que tiene el mismo nombre.
this.from = from; Esta instrucción, como ahora vemos, tiene el siguiente efecto: campo denominado from = parámetro denominado from; En otras palabras, asigna el valor del parámetro al campo que tiene el mismo nombre. Si un cierto nombre describe perfectamente el uso, resulta razonable emplearlo en ambos casos y aceptar la complicación de utilizar la palabra clave this dentro de la asignación para resolver el conflicto de nombres.
El depurador no solo nos permite interrumpir la ejecución del programa e inspeccionar las variables, sino que también nos permite avanzar en la ejecución lentamente. Al estar parados en un punto de interrupción, si hacemos clic en el botón Step (paso) se ejecuta una única línea de código y luego la ejecución vuelve a detenerse.
(Recuerde: construir un objeto hace dos cosas, crear un objeto y ejecutar el constructor). Invocar el constructor funciona de forma muy similar a la invocación de métodos

4 agrupación de objeos

El principal objeto de este capítulo es presentar algunas de las formas en las que pueden agruparse objetos para formar colecciones. En particular, hablaremos de la clase ArrayList como ejemplo de colecciones de tamaño flexible. Estrechamente asociada con las colecciones se encuentra la necesidad de iterar a lo largo de los elementos que esas colecciones contienen. Con ese propósito, presentaremos tres nuevas estructuras de control: dos versiones del bucle for y el bucle while.

Vimos entonces que la abstracción nos permite simplificar un problema, identificando componentes discretos que puedan contemplarse como un todo, en lugar de tener que preocuparnos por sus detalles. Analizaremos este principio en acción cuando comencemos a hacer uso de las clases de librería disponibles en Java. Aunque estas clases no son, estrictamente hablando, parte del lenguaje, algunas de ellas están íntimamente asociadas con la escritura de la mayor parte de los programas Java, por lo que a menudo se piensa en ellas como en una parte más del lenguaje. La mayoría de la gente que escribe programas Java comprueba constantemente las librerías para ver si alguien ha escrito ya una clase que ellos puedan aprovechar. De esta forma, se ahorran una gran cantidad de esfuerzo, que puede emplearse mejor en trabajar en otras partes del programa. El mismo principio se aplica en la mayoría de los demás lenguajes de programación, que también suelen disponer de librerías de clases útiles. Por tanto, merece la pena familiarizarse con el contenido de la librería y saber cómo usar las clases más comunes. La potencia de la abstracción reside en que, para usar una clase de manera efectiva, normalmente no nos hace falta conocer muchos detalles (¡de hecho, acaso ninguno!) acerca de las interioridades de la clase. Si utilizamos una clase de la librería, lo que haremos será escribir código que cree instancias de esa clase, después de lo cual nuestros objetos podrán interactuar con los objetos de la librería.
la colección como abstracción: la idea de colección, el concepto de agrupar cosas para poder referirnos a ellas y manejarlas de manera conjunta. En un contexto de programación, la abstracción colección se convierte en una clase de un cierto tipo, y las operaciones serían los métodos de esa

clase. Una colección (mi colección de música) sería una instancia de la clase. Además, los elementos almacenados en una instancia de colección serían, ellos mismos, objetos.
Hasta ahora, no hemos visto ninguna característica de Java que nos permita agrupar un número arbitrario de elementos. Quizá podríamos definir una clase con un grupo muy extenso de campos individuales, para abarcar un número fijo, pero muy elevado, de elementos. Sin embargo, los programas suelen necesitar una solución que sea más general. Sería adecuada aquella que no nos exigiera saber de antemano cuántos elementos vamos a agrupar, y que no nos obligara a fijar un límite superior para dicho número. la forma más simple posible de agrupar objetos, mediante una lista secuencial desordenada de tamaño flexible: ArrayList.

Vamos a escribir una clase que nos ayude a organizar nuestros archivos de música almacenados en una computadora. Nuestra clase no almacenará en realidad los detalles de los archivos; en su lugar, lo que hará será delegar esa responsabilidad en la clase estándar de librería ArrayList, que nos ahorrará mucho trabajo. Entonces, ¿por qué necesitamos escribir una clase? Un punto importante que hay que tener en mente al tratar con las clases de librería es que estas no se han escrito para ningún escenario de aplicación concreto, son clases de propósito general. Esto significa que quienes proporcionan las operaciones específicas de cada escenario son las clases que escribamos para utilizar las clases de librería.
Librerías de clases Una de las características de los lenguajes orientados a objetos que les dota de más potencia es que a menudo suelen estar acompañados por librerías de clases. Estas librerías suelen contener varios cientos o miles de clases distintas, que han demostrado ser útiles para los desarrolladores en un amplio rango de proyectos distintos. Java denomina a sus librerías paquetes. Las clases de librería se emplean exactamente de la misma forma que utilizaríamos nuestras clases. Las instancias se construyen utilizando new y las clases tienen campos, constructores y métodos

La primera línea del archivo de clase ilustra la forma en la que obtenemos acceso a una clase de librería en Java: mediante una instrucción de importación (import): import java.util.ArrayList;
Esto hace que la clase ArrayList del paquete java.util esté disponible a la hora de definir nuestra clase. Las instrucciones de importación deben colocarse siempre antes de las instrucciones de clases en un archivo. Una vez importado desde un paquete de esta manera un archivo de clase, podemos utilizar dicha clase como si fuera una de nuestras propias clases.
cuando señalamos que ArrayList es una clase de colección de propósito general, es decir, que no está restringida en lo que respecta a los tipos de objeto que puede almacenar. Sin embargo, cuando creamos un objeto ArrayList, tenemos que ser específicos acerca del tipo de objetos que se almacenarán en esa instancia concreta. Podemos almacenar cualquier tipo que decidamos, pero es necesario designar dicho tipo al declarar una variable ArrayList. Las clases como ArrayList, que se parametrizan con un segundo tipo, se denominan clases genéricas.
Al utilizar colecciones, por tanto, siempre tenemos que especificar dos tipos: el tipo de la propia colección (en este caso, ArrayList) y el tipo de los elementos que pretendemos almacenar en esa colección (que aquí es String). Podemos leer la definición completa de tipo ArrayList$< String >$ como "una colección ArrayList de objetos de tipo String".
Observe que al crear la instancia ArrayList hemos escrito la siguiente instrucción: files = new ArrayList$< String >$(); Se trata de la denominada notación diamante (debido a que los dos corchetes en ángulo crean una forma de rombo) y parece poco habitual . Antes hemos visto que la instrucción new adopta la forma siguiente: new nombre-tipo (parámetros)
La primera versión, con la notación diamante (que deja la mención del tipo String) es simplemente un modo abreviado utilizado por comodidad. Si la creación del objeto colección se combina con una asignación, el ordenador podrá deducir el tipo de los elementos de colección a partir del tipo de variable del lado izquierdo de la asignación, y Java nos permite no tener que definirla de nuevo. El tipo de elemento se infiere automáticamente a partir del tipo de variable.
La clase ArrayList define numerosos métodos, pero nosotros, para dar soporte a la funcionalidad que nos hace falta, por el momento solo utilizaremos cuatro de ellos: add, size, get y remove.

Estructuras de objetos con colecciones: Para entender cómo opera un objeto colección como ArrayList resulta útil examinar un diagrama de objetos. Existen al menos tres características de la clase ArrayList que debemos considerar:
- Es capaz de incrementar su capacidad interna según sea necesario: a medida que se añaden nuevos elementos, se limita a crear espacio para ellos.
- Mantiene su propio contador privado, el número de elementos que almacena en cada instante. Su método size devuelve el valor de ese contador.
- Mantiene el orden de los elementos que se inserten en la lista. El método add almacena cada nuevo elemento al final de la lista. Posteriormente, podemos extraerlos en el mismo orden.
Todo el trabajo complicado se realiza dentro del objeto ArrayList. Esta es una de las grandes ventajas de utilizar clases de librería. Alguien ha invertido tiempo y esfuerzo para implementar algo útil y nosotros tenemos acceso a esa funcionalidad de manera prácticamente gratuita sin más que utilizar esa clase. Por el momento, no necesitamos preocuparnos por cómo es capaz un ArrayList de soportar esas características. Nos basta con ser capaces de apreciar lo útil que es esta capacidad. Recuerde: esta supresión de los detalles es uno de los beneficios que la abstracción nos proporciona; implica que podemos utilizar ArrayList para escribir cualquier número de clases distintas que necesiten almacenar un número arbitrario de objetos.

La nueva notación que utiliza los corchetes angulares merece alguna explicación adicional. El tipo de nuestro campo files se ha declarado como ArrayList$< String >$ La clase que utilizamos aquí se llama simplemente ArrayList, pero requiere que se especifique un segundo tipo como parámetro cuando se usa para declarar campos u otras variables. Las clases que requieren un parámetro de tipo como este se denominan clases genéricas. A diferencia de otras clases que hemos visto hasta ahora, las clases genéricas no definen un único tipo en Java, sino que pueden definir muchos tipos.
Cada ArrayList particular es un tipo distinto, que puede utilizarse en las declaraciones de campos, parámetros y valores de retorno.
los elementos almacenados en las colecciones ArrayList tienen una numeración o posicionamiento implícito que comienza en 0. La posición de un objeto dentro de una colección se conoce comúnmente como el nombre de índice. Al primer elemento añadido a una colección se le da el número de índice 0, al segundo se le da el número de índice 1, y así sucesivamente. Los métodos listFile y removeFile ilustran la forma en que se utiliza el número de índice para acceder a un elemento de un ArrayList: uno lo hace a través del método get y el otro a través del método remove.
La clase ArrayList tiene un método remove que toma como parámetro el índice del objeto que hay que eliminar. Un detalle del proceso de eliminación del que debemos ser conscientes es que puede cambiar los valores de índice en los que están almacenados otros objetos de la colección. Si se elimina un objeto con un número de índice bajo, entonces la colección desplazará una posición todos los elementos posteriores con el fin de rellenar el hueco. Como consecuencia, sus números de índice disminuirán en una unidad. es posible insertar elementos en un ArrayList en una posición distinta del final de la lista. Esto quiere decir que los elementos que ya se encuentren en la lista podrían ver sus números de índice incrementados cuando se añada un nuevo elemento.

bucles: el uso de instrucciones de bucle, que también se conocen con el nombre de estructuras iterativas de control. El primer bucle que presentaremos para enumerar los archivos es un bucle especial que se utiliza con colecciones y que elimina completamente la necesidad de utilizar una variable de índice: se denomina bucle for-each. Un bucle for-each es una de las formas de llevar a cabo repetidamente un conjunto de acciones sobre los elementos de una colección
for(TipoElemento elemento : colección) { cuerpo del bucle }
El principal elemento nuevo de Java es la palabra for. El lenguaje Java tiene dos variantes del bucle for: una es el bucle for-each, que es el que estamos analizando aquí, y la otra se denomina simplemente bucle for.
Un bucle for-each tiene dos partes: una cabecera del bucle (la primera línea de la instrucción de bucle) y un cuerpo de bucle situado a continuación de la cabecera. El cuerpo contiene aquellas instrucciones que queremos ejecutar una y otra vez. El bucle for-each obtiene su nombre de la forma en que podemos

interpretar su sintaxis: si leemos la palabra clave for como “for each” (“para cada”) y los dos puntos de la cabecera del bucle como “in” (“en”). Analicemos el bucle con un poco más de detalle. La palabra clave for inicia el bucle. Va seguida de una pareja de paréntesis, dentro de los cuales se definen los detalles del bucle. El primero de esos detalles es la declaración que declara una nueva variable local que se utilizará para almacenar sucesivamente los distintos elementos de la lista. A esta variable la denominamos variable de bucle. Podemos elegir el nombre que queramos para esta variable, al igual que sucede con cualquier otra; El tipo de la variable de bucle debe coincidir con el tipo de elemento declarado para la colección que vayamos a utilizar.
A continuación, aparece un carácter de dos puntos y luego la variable que contiene la colección que queremos procesar. Para esta colección, cada elemento será asignado por turno a la variable de bucle; y para cada una de esas asignaciones, se ejecuta una vez el cuerpo del bucle. En el cuerpo del bucle, utilizamos la variable de bucle para hacer referencia a cada elemento.
El bucle for-each se emplea siempre para iterar a través de una colección. Nos proporciona una forma de acceder a cada elemento de la colección sucesivamente, uno por uno, y procesar esos elementos de la forma que deseemos. Podemos decidir llevar a cabo las mismas acciones con cada elemento (como hicimos al imprimir la lista completa) o podemos ser selectivos y filtrar la lista (como hicimos al imprimir solo un subconjunto de la colección). El cuerpo del bucle puede ser todo lo complicado que queramos.
No obstante, esta sencillez esencial lleva aparejadas necesariamente algunas limitaciones. Por ejemplo, una restricción es que no podemos modificar lo que está almacenado en la colección mientras iteramos a través de ella: ni añadir nuevos elementos ni eliminar elementos de la colección. Sin embargo, esto no significa que no podamos cambiar el estado de los objetos que ya están dentro de la colección
También hemos visto que el bucle for-each no nos proporciona un valor de índice para los elementos de la colección. Si deseamos uno, tendremos que declarar y mantener nuestra propia variable local. La razón tiene que ver de nuevo con la abstracción. Al tratar con colecciones e iterar a través de ellas, resulta útil tener presentes dos consideraciones:
Un bucle for-each proporciona una estructura general de control para iterar a través de diferentes tipos de colecciones.
Existen algunos de tipos de colecciones que no asocian de manera natural índices enteros con los elementos que almacenan
Por tanto, el bucle for-each abstrae la tarea de procesar una colección completa elemento a elemento y es capaz de manejar diferentes tipos de colección. No necesitamos saber los detalles de cómo manipula las colecciones.

le recomendamos emplear un bucle for-each solo si está seguro de querer procesar la colección completa. Dicho de otra forma, una vez que el bucle comience, sabremos con seguridad cuántas veces se va a ejecutar el cuerpo del bucle; ese número de veces será igual al tamaño de la colección. Este estilo se denomina en ocasiones iteración definida. Para aquellas tareas en las que queramos detener anticipadamente el procesamiento de la colección hay otros bucles más apropiados que se pueden utilizar como, por ejemplo, el bucle while, que presentaremos a continuación. En estos casos, el número de veces que se ejecutará el cuerpo del bucle es menos preciso; normalmente, dependerá de lo que suceda durante la iteración. Este estilo se denomina en ocasiones iteración indefinida. Un bucle for-each proporciona una iteración definida; dado el estado de una colección concreta, el cuerpo del bucle se ejecutará un número de veces que se corresponde exactamente con el tamaño de dicha colección. Ahora bien, hay muchas situaciones en las que queremos repetir una serie de acciones pero en las que no podemos predecir de antemano exactamente cuántas veces será. Un bucle for-each no nos sirve de ayuda en estos casos

ieración indefinida: la acción se repetirá un número de veces no predecible hasta que se complete la tarea. frecuentemente nos encontraremos con situaciones en las que querremos seguir haciendo una determinada cosa hasta que la repetición deje de ser necesaria. De hecho, estas situaciones son tan comunes que la mayoría de los lenguajes de programación proporcionan al menos una (y normalmente más de una) estructura de bucle para expresarlas. Un bucle while consta de una cabecera y de un

cuerpo; el cuerpo está pensado para ser ejecutado de manera repetida. He aquí la estructura de un bucle while donde condición booleana y cuerpo del bucle son pseudocódigo, pero todo lo restante es la sintaxis de Java:
while (condición booleana) {
cuerpo del bucle
}
El bucle se inicia con la palabra clave while, seguida de una condición booleana. La condición es la que controla, en último término, cuántas veces se iterará un bucle concreto. La condición se evalúa cuando el control del programa alcanza por primera vez el bucle, y vuelve a evaluarse después de ejecutar cada vez el cuerpo del bucle. Esto es lo que da al bucle while su carácter indefinido: ese proceso de reevaluación. Si la condición se evalúa como true, entonces se ejecuta el cuerpo del bucle; y una vez que la condición se evalúe como false, se da por terminada la iteración. Entonces el programa se salta el cuerpo del bucle y la ejecución continúa con lo que haya a continuación del bucle.
ventajas: en primer lugar, el bucle while no necesita estar relacionado con una colección (podemos construir un bucle con cualquier condición que podamos escribir en forma de expresión booleana); en segundo lugar, aunque utilicemos el bucle para procesar una colección, es posible que no necesitemos procesar todos los elementos; en lugar de ello, podríamos detenernos anticipadamente, si así lo deseamos, e incluir otra componente dentro de la condición del bucle que exprese por qué querríamos terminar el bucle. Por supuesto, estrictamente hablando, lo que la condición del bucle expresa en realidad es por qué querríamos continuar, y es la negación de esa condición la que hace que el bucle se detenga. Una ventaja de tener una variable de índice explícita es que podemos utilizar su valor tanto dentro como fuera del bucle, lo que no podíamos hacer en los ejemplos de for-each. Tener una variable de índice local puede ser muy importante al realizar búsquedas en una lista, porque puede proporcionar información sobre dónde estaba ubicado el elemento y podemos hacer que esa información siga estando disponible una vez que el bucle haya finalizado.

búsquedas: La característica clave de una búsqueda es que implica una iteración indefinida; así tiene que ser necesariamente, porque si supiéramos exactamente dónde buscar, no nos haría falta realizar ninguna búsqueda. En lugar de ello, lo que tenemos que hacer es iniciar una búsqueda y luego nos hará falta un número desconocido de iteraciones antes de completarla. Esto implica que un bucle for-each es inapropiado para las búsquedas, porque siempre llevará a cabo su conjunto completo de iteracione En situaciones reales de búsqueda, tenemos que tener en cuenta que la búsqueda puede fallar: es posible que nos quedemos sin lugares en los que buscar. Eso quiere decir que normalmente tendremos que tomar en consideración dos posibilidades de finalización a la hora de escribir un bucle de búsqueda: -La búsqueda tiene éxito después de un número indefinido de iteraciones.
-La búsqueda falla después de agotar todas las posibilidades.

Debemos tener en cuenta las dos posibilidades a la hora de escribir la condición del bucle. Como la condición del bucle debe evaluarse como true si queremos iterar otra vez más, cada uno de los criterios de finalización debe poder hacer, por sí mismo, que la condición se evalúe como false para detener el bucle.
El hecho de que terminemos por analizar la lista completa en aquellos casos en los que la búsqueda falla no hace que las búsquedas fallidas constituyan un ejemplo de iteración definida. La característica clave de la iteración definida es que podemos determinar el número de iteraciones en el momento de iniciarse el bucle. Ese no será nunca el caso cuando estemos haciendo una búsqueda.

También necesitamos añadir una segunda parte a la condición que indique si hemos encontrado ya el elemento de búsqueda y que detenga la búsqueda en caso de que lo hayamos hecho.
Una variable denominada searching (o, por ejemplo, missing) configurada inicialmente con el valor true haría que la búsqueda continuara hasta que la variable se configurara como false dentro del bucle, después de haber encontrado el elemento. Una variable denominada found, configurada inicialmente como false y utilizada en la condición como !found haría que la búsqueda continuara hasta que la variable se configurara como true después de encontrar el elemento. Recuerde que toda la condición

debe evaluarse como true si queremos continuar buscando, y que debe evaluarse como false si queremos dejar de buscar, por la razón que sea.

Los bucles no se utilizan solo con colecciones. Existen muchas situaciones en las que queremos repetir un bloque de instrucciones en el que no interviene para nada ninguna colección. Los bucles for-each solo se pueden usar para iterar a través de colecciones.

Una de las ventajas de la orientación a objetos es que nos permite diseñar clases que modelen bastante fielmente los comportamientos y estructura inherentes de las entidades del mundo real que estemos intentando representar. Esto se consigue escribiendo clases cuyos campos y métodos se correspondan con los de los atributos

El tipo iterador: Utiliza un bucle while para realizar la iteración y un objeto Iterator en lugar de una variable de índice entera para controlar la posición dentro de la lista. Tenemos que ser muy cuidadosos con la denominación en este punto, porque Iterator (observe la mayúscula I) es un tipo de Java, pero también nos encontraremos con un método denominado iterator (observe la minúscula i). Examinar todos los elementos de una colección es tan común, que ya hemos visto que existe un estructura de control especial (el bucle for-each) que está diseñada a propósito para esta tarea. Además, las distintas clases de librería para colecciones de Java proporcionan un tipo común diseñado a medida para soportar la iteración, y ArrayList es típica a este respecto. El método iterator de ArrayList devuelve un objeto Iterator. Iterator también está definido en el paquete java.util, así que debemos añadir una segunda instrucción de importación al archivo de clase para poder utilizarlo:
import java.util.ArrayList;
import java.util.Iterator;
Un Iterator proporciona simplemente tres métodos, y dos de ellos se usan para iterar a través de una colección: hasNext y next. Ninguno de ellos admite parámetros, pero ambos tienen tipos de retorno definidos, de modo que se usan en expresiones.
La forma en que normalmente utilizamos un Iterator puede describirse en pseudocódigo como sigue:

Iterator $<$ *TipoElemento* $>$ it = myCollection.iterator();
while(it.hasNext()){
llamar it.next() *para obtener el siguiente elemento*
hacer algo con ese elemento }

utilizamos primero el método iterator de la clase ArrayList para obtener un objeto Iterator. Observe que Iterator es también un tipo genérico, por lo que lo parametrizamos con el tipo de los elementos contenidos en la colección a través de la cual estamos iterando. A continuación, empleamos ese Iterator para comprobar repetidamente si hay más elementos, mediante it.hasNext(), y para obtener el siguiente elemento, mediante it.next(). Un punto importante que hay que resaltar es que es al objeto Iterator al que le pedimos que devuelva el siguiente elemento y no al objeto colección. De hecho, tendemos a no referirnos directamente en absoluto a la colección dentro del cuerpo del bucle; toda la interacción con la colección se realiza a través del Iterator.
Un aspecto concreto que hay que resaltar acerca de esta última versión es que utilizamos un bucle while, pero no necesitamos preocuparnos de la variable index. Esto se debe a que Iterator controla el punto en el que nos encontramos dentro de la colección, de modo que sabe si quedan más elementos (hasNext) y qué elemento devolver (next) si es que todavía quedan.

Una de las claves para comprender cómo funciona Iterator es que la llamada a next hace que el objeto Iterator devuelva el siguiente elemento de la colección y luego avance más allá de ese elemento. Por tanto, las llamadas sucesivas a next en un Iterator siempre devolverán elementos diferentes; no se puede volver al elemento anterior después de haber invocado next. En algún momento, el Iterator alcanzará el final de la colección y devolverá false al hacerse una llamada a hasNext. Una vez que hasNext ha devuelto false, sería un error tratar de invocar next sobre ese objeto Iterator concreto; de

hecho, el objeto Iterator habrá sido "agotado" y ya no tendrá ninguna utilidad.

Eliminación de elemnetos:
Si intentamos modificar la colección utilizando uno de los métodos remove de la misma mientras estamos en mitad de una iteración, el sistema nos dará un error (denominado ConcurrentModificationException). Sucede así porque cambiar la colección en mitad de una iteración tiene el potencial de confundir la situación enormemente. por lo que simplemente se prohíbe la utilización del método remove de la colección durante una iteración del bucle for-each.
La solución apropiada para efectuar la eliminación mientras estamos iterando consiste en utilizar un Iterator. Su tercer método (además de hasNext y next) es remove. No admite ningún parámetro y tiene un tipo de retorno void. Invocar remove hará que sea eliminado el elemento devuelto por la llamada más reciente a next.
Iterator $< Track >$ it = tracks.iterator();
while(it.hasNext()){
Track t = it.next();
String artist = t.getArtist();
if (artist.equals(artistToRemove)) {
it.remove();
}
}
De nuevo, observe que no usamos la variable de colección tracks en el cuerpo del bucle. Aunque tanto ArrayList como Iterator tienen métodos remove, debemos utilizar el método remove de Iterator, no el de ArrayList. Utilizar el método remove de Iterator es menos flexible: no podemos eliminar elementos arbitrarios, sino solo el último elemento extraído por el método next de Iterator. Por otro lado, sí que se permite utilizar el método remove de Iterator durante una iteración. Dado que el propio Iterator está informado de la eliminación (y se encarga de llevarla a cabo por nosotros), puede mantener apropiadamente la iteración sincronizada con la colección. Dicha eliminación no es posible con el bucle for-each, porque no disponemos ahí de un Iterator con el que trabajar.
En este caso, necesitamos utilizar el **bucle while con un Iterator.**
Técnicamente, también podemos eliminar elementos usando el método get de la colección con un índice para la iteración. Sin embargo, no le recomendamos que haga esto, porque los índices de los elementos pueden cambiar cuando añadimos o quitamos elementos y es muy fácil conseguir que la iteración opere con índices incorrectos, cuando modificamos la colección durante la iteración. El uso de un Iterator nos protege frente a tales errores.

(hemos visto cómo podemos utilizar un objeto ArrayList, creado a partir de una clase de la librería de clases, para almacenar un número arbitrario de objetos dentro de una colección. No tenemos que decidir de antemano cuántos objetos vamos a almacenar y el objeto ArrayList lleva automáticamente a la cuenta del número de elementos que almacena.)

Un principio muy importante es que, si una variable contiene el valor null, no se debe realizar ninguna llamada a método con ella. La razón debería estar clara: como los métodos pertenecen a objetos, no podemos invocar un método si la variable no hace referencia a un objeto. Esto quiere decir que en ocasiones tenemos que usar una instrucción if para comprobar si una variable contiene null o no antes de invocar un método sobre dicha variable. Si no se hace esta comprobación, se obtendrá el error de tiempo de ejecución NullPointerException, que es muy común.

objetos anónimos
El método enterLot de Auction ilustra un concepto bastante común: los objetos anónimos. Podemos ver esto en la siguiente instrucción:
lots.add(new Lot(nextLotNumber, description));
Aquí estamos haciendo dos cosas:
-Estamos creando un nuevo objeto Lot.

-También estamos pasando este nuevo objeto al método add de ArrayList.
lo que hacemos es crear un objeto anónimo, un objeto sin nombre, y pasárselo directamente al método que va a utilizarlo.

Utilización de colecciones:
La clase de colección ArrayList (y otras como ella) constituye una herramienta de programación importante, porque muchos problemas de programación implican trabajar con colecciones de objetos de tamaño variable.

5 comportamientos más sofisticados

Documentación para clases de librería: La librería Java estándar es enorme. Está formada por miles de clases, cada una de las cuales tiene muchos métodos, que a su vez pueden tener o no parámetros, contener o no tipos de retorno. Es imposible memorizar todos los métodos y los detalles correspondientes a cada uno. En lugar de ello, lo que un buen programador Java debe hacer es:
-Conocer por su nombre algunas de las clases más importantes y sus métodos (ArrayList es una de esas clases importantes).
-Saber localizar información acerca de esas clases y buscar los correspondientes detalles (como, por ejemplo, métodos y parámetros).
startWith es un método dentro de la clase String, lee si el estring empieza con tales caracteres que introduzcas. La clase String es una de las clases de la librería estándar de clases Java. Podemos conocer más detalles acerca de la misma leyendo la documentación de librería para la clase String. Para ello, seleccione el elemento Java Class Libraries del menú Help de BlueJ. Se abrirá un explorador web mostrando la página principal de la documentación de la API (Application Programming Interface, Interfaz de programación de aplicaciones) de Java
Verá que la documentación incluye diferentes elementos de información. Entre otros, se incluyen los siguientes:
-El nombre de la clase.
-Una descripción general del propósito de la clase.
-Una lista de los constructores y métodos de la clase.
-Los parámetros y los tipos de retorno para cada constructor y método.
-Una descripción del propósito de cada constructor y método.
Esta información, tomada conjuntamente, se denomina interfaz de una clase. Observe que la interfaz no muestra el código fuente que implementa la clase. Si una clase está bien descrita (es decir, si su interfaz está bien escrita), entonces el programador no necesita ver el código fuente para ser capaz de utilizar la clase. Con ver la interfaz, tenemos toda la información necesaria. Esto es de nuevo un ejemplo del concepto de abstracción.

El código fuente subyacente, que es el que hace que la clase funcione, se conoce como implementación de la clase. Normalmente, un programador trabaja en la implementación de una clase a la vez que hace uso de otras diversas clases a través de sus interfaces. Esta distinción entre la interfaz y la implementación es un concepto muy importante, que volverá a aparecer una y otra en este capítulo y en capítulos posteriores.
La interfaz de un método consta de la signatura del método y de un comentario (mostrado aquí en cursiva). La signatura de un método incluye (en este orden):
-Un modificador de acceso (que aquí es public), del que hablaremos más adelante.
-El tipo de retorno del método (en este caso int).
-El nombre del método.
-Una lista de parámetros (que en este ejemplo está vacía); el nombre y los parámetros reciben también conjuntamente el nombre de signatura del método.
La interfaz proporciona todo lo que necesitamos conocer para hacer uso de este método. La documentación de la clase String nos dice que dispone de un método denominado trim para eliminar espacios al principio y al final de la cadena de caracteres.

Un detalle importante acerca de los objetos String es que son inmutables; es decir, no pueden modificarse después de haberlos creado. Fíjese especialmente en que el método trim, por ejemplo, devuelve una nueva cadena de caracteres, no modifica la cadena original. Preste especial atención al siguiente comentario de "Error común".
Error común Es un error común en Java tratar de modificar una cadena. Por ejemplo, escribiendo
input.toUpperCase();
Esto es incorrecto (las cadenas de caracteres no pueden modificarse), aunque lamentablemente no produce ningún error. La instrucción simplemente no tienen ningún efecto, y la cadena de entrada no será modificada. El método toUpperCase, así como otros métodos de cadena, no modifica la cadena original, sino que devuelve una nueva cadena que es similar a la original, pero con algunos cambios aplicados (en este caso, los caracteres se han pasado a mayúscula). Si queremos modificar nuestra variable de entrada, entonces tenemos que asignar otra vez este nuevo objeto a la variable (descartando la original), como en el siguiente ejemplo:
input = input.toUpperCase();
El nuevo objeto también podría asignarse a otra variable o procesarse de alguna otra manera.

Después de estudiar la interfaz del método trim, podemos ver que se pueden eliminar los espacios de una cadena de entrada con la siguiente línea de código:
input = input.trim();
Este código solicitará al objeto String almacenado en la variable input que cree una nueva cadena, similar a la anterior, pero sin los espacios iniciales y finales. El nuevo objeto String se almacena entonces en la variable input, porque no tenemos ningún uso adicional que dar a la cadena de caracteres anterior. Por tanto, después de esta línea de código, input hace referencia a una cadena que no tiene espacios ni al principio ni al final.
Comporbación de la Igualdad entre cadenas El operador de igualdad (==) comprueba si cada lado del operador hace referencia al mismo objeto, no si tienen el mismo valor. Son dos cosas completamente distintas.
La solución es utilizar el método **equals**, definido en la clase String. Este método comprueba correctamente si el contenido de dos objetos String coincide.

Adición de comportamiento aleatorio
Aleatorio y seudoaleatorio La generación de números aleatorios en una computadora no es tan fácil de realizar, de hecho, como inicialmente podría pensarse. Dado que las computadoras operan de una forma bien definida y determinista, que descansa en el hecho de que todos los cálculos son predecibles y repetibles, proporcionan poco espacio para un comportamiento realmente aleatorio. Los investigadores han propuesto, a lo largo del tiempo, muchos algoritmos para generar secuencias de números aparentemente aleatorias. Estos números normalmente no son realmente aleatorios, sino que se generan de acuerdo con una serie muy complicada de reglas. Por ello se los denomina números seudoaleatorios. En un lenguaje como Java, la generación de números seudoaleatorios está implementada, afortunadamente, en una clase de librería, por lo que lo único que tenemos que hacer para obtener un número seudoaleatorio es hacer algunas llamadas a la librería.
La clase Random:
Para generar un número aleatorio, tenemos que:
-Crear una instancia de la clase Random.
-Hacer una llamada a un método de dicha instancia para obtener un número.
Al examinar la documentación, vemos que hay varios métodos denominados nextAlgo para generar valores aleatorios de distintos tipos. El que genera un número aleatorio entero se denomina nextInt.
El siguiente fragmento ilustra el código necesario para generar e imprimir un número aleatorio entero:
Random randomGenerator;
randomGenerator = new Random();
int index = randomGenerator.nextInt();
System.out.println(index);
Este fragmento de código crea una nueva instancia de la clase Random y la almacena en la variable

randomGenerator. A continuación, llama al método nextInt para recibir un número aleatorio, lo almacena en la variable index y al final lo imprime. Su clase solo debe crear un instancia de la clase Random (en su constructor) y almacenarla en un campo. No cree una nueva instancia de Random cada vez que desee generar un nuevo número.

Números aleatorios con rango limitado:
Los números aleatorios que hemos visto hasta ahora se generaban a partir del rango completo de enteros Java (–2147483648 a 2147483647). Eso está bien para un experimento, pero rara vez resulta útil. Más frecuentemente, lo que querremos es obtener números aleatorios dentro de un rango limitado específico.
La clase Random también ofrece un método para satisfacer esta necesidad. Se llama nextInt, pero tiene un parámetro para especificar el rango de números que nos gustaría usar.
Al utilizar un método que genere números aleatorios a partir de un rango especificado, hay que tener cuidado de comprobar si los límites son inclusivos o exclusivos. El método nextInt (int n) en la clase Random de la librería Java, por ejemplo, especifica que genera un número comprendido entre 0 (inclusive) y n (exclusive). Esto significa que el valor 0 está incluido en los posibles resultados, mientras que el valor especificado para n no lo está. El número más alto que puede devolver una de esas llamadas es n–1.

Generación de respuestas aleatorias:
-Declarar un campo de tipo Random para almacenar el generador de números aleatorios.
-Declarar un campo de tipo ArrayList para almacenar nuestras posibles respuestas.
-Crear los objetos Random y ArrayList en el constructor de Responder.
-Rellenar la lista de respuestas con algunas frases.
-Seleccionar y devolver una frase aleatoria cuando se invoque generateResponse.

```
public String generateResponse(){
int index = randomGenerator.nextInt(responses.size());
return responses.get(index);
}
```

La primera línea del código en este método hace tres cosas:
-Calcula el tamaño de la lista de respuestas llamando a su método size.
-Calcula el tamaño de la lista de respuestas llamando a su método size.
-Almacena ese número aleatorio en la variable local index.
Es importante observar que este segmento de código generará un número aleatorio en el rango de 0 a listSize–1 (inclusive). Esto encaja perfectamente con los índices legales para un ArrayList. Recuerde que el rango de índices para un ArrayList de tamaño listSize va de 0 a listSize–1. Por tanto, el número aleatorio calculado nos da un índice perfecto para acceder aleatoriamente a uno de los elementos de la lista completa.
La última línea del método:
-Extrae la respuesta situada en la posición index utilizando el método get.
-Devuelve la cadena seleccionada como resultado del método, con la instrucción return.
Si no tiene cuidado, su código podría generar un número aleatorio que quede fuera del rango de índices válidos del objeto ArrayList. Cuando luego intente utilizarlo como índice para acceder a un elemento de la lista, obtendrá un error IndexOutOfBoundsException.

Lactura de la documentación de las clases parametrizadas
Hasta ahora, le hemos pedido que examine la documentación de la clase String del paquete java.lang y de la clase Random del paquete java.util. Puede que haya observado al hacer esto que algunos nombres de clases en la lista contenida en la documentación tienen un aspecto ligeramente distinto, como por ejemplo ArrayList $< E >$ o HashMap$< K, V >$. Es decir, el nombre de la clase va seguido por una cierta información adicional que aparece entre corchetes angulares. Las clases de este estilo se denominan clases parametrizadas o clases genéricas.
La información encerrada en los corchetes angulares nos dice que al utilizar estas clases debemos sum-

inistrar uno o más nombres de tipo entre corchetes angulares para completar la definición.
Ya hemos visto aplicada esta idea en el Capítulo 4, donde hemos utilizado ArrayList parametrizándola con nombres de tipo como String. También pueden parametrizarse con cualquier otro tipo:
private ArrayList< *String* > notes;
private ArrayList< *Student* > students;
Dado que podemos parametrizar un ArrayList con cualquier otro tipo de clase que elijamos, este hecho se refleja en la documentación de la API. Por tanto, si examinamos la lista de métodos de ArrayList< E >, podremos ver métodos como:
boolean add(E o)
E get(int index)
Esto nos dice que el tipo de objetos que podemos añadir a un ArrayList (con add) depende del tipo utilizado para parametrizarla y que el tipo de los objetos devueltos por su método get depende de la misma manera de ese tipo empleado en la parametrización. De hecho, si creamos un objeto ArrayList< *String* >, lo que la documentación nos dice es que el objeto tiene los siguientes dos métodos:
boolean add(String o)
String get(int index)
mientras que si creamos un objeto ArrayLis*t* < *Student* >, entonces tendrá los otros dos métodos:
boolean add(Student o)
Student get(int index)

5.1 Paquetes de imporacion

import java.util.ArrayList;
import java.util.Random;
Las clases Java que están almacenadas en la librería de clases no están disponibles automáticamente para ser utilizadas. En lugar de ello, debemos indicar en nuestro código fuente que nos gustaría utilizar una clase de la librería. Esto se denomina importar la clase y se hace mediante la instrucción import. La instrucción import tiene el formato:
import nombre-clase-cualificado;
Dado que la librería Java contiene varios miles de clases, hace falta una cierta estructura en la organización de la librería para facilitar el manejo de ese gran número de clases. Java utiliza paquetes para clasificar las clases de librería en grupos de clases relacionadas. Los paquetes están anidados (es decir, los paquetes pueden contener otros paquetes).

Las clases ArrayList y Random se encuentran ambas en el paquete java.util. Esta información puede encontrarse en la documentación de la clase. El nombre completo o nombre cualificado de una clase es el nombre de su paquete, seguido por un punto y por el nombre de la clase. Por tanto, los nombres cualificados de las dos clases que hemos usado aquí son: java.util. ArrayList y java.util.Random.
Java también nos permite importar paquetes completos con instrucciones de la forma
import nombre-paquete.*;
Por tanto, la siguiente instrucción importaría todos los nombres de clase del paquete java.util:
import java.util.*;
Enumerar por separado todas las clases utilizadas, como en nuestra primera versión, requiere algo más de trabajo de escritura, pero resulta adecuado desde el punto de vista de la documentación. Indica claramente qué clases están siendo utilizadas realmente por nuestra clase. Por tanto, en este libro tenderemos a utilizar el estilo del primer ejemplo, enumerando por separado todas las clases importadas.
Existe una excepción a estas reglas: algunas clases se emplean tan frecuentemente que casi todas las demás clases tendrán que importarlas. Estas clases se han incluido en el paquete java.lang, y este paquete se importa de manera automática en todas las clases. Por tanto, no necesitamos escribir instrucciones de importación para las clases contenidas en java.lang. La clase String es un ejemplo de ese tipo de clases.

5.2 Utilización de mapas para asociaciones

HashMap es una especialización de Map, que también está documentada. Se encontrará con que tiene que leer la documentación de ambas clases para comprender lo que es un HashMap y cómo funciona. Un mapa es una colección de parejas clave/valor de objetos. Como con ArrayList, un mapa puede almacenar un tipo flexible de entradas. Una diferencia entre ArrayList y Map es que con Map cada entrada no es un objeto, sino una pareja de objetos. Esta pareja está formada por un objeto clave y un objeto valor. En lugar de buscar entradas en esta colección utilizando un índice entero (como hicimos con ArrayList), empleamos el objeto clave para buscar el objeto valor.
Un ejemplo de mapa en la vida cotidiana sería una guía telefónica. La guía telefónica contiene entradas y cada entrada es una pareja: un nombre y un número de teléfono. Utilizamos la guía telefónica buscando un nombre y leyendo el número de teléfono asociado. No empleamos ningún índice (la posición de la entrada en la guía) para averiguar el número de teléfono. Un mapa se puede organizar de tal manera que sea fácil buscar el valor correspondiente a una clave. En el caso de la guía telefónica, esto se hace mediante la ordenación alfabética. Si se almacenan las entradas en orden alfabético de sus claves, resulta sencillo localizar la clave y consultar el valor asociado. La búsqueda inversa (localizar la clave correspondiente a un valor, es decir, encontrar el nombre para un número de teléfono dado) no es tan simple con ayuda de un mapa. Al igual que con una guía telefónica, es posible realizar una búsqueda inversa en un mapa, pero se necesita un tiempo relativamente largo. Por tanto, los mapas son ideales para búsquedas en una sola dirección, en las que conocemos la clave de búsqueda y necesitamos saber el valor asociado con esa clave.

HashMap es una implementación específica de Map. Los métodos más importantes de la clase HashMap son put y get. El método put inserta una entrada en el mapa, mientras que get extrae el valor correspondiente a una clave especificada. El siguiente fragmento de código crea un HashMap y inserta en él tres entradas. Cada entrada es una pareja clave/valor compuesta por un nombre y un número de teléfono.
HashMap< $String, String$ > contacts = new HashMap<>();
contacts.put("Charles Nguyen", "(531) 9392 4587");
contacts.put("Lisa Jones", "(402) 4536 4674");
contacts.put("William H. Smith", "(998) 5488 0123");
Como hemos visto con ArrayList, al declarar una variable HashMap y al crear un objeto Hash- Map, tenemos que indicar qué tipo de objetos se almacenarán en el mapa y, adicionalmente, qué tipo de objetos se emplearán como clave. Para la guía de teléfonos utilizaríamos cadenas de caracteres tanto para las claves como para los valores, pero en otros casos ambos tipos serán diferentes.
cuando se crean objetos de clases genéricas y se les asigna una variable, es preciso especificar los tipos genéricos en este caso < $String, String$ >) solo una vez en el lado izquierdo de la asignación, y puede utilizarse el operador diamante en la construcción de objeto de la derecha; los tipos genéricos utilizados para la construcción de objeto se copian entonces de la declaración de variable.
El siguiente código encuentra el número de teléfono de Lisa Jones y lo imprime.
String number = phoneBook.get("Lisa Jones");
System.out.println(number);
Observe que pasamos la clave (el nombre "Lisa Jones") al método get para recibir el valor (el número de teléfono).

5.3 Utilización de conjuntos

La librería estándar Java incluye diferentes variantes de conjuntos implementados en clases distintas. La clase que vamos a utilizar aquí se denomina HashSet.
Los dos tipos de funcionalidad que necesitamos son la capacidad de introducir elementos en el conjunto y de extraer esos elementos posteriormente. Afortunadamente, estas tareas apenas contienen nada que nos resulte nuevo.
import java.util.HashSet;
HashSet< $String$ > mySet = new HashSet<>();

mySet.add("one");
mySet.add("two");
mySet.add("three");
Veamos cómo se iteraría a través de los elementos:
for(String item : mySet){
hacer algo con ese elemento }
En resumen, la utilización de colecciones en Java es bastante similar para los distintos tipos de colección. Una vez que se comprende cómo utilizar una de ellas, se pueden usar todas. Las diferencias radican, realmente, en el comportamiento de cada colección. Por ejemplo, una lista mantendrá todos los elementos que se introduzcan en el orden deseado, proporcionará acceso a esos elementos mediante un índice y puede contener el mismo elemento varias veces. Un conjunto, por el contrario, no mantiene ningún orden específico (los elementos pueden ser devueltos en un bucle for-each en un orden distinto de aquel en el que fueron introducidos) y garantiza que cada elemento se introduzca en el conjunto como máximo una vez. Introducir un elemento una segunda vez simplemente no tiene ningún efecto.

5.4 División de cadenas de caracteres

el método split, que es un método estándar de la clase String. El método split puede dividir una cadena en una serie de subcadenas separadas y devolverlas en una matriz de cadenas. (Se hablará más en detalle de matrices en el capítulo siguiente). El parámetro del método split define cuál es el tipo de caracteres según los cuales hay que dividir la cadena original. Lo que hemos hecho es definir que queremos cortar nuestra cadena por cada carácter de espaciado:
String inputline = reader.nextLine().trim().toLowerCase();
String wordArray = inputline.split(" ");

5.5 Autoboxing y clases envolventes

con una parametrización adecuada, las clases de colección pueden almacenar objetos de cualquier tipo. Sin embargo, persiste un problema: Java maneja algunos tipos que no son tipos de objeto.
Como sabemos, los tipos sencillos (como int, boolean y char) son diferentes de los tipos de objeto. Sus valores no son instancias de clases, y normalmente no sería posible añadirlos en una colección.
Esto supone un problema. Se dan situaciones en las que, por ejemplo, podríamos querer crear una lista de valores int o un conjunto de valores char. La solución Java a este problema proviene de las clases envolventes (wrappers). Cada tipo primitivo de Java tiene una clase envolvente correspondiente que representa el mismo tipo, pero que es un tipo de objeto real. Por ejemplo, la clase envolvente para int recibe el nombre de Integer. Las instrucciones siguientes envuelven explícitamente el valor de la variable primitiva int ix en un objeto Integer:
Integer iwrap = new Integer(ix);

Así, por ejemplo, iwrap podría almacenarse fácilmente en una colección ArrayList< *Integer* >. Sin embargo, el almacenamiento de valores primitivos en una colección de objetos se facilita aún más mediante una característica del compilador denominada autoboxing.
Siempre que se utilice un valor de un tipo de primitiva en un contexto que necesite una clase envolvente, el compilador envuelve automáticamente el valor de tipo primitivo como un objeto envolvente adecuado. Esto significa que los valores de tipo primitivo pueden añadirse directamente a una colección:
private ArrayList< *Integer* > markList;
public void storeMarkInList(int mark) {
markList.add(mark);
}
La operación inversa (unboxing) se realiza también de forma automática, con lo cual la recuperación desde una colección tendría un aspecto como el siguiente:
int firstMark = markList.remove(0);
El autoboxing se aplica también siempre que se transfiera un valor de tipo primitivo como un parámetro

a un método que expera un tipo envolvente, y cuando se almacena un valor de tipo primitivo en una variable de tipo envolvente. De forma semejante, el unboxing se aplica cuando se pasa un valor de tipo envolvente como parámetro a un método que espera un valor de tipo primitivo, y cuando se almacena en una variable de tipo primitivo. Conviene señalar que el resultado es casi equivalente a almacenar los tipos primitivos en colecciones. Sin embargo, el tipo de la colección aún debe declararse como un tipo envolvente (p. ej., ArrayList< $Integer$ >, no ArrayList< int >).

5.6 Escritura de la documentación de las clases

Cuando trabaje con sus propios proyectos, es importante que escriba la documentación de sus clases a medida que desarrolle el código fuente. Los sistemas Java incluyen una herramientas denominada javadoc que puede utilizarse para generar la descripción de esas interfaces a partir del código fuente. La documentación de la librería estándar que hemos utilizado, por ejemplo, fue creada a partir del código fuente de las clases mediante javadoc.
El entorno BlueJ utiliza javadoc para permitirnos crear documentación para nuestras clases de dos formas distintas para: -Ver la documentación para una única clase pasando el selector emergente situado en la parte superior derecha de la ventana del editor de Source Code a Documentation, o seleccionando Toggle Documentation View (Cambiar a vista de documentación) en el menú Tools (Herramientas) del editor.
-Usar la función Project Documentation disponible en el menú Tools de la ventana principal para generar la documentación correspondiente a todas las clases del proyecto.

Elementos de la documentación de una clase
La documentación de una clase debe incluir al menos:
-El nombre de la clase.
-Un comentario que describa el propósito global y las características de la clase.
-Un número de versión.
-El nombre del autor (o autores).
-La documentación para cada constructor y cada método.
La documentación de cada constructor y cada método debe incluir:
-El nombre del método. -El tipo de retorno.
- Los nombres y tipos de los parámetros
-Una descripción del propósito y función del método.
- Una descripción de cada parámetro.
-Una descripción del valor devuelto.
Además, cada proyecto completo ha de tener un comentario global del proyecto, que a menudo estará contenido en un archivo "ReadMe" (Léame). En BlueJ, se puede acceder a este comentario a través de la nota de texto mostrada en la esquina superior izquierda del diagrama de clases.
El símbolo de inicio del comentario tiene que tener dos asteriscos para que sea reconocido como un comentario javadoc. Dicho comentario, si precede inmediatamente a la declaración de la clase, se lee como un comentario de la clase. Si el comentario está justo encima de la signatura de un método se considera un comentario del método.
Los detalles exactos de cómo se produce y formatea la documentación difieren en los distintos lenguajes y entornos de programación. Sin embargo, el contenido debe ser más o menos siempre el mismo.
En Java, con javadoc hay disponibles varios símbolos clave especiales para dar formato a la documentación. Estos símbolos clave comienzan con el símbolo @ e incluyen: @version @author @param @return

5.7 Public y private

Los modificadores de acceso son las palabras clave public o private situadas al principio de las declaraciones de campo y de las signaturas de método. Los campos, los constructores y los métodos pueden ser públicos o privados, aunque hasta ahora hemos visto campos privados y constructores y métodos públicos.

Los modificadores de acceso definen la visibilidad de un campo, un constructor o un método. Por ejemplo, si un método es público, se puede invocar desde la misma clase o desde cualquier otra. Por el contrario, los métodos privados solo pueden invocarse desde la misma clase en la que están declarados. No son visibles para otras clases.
Recuerde: la interfaz de una clase es el conjunto de detalles que necesita ver cualquier otro programador que la use. Proporciona información acerca de cómo utilizar la clase. La interfaz incluye las signaturas de los constructores y los métodos, además de una serie de comentarios. También se le denomina parte pública de una clase. Su propósito es definir lo que la clase hace.
La implementación es la sección de una clase que define precisamente cómo funciona la clase. Los cuerpos de los métodos, que contienen las instrucciones Java y la mayoría de los campos son parte de la implementación. La implementación también se denomina parte privada de una clase. El usuario de una clase no necesita conocer su implementación. De hecho, hay buenas razones por las que a un usuario debería impedírsele conocer la implementación (o al menos hacer uso de dicho conocimiento). Este principio se conoce como ocultamiento de la información.
La palabra clave public declara un elemento de una clase (un campo o un método) como parte de la interfaz (es decir, que es públicamente visible); la palabra clave private declara que es parte de la implementación (es decir, que está oculto frente a posibles accesos externos).
Muy a menudo, el programador de mantenimiento debe modificar o ampliar la implementación de una clase, con el fin de realizar mejoras o de corregir errores. Idealmente, cambiar la implementación de una clase no debería hacer necesario que se modifiquen otras clases. Esta cuestión se conoce también con el nombre de acoplamiento. Si los cambios en una parte de un programa no hacen necesario realizar cambios en otra parte de un programa, decimos que existe un acoplamiento bajo o débil. El acoplamiento débil es positivo porque facilita notablemente el trabajo de un programador de mantenimiento. En vez de tener que entender y modificar muchas clases, tal vez solo necesite entender y modificar una única clase. Por ejemplo, si un programador de sistemas Java aporta una mejora en la implementación de la clase ArrayList, confiamos en que no nos obligue a modificar aquellas partes de nuestro código donde se use dicha clase. En principio, debería ser así, porque no hemos hecho ninguna referencia a la implementación de ArrayList dentro de nuestro propio código.
Por tanto, para ser más precisos, la regla de que a un usuario no debería permitírsele conocer las interioridades de una clase no se refiere al programador de otra clase, sino a la propia clase. Normalmente, no es ningún problema que un programador conozca los detalles de implementación, pero las clases que ese programador desarrolle no deberían "conocer" (no deberían ser dependientes de) los detalles internos de otra clase. El programador de ambas clases puede ser incluso la misma persona, pero las clases deben estar débilmente acopladas.

Otra buena razón para tener un método privado es cuando hace falta una tarea (como subtarea) en varios de los métodos de una clase. En lugar de escribir el código múltiples veces, podemos escribirlo una sola vez como un único método privado y luego invocarlo desde varios lugares distintos.
En Java, los campos también pueden declararse como privados o públicos. Hasta ahora, no hemos visto ejemplos de campos públicos y hay una buena razón para ello. Declarar campos como públicos viola el principio de ocultamiento de la información. Hace que una clase que dependa de dicha información sea vulnerable a los fallos de operación, en caso de que la implementación cambie. Aun cuando el lenguaje Java nos permite declarar campos públicos, consideramos que este es un mal estilo de programación, así que no haremos uso de dicha opción.
Una razón adicional para mantener privados los campos es que otorga a los objetos un mayor grado de control sobre su propio estado. Si canalizamos el acceso a un campo privado a través de métodos selectores y mutadores, los objetos tendrán la posibilidad de garantizar que el campo no se configure nunca con un valor que sea incoherente con su estado global.

5.8 Palabra clabe static

La palabra clave static es la sintaxis de Java para definir variables de clase. Las variables de clase son campos que se almacenan en la propia clase, no en un objeto. En esto se diferencian fundamentalmente de las variables de instancia

Como resultado, habrá siempre una única copia de esta variable, independientemente del número de instancias que creemos. El código fuente de la clase puede acceder (leer y configurar) este tipo de variable exactamente igual que en una variable de instancia. A la variable de clase se puede acceder desde cualquiera de las instancias de la clase. Como resultado, todos los objetos de esa clase compartirán dicha variable.
Las variables de clase se emplean frecuentemente cuando tenemos un valor que debe ser siempre el mismo para todas las instancias de una clase. En lugar de almacenar una copia del mismo valor en cada objeto, lo que sería un desperdicio de espacio y podría ser difícil de coordinar, se utiliza un único valor compartido por todas las instancias. Java también soporta los métodos de clase (también conocidos como métodos estáticos), que son métodos que pertenecen a una clase.

Constantes
Un uso frecuente de la palabra clave static tiene lugar en la definición de constantes. Las constantes son similares a las variables, pero no pueden cambiar de valor durante la ejecución de una aplicación. En Java, las constantes se definen mediante la palabra clave final:
private final int SIZE = 10;
Aquí, hemos definido una constante denominada SIZE con el valor 10. Observe que las declaraciones de constantes tienen un aspecto similar a las declaraciones de campos, con dos diferencias:
-Incluyen la palabra clave final antes del nombre del tipo.
-Han de inicializarse con un valor en el momento de la declaración.
Si se pretende que un valor no cambie nunca, es una buena idea declararlo como final. Esto garantiza que no pueda ser modificado posteriormente de manera accidental. Cualquier intento de modificar un campo constante provocará un mensaje de error en tiempo de compilación. Las constantes se suelen escribir en mayúsculas
En la práctica es frecuente que las constantes se apliquen a todas las instancias de una clase. En este tipo de situación, lo que hacemos es declarar constantes de clase. Las constantes de clase son campos de clase constantes. Se declaran con una combinación de las palabras clave static y final.
private static final int SIZE = 10;

Métodos de clase o estáticos
Hasta ahora, todos los métodos que hemos visto han sido de tipo instancia: se invocan en una instancia de una clase. La distinción entre los métodos de clase y los de instancia reside en que los primeros pueden invocarse sin ninguna instancia: basta con tener la clase.
Para definir un método de clase se añade la palabra clave static delante del nombre de tipo en la cabecera del método:
public static int getNumberOfDaysThisMonth() { ... }
continuación puede llamarse al método especificando el nombre de la clase en la que se ha definido, delante del punto en la notación habitual. Si, por ejemplo, el método anterior se define en una clase denominada Calendar, se invoca mediante el código siguiente:
int days = Calendar.getNumberOfDaysThisMonth();
Como puede verse, el nombre de la clase se utiliza delante del punto; no se ha creado ningún objeto.
Dado que los métodos de clase están asociados con una clase y no con una instancia, presentan dos limitaciones importantes. La primera es que un método de clase tal vez no pueda acceder a todos los campos de instancia definidos en la clase. Este resultado es lógico, ya que los campos de instancia se asocian con objetos individuales. Así pues, los métodos de clase tienen restricciones para acceder a las variables de clase desde su clase. La segunda limitación se parece a la primera: un método de clase no puede invocar a un método de instancia desde la clase. Tan solo puede llamar a otros métodos de clase definidos en su clase.

5.9 El método principal

Si queremos iniciar una aplicación Java sin BlueJ, debemos usar un método de clase. En BlueJ, normalmente creamos un objeto e invocamos a uno de sus métodos, pero sin BlueJ, la aplicación empieza sin que exista ningún objeto. Las clases son las únicas entidades con las que contamos

inicialmente, por lo cual el primer método al que debemos invocar es un método de clase. La definición en Java para iniciar aplicaciones es bastante sencilla: el usuario especifica la clase que debe iniciarse, y el sistema Java invocará un método denominado main en esta clase. Este método debe tener una signatura muy determinada:
public static void main(String[] args)
Si en esa clase no existiera tal método se comunicaría un error.

6 Colecciones de tamaño fijo

El presente capítulo trata sobre colecciones que no son flexibles, sino que poseen una capacidad fija; en el punto en que se crea el objeto de colección, tenemos que especificar el número máximo de elementos que puede almacenar, que no podrá modificarse. A primera vista podría parecer una restricción innecesaria, pero en algunas aplicaciones conocemos con antelación el número exacto de elementos que deseamos almacenar en una colección, y ese número suele mantenerse fijo en todo el tiempo de vida de la colección. En tales circunstancias, tenemos la opción de elegir utilizar un objeto de colección de tamaño fijo especializado para almacenar los elementos.

Matrices: Una colección de tamaño fijo se denomina matriz. Aunque el tamaño fijo de las matrices puede ser una desventaja significativa en muchas situaciones, tienen a cambio dos ventajas frente a las clases de colección de tamaño flexible:
-El acceso a los elementos almacenados en una matriz suele ser más eficiente que el acceso a los elementos de una colección de tamaño flexible comparable.
-Las matrices pueden almacenar tanto objetos como valores de tipo primitivo. Las colecciones de tamaño flexible solo pueden almacenar objetos
Otra característica distintiva de las matrices es que tiene un soporte sintáctico especial en Java; se puede acceder a ellas con ayuda de una sintaxis personalizada que difiere de las llamadas tradicionales a métodos. La razón es principalmente histórica: las matrices son la estructura de colección más antigua utilizada en los lenguajes de programación, y la sintaxis para tratar con matrices se ha ido desarrollando a lo largo de muchas décadas. Java utiliza la misma sintaxis establecida en otros lenguajes de programación para hacer más sencillas las cosas para aquellos programadores que ya estén empleando matrices, aun cuando este tratamiento no sea coherente con el resto de la sintaxis del lenguaje.
La característica distintiva de la declaración de una variable de tipo matriz es una pareja de corchetes que forman parte del nombre del tipo: int[]. Esto indica que la variable hourCounts es del tipo matriz de enteros. Decimos en este caso que int es el tipo base de esta matriz concreta, lo que significa que el objeto matriz almacenará valores de tipo int
La forma general de construcción de un objeto matriz es: new tipo[expresión-entera] La elección de tipo especifica el tipo de elemento que se almacenará en la matriz. La expresiónentera especifica el tamaño de la matriz; es decir, el número fijo de elementos que puede almacenarse en ella.
Cuando se asigna un objeto matriz a una variable matriz, el tipo del objeto matriz debe corresponderse con el tipo declarado para la variable
A los elementos individuales de una matriz se accede indexando la matriz. Un índice es una expresión entera escrita entre corchetes y que se coloca después del nombre de una variable de matriz. Los valores válidos para una expresión de índice dependen de la longitud de la matriz con la que se esté trabajando. Al igual que sucede con otras colecciones, los índices de una matriz siempre comienzan en cero y van hasta una unidad menos que la longitud de la matriz.
La utilización de un índice de matriz en el lado izquierdo de una instrucción de asignación es el equivalente, dentro de las matrices, a un método mutador (o método set), porque se modificará el contenido de la matriz. La utilización de un índice de matriz en cualquier otro lugar es el equivalente de un método selector (o método get).

6.1 El blucle for

Java define dos variantes de bucles for, indicándose ambas mediante la palabra clave for en el código fuente. la primera de esas variantes, el bucle for-each, que es un método conveniente para iterar a

través de una colección de tamaño flexible. La segunda variante, el bucle for, es una estructura de control iterativo alternativa2, que es particularmente apropiada cuando:
-queremos ejecutar un cierto conjunto de instrucciones un número fijo de veces,
-necesitamos una variable dentro del bucle cuyo valor cambie en una cantidad fija, incrementándose normalmente en 1, en cada iteración.
El bucle for está bien adaptado a aquellas situaciones en las que se necesita una iteración definida. Por ejemplo, es común emplear un bucle for cuando queremos hacer algo con todos los elementos de una matriz, como imprimir el contenido de cada elemento. Esto encaja con el criterio, ya que el número fijo de veces se corresponde con la longitud de la matriz y nos hace falta una variable para proporcionar un índice incremental para la matriz.
Un bucle for tiene la siguiente forma general:
for(inicialización; condición; acción post-cuerpo) {
instrucciones que hay que repetir
}
Cuando comparamos este bucle for con el bucle for-each, observamos que la diferencia sintáctica se encuentra en la sección situada entre los paréntesis, en la cabecera del bucle. En este bucle for, los paréntesis contienen tres secciones independientes, separadas por caracteres de punto y coma.
Aun cuando el bucle for se emplea a menudo para iteración definida, el hecho de que esté controlado por una expresión booleana de carácter general indica que se aproxima más al bucle while que al bucle for-each.
En general, cuando queremos acceder a todos los elementos de una matriz, la cabecera del bucle for tendrá el siguiente formato general:
for(int index = 0; index < array.length; index++)

Existe un uso especial del bucle for con un Iterator cuando deseamos hacer algo así. Suponga que deseáramos eliminar de nuestro organizador de música todas las canciones de un artista concreto.podemos utilizar un bucle for de la manera siguiente:
for(Iterator< $Track$ > it = tracks.iterator(); it.hasNext();) {
Track t = it.next();
if(track.getArtist().equals(artist)) {
t.remove();
}
}
El aspecto importante aquí es que no hay ninguna acción post-cuerpo del bucle; nos hemos limitado a dejarla en blanco. Esto es perfectamente legal, pero seguimos teniendo que incluir el punto y coma después de la condición del bucle. Utilizando un bucle for en lugar de un bucle while, queda algo más claro que pretendemos examinar todos los elementos de la lista.

6.2 El operador condicional

if(condición) {
hacer algo;
} else {
hacer algo similar;
}
Este operador se utiliza para seleccionar uno de los dos valores alternativos, basándose en la evaluación de una expresión booleana y la forma general es:
condición ? valor1 : valor2
Si la condición es true, entonces el valor de la expresión completa será valor1, y en caso contrario será valor2.
Códigos de Wolfram. En 1983, Stephen Wolfram publicó un estudio con los 256 autómatas celulares elementales posibles. Propuso un sistema numérico para definir la conducta de cada tipo de autómata, y el código asignado a cada uno recibe el nombre de código de Wolfram. A partir del código numérico, es muy sencillo deducir la regla aplicable para los cambios de estado, dados los valores de una célula

y de sus dos vecinas, dado que el mismo código codifica las reglas. Véase el Ejercicio 7.36, mostrado más adelante, sobre el funcionamiento en la práctica, o busque en la web los términos "elementary cellular automata" (autómatas celulares elementales).

La sintaxis para declarar una variable de matriz de más de una dimensión es una extensión del caso unidimensional: un par de corchetes vacíos por cada dimensión:
Cell[][] cells;
De forma semejante, la creación del objeto matriz mediante new especifica la longitud de cada dimensión:
cells = new Cell[numRows][numCols];

7 Diseño de clases

Los malos diseños tienen más que ver con las decisiones que tomamos a la hora de resolver un problema concreto. No podemos utilizar como excusa para hacer un mal diseño el argumento de que no había otra manera de resolver el problema.

7.1 Acoplamiento y cohesión

Hay dos términos que son fundamentales a la hora de hablar de la calidad de un diseño de clase: el acoplamiento y la cohesión.
El término acoplamiento hace referencia al grado de interconexión de las clases. lo que buscamos es diseñar nuestra aplicación como un conjunto de clases en cooperación, que se comunican a través de interfaces bien definidas. El grado de acoplamiento indica lo estrechamente conectadas que están las clases. Lo que buscamos es un grado bajo de acoplamiento, o un acoplamiento débil.

El grado de acoplamiento determina lo difícil que es realizar cambios en una aplicación. En una estructura de clases estrechamente acoplada, un cambio en una clase puede hacer necesario introducir cambios también en otras clases. Esto es precisamente lo que tratamos de evitar, porque el efecto de realizar un pequeño cambio puede propagarse rápidamente en cascada a través de toda la aplicación. Además, localizar todos los lugares en los que es necesario hacer cambios y llevar a la práctica esos cambios puede resultar difícil y requerir mucho tiempo. En un sistema débilmente acoplado, por el contrario, podemos cambiar una clase sin efectuar ningún cambio en las clases restantes, y la aplicación seguirá funcionando correctamente.

El término cohesión se relaciona con el número y la diversidad de las tareas de las que es responsable cada unidad de una aplicación. La cohesión es relevante tanto para unidades formadas por una sola clase, como para métodos individuales
Idealmente, cada unidad de código debe ser responsable de una tarea coherente (es decir, una tarea que pueda ser vista como una unidad lógica). Cada método debería implementar una operación lógica, y cada clase debería representar un tipo de entidad. La principal razón que subyace al principio de la cohesión es la reutilización: si un método o clase es responsable de una única cosa bien definida, entonces es mucho más probable que pueda utilizarse de nuevo en un contexto distinto. Una ventaja complementaria de adherirse a este principio es que, cuando haga falta realizar modificaciones en algún aspecto de la aplicación, es probable que encontremos todas las piezas relevantes dentro de una misma unidad.

La duplicación de código es un indicador de un mal diseño. El problema con la duplicación de código es que cualquier modificación en una versión debe ser realizado también en la otra, si queremos evitar las incoherencias. Esto incrementa la cantidad de trabajo que el programador de mantenimiento tiene que realizar e introduce el peligro de que aparezcan errores.
Normalmente, la duplicación de código es un síntoma de una mala cohesión.

7.2 Encapsulación para reducir el acoplamiento

La directriz referida a la encapsulación (ocultar la información de la implementación a ojos de otras clases) sugiere que solo debe hacerse visible para el exterior la información acerca de lo que hace una clase, no la información acerca de cómo lo hace. Esto tiene una gran ventaja: si ninguna otra clase sabe cómo está almacenada nuestra información, entonces podemos cambiar fácilmente el modo en que está almacenada sin por ello hacer que otras clases dejen de funcionar.
Podemos obligar a esta separación entre lo que se hace y cómo se hace definiendo los campos como privados y utilizando un método selector para acceder a ellos.

7.3 Diseño dirigido por responsabilidad

Hemos visto en la sección anterior que el hacer uso de una encapsulación adecuada reduce el acoplamiento y puede disminuir significativamente la cantidad de trabajo necesario para llevar a cabo cambios en una aplicación. Sin embargo, la encapsulación no es el único factor que influye en el grado de acoplamiento. Otro aspecto es el conocido con el nombre de diseño dirigido por responsabilidad.
El diseño dirigido por responsabilidad expresa la idea de que cada clase debe ser responsable de gestionar sus propios datos. A menudo, cuando necesitamos añadir alguna nueva funcionalidad a una aplicación, tenemos que preguntarnos a nosotros mismos en qué clase deberíamos añadir un método para implementar esa nueva función. ¿Qué clase debería ser responsable de la tarea? La respuesta es que la clase responsable de almacenar unos determinados datos debería ser también responsable de manipularlos. Lo bien que se utilice el diseño dirigido por responsabilidad influye en el grado de acoplamiento y, por tanto, influye de nuevo en la facilidad con la que se puede modificar o ampliar una aplicación.
Otro aspecto de los principios de acoplamiento y responsabilidad es el de la localidad de los cambios. Lo que queremos es crear un diseño de clases que facilite los cambios posteriores, haciendo que los efectos de un cambio sean locales. Idealmente, solo deberíamos tener que cambiar una única clase para realizar una modificación. En ocasiones, será necesario modificar varias clases, pero entonces nuestro objetivo será que se trate del menor número de clases posible. Además, los cambios necesarios en otras clases deberán ser obvios, fáciles de detectar y sencillos de llevar a cabo.
En buena medida, podemos conseguir esto siguiendo unas buenas reglas de diseño, como por ejemplo emplear un diseño dirigido por responsabilidad y tratar de conseguir un acoplamiento débil y una alta cohesión. Además, sin embargo, deberíamos tener en mente las futuras modificaciones y ampliaciones en el momento de crear nuestras aplicaciones. Es importante prever que un cierto aspecto de nuestro programa puede cambiar de cara a hacer que dicho cambio sea lo más fácil posible.

Acomplamiento implícito: El acoplamiento implícito es una situación en la que una clase depende de la información interna de otra, pero dicha dependencia no es inmediatamente obvia. El acoplamiento fuerte en el caso de los campos públicos no era bueno, pero al menos era obvio: si cambiamos los campos públicos en una clase y nos olvidamos de la otra, la aplicación no podrá compilarse y el compilador nos indicará que hay un problema. Sin embargo, en los casos de acoplamiento implícito, la omisión de un cambio necesario puede no ser detectada.
Se trata de un problema fundamental, porque cada vez que añadimos un comando, tenemos que cambiar el texto de ayuda y es bastante sencillo olvidarse de hacer este cambio. El programa se compila y se ejecuta y todo parece ir correctamente. El programador de mantenimiento podría creer que su tarea ha finalizado y lanzar comercialmente un programa que ahora contendrá un error. Este es un ejemplo de acoplamiento implícito. Cuando los comandos cambian, el texto de ayuda debe modificarse (acoplamiento), pero no hay nada en el código fuente del programa que indique claramente esta dependencia (y es por ello que es implícita). Una clase bien diseñada evitaría esta forma de acoplamiento siguiendo la regla del diseño dirigido por responsabilidad.

principio de cohesion: cada unidad de código debería ser siempre responsable de una, y solo una, tarea. Cuando hablamos de la cohesión de métodos, queremos expresar que, idealmente, cada método

debe ser responsable de una, y solo una, tarea bien definida. El principio de cohesión se puede aplicar a clases y métodos: tanto unas como otros deben mostrar un alto grado de cohesión.
Sin embargo, es mucho más fácil entender lo que hace un segmento de código y es también más fácil realizar modificaciones en el mismo, si se utilizan métodos cortos y bien cohesionados. Con la estructura de métodos elegida, todos los métodos son relativamente cortos y fáciles de entender, y sus nombres indican sus propósito de forma bastante clara. Estas características representan una ayuda muy valiosa para el programador de mantenimiento.

Cohesión de clase: La regla de la cohesión de las clases afirma que cada clase debería representar una única entidad bien definida dentro del dominio del problema.
Son varias las maneras en que una alta cohesión beneficia a un diseño. Las dos más importantes son la legibilidad y la reutilización.

7.4 Refactorización

Al diseñar aplicaciones, debemos tratar de planificar por adelantado, anticipando los posibles cambios que puedan realizarse en el futuro y creando clases y métodos altamente cohesionados y débilmente acoplados que faciliten las modificaciones. Este es un objetivo muy loable, pero por supuesto no siempre podemos anticipar todas las futuras adaptaciones y tampoco es factible prepararse para todas las posibles ampliaciones que podamos imaginar. Esta es la razón de que la refactorización sea importante.

La refactorización es la actividad consistente en reestructurar las clases y método existentes, para adaptarlos a los cambios en la funcionalidad y en los requisitos. A menudo, a lo largo de la vida de una aplicación se suele ir añadiendo gradualmente funcionalidad. Un efecto común es que, como consecuencia directa, la longitud de los métodos y de las clases crece lentamente. Resulta tentador para un programador de mantenimiento añadir código adicional a las clases o método existentes. Sin embargo, si seguimos haciendo esto durante un cierto tiempo, el grado de cohesión se reducirá. Cuando se añade más y más código a un método o una clase, es probable que llegue un momento en el que ese método o esa clase representen más de una tarea o entidad claramente definida. La refactorización consiste en repensar y rediseñar las estructuras de clases y métodos. El efecto más común es que las clases se dividan en dos o que los métodos se dividan en dos o más métodos. La refactorización puede incluir también la unión de varias clases o métodos en uno, aunque esto suele ser bastante menos común que la división.

Antes de proporcionar un ejemplo de refactorización, tenemos que reflexionar sobre el hecho de que, al refactorizar un programa, estamos proponiendo normalmente realizar cambios potencialmente grandes en algo que ya funciona. Cuando se modifica algo, hay una cierta probabilidad de que se produzcan errores. Por tanto, es importante actuar con cautela y, antes de refactorizar, debemos asegurarnos de que exista un conjunto de pruebas para la versión actual del programa. Si esas pruebas no existen, entonces debemos decidir primero cómo se puede probar razonablemente la funcionalidad del programa, y dejar constancia de esas pruebas (por ejemplo, anotándolas por escrito) de modo que podamos repetir esas mismas pruebas posteriormente.

Idealmente, la refactorización debe realizarse en dos etapas:
La primera etapa consiste en refactorizar para mejorar la estructura interna del código, pero sin realizar ningún cambio en la funcionalidad de la aplicación. En otras palabras, el programa debería, al ejecutarse, comportarse de la misma forma exacta que antes. Una vez completada esta etapa, habrá que repetir las pruebas previamente decididas para verificar que no hemos introducido errores inadvertidamente.
La segunda etapa puede acometerse una una vez que hayamos restablecido la funcionalidad base en la versión refactorizada. Entonces estaremos en disposición de mejorar el programa. Una vez que lo hayamos hecho, por supuesto, habrá que realizar pruebas con la nueva versión.
Hacer varios cambios al mismo tiempo (refactorizar y añadir nuevas características) hace que sea más difícil localizar las fuentes de error en caso de que introduzcan errores.

8 Objetos don un buen comportamiento