

Módulo 4

Introducción a clases, objetos y métodos

HABILIDADES FUNDAMENTALES

- 4.1 Conozca los fundamentos de las clases
- 4.2 Comprenda cómo se crean los objetos
- 4.3 Comprenda cómo se asignan las variables de referencia
- 4.4 Cree métodos, valores devueltos y parámetros de uso
- 4.5 Use la palabra clave **return**
- 4.6 Regrese un valor de un método
- 4.7 Agregue parámetros a un método
- 4.8 Utilice constructores
- 4.9 Cree constructores con parámetros
- 4.10 Comprenda **new**
- 4.11 Comprenda la recolección de basura y los finalizadores
- 4.12 Use la palabra clave **this**

Antes de que proceda con su estudio de Java, necesita aprender acerca de las clases. Las clases son la esencia de Java pues representan los cimientos sobre los que todo el lenguaje Java está construido. Así pues, la clase define la naturaleza de un objeto. Como tal, la clase forma la base de la programación orientada a objetos en Java. Dentro de una clase están definidos los datos y el código que actúan sobre tales datos. El código está contenido en métodos. Debido a que en Java son fundamentales, en este módulo se presentarán clases, objetos y métodos. La comprensión básica de estas características le permitirá escribir programas más complejos y entender mejor ciertos elementos clave de Java que se describen en el siguiente módulo.

HABILIDAD
FUNDAMENTAL

4.1

Fundamentos de las clases

Desde el inicio de este libro hemos estado empleando clases debido a que toda la actividad en un programa de Java ocurre dentro de una clase. Por supuesto, sólo se han utilizado clases extremadamente simples y aún no se ha aprovechado la mayor parte de sus funciones. Como verá, las clases son sustancialmente más poderosas que las clases limitadas que se han presentado hasta el momento.

Empecemos por revisar los fundamentos: una clase es una plantilla que define la forma de un objeto y especifica los datos y el código que operarán sobre esos datos. Java usa una especificación de clase para construir *objetos*. Los objetos son *instancias* de una clase. Por lo tanto, una clase es, en esencia, un conjunto de planos que especifican cómo construir un objeto. Es importante que el siguiente tema quede claro: una clase es una abstracción lógica. No es sino hasta que se crea un objeto de esa clase que una representación física de dicha clase llega a existir en la memoria.

Otro tema: recuerde que a los métodos y las variables que constituyen una clase se les denomina *miembros* de la clase. Los miembros de datos también son conocidos como *variables de instancia*.

La forma general de una clase

Cuando define una clase, declara su forma y su naturaleza exactas, lo cual lleva a cabo al especificar las variables de instancia que contiene y los métodos que operan sobre estas variables. Aunque es posible que las clases muy simples contengan solamente métodos o variables de instancia, casi todas las clases reales contienen ambos.

Una clase se crea empleando la palabra clave **class**. A continuación se muestra la forma general de una definición de clase:

```
class nombreclase {  
    // declare variables de instancia  
    tipo var1;  
    tipo var2;  
    //...  
    tipo varN;  
  
    // declare métodos  
    tipo método1(parámetros) {
```

```
// cuerpo del método
}
tipo método2(parámetros) {
    // cuerpo del método
}
// ...
tipo métodoN(parámetros) {
    // cuerpo del método
}
}
```

Aunque no hay reglas sintácticas que así lo dicten, una clase bien diseñada debe definir una y sólo una entidad lógica. Por ejemplo, una clase que almacena nombres y números de teléfono no almacenará información acerca de la bolsa de valores, el promedio de precipitación pluvial, los ciclos de las manchas solares u otra información no relacionada. Lo importante aquí es que una clase bien diseñada agrupa información conectada de manera lógica. Si coloca información no relacionada en la misma clase, ¡desestructurará rápidamente su código!

Hasta este momento, las clases que se han estado empleando sólo tienen un método: **main()**. Pronto verá cómo crear otras clases. Sin embargo, tome en cuenta que la forma general de una clase no especifica un método **main()**. Sólo se requiere éste si la clase es el punto de partida de su programa. Además, los applets no requieren un **main()**.

Definición de una clase

Para ilustrar las clases, desarrollaremos una clase que encapsule información acerca de automotores como coches, camionetas y camiones. Esta clase será **Automotor** y almacenará tres elementos de información acerca de un vehículo: el número de pasajeros que puede tener, la capacidad del tanque de combustible y su consumo promedio de gasolina (en kilómetros por litro).

A continuación se presenta la primera versión de **Automotor**. Define tres variables de instancia: **pasajeros**, **tanquegas** y **kpl**. Observe que **Automotor** no contiene métodos, por lo tanto, se trata actualmente de una clase que sólo contiene datos. (En secciones posteriores se le agregarán métodos.)

```
class Automotor {
    int pasajeros;    // número de pasajeros
    int tanquegas;    // capacidad del tanque en litro
    int kpl;          // consumo de gasolina en km por litro
}
```

Una definición de **class** crea un nuevo tipo de datos. En este caso, al nuevo tipo de datos se le llama **Automotor**. Usará este nombre para declarar objetos de tipo **Automotor**. Recuerde que una definición de **class** es sólo una descripción de tipo, por lo que no crea un objeto real. Por consiguiente, el código anterior no hace que ningún objeto de **Automotor** cobre vida.

Para crear realmente un objeto de **Automotor**, debe usar una instrucción como la siguiente:

```
Automotor minivan = new Automotor(); // crea un objeto de Automotor llamado minivan
```

Después de que esta instrucción se ejecuta, **minivan** será una instancia de **Automotor** y, por lo tanto, tendrá una realidad “física”. Por el momento, no se preocupe de los detalles de esta instrucción.

Cada vez que cree la instancia de una clase, estará creando un objeto que contenga su propia copia de cada variable de instancia definida por la clase. En consecuencia, todos los objetos de **Automotor** contendrán sus propias copias de las variables de instancia **pasajeros**, **tanquegas** y **kpl**. Para acceder a estas variables, usará el operador de punto (.). El *operador de punto* vincula el nombre de un objeto con el de un miembro. Ésta es la forma general del operador de punto:

objeto.miembro

El objeto se especifica a la izquierda y el miembro a la derecha. Por ejemplo, para asignar a la variable **tanquegas** de **minivan** el valor 60, use la siguiente instrucción:

```
minivan.tanquegas = 60;
```

En general, puede usar el operador de punto para acceder a variables de instancia y métodos.

He aquí un programa completo que usa la clase **Automotor**:

```
/* Programa que usa la clase Automotor.

   Llame a este archivo AutomotorDemo.java
*/
class Automotor {
    int pasajeros;    // número de pasajeros
    int tanquegas;    // capacidad del tanque en litro
    int kpl;          // consumo de gasolina en km por litro
}

// Esta clase declara un objeto de tipo Automotor.
class AutomotorDemo {
    public static void main(String args[]) {
        Automotor minivan = new Automotor();
        int rango;

        // asigna valores a campos de minivan
        minivan.pasajeros = 7;
        minivan.tanquegas = 60; ← Observe el uso del operador de punto
        minivan.kpl = 6;      para acceder a un miembro.

        // calcula el rango suponiendo un tanque lleno de gas
        rango = minivan.tanquegas * minivan.kpl;
```

```

        System.out.println("Una minivan puede transportar " + minivan.pasajeros +
                           " pasajeros con un rango de " + rango);
    }
}

```

Debe llamar con el nombre de **AutomotorDemo.java** al archivo que contiene este programa porque el método **main()** está en la clase **AutomotorDemo**, no en la clase **Automotor**. Cuando compile este programa, encontrará que se han creado dos archivos **.class**, uno para **Automotor** y otro para **AutomotorDemo**. El compilador de Java coloca automáticamente cada clase en su propio archivo **.class**. No es necesario que ambas clases estén en el mismo archivo fuente. Puede poner cada clase en sus propios archivos, llamados **Automotor.java** y **AutomotorDemo.java**, respectivamente.

Para ejecutar este programa, debe ejecutar **AutomotorDemo.class**. Se despliega la siguiente salida:

```
Una minivan puede transportar 7 pasajeros con un rango de 360
```

Antes de seguir adelante, revisemos un principio fundamental: cada objeto tiene sus propias copias de las variables de instancia definidas para su clase. Por lo tanto, el contenido de las variables en un objeto puede diferir del contenido de las variables en otro. No existe una conexión entre los dos objetos excepto por el hecho de que ambos objetos son del mismo tipo. Por ejemplo, si tiene dos objetos **Automotor**, cada uno tiene su propia copia de **pasajeros**, **tanquegas** y **kpl**, y el contenido de éstos puede diferir entre los dos objetos. El siguiente programa lo demuestra. (Observe que la clase con **main()** ahora se llama **DosAutomotores**.)

```

// Este programa crea dos objetos de Automotor.

class Automotor {
    int pasajeros;    // número de pasajeros
    int tanquegas;    // capacidad del tanque en litros
    int kpl;          // consumo de gasolina en km por litro
}

// Esta clase declara un objeto de tipo Automotor.
class DosAutomotores {
    public static void main(String args[]) {
        Automotor minivan = new Automotor();
        Automotor carrodepor = new Automotor();

        int rango1, rango2;

        // asigna valores en campos de minivan
        minivan.pasajeros = 7;
        minivan.tanquegas = 60;
        minivan.kpl = 6;
    }
}

```

Recuerde que **minivan** y **carrodepor** hacen referencia a objetos separados

```
// asigna valores en campos de carrodepor
carrodepor.pasajeros = 2;
carrodepor.tanquegas = 50;
carrodepor.kpl = 4;

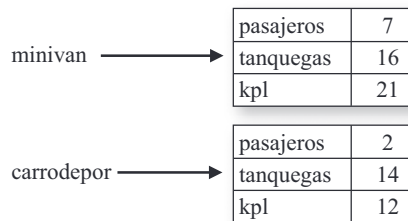
// calcula el rango suponiendo un tanque lleno de gas
rango1 = minivan.tanquegas * minivan.kpl;
rango2 = carrodepor.tanquegas * carrodepor.kpl;

System.out.println("Una minivan puede transportar " + minivan.pasajeros +
    " pasajeros con un rango de " + rango1);
System.out.println("Un carro deportivo puede transportar " + carrodepor.
pasajeros +
    " pasajeros con un rango de " + rango2);
}
}
```

Aquí se muestra la salida producida por este programa:

```
Una minivan puede transportar 7 pasajeros con un rango de 1260
Un carro deportivo puede transportar 2 pasajeros con un rango de 200
```

Como verá, los datos de **minivan** están completamente separados de los contenidos en **carrodepor**. La siguiente ilustración describe esta situación.



Comprobación de avance

1. ¿Cuáles son las dos cosas que una clase contiene?
2. ¿Cuál operador se usa para acceder a los miembros de una clase a través de un objeto?
3. Cada objeto tiene su propia copia de las _____ de la clase.

-
1. Código y datos. En Java, esto significa métodos y variables de instancia.
 2. El operador de punto.
 3. Variables de instancia.

HABILIDAD
FUNDAMENTAL

4.2

Cómo se crean los objetos

En los programas anteriores se usó la siguiente línea para declarar un objeto de tipo **Automotor**:

```
Automotor minivan = new Automotor();
```

Esta declaración tiene dos funciones. En primer lugar, declara una variable llamada **minivan** del tipo de clase **Automotor**. Esta variable no define un objeto, sino que simplemente es una variable que puede *hacer referencia* a un objeto. En segundo lugar, la declaración crea una copia del objeto y asigna a **minivan** una referencia a dicho objeto. Esto se hace al usar el operador **new**, el cual asigna de forma dinámica (es decir, en tiempo de ejecución) memoria para un objeto y devuelve una referencia a él. Esta referencia es, más o menos, la dirección en memoria del objeto asignado por **new**. Luego, esta referencia se almacena en una variable. Por lo tanto, en Java, todos los objetos de clase deben asignarse dinámicamente.

Los dos pasos combinados de la instrucción anterior pueden reescribirse de esta manera para mostrar cada paso individualmente.

```
Automotor minivan; // declara referencia a un objeto
Minivan = new Automotor(); // asigna un objeto de Automotor
```

La primera línea declara **minivan** como referencia a un objeto de tipo **Automotor**. Por lo tanto, **minivan** es una variable que puede hacer referencia a un objeto, pero no es un objeto en sí. En este sentido, **minivan** contiene el valor **null**, que significa que no alude a un objeto. La siguiente línea crea un nuevo objeto de **Automotor** y asigna a **minivan** una referencia a él. Ahora, **minivan** está vinculado con un objeto.

HABILIDAD
FUNDAMENTAL

4.3

Variables de referencia y asignación

En una operación de asignación, las variables de referencia a objetos actúan de manera diferente a las variables de un tipo primitivo, como **int**. Cuando asigna una variable de tipo primitivo a otra, la situación es sencilla: la variable de la izquierda recibe una *copia* del *valor* de la variable de la derecha. Cuando asigna la variable de referencia de un objeto a otro, la situación se vuelve un poco más complicada porque usted cambia el objeto al cual se refiere la variable de referencia. El efecto de esta diferencia puede arrojar algunos resultados contraproducentes. Por ejemplo, considere el siguiente fragmento:

```
Automotor carro1 = new Automotor();
Automotor carro2 = carro1;
```

A primera vista, resulta fácil pensar que **carro1** y **carro2** aluden a objetos diferentes, pero no es así, pues ambos se refieren al *mismo* objeto. La asignación de **carro1** a **carro2** simplemente hace que

carro2 se refiera al mismo objeto que **carro1**. Por lo tanto, **carro1** y **carro2** pueden actuar sobre el objeto. Por ejemplo, después de que se ejecuta la asignación

```
carro1.kpl = 9;
```

estas dos instrucciones **println()**

```
System.out.println(carro1.kpl);
System.out.println(carro2.kpl);
```

despliegan el mismo valor: 9.

Aunque **carro1** y **carro2** hacen referencia al mismo objeto, no están vinculados de ninguna otra manera. Por ejemplo, una asignación posterior a **carro2** simplemente cambia el objeto al que **carro2** hace referencia. Por ejemplo,

```
Automotor carro1 = new Automotor();
Automotor carro2 = carro1;
Automotor carro1 = new Automotor();
```

```
carro2 = carro3; // ahora carro2 y carro3 hacen referencia al mismo objeto.
```

Después de que esta secuencia se ejecuta, **carro2** hace referencia al mismo objeto que **carro3**. El objeto al que hace referencia **carro1** permanece sin cambio.



Comprobación de avance

1. Explique lo que ocurre cuando una variable de referencia a un objeto se asigna a otro objeto.
2. Suponiendo una clase llamada **miClase**, muestre cómo se crea un objeto llamado **ob**.

HABILIDAD
FUNDAMENTAL

4.4

Métodos

Como ya se explicó, las variables de instancia y los métodos son los elementos de las clases. Hasta ahora, la clase **Automotor** contiene datos, pero no métodos. Aunque las clases que sólo contienen datos son perfectamente válidas, la mayor parte de las clases tiene métodos. Los métodos son subrutinas que manipulan los datos definidos por la clase y, en muchos casos, proporcionan acceso a esos datos. En la mayoría de los casos, otras partes de su programa interactuarán con una clase a través de sus métodos.

1. Cuando una variable de referencia a objeto se asigna a otro objeto, ambas variables hacen referencia al mismo objeto. *No se hace una copia del objeto.*
2. `MiClase on = new MiClase();`

Un método contiene una o más instrucciones. En un código de Java bien escrito, cada método sólo realiza una tarea y tiene un nombre. Este nombre es el que se utiliza para llamar al método. En general, puede asignarle a un método el nombre que usted desee. Sin embargo, recuerde que **main()** está reservado para el método que empieza la ejecución de su programa. Además, no debe usar palabras clave de Java para los nombres de métodos.

Al denotar métodos en texto, se ha usado y se seguirá usando en este libro una convención que se ha vuelto común cuando se escribe acerca de Java: un método tendrá un paréntesis después de su nombre. Por ejemplo, si el nombre de un método es **obtenerval**, se escribirá **obtenerval()** cuando su nombre se use en una frase. Esta notación le ayudará a distinguir los nombres de variables de los nombres de métodos en este libro.

La forma general de un método se muestra a continuación:

```
tipo-dev nombre(lista-parámetros) {  
    // cuerpo del método  
}
```

Aquí, *tipo-dev* especifica el tipo de datos que es devuelto por el método. Éste puede ser cualquier tipo válido, incluyendo tipos de clase que usted puede crear. Si el método no devuelve un valor, su tipo devuelto deberá ser **void**. El nombre del método está especificado por *nombre*. Éste puede ser cualquier identificador legal diferente al que ya usan otros elementos dentro del alcance actual. *Lista-parámetros* es una secuencia de pares tipo-identificador separados por comas. Los parámetros son, en esencia, variables que reciben el valor de los *argumentos* que *pasaron* al método al momento de que éste fue llamado. Si el método no tiene parámetros, la lista de parámetros estará vacía.

Adición de un método a la clase Automotor

Como se acaba de explicar, los métodos de una clase suelen manipular y proporcionar acceso a los datos de la clase. Con esto en mente, recuerde que **main()** en los ejemplos anteriores calculó el rango de un automotor al multiplicar su consumo de gasolina por la capacidad de su tanque. Aunque es técnicamente correcta, ésta no es la mejor manera de manejar este cálculo. La propia clase **Automotor** maneja mejor el cálculo del rango de un automotor. La razón de que esta conclusión sea fácil de comprender es que el rango de un vehículo depende de la capacidad del tanque y la tasa de consumo de gasolina. Ambas cantidades están encapsuladas en **Automotor**. Al agregar a **Automotor** un método que calcule el rango, mejorará su estructura orientada a objetos.

Para agregar un método a **Automotor**, debe especificarlo dentro de una declaración de **Automotor**. Por ejemplo, la siguiente versión de **Automotor** contiene un método llamado **rango()** que despliega el rango del vehículo.

```
// Agrega rango a Automotor.  
  
class Automotor {  
    int pasajeros;    // número de pasajeros  
    int tanquegas;    // capacidad del tanque en litro
```

```

int kpl;           // consumo de gasolina en km por litro

// Despliega el rango.
void rango() { ← El método rango() está contenido dentro de la clase Automotor.
    System.out.println("El rango es " + tanquegas * kpl);
}
}

```

↑ ↑
 Observe que **tanquegas** y **kpl** se usan de manera directa, sin el operador de punto.

```

class AgregarMet {
    public static void main(String args[]) {
        Automotor minivan = new Automotor();
        Automotor carrodepor = new Automotor();

        int rango1, rango2;

        // asigna valores en campos de minivan
        minivan.pasajeros = 7;
        minivan.tanquegas = 60;
        minivan.kpl = 6;

        // asigna valores en campos de carrodepor
        carrodepor.pasajeros = 2;
        carrodepor.tanquegas = 50;
        carrodepor.kpl = 4;

        System.out.print("Una minivan puede transportar " + minivan.pasajeros +
            ". ");

        minivan.rango(); // despliega rango de minivan

        System.out.print("Un carro deportivo puede transportar " + carrodepor.
            pasajeros + ".");

        carrodepor.rango(); // despliega rango de carrodepor.
    }
}

```

Este programa genera la siguiente salida:

```

Una minivan puede transportar 7 pasajeros con un rango de 1260
Un carro deportivo puede transportar 2 pasajeros con un rango de 200

```

Revisemos los elementos clave de este programa empezando por el propio método **rango()**. La primera línea de **rango()** es

```
void rango() {
```

La línea declara un método llamado **rango** que no tiene parámetros. El tipo que devuelve es **void**; así que **rango()** no devuelve un valor al método que lo llama. La línea termina con la llave de apertura del método del cuerpo.

El cuerpo de **rango()** es la siguiente línea única:

```
System.out.println("El rango es " + tanquegas * kpl);
```

La instrucción despliega el rango del vehículo al multiplicar **tanquegas** por **kpl**. Debido a que cada objeto de tipo **Automotor** tiene su propia copia de **tanquegas** y **kpl**, cuando se llama a **rango()**, el cálculo del rango usa copias de esas variables en el objeto que llama.

El método **rango()** termina cuando se encuentra su llave de cierre. Esto hace que el control del programa se transfiera de nuevo al método que hace la llamada.

Ahora, revise esta línea de código a partir del interior de **main()**:

```
minivan.rango();
```

Esta instrucción invoca el método **rango()** de **minivan**, es decir, llama a **rango()** relacionado con el objeto **minivan** empleando el nombre del objeto seguido del operador de punto. Cuando se llama a un método, el control del programa se transfiere al método. Cuando el método termina, el control se regresa al que llama, y la ejecución se reanuda con la línea de código que sigue a la llamada.

En este caso, la llamada a **minivan.rango()** despliega el rango del vehículo definido por **minivan**. De manera similar, la llamada a **carrodepor.rango()** despliega el rango del vehículo definido por **carrodepor**. Cada vez que **rango()** es invocado, éste despliega el rango del objeto especificado.

Debe tomarse en cuenta un elemento muy importante dentro del método **rango()**: hay referencias directas a las variables de instancia **tanquegas** y **kpl**; no es necesario antecederlas con un nombre de objeto o un operador de punto. Cuando un método usa una variable de instancia que está definida por su clase, lo hace directamente, sin la referencia explícita a un objeto y sin el uso del operador de punto. Si lo reflexiona, resulta fácil comprenderlo. Un método siempre se invoca en relación con algún objeto de su clase. Una vez que esta invocación ha ocurrido, se conoce al objeto. Por lo tanto, dentro de un método no es necesario especificar el objeto por segunda ocasión. Esto significa que **tanquegas** y **kpl** dentro de **rango()** hacen referencia implícita a las copias de estas variables encontradas en el objeto que invoca a **rango()**.

HABILIDAD
FUNDAMENTAL

4.5

Regreso de un método

En general, hay dos condiciones que causan el regreso de un método: la primera, como se muestra en el método **rango()**, es cuando se encuentra la llave de cierre del método; y la segunda es cuando se ejecuta una instrucción **return**. Hay dos formas de **return**: para usarla en métodos **void** (los que no devuelven un valor) y para valores de regreso. La primera forma se examina a continuación. En la siguiente sección se explicará cómo devolver valores.

En un método **void**, puede usar la terminación inmediata de un método empleando esta forma de **return**:

```
return;
```

Cuando se ejecuta esta instrucción, el control del programa regresa al método que llama, omitiendo cualquier código restante del método. Por ejemplo, revise este método:

```
void miMet() {
    int i;

    for(i=0; i<10; i++) {
        if(i == 5) return; // se detiene en 5
        System.out.println();
    }
}
```

Aquí, el bucle **for** sólo se ejecutará de 0 a 5 porque una vez que **i** es igual a 5, el método regresa.

Se permite tener varias instrucciones de regreso en un método, sobre todo cuando hay dos o más rutas fuera de él. Por ejemplo:

```
void miMet() {
    // ...
    if(hecho) return;
    // ...
    if(error) return;
}
```

Aquí, el método regresa si se cumple o si un error ocurre. Sin embargo, tenga cuidado, porque la existencia de muchos puntos de salida de un método puede destruir su código; así que evite su uso de manera casual. Un método bien diseñado se caracteriza por puntos de salida bien definidos.

En resumen: un método **void** puede regresar mediante una de las siguientes formas: ya sea que se llegue a su llave de cierre o que se ejecute una instrucción **return**.

HABILIDAD
FUNDAMENTAL

4.6

Devolución de un valor

Aunque los métodos con un tipo de devolución **void** no son raros, casi todos los métodos devolverán un valor. En realidad, la capacidad de regresar un valor es una de las características más útiles de un método. Un ejemplo de un valor devuelto es cuando usamos la función **sqrt()** para obtener una raíz cuadrada.

Los valores devueltos se usan para diversos fines en programación. En algunos casos, como con **sqrt()**, el valor devuelto contiene el resultado de algunos cálculos. En otros casos, el valor devuelto puede indicar simplemente éxito o fracaso. En otros más, puede contener un código de estatus. Cualquiera que sea el propósito, el uso de valores de retorno de método constituye una parte integral de la programación en Java.

Los métodos regresan un valor a la rutina que llaman empleando esta forma de **return**:

```
return valor;
```

Aquí, *valor* es el valor devuelto.

Puede usar un valor devuelto para mejorar la implementación de **rango()**. En lugar de desplegar el rango, un método mejor consiste en hacer que **rango()** calcule el rango y devuelva este valor. Una de las ventajas de este método es que puede usar el valor para otros cálculos. El siguiente ejemplo modifica **rango()** para que devuelva el rango en lugar de desplegarlo.

```
// Uso de un valor devuelto.

class Automotor {
    int pasajeros;    // número de pasajeros
    int tanquegas;    // capacidad del tanque en litro
    int kpl;          // consumo de gasolina en km por litro

    // Devuelve el rango.
    int rango() {
        return kpl * tanquegas; ← Devuelve el rango dado para un automotor determinado.
    }
}


class RetMet {
    public static void main(String args[]) {
        Automotor minivan = new Automotor();
        Automotor carrodepor = new Automotor();

        int rangol, rango2;

        // asigna valores en campos de minivan
        minivan.pasajeros = 7;
        minivan.tanquegas = 60;
        minivan.kpl = 6;

        // asigna valores en campos de carrodepor
        carrodepor.pasajeros = 2;
        carrodepor.tanquegas = 50;
        carrodepor.kpl = 4;
```

```
// obtiene los rangos
rango1 = minivan.rango();
rango2 = carrodepor.rango();
```



Asigna el valor devuelto a una variable.

```
System.out.println("Una minivan puede transportar " + minivan.pasajeros +
    " pasajeros con rango de " + rango1 + " km");
```

```
System.out.println("Un carro deportivo puede transportar " + carrodepor.
    pasajeros + " pasajeros con rango de " + rango2 + " km");

}
}
```

Ésta es la salida:

```
Una minivan puede transportar 7 pasajeros con rango de 1260 km
Un carro deportivo puede transportar 2 pasajeros con un rango de 200 km
```

En el programa, observe que cuando se llama a **rango()**, éste se pone a la derecha de la instrucción de asignación. A la izquierda está una variable que recibirá el valor devuelto por **rango()**. Por lo tanto, después de que se ejecuta

```
rango1 = minivan.rango();
```

el rango del objeto **minivan** se almacena en **rango1**.

Observe que **rango()** ahora tiene un tipo de retorno **int**. Esto significa que devolverá un valor entero. El tipo devuelto por un método es importante porque el tipo de datos debe ser compatible con el tipo devuelto que el método especifica. Así que, si quiere que un método devuelva datos de tipo **double**, su tipo devuelto debe ser **double**.

Aunque el programa anterior es correcto, no está escrito con la eficiencia que podría tener. De manera específica, no son necesarias las variables **rango1** o **rango2**. Es posible usar directamente una llamada a **rango()** en la instrucción **println()** como se muestra aquí:

```
System.out.println("Una minivan puede transportar " + minivan.pasajeros +
    " pasajeros con rango de " + minivan.rango() + " km");
```

En este caso, cuando se ejecuta **println()**, se llama automáticamente a **minivan.rango()** y su valor se pasará a **println()**. Más aún, puede usar una llamada a **rango()** cada vez que se requiera el rango de un objeto de **Automotor**. Por ejemplo, esta instrucción compara los rangos de los dos vehículos:



```
if(a1.rango() > a2.rango()) System.out.println("a1 tiene un rango mayor");
```

Uso de parámetros

Es posible pasar uno o más valores a un método cuando éste es llamado. Como se explicó, un valor que pasa a un método es un *argumento*. Dentro del método, la variable que recibe el argumento es un *parámetro*. Los parámetros se declaran dentro del paréntesis que sigue al nombre del método. La sintaxis de declaración del parámetro es la misma que la que se usa para las variables. Un parámetro se encuentra dentro del alcance de su método y, además de las tareas especiales de recibir un argumento, actúa como cualquier otra variable local.

He aquí un ejemplo simple en el que se utiliza un parámetro. Dentro de la clase **RevNum**, el método **esPar()** devuelve **true** si el valor que se pasa es par. De lo contrario, devuelve **false**. Por lo tanto, **esPar()** tiene un tipo de regreso **boolean**.

```
// Un ejemplo simple en el que se utiliza un parámetro.
```

```
class RevNum {  
    // devuelve true si x es par  
    boolean esPar(int x) {  Aquí, x es un parámetro entero de esPar().  
        if((x%2) == 0) return true;  
        else return false;  
    }  
}  
  
class ParamDemo {  
    public static void main(String args[]) {  
        RevNum e = new RevNum();  
         Pasa argumentos a esPar().  
        if(e.esPar(10)) System.out.println("10 es par.");  
        if(e.esPar(9)) System.out.println("9 es par.");  
        if(e.esPar(8)) System.out.println("8 es par.");  
    }  
}
```

He aquí la salida producida por el programa:

```
10 es par  
8 es par
```

En el programa, se llama tres veces a **esPar()**, y cada vez que esto sucede se pasa un valor diferente. Echemos un vistazo de cerca al proceso. En primer lugar, observe cómo se llama a **esPar()**:

el argumento se especifica entre los paréntesis y cuando se llama a **esPar()** por primera vez, se pasa el valor 10. De ahí que, cuando se empieza a ejecutar **esPar()**, el parámetro **x** recibe el valor 10. En la segunda llamada, el argumento es 9, y **x** entonces tiene el valor 9. En la tercera llamada, el argumento es 8, que es el valor que recibe **x**. Lo que debe notar es que el valor que se pasa como argumento cuando se llama a **esPar()** es el valor recibido por su parámetro, en este caso, **x**.

Un método puede tener más de un parámetro. Simplemente declare cada parámetro separándolos mediante una coma. Por ejemplo, la clase **Factor** define un método llamado **esFactor()** que determina si el primer parámetro es factor del segundo.

```
class Factor {
    boolean esFactor(int a, int b) {
        if( (b % a) == 0) return true;
        else return false;
    }
}

class esFact {
    public static void main(String args[]) {
        Factor x = new Factor();
        if(x.esFactor(2, 20)) System.out.println("2 es factor");
        if(x.esFactor(3, 20)) System.out.println("esto no se muestra");
    }
}
```

← Este método tiene 2 parámetros.

→ Pasa dos argumentos a **esFactor()**.

Observe que cuando se llama a **esFactor()**, los argumentos también están separados por comas.

Cuando se usan varios parámetros, cada uno especifica su propio tipo, el cual puede diferir de los demás. Por ejemplo, lo siguiente es perfectamente válido:

```
int MiMet(int a, double b, float c) {
    // ...
}
```

Adición de un método con parámetros a un automotor

Puede usar un método con parámetros para agregar una nueva función a la clase **Automotor**: la capacidad de calcular la cantidad de gasolina necesaria para recorrer una distancia determinada. A este nuevo método se le llamará **gasnecesaria()**. Este método toma el número de kilómetros que quiere manejar y devuelve el número de litros requeridos de gasolina. El método **gasnecesaria()** se define así:

```
double gasnecesaria (int miles) {
    return (double) miles / kpl
}
```


Tome en cuenta que este método regresa un valor de tipo **double**. Esto es útil ya que la cantidad de gasolina necesaria para una distancia tal vez no sea un número constante.

La clase **Automotor** completa que incluye tanquegas se muestra a continuación

```
/*
    Agrega un método con parámetros que calcula la
    gasolina necesaria para una distancia determinada.
*/

class Automotor {
    int pasajeros;    // número de pasajeros
    int tanquegas;    // capacidad del tanque en litro
    int kpl;          // consumo de gasolina en km por litro

    // Devuelve el rango.
    int rango() {
        return kpl * tanquegas;
    }

    // calcula la gasolina necesaria para una distancia.
    double gasnecesaria(int kilómetros) {
        return (double) miles / kpl;
    }
}

class CalcGas {
    public static void main(String args[]) {
        Automotor minivan = new Automotor();
        Automotor carrodepor = new Automotor();
        double litros;
        int dist = 252;

        // asigna valores en campos de minivan
        minivan.pasajeros = 7;
        minivan.tanquegas = 60;
        minivan.kpl = 6;

        // asigna valores en campos de carrodepor
        carrodepor.pasajeros = 2;
        carrodepor.tanquegas = 50;
        carrodepor.kpl = 4;
    }
}
```

```
litros = minivan.gasnecesaria(dist);

System.out.println("Para recorrer " + dist + " km una minivan necesita " +
    litros + " litros de gasolina.");

litros = carrodepor.gasnecesaria(dist);

System.out.println("Para recorrer " + dist + " km un carro deportivo
necesita " + litros + " litros de gasolina.");

}
}
```

Ésta es la salida del programa:

```
Para recorrer 252 km una minivan necesita 42.0 litros de gasolina.
Para recorrer 252 km un carro deportivo necesita 63.0 litros de gasolina.
```



Comprobación de avance

1. ¿Cuándo se debe acceder a una variable de instancia o a un método mediante la referencia a un objeto empleando el operador de punto? ¿Cuándo puede usarse directamente una variable o un método?
2. Explique la diferencia entre un argumento y un parámetro.
3. Explique las dos maneras en las que un método puede regresar a su llamada.

-
1. Cuando se accede a una variable de instancia mediante un código que no es parte de la clase en la que está definida la variable de instancia, debe realizarse mediante un objeto, con el uso de un operador de punto. Sin embargo, cuando se accede a una variable de instancia con un código que es parte de la misma clase que la variable de instancia, es posible hacer referencia directa a dicha variable. Lo mismo se aplica a los métodos.
 2. Un *argumento* es un valor que se pasa a un método cuando es invocado. Un *parámetro* es una variable definida por un método que recibe el valor de un argumento.
 3. Es posible hacer que un método regrese con el uso de la instrucción **return**. Si el método cuenta con un tipo de regreso **void**, entonces el método también regresará cuando se llegue a sus llaves de cierre. Los métodos que no son **void** deben devolver un valor, de modo que regresar al alcanzar la llave de cierre no es una opción.

Proyecto 4.1 Creación de una clase Ayuda

`ClaseAyudaDemo.java` Si se intentara resumir en una frase la esencia de una clase, éste sería el resultado: una clase encapsula la funcionalidad. Por supuesto, el truco consiste en saber dónde termina una “funcionalidad” y dónde empieza otra. Como regla general, seguramente usted desea que sus clases sean los bloques de construcción de una aplicación más grande. Para ello, cada clase debe representar una sola unidad funcional que realice acciones claramente delineadas; así que usted querrá que sus clases sean lo más pequeñas posibles, ¡pero no demasiado pequeñas! Es decir, las clases que contienen una funcionalidad extraña confunden y destruyen al código, pero las clases que contienen muy poca funcionalidad están fragmentadas. ¿Cuál es el justo medio? Éste es el punto en el que la *ciencia* de la programación se vuelve el *arte* de la programación. Por fortuna, la mayoría de los programadores se dan cuenta de que ese acto de equilibrio se vuelve más fácil con la experiencia.

Para empezar a obtener esa experiencia, convierta el sistema de ayuda del proyecto 3.3 del módulo anterior en una clase Ayuda. Examinemos el porqué ésta es una buena idea. En primer lugar, el sistema de ayuda define una unidad lógica. Simplemente despliega la sintaxis de las instrucciones de control de Java. Por consiguiente, su funcionalidad es compacta y está bien definida. En segundo lugar, colocar la ayuda en una clase es un método estéticamente agradable. Cada vez que quiera ofrecer el sistema de ayuda a un usuario, simplemente iniciará un objeto de sistema de ayuda. Por último, debido a que la ayuda está encapsulada, ésta puede actualizarse o cambiarse sin que ello ocasione efectos secundarios indeseables en los programas que la utilizan.

Paso a paso

1. Cree un nuevo archivo llamado **ClaseAyudaDemo.java**. Si desea guardar lo que ya tiene escrito, copie el archivo del proyecto 3.3, llamado **Ayuda3.java**, en **ClaseAyudaDemo.java**.
2. Para convertir el sistema de ayuda en una clase, primero debe determinar con precisión lo que constituye el sistema de ayuda. Por ejemplo, en **Ayuda3.java** hay un código para desplegar un menú, ingresar opciones del usuario, revisar respuestas válidas y desplegar información acerca del elemento seleccionado. El programa también se recorre en bucle hasta que se oprime la letra q. Es evidente entonces que el menú, la revisión de una respuesta válida y el despliegue de la información integran al sistema de ayuda, y no la manera en que se obtiene la entrada del usuario y si deben o no procesarse solicitudes repetidas. Por lo tanto, creará una clase que despliegue la información de ayuda y el menú de ayuda, y que compruebe que se lleve a cabo una selección válida. Sus métodos se denominarán **ayudactiva()**, **mostrarmenu()** y **escorrecta()**, respectivamente.

(continúa)

3. Cree el método **ayudactiva()** como se muestra aquí:

```
void ayudactiva(int que) {
    switch(que) {
        case '1':
            System.out.println("if:\n");
            System.out.println("if(condición) instrucción;");
            System.out.println("instrucción else;");
            break;
        case '2':
            System.out.println("switch:\n");
            System.out.println("switch(expresión) {");
            System.out.println("    constante case:");
            System.out.println("    secuencia de instrucciones");
            System.out.println("    break;");
            System.out.println("    // ...");
            System.out.println("}");
            break;
        case '3':
            System.out.println("for:\n");
            System.out.print("for(inic; condición; iteración)");
            System.out.println(" instrucción;");
            break;
        case '4':
            System.out.println("while:\n");
            System.out.println("while(condición) instrucción;");
            break;
        case '5':
            System.out.println("do-while:\n");
            System.out.println("do {");
            System.out.println("    instrucción;");
            System.out.println("} while (condición);");
            break;
        case '6':
            System.out.println("break:\n");
            System.out.println("break; o break etiqueta;");
            break;
        case '7':
            System.out.println("continue:\n");
            System.out.println("continue; o continue etiqueta;");
            break;
    }
    System.out.println();
}
```

4. A continuación, cree el método **mostrarmenu()**:

```
void mostrarmenu() {
    System.out.println("Ayuda habilitada:");
}
```

```
System.out.println(" 1. if");
System.out.println(" 2. switch");
System.out.println(" 3. for");
System.out.println(" 4. while");
System.out.println(" 5. do-while");
System.out.println(" 6. break");
System.out.println(" 7. continue\n");
System.out.print("Elija una (q para salir): ");
}
```

5. Cree el método **esincorrecta()** que se muestra aquí:

```
boolean esincorrecta(int ch) {
    if(ch < '1' | ch > '7' & ch != 'q') return false;
    else return true;
}
```

6. Ensamble los métodos anteriores en la clase **Ayuda**, mostrada aquí:

```
class Ayuda {
    void ayudactiva(int que) {
        switch(que) {
            case '1':
                System.out.println("if:\n");
                System.out.println("if(condición) instrucción;");
                System.out.println("instrucción else;");
                break;
            case '2':
                System.out.println("switch:\n");
                System.out.println("switch(expresión) {");
                System.out.println("    constante case:");
                System.out.println("    secuencia de instrucciones");
                System.out.println("    break;");
                System.out.println("    // ...");
                System.out.println("}");
                break;
            case '3':
                System.out.println("for:\n");
                System.out.print("for(inic; condición; iteración)");
                System.out.println(" instrucción;");
                break;
            case '4':
                System.out.println("while:\n");
                System.out.println("while(condición) instrucción;");
                break;
            case '5':
                System.out.println("do-while:\n");
```

(continúa)

```

        System.out.println("do {");
        System.out.println("    instrucción;");
        System.out.println("} while (condición);");
        break;
    case '6':
        System.out.println("break:\n");
        System.out.println("break; o break etiqueta;");
        break;
    case '7':
        System.out.println("continue:\n");
        System.out.println("continue; o continue etiqueta;");
        break;
    }
    System.out.println();
}

void mostrarmenú() {
    System.out.println("Ayuda habilitada:");
    System.out.println("  1. if");
    System.out.println("  2. switch");
    System.out.println("  3. for");
    System.out.println("  4. while");
    System.out.println("  5. do-while");
    System.out.println("  6. break");
    System.out.println("  7. continue\n");
    System.out.print("Elija una (q para salir): ");
}

boolean escorrecta(int ch) {
    if(ch < '1' | ch > '7' & ch != 'q') return false;
    else return true;
}
}

```

- 7.** Por último, reescriba el método **main()** del proyecto 3.3, de modo que use la nueva clase **Ayuda**. Llame a esta clase **ClaseAyudaDemo.java**. A continuación se muestra el listado completo:

```

/*
    Proyecto 4.1

    Convierte el sistema de ayuda del proyecto 3.3
    en una clase Ayuda.
*/

class Ayuda {

```

```
void ayudactiva(int que) {
    switch(que) {
        case '1':
            System.out.println("if:\n");
            System.out.println("if(condición) instrucción;");
            System.out.println("instrucción else;");
            break;
        case '2':
            System.out.println("switch:\n");
            System.out.println("switch(expresión) {");
            System.out.println("    constante case:");
            System.out.println("    secuencia de instrucciones");
            System.out.println("    break;");
            System.out.println("    // ...");
            System.out.println("}");
            break;
        case '3':
            System.out.println("for:\n");
            System.out.println("for(inic; condición; iteración)");
            System.out.println("instrucción;");
            break;
        case '4':
            System.out.println("while:\n");
            System.out.println("while(condición) instrucción;");
            break;
        case '5':
            System.out.println("do-while:\n");
            System.out.println("do {");
            System.out.println("    instrucción;");
            System.out.println("} while (condición);");
            break;
        case '6':
            System.out.println("break:\n");
            System.out.println("break; o break etiqueta;");
            break;
        case '7':
            System.out.println("continue:\n");
            System.out.println("continue; o continue etiqueta;");
            break;
    }
    System.out.println();
}

void mostrarmenú() {
    System.out.println("Ayuda habilitada:");
}
```

(continúa)

```

        System.out.println("  1. if");
        System.out.println("  2. switch");
        System.out.println("  3. for");
        System.out.println("  4. while");
        System.out.println("  5. do-while");
        System.out.println("  6. break");
        System.out.println("  7. continue\n");
        System.out.print("Elija una (q para salir): ");
    }

    boolean escorrecta(int ch) {
        if(ch < '1' | ch > '7' & ch != 'q') return false;
        else return true;
    }

}

class ClaseAyudaDemo {
    public static void main(String args[])
        throws java.io.IOException {
        char inciso;
        Ayuda objayuda = new Ayuda();

        for(;;) {
            do {
                objayuda.mostrarmenú();
                do {
                    inciso = (char) System.in.read();
                } while(inciso == '\n' | inciso == '\r');

            } while( !objayuda.escorrecta(inciso) );

            if(inciso == 'q') break;

            System.out.println("\n");

            objayuda.ayudactiva(inciso);
        }
    }
}

```

Al probar el programa encontrará que funciona igual que antes. La ventaja de este método es que ahora tiene un componente de sistema de ayuda que puede reutilizarse cuando sea necesario.

Constructores

En los ejemplos anteriores, las variables de instancia de cada objeto **Automotor** tienen que establecerse manualmente mediante una secuencia de instrucciones como:

```
minivan.pasajeros = 7;  
minivan.tanquegas = 60;  
minivan.kpl = 6;
```

Un método como éste nunca se usaría en un código escrito profesionalmente en Java. Aparte de ser propenso a error (podría olvidar establecer alguno de los campos), existe una mejor manera de realizar esta tarea: el constructor.

Un *constructor* inicializa un objeto cuando este último se crea. Tiene el mismo nombre que su clase y es sintácticamente similar a un método. Sin embargo, los constructores no tienen un tipo de regreso explícito. Por lo general, usará un constructor para dar valores iniciales a las variables de instancia definidas por la clase, o para realizar cualquier otro procedimiento de inicio que se requiera para crear un objeto completamente formado.

Todas las clases tienen constructores, ya sea que los defina o no, porque Java proporciona automáticamente un constructor predeterminado que inicializa todas las variables de miembros en cero. Sin embargo, una vez que haya definido su constructor, el constructor predeterminado ya no se utilizará.

He aquí un ejemplo simple en el que se utiliza un constructor.

```
// Un constructor simple.  
  
class MiClase {  
    int x;  
  
    MiClase() { ← Éste es el constructor de MiClase.  
        x = 10;  
    }  
}  
  
class ConsDemo {  
    public static void main(String args[]) {  
        MiClase t1 = new MiClase();  
        MiClase t2 = new MiClase();  
  
        System.out.println(t1.x + " " + t2.x);  
    }  
}  
  
En este ejemplo, el constructor de MiClase es  
MiClase() {  
    x = 10;  
}
```

Este constructor asigna el valor 10 a la variable de instancia **x** de **MiClase**. Este constructor es llamado por **new** cuando un objeto se crea. Por ejemplo, en la línea

```
MiClase t1 = new MiClase();
```

se llama al constructor **MiClase()** en el objeto **t1**, dando a **t1.x** el valor 10. Lo mismo aplica para **t2**. Después de la construcción, **t2.x** tiene el valor de 10. Por lo tanto, la salida del programa es

```
10 10
```

HABILIDAD
FUNDAMENTAL

4.9

Constructores con parámetros

En el ejemplo anterior se empleó un constructor sin parámetros. Aunque esto resulta adecuado para algunas situaciones, lo más común será que necesite un constructor que acepte uno o más parámetros. Los parámetros se agregan a un constructor de la misma manera en la que se agregan a un método: sólo se les declara dentro del paréntesis después del nombre del constructor. Por ejemplo, en el siguiente caso, a **MiClase** se le ha dado un constructor con parámetros:

```
// Un constructor con parámetros.

class MiClase {
    int x;

    MiClase(int i) { ← El constructor tiene un parámetro.
        x = i;
    }
}

class ConsParamDemo {
    public static void main(String args[]) {
        MiClase t1 = new MiClase(10);
        MiClase t2 = new MiClase(88);

        System.out.println(t1.x + " " + t2.x);
    }
}
```

Ésta es la salida de este programa:

```
10 88
```

En esta versión del programa, el constructor **MiClase()** define un parámetro llamado **i**, el cual se usa para inicializar la variable de instancia **x**. Por lo tanto, cuando se ejecuta la línea

```
MiClase t1 = new MiClase(10);
```

se pasa el valor 10 a **i**, que luego se asigna a **x**.

Adición de un constructor a la clase Automotor

Podemos mejorar la clase **Automotor** agregando un constructor que inicialice automáticamente los campos pasajeros, **tanquegas** y **kpl** cuando se construye un objeto. Ponga especial atención en la manera en que se crean los objetos de **Automotor**.

```
// Agregar un constructor.

class Automotor {
    int pasajeros;    // número de pasajeros
    int tanquegas;    // capacidad del tanque en litro
    int kpl;          // consumo de gasolina en km por litro

    // Este es un constructor para Automotor.
    Automotor(int p, int f, int m) { ← Constructor de Automotor.
        pasajeros = p;
        tanquegas = f;
        kpl = m;
    }

    // Regresa el rango.
    int rango() {
        return kpl * tanquegas;
    }

    // calcula la gasolina necesaria para una distancia.
    double gasnecesaria(int km) {
        return (double) km / kpl;
    }
}

class ConsAutDemo {
    public static void main(String args[]) {

        // construye automotores completos
        Automotor minivan = new Automotor(7, 60, 6);
        Automotor carrodepor = new Automotor(2, 50, 3);
        double litros;
        int dist = 252;

        litros = minivan.gasnecesaria(dist);

        System.out.println("Para recorrer " + dist + " kms una minivan necesita " +
            litros + " litros de gasolina.");
    }
}
```

```

        litros = carrodepor.gasnecesaria(dist);

        System.out.println("Para recorrer " + dist + " kms un carro deportivo
                           necesita " + litros + " litros de gasolina.");
    }
}

```

Cuando se crean, tanto **minivan** como **carrodepor** se inicializan con el constructor **Automotor()**. Cada objeto es inicializado como se especifica en los parámetros de su constructor. Por ejemplo, en la siguiente línea:

```
Automotor minivan = new Automotor(7, 60, 6);
```

Los valores 7, 60 y 6 se pasan al constructor **Automotor()** cuando **new** crea el objeto. Por lo tanto, las copias de **minivan** de **pasajeros**, **tanquegas** y **kpl** contendrán los valores 7, 60 y 6, respectivamente. La salida de este programa es la misma que en la versión anterior.



Comprobación de avance

1. ¿Qué es un constructor y cuándo se ejecuta?
2. ¿Un constructor tiene un tipo de regreso?

HABILIDAD
FUNDAMENTAL

4.10

Nueva visita al operador new

Ahora que sabe más de las clases y sus constructores, echemos un vistazo al operador **new**. Este operador tiene esta forma general:

```
var-clase = new nombre-clase();
```

En este caso, *var-clase* es una variable del tipo de clase que se está creando. El *nombre-clase* es el nombre de la clase de la que se está creando una instancia. El nombre de la clase seguida de un paréntesis especifica el constructor de la clase. Si una clase no define su propio constructor, **new** usará el constructor predeterminado que Java proporciona. De manera que **new** puede usarse para crear un objeto de cualquier tipo de clase.

1. Un constructor es un método que se ejecuta cuando el objeto de una clase se inicializa. Un constructor se usa para inicializar el objeto que se está creando.
2. No.

Pregunte al experto

P: ¿Porqué no necesito usar `new` para variables de tipos primitivos, como `int` o `float`?

R: Los tipos primitivos de Java no están implementados como objetos sino que, debido a razones de eficiencia, están implementados como variables “normales”. Una variable de tipo primitivo contiene en realidad el valor que usted le ha proporcionado. Como se explicó, las variables de objeto son referencias al objeto. Esta capa de direccionamiento (así como otras funciones del objeto) agregan una carga extra a un objeto que se evita en un tipo primitivo.

Debido a que la memoria es finita, tal vez `new` no pueda asignar memoria a un objeto por falta de memoria suficiente. Si esto sucede, ocurrirá una excepción en tiempo de ejecución. (Aprenderá a manejar ésta y otras excepciones en el módulo 9.) En el caso de los programas que vienen de ejemplo en este libro, no tendrá que preocuparse por quedarse sin memoria, pero debe considerar esta posibilidad en los programas reales que escriba.

HABILIDAD
FUNDAMENTAL

4.11

Recolección de basura y finalizadores

Como se ha visto, los objetos se asignan, empleando el operador `new`, de manera dinámica a partir de un almacén de memoria libre. Como se explicó, la memoria no es infinita por lo que la memoria libre puede agotarse. Por lo tanto, es posible que `new` falle porque hay insuficiente memoria libre para crear el objeto deseado. Por tal motivo, un componente clave de cualquier esquema de asignación dinámica es la recuperación de memoria libre a partir de objetos no empleados, lo que deja memoria disponible para una reasignación posterior. En muchos lenguajes de programación, la liberación de la memoria asignada previamente se maneja de manera manual. Por ejemplo, en C++ usted usa el operador `delete` para liberar la memoria que fue asignada. Sin embargo, Java usa un método diferente, más libre de problemas: la *recolección de basura*.

El sistema de recolección de basura de Java reclama objetos automáticamente (lo cual ocurre de manera transparente, tras bambalinas, sin intervención del programador). Funciona así: cuando no existen referencias a un objeto, se supone que dicho objeto ya no es necesario por lo que se libera la memoria ocupada por el objeto. Esta memoria reciclada puede usarse entonces para asignaciones posteriores.

La recolección de basura sólo ocurre de manera esporádica durante la ejecución de su programa. No ocurrirá por el solo hecho de que existan uno o más objetos que ya no se usen. Por razones de eficiencia, la recolección de basura por lo general sólo se ejecutará cuando se cumplan dos condiciones: cuando haya objetos que reciclar y cuando sea necesario reciclarlos. Recuerde que la recolección de basura ocupa tiempo, de modo que el sistema en tiempo de ejecución de Java sólo la lleva a cabo cuando es necesaria. Por consiguiente, no es posible saber con precisión en qué momento tendrá lugar la recolección de basura.

El método `finalize()`

Es posible definir un método que sea llamado antes de que la recolección de basura se encargue de la destrucción final de un objeto. A este método se le llama **`finalize()`** y se utiliza con el fin de asegurar que un objeto terminará limpiamente. Por ejemplo, podría usar **`finalize()`** para asegurarse de que se cierre un archivo abierto que es propiedad de un determinado objeto.

Para agregar un finalizador a una clase, simplemente debe definir el método **`finalize()`**. En tiempo de ejecución, Java llama a ese método cada vez que está a punto de reciclar un objeto de esa clase. Dentro del método **`finalize()`** usted especificará las acciones que deben realizarse antes de que un objeto se destruya.

El método **`finalize()`** tiene la siguiente forma general:

```
protected void finalize()  
{  
    //aquí va el código de finalización  
}
```

En este caso, la palabra clave **`protected`** es un especificador que evita que un código definido fuera de su clase acceda a **`finalize()`**. Éste y los demás especificadores de acceso se explicarán en el módulo 6.

Resulta importante comprender que se llama a **`finalize()`** justo antes de la recolección de basura. No se le llama cuando, por ejemplo, un objeto sale del alcance. Esto significa que usted no puede saber cuando (o incluso si) **`finalize()`** se ejecutará. Por ejemplo, si su programa termina antes de que la recolección de basura ocurra, **`finalize()`** no se ejecutará. Por lo tanto, éste debe usarse como procedimiento de “respaldo” para asegurar el manejo apropiado de algún recurso, o bien, para aplicaciones de uso especial, y no como el medio que su programa emplea para su operación normal.

Pregunte al experto

P: Sé que C++ define a los llamados destructores, los cuales se ejecutan automáticamente cuando se destruye un objeto. ¿`Finalize()` es parecido a un destructor?

R: Java no tiene destructores. Aunque es verdad que el método **`finalize()`** se aproxima a la función de un destructor, no son lo mismo. Por ejemplo, siempre se llama a un destructor de C++ antes de que un objeto salga del alcance; sin embargo, no es posible saber en qué momento se llamará a **`finalize()`** para un objeto específico. Francamente, como Java emplea la recolección de basura, un destructor no resulta muy necesario.

Proyecto 4.2 Demostración de la finalización

Finalize.java

Debido a que la recolección de basura se aplica de manera esporádica y secundaria, no resultará trivial la demostración del método **finalize()**. Recuerde que se llama a **finalize()** cuando un objeto está por ser reciclado. Como ya se explicó, los objetos no necesariamente se reciclan en cuanto dejan de ser necesarios. El recolector de basura espera hasta que pueda realizar eficientemente su recolección que, por lo general, es cuando hay muchos objetos sin usar. En este sentido, para demostrar el método **finalize()**, a menudo necesitará crear y destruir una gran cantidad de objetos. Y eso es precisamente lo que llevará a cabo en este proyecto.

Paso a paso

1. Cree un nuevo archivo llamado **Finalize.java**.
2. Cree la clase **FDemo** que se muestra aquí:

```
class FDemo {
    int x;

    FDemo(int i) {
        x = i;
    }

    // llamado cuando se recicla el objeto
    protected void finalize() {
        System.out.println("Finaliza " + x);
    }

    // genera un ob que se destruye de inmediato
    void generator(int i) {
        FDemo o = new FDemo(i);
    }
}
```

El constructor asigna a la variable de instancia **x** un valor conocido. En este ejemplo, **x** se usa como ID de un objeto. El método **finalize()** despliega el valor de **x** cuando se recicla un objeto. De especial interés resulta **generator()** pues este método crea y descarta de inmediato un objeto de **FDemo**. En el siguiente paso observará cómo se lleva a cabo esto.

(continúa)

3. Cree la clase **Finalize** que se muestra aquí:

```
class Finalize {
    public static void main(String args[]) {
        int cuenta;

        FDemo ob = new FDemo(0);

        /* Ahora genera gran cantidad de objetos. En
           cierto momento se recolecta la basura.
           Nota: tal vez necesite aumentar la cantidad
           de objetos generados para forzar la
           recolección de basura. */

        for(cuenta=1; cuenta < 100000; cuenta++)
            ob.generator(cuenta);
    }
}
```

Esta clase crea un objeto inicial de **FDemo** llamado **ob**. Luego, usando **ob**, crea 100 000 objetos llamando a **generator()** sobre **ob**. Esto tiene el efecto neto de crear y descartar 100 000 objetos. En varios puntos a mitad de este proceso, la recolección de basura tiene lugar. Varios factores determinan con precisión la frecuencia y el momento; entre estos factores están la cantidad inicial de memoria libre y el sistema operativo. Sin embargo, en algún punto, usted empezará a ver los mensajes generados por **finalize()**. Si no ve los mensajes, pruebe a aumentar el número de objetos que se está generando al aumentar la cuenta en el bucle **for**.

4. He aquí el programa **finalize.java** completo:

```
/*
    Proyecto 4-2
    Demuestra el método finalize().
*/

class FDemo {
    int x;

    FDemo(int i) {
        x = i;
    }

    // llamado cuando el objeto se recicla
    protected void finalize() {
        System.out.println("Finaliza " + x);
    }
}
```



```

// genera un ob que se destruye de inmediato
void generator(int i) {
    FDemo o = new FDemo(i);
}

}

class Finalize {
    public static void main(String args[]) {
        int cuenta;

        FDemo ob = new FDemo(0);

        /* Ahora genera gran cantidad de objetos. En
           cierto momento se recolecta la basura.
           Nota: tal vez necesite aumentar la cantidad
           de objetos generados para forzar la
           recolección de basura. */

        for(cuenta=1; cuenta < 100000; cuenta++)
            ob.generator(cuenta);
    }
}

```

HABILIDAD
FUNDAMENTAL

4.12

La palabra clave this

Antes de concluir este módulo, es necesario introducir **this**. Cuando se llama a un método, se pasa automáticamente un argumento implícito que es una referencia al objeto que invoca (es decir, el objeto en el que se llama al método). A esta referencia se le denomina **this**. Para comprender **this**, primero considere un programa que cree una clase llamada **Pot**, la cual calcula el resultado de un número que esté elevado a alguna potencia entera:

```

class Pot {
    double b;
    int e;
    double val;

    Pot(double base, int exp) {
        b = base;
        e = exp;

        val = 1;
        if(exp==0) return;
        for( ; exp>0; exp--) val = val * base;
    }
}

```

4

Introducción a clases, objetos y métodos

Proyecto 4.2

Demostración de la finalización

```

    }

    double obtener_pot() {
        return val;
    }
}

class DemoPot {
    public static void main(String args[]) {
        Pot x = new Pot(4.0, 2);
        Pot y = new Pot(2.5, 1);
        Pot z = new Pot(5.7, 0);

        System.out.println(x.b + " elevado a la " + x.e +
                           " potencia es " + x.obtener_pot());
        System.out.println(y.b + " elevado a la " + y.e +
                           " potencia es " + y.obtener_pot());
        System.out.println(z.b + " elevado a la " + z.e +
                           " potencia es " + z.obtener_pot());
    }
}

```

Como ya lo sabe, dentro de un método, es posible acceder directamente a los otros miembros de una clase, sin necesidad de ninguna calificación de objeto o clase; así que, dentro de **obtener_pot()**, la instrucción

```
return val;
```

significa que se devolverá la copia de **val** asociada con el objeto al que invoca. Sin embargo, es posible escribir la misma instrucción de la siguiente manera:

```
return.this.val;
```

En este caso, **this** alude al objeto en el que se llama a **obtener_pot()**. Por consiguiente, **this.val** alude a esa copia del objeto de **val**. Por ejemplo, si se ha invocado a **obtener_pot()** en **x**, entonces **this** en la instrucción anterior hubiera hecho referencia a **x**. Escribir la instrucción sin el uso de **this** constituye en realidad un atajo.

He aquí la clase **Pot** completa, la cual está escrita con la referencia a **this**:

```

class Pot {
    double b;
    int e;
    double val;

    Pot(double base, int exp) {
        this.b = base;
    }
}

```

```

        this.e = exp;

        this.val = 1;
        if(exp==0) return;
        for( ; exp>0; exp--) this.val = this.val * base;
    }

    double obtener_pot() {
        return this.val;
    }
}

```

En realidad, ningún programador de Java escribiría **Pot** como se acaba de mostrar ya que no se gana nada, así que la forma estándar es más fácil. Sin embargo, **this** tiene algunos usos importantes. Por ejemplo, la sintaxis de Java permite que el nombre de un parámetro o de una variable local sea igual al nombre de una variable de instancia. Cuando esto sucede, el nombre local *oculta* la variable de instancia. Puede obtener acceso a la variable de instancia oculta si hace referencia a ella mediante **this**. Por ejemplo, aunque no se trata de un estilo recomendado, la siguiente es una manera sintácticamente válida de escribir el constructor **Pot()**.

```

Pot(double b, int e) {
    this.b = b;
    this.e = e;
    ▲────────────────────────────────── Esto se refiere a la variable de
    val = 1;                             instancia b, no al parámetro.
    if(e==0) return;
    for( ; e>0; e--) val = val * b;
}

```

En esta versión, los nombres de los parámetros son los mismos que los de las variables de instancia, pero estas últimas están ocultas. Sin embargo, **this** se usa para “descubrir” las variables de instancia.

✓ Comprobación de dominio del módulo 4

1. ¿Cuál es la diferencia entre una clase y un objeto?
2. ¿Cómo se define una clase?
3. ¿De qué tiene cada objeto una copia propia?
4. Empleando dos instrucciones separadas, muestre cómo declarar un objeto llamado **contador** de una clase llamada **MiContador**.

5. Muestre cómo se declara un método llamado **MiMet()** si tiene un tipo de retorno **double** y dos parámetros **int** llamados **a** y **b**.
6. Si un método regresa un valor, ¿cómo debe regresar?
7. ¿Qué nombre tiene un constructor?
8. ¿Qué función tiene **new**?
9. ¿Qué es la recolección de basura y cómo funciona? ¿Qué es **finalize()**?
10. ¿Qué es **this**?
11. ¿Un constructor puede tener uno o más parámetros?
12. Si un método no regresa un valor, ¿cuál debe ser su tipo de regreso?