Notas de Introducción al Lenguaje de Programación Java

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Septiembre 2004

Índice general

Capítulo 1

Introducción

Java es un lenguaje de programación de alto nivel, orientado a objetos, multi-hebra, usado para escribir tanto programas autocontenidos como *"applets"*, estos últimos como parte de los documentos HTML (*hypertext markup language*) que se encuentran disponibles en páginas WWW. Los *applets* permiten a las páginas de red tener un contenido ejecutable accesible con un navegador habilitado con Java.

Java tiene varias ventajas y características útiles para el programador:

- Una vez que el compilador e intérprete de Java han sido portados a una nueva plataforma, un programa Java puede ejecutarse en tal plataforma sin cambios.
- Java es un lenguaje fuertemente tipificado: el compilador hace varias verificaciones, tal como comparar los tipos de los argumentos que se pasan a los métodos para asegurar que se ajustan a los tipos de los parámetros formales.
- El intérprete hace muchas verificaciones durante el tiempo de ejecución, tal como asegurar que el índice de un arreglo no salga de sus limites, o que una referencia a un objeto no sea nula.
- No son posibles ni "fugas de memoria" (*memory leak*, un bloque de memoria que todavía se considera reservado en memoria aun cuando no haya referencias a él) ni "apuntadores colgantes" (*dangling pointers*, una referencia a un bloque de memoria que ha sido liberado), ya que el recolector de basura (*garbage collector*) de Java recolecta toda la memoria ocupada por objectos inalcanzables. No es necesario un procedimiento `free()`, susceptible a errores.

- No hay aritmética de apuntadores: las únicas operaciones permitidas sobre las referencias a un objeto son asignación de otra referencia a objeto y comparación con otra referencia. Por lo tanto, el programador tiene menor posibilidad de escribir código susceptible a errores que modifique un apuntador.

- Java viene acompañado de una biblioteca de herramientas para la creación de ventanas, conocida como AWT, que se utiliza por aplicaciones y *applets* para la implementación de interfaces gráficas de usuario.

Se encuentran disponibles muchos libros que describen el lenguaje Java (y que se han usado como referencias para las presentes notas) como son:

1. *The Java Programming Language.*
   Ken Arnold and James Gosling.
   Addison-Wesley, 1996.

2. *Java Essentials for C and C++ Programmers.*
   Barry Boone.
   Addison-Wesley, 1996.

3. *Core Java.*
   Gary Cornell and Cay S. Horstsmann.
   Prentice-Hall, 1996.

4. *Java in a Nutshell.*
   David Flanagan.
   O'Reilly, 1996.

Varios vienen acompañados de un CD-ROM que contiene los ejemplos de los programas del libro y una copia del *Java Development Kit* (JDK) de Sun Microsystems. Más aun, el JDK y su documentación asociada pueden obtenerse en forma gratuita en el sitio `ftp.javasoft.com`, mantenido por JavaSoft, que es una subsidiaria de Sun Microsystems.

El JDK consiste de un compilador, un intérprete (JVM o *Java Virtual machine*), y las bibliotecas de clases del sistema. Existen varias versiones de JDK para los más populares y conocidos sistemas operativos (MS-Windows, Linux, Solaris, MacOS System, etc.). Para información sobre nuevos desarrollos sobre JDK puede hallarse en la página de JavaSoft, en: `http://www.javasoft.com` y `http://www.sun.com`.

Estas notas dan una introducción al lenguaje de programación Java. Su objetivo es servir al estudiante como una referencia sencilla y rápida al lenguaje. Su enfoque considera primero presentar elementos de programación básica (como son identificadores, tipos, variables arreglos, operaciones, enunciados de control, etc.) que Java tiene en común con otros lenguajes de programación. En seguida, se presenta cómo Java presenta los conceptos de programación Orientada a Objetos (principalmente, clases, métodos y herencia). Finalmente, se introducen varios tópicos especiales exclusivos del lenguaje Java, que utilizan la programación básica y orientada a objetos (temas como paquetes, interfaces, excepciones, hebras de control y ventanas). Sin embargo, estas notas no describen las bibliotecas de clases de Java. Estas están extensivamente descritas en la documentación de JDK que se encuentra disponible en las páginas de Web anteriormente mencionadas.

Jorge L. Ortega Arjona
Septiembre 2004

Capítulo 2

Programación Básica en Java

2.1. Codificación Java

Un programa en Java consiste de una serie de declaraciones de clases e interfaces, cada una de las cuales se compone de declaraciones de métodos y variables. Estas declaraciones se presentan formando un texto o código que es almacenado en un archivo fuente. Aun cuando las declaraciones de varias clases y/o interfaces pueden ser almacenadas en el mismo archivo fuente, típicamente cada declaración se almacena en un archivo separado. Esto no se debe solo a preferencia, sino que también se relaciona con la forma en que las clases de Java se cargan cuando un programa se ejecuta.

2.1.1. Archivos `.java`

Los archivos fuente siempre tienen nombres con la terminación `.java`. La primera parte del nombre del archivo es el nombre de la clase o interfaz declarada en el propio archivo. Por ejemplo, una clase `Test` se almacenaría en un archivo `Test.java`. Si un archivo contiene la declaración de más de una clase o interfaz, el archivo debe ser nombrado a partir de una de ellas, la cual debe ser la única clase o interfaz pública (`public`) en el archivo.

2.1.2. Comentarios

Los comentarios en Java tienen tres formas:

```
// Este es un comentario de una sola linea
/* Este es un comentario
        multilinea */
```

```
/** Este es un comentario
        para documentacion */
```

Los comentarios para documentación se reconocen mediante la herramienta `javadoc`, la cual lee los comentarios de documentación y automáticamente genera páginas HTML conteniendo una versión limpiamente formateada de la información de los comentarios. Cada página documenta una clase y lista cada variable miembro junto con cualquier información o comentario provisto. Las bibliotecas de Java se documentan usando los comentarios para documentación y la herramienta `javadoc`.

Los comentarios para documentación empiezan con una marca `/**` y terminan con `*/`. Pueden extenderse por múltiples líneas, pero no pueden ser anidados. Sólo los comentarios para declaración inmediatamente antes de la declaración de clases, interfaces, variables miembro y métodos se reconocen por `javadoc` como comentarios para documentación. En cualquier otro lado, un comentario para documentación se considera solo como un comentario multilínea.

Dentro de un comentario para documentación pueden aparecer varias etiquetas que permiten procesar la información del comentario en formas específicas por la herramienta `javadoc`. Cada etiqueta se marca por un símbolo `@`, y debe comenzar en una línea aparte. Las siguientes etiquetas se encuentran disponibles:

- `@author`. Nombre del o los autores del código que se comenta. El nombre del autor se escribe simplemente como texto:

  ```
  @author Daniel Lopez
  ```

  Pueden utilizarse varias etiquetas con diferentes autores, o varios nombres pueden ser considerados en la misma etiqueta:

  ```
  @author Daniel Lopez, Javier Jimenez
  ```

- `@deprecated`. Se usa para indicar que la siguiente clase o método se ha cambiado de una versión anterior del código, y será removida en versiones futuras. La etiqueta puede ser seguida de una corta explicación:

```
@deprecated  No sera disponible en siguientes versiones
```

Idealmente, `@deprecated` debe seguirse de una etiqueta `@see` que dirija al lector al punto de reemplazo. Esta etiqueta es adicionalmente reconocida por el compilador de Java, lo que genera un mensaje de advertencia (*warning*) si el código se utiliza.

- `@exception`. Provee de información sobre las excepciones que pueden arrojarse a partir de un método. Un nombre de excepción puede aparecer después de la etiqueta seguida de un comentario de texto.

```
@exception IndexOutOfBoundsException Intento de accesar
          un elemento invalido
```

Un comentario para documentación puede contener varias etiquetas `@exception`.

- `@param`. Provee información sobre parámetros de métodos y constructores. La etiqueta es seguida del nombre del parámetro y de un comentario:

```
@param size Tamano de una estructura de datos
```

Estas etiquetas solo deben aparecer en comentarios precediendo métodos y constructores. Idealmente, debe haber una etiqueta por cada parámetro, presentándose en el mismo orden de los parámetros.

- `@return`. Documenta el valor de retorno de un método.

```
@return El indice de un elemento identificado
```

El comentario es texto simple. Un comentario para documentación debe solo contener una etiqueta `@return`, ya que solo puede haber un solo valor de retorno.

- `@see`. Provee una referencia cruzada a otra clase, interfaz, método, variable o URL. Las siguientes son referencias válidas:

```
@see java.lang.Integer
@see Integer
@see Integer#intValue
@see Integer#getInteger(String)
@see <a href="info.html">Vease aqui para mayor informacion</a>
```

Las clases e interfaces pueden ser referenciadas tanto por su nombre como por el nombre completo del paquete al que pertenecen. Las variables y métodos miembro pueden referenciarse mediante añadir su nombre al nombre de la clase siguiendo el símbolo `#`. Los URLs pueden aparecer si se hace un formato utilizando las etiquetas `<a>...</a>` de HTML. Múltiples etiquetas `@see` pueden aparecer en el mismo comentario.

- `@since`. Se usa para establecer cuándo una característica particular fue incluida (por ejemplo, desde cuando se ha hecho disponible). La etiqueta se sigue por un texto dando la información requerida:

  ```
  @since JDK1.0
  ```

- `@version`. Se utiliza para dar información sobre la versión de la revisión actual del código siendo comentado. El formato para la información sobre la versión no está especificada, y se deja al programador. Una convención no oficial que se está utilizando cada vez más es aquélla en que el número de la versión se sigue por la fecha de liberación (*release date*):

  ```
  @version 1.2 20.8.1997
  ```

  Solo una etiqueta de versión puede aparecer dentro de un comentario.

El texto incluido en un comentario para documentación puede también marcarse con etiquetas HTML usadas para controlar la apariencia del texto, incluyendo `<code>...</code>` (que delimita el texto utilizando la letra para código de programación), y `<p>` (que indica el comienzo de un nuevo párrafo, y que se utiliza frecuentemente también como delimitador de la forma `<p>...</p>`). Consúltese libros sobre HTML para detalles referentes a la notación de ese lenguaje.

La siguiente declaración de una clase ilustra el uso de los comentarios para documentación:

```
/** Esta es una clase de prueba para mostrar el uso de comentarios para
documentacion.
@author Daniel Lopez
@see java.lang.String
@version 1.0 10.8.1997
*/

public class Comment1 {

        /** Retorna el argumento multiplicado por 2.
        @param x El valor entero a ser duplicado.
        @return El argumento del metodo multiplicado por 2.
        @deprecated Sera removido en la siguiente version de la clase.
        @see #multiplyByTwo
        */
        public int times2 (int x) {
                return x*2;
        }

        /** Retorna el argumento multiplicado por 2, reemplazando
        la version anterior de este metodo.
        @param x El valor entero a ser duplicado.
        @return El argumento del metodo multiplicado por 2.
        */
        public int multiplyByTwo (int x) {
                return 2*x;
        }

        /** La funcion main usada para probar la clase.
        @param args argumentos de linea de comando (no usados)
        */
        public static void main(String[] args) {
                Comment1 test = new Comment1();
                int n = test.times2(5);
                n = test.multiplyByTwo(10);

                System.out.println(s);
        }

        /** Una cadena de caracteres utilizada por la funcion
        <code>main</code>.
        */

        static String s = "Hello";
}
```

2.2. Identificadores

Un identificador (o simplemente un nombre) en Java es básicamente una secuencia de caracteres alfabéticos o dígitos que comienza con un caracter alfabético. Los identificadores pueden incluir también el subrayado (*underscore*: _) y `$` (aun cuando `$` no debe ser utilizado en código normal).

Ejemplos son:

`result`, `PI`, `xyz123`, `unNombreMuyLargoQueResultaDificilDeEscribir`

Los identificadores no pueden incluir espacios ni ser ninguna de las palabras clave o literales de Java, ya que tales palabras clave o literales tienen un significado particular dentro del lenguaje. Estas son:

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

Algunos compiladores de Java también reconocen `const` y `goto` como palabras clave o reservadas. Además, unos cuantos valores literales tienen nombres que deben considerarse también como reservados. Estos son `true`, `false` y `null`.

2.3. Valores

Los valores en Java pueden ser de los siguientes tipos:

- Números enteros: `int` y `long`
- Números de punto flotante: `float` y `double`
- Valores booleanos: `boolean`
- Caracteres: `char`
- Cadenas de caracteres: `String`
- Referencia nula: `null`

Un valor puede ser utilizado en cualquier expresión donde su uso sea del tipo correcto.

2.3.1. Valores enteros

Los valores decimales de tipo `int` (32 bits) pueden escribirse como sigue:

```
0, 123, -456, 555665, 2354545, -3456345
```

Los valores octales de tipo `int` son precedidos por un cero (`0`), seguido por dígitos en el rango de 0 a 7:

```
00, 0123, 0777, -045233, 023232, -01
```

Los valores hexadecimales del tipo `int` se preceden por `0x` o `0X`. Los dígitos hexadecimales pueden representarse por cifras del 0 al 9 y de A a F (o a a a f):

```
0x0, 0X12F, -0xffed, 0XFFFF, 0x12def1, 0xff342232
```

Los valores de tipo `long` (64 bits) se denotan por añadirles una `L` o `l` a cualquier representación de un entero:

```
0L, 0l, 1223434545L, 0xffeeL, 077L, -345435431,
-0451, -0XC0B0L
```

2.3.2. Valores de punto flotante

La representación de números de punto flotante incluyen un punto decimal. Por defecto, todo número con punto decimal se considera de tipo `double`:

```
1.2345, 1234.432353, 0.1, 3.4, .3, 1., -23.4456
```

Sólo si el número cuenta con el sufijo `F` o `f` será de tipo `float`:

```
1.23f, 2.34F, 0f, .2F, -4.5687F, -4423.223f
```

Un sufijo `D` o `d` puede de forma similar implicar tipo `double`.

Los números de punto flotante pueden ser escritos utilizando el formato con exponente, mediante el uso de `e` o `E`, considerando las reglas anteriormente señaladas al añadir `f`, `F`, `d` y `D`:

```
4.54003e+24, 5.3453E-22, 1.2e5f, -1.978748E+33D, 2.5444e+9f
```

Los valores de punto flotante no pueden ser escritos utilizando notaciones octal o hexadecimal.

2.3.3. Valores booleanos

Existen dos valores booleanos: `true` y `false`. Estos son considerados como palabras clave, por lo que tales nombres no pueden ser utilizados con otro propósito dentro de un programa.

2.3.4. Valores caracter

Un valor caracter representa un solo caracter, el cual es limitado por comillas sencillas:

```
'a', 'A', '!', '3', '{', ' '
```

Los caracteres que no tienen representación visible (como por ejemplo, línea aparte o tabulador) requieren una secuencia que se denota mediante el uso de \ seguido por un caracter:

- '`\b`': borrado de caracter (*backspace*)
- '`\f`': alimentación de página (*formfeed*)
- '`\n`': línea aparte (*newline*)
- '`\r`': retorno de carro (*carriage return*)
- '`\t`': tabulador (*tab*)

Algunos signos de puntuación requieren también secuencias especiales:

- '`\\`': diagonal invertida (*backslash*)
- '`\'`': comilla sencilla (*single quote*)

Las secuencias pueden utilizarse también para especificar código octal representando caracteres directamente. Los códigos válidos van de 000 a 377, y no requieren ser precedidos por el cero:

```
\123, \167, \023
```

Las secuencias Unicode también pueden ser utilizadas como valores caracter. Nótese, sin embargo, que dado que las secuencias son procesadas primero, pueden darse algunas sorpresas. Por ejemplo, si en Unicode el caracter representando alimentación de línea aparece en un valor caracter (`'\u000a'`) será traducido a un fin de línea en el texto del programa, y no la alimentación de línea que se esperaba.

2.3.5. Valores de cadenas de caracteres

Los valores de cadenas de caracter se encierran en comillas dobles, y se almacenan como objetos `String` cuando el programa se ejecuta. Pueden incluir caracteres especiales (comúnmente, la línea aparte):

```
"Esta es una cadena", "Hola Mundo\n", "Uno\tDos\tTres"
```

2.3.6. Valor nulo

Este valor aparece simplemente como `null`, y como los valores booleanos, se considera efectivamente como una palabra reservada.

2.4. Tipos

Un tipo es un nombre que representa un conjunto de valores. Java tiene dos categorías de tipos: tipos primitivos y tipos de referencia. Por cada tipo hay un conjunto de operadores para los valores de tal tipo (por ejemplo, `+`, `-`, `*`, etc.).

Java es un lenguaje fuertemente tipificado, lo que significa que toda expresión, valor y objeto debe tener un tipo, y que el uso de tipos debe ser consistente. El compilador de Java refuerza el uso correcto de tipos mediante una rigurosa verificación de tipos.

2.4.1. Tipos Primitivos

Los tipos primitivos reflejan los tipos básicos que se soportan directamente por el procesador de una computadora típica, haciéndolos los tipos más eficientes a ser usados. Las operaciones sobre estos tipos usualmente se mapean directamente a instrucciones del procesador. Dicho eso, Java define cuidadosamente una representación estándar para cada uno de estos tipos, de tal modo que se comporten exactamente igual en cualquier máquina que soporte la Máquina Virtual de Java (*Java Virtual Machine*, o JVM). La mayoría de los procesadores actuales consideran tales representaciones directamente. Los tipos primitivos se representan como sigue:

- `boolean`. El tipo `boolean` simplemente tiene dos valores: `true` y `false`.
- `byte`. Enteros de 8 bits signados con complemento a 2, con rango de valores entre -128 y 127.
- `short`. Enteros de 16 bits signados con complemento a 2, con rango de valores entre -32768 y 32767.
- `int`. Enteros de 32 bits signados con complemento a 2, con rango de valores entre -2147483648 y 2147483647.
- `long`. Enteros de 64 bits signados con complemento a 2, con rango de valores entre -9223372036854775808 y 9223372036854775807.
- `char`. Valores de 16 bits no signados, de 0 a 65535, representando caracteres Unicode.
- `float`. Valores de punto flotante de precisión sencilla, con formato IEEE 754 de 32 bits, con un rango de valores entre 1.40239846e-45 y 3.40282347e+38.
- `double`. Valores de punto flotante de doble precisión, con formato IEEE 754 de 64 bits, con un rango de valores entre 4.9406564581246544e-324 y 1.79769313486231570e+308

Hay otros tres valores especiales de tipo flotante: "infinito positivo" (*positive infinity*), "infinito negativo" (*negative infinity*) y "no un número" (*not a number*, o `NaN`). Estos se generan cuando una operación de punto flotante sobrepasa la capacidad de la computadora (*overflow*), u ocurre una división entre cero.

Los operadores definidos para cada uno de estos tipos primitivos se describen más adelante.

La biblioteca de clases de Java provee (en el paquete `java.lang`) una clase por cada tipo primitivo descrito anteriormente, lo que permite que estos tipos se representen mediante objetos. Las clases son `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float` y `Double`.

Existe aun un tipo más que pertenece a los tipos primitivos del lenguaje. Este es el tipo `void` (`void` es una palabra reservada). No existen valores de tipo `void`, así que no tiene representación. El único uso de `void` es para la declaración de métodos que no retornan ningún resultado (o para ser mas precisos, retornan un valor de tipo `void` para el cual no hay representación, y por lo tanto, no requiere forma de almacenamiento en memoria).

2.4.2. Tipos de Referencia

En Java, hay tres tipos de referencia: clases, interfaces y arreglos. Toda expresión que haga referencia a un objeto tiene un tipo de referencia. Los objetos en sí mismos pueden conformarse de uno o más tipos de referencia.

- Una clase se crea cuando se declara, utilizando la palabra clave `class`. El nombre del tipo será el mismo que el nombre de la clase.
- Una interfaz se crea cuando se declara, utilizando la palabra clave `interface`. El nombre del tipo será el mismo que el nombre de la interfaz.
- Un arreglo se especifica cuando se declara un nombre determinado por un tipo, precedido (o en ocasiones, seguido) de corchetes (`[]`). Por ejemplo, `int [] i;` tiene un tipo `array` de `int`.

Los nombre de los tipos son en realidad identificadores.

Por último, el tipo de referencia `String` se da para un reconocimiento especial dentro del lenguaje debido a la necesidad de soportar valores de cadenas de caracteres, aun cuando existe una clase `String` en las bibliotecas de clases.

2.4.3. Conversión Automática de Tipos

Java provee de una variedad de conversiones automáticas entre tipos que permite una mayor flexibilidad al escribir expresiones. Por ejemplo: es posible asignar un valor de tipo `int` a una variable de tipo `long`. Esto permite que todo `int` se convierta en `long` sin pérdida de información. La asignación opuesta, de `long` a `int`, no se permite pues puede potencialmente perderse información, ya que el rango de `long` es mayor que el rango de `int`. Las conversiones como ésta pueden forzarse utilizando el mecanismo de `cast` en las expresiones, siempre y cuando el programador sea consciente de que potencialmente puede perderse información.

Los siguientes tipos de conversión automática se soportan:

- Conversiones de Ampliación de Primitivas: Estas convierten de un tipos a otro considerando que no hay pérdida de información. Las conversiones permitidas son:

`byte`	`short`, `int`, `long`, `float` o `double`
`short`	`int`, `long`, `float` o `double`
`int`	`long`, `float` o `double`
`long`	`float` o `double`
`float`	`double`

- Conversiones de Ampliación de Referencias: Estas permiten que una referencia de una subclase sea tratada como una referencia del tipo de su superclase.
- Conversiones de Cadenas de Caracteres: cuando el operador `+` (concatenación de cadenas) tiene un argumento de tipo `String`, cualquier otro argumento puede ser convertido a tipo `String`.

Las conversiones se realizan principalmente durante operaciones de asignación, y cuando los valores se pasan como parámetros o argumentos de un método. Esto permite al programador escribir enunciados como:

```
short a = 12;
long b = a;
```

Es posible hacer esto aun cuando a primera vista `a` y `b` son de tipos diferentes. Cuando este código es compilado, el compilador insertará las conversiones necesarias automáticamente y sin comentarios. La mayor parte del tiempo esto no causará problemas, pero ocasionalmente puede generar sorpresas, particularmente en el caso de las conversiones de referencias.

Se habla del uso de conversiones cuando se utiliza la frase "compatibilidad por asignación" (*assignment compatible*), de tal forma que en el ejemplo inicial, todo `int` es compatible por asignación con todo `long`.

2.5. Entornos (*Scope*)

Un programa puede tener los siguientes entornos:

- *Entorno Global*. Los nombres declarados en este entorno son accesibles para cualquier enunciado dentro de todo el programa. Solo los nombres de "paquetes" (*packages*) son globales.
- *Entorno de Paquetes*. Un paquete (*package*) define un entorno y una unidad de encapsulación. Los nombres declarados dentro de un paquete pueden ser globales para el paquete mismo, y pueden hacerse públicos (`public`), de tal modo que puedan ser accesados por los usuarios del paquete. Los paquetes no pueden ser anidados, es decir, estar uno dentro de otro, pero un paquete puede usar cualquier otro.
- *Entorno de Unidad de Compilación*. Una unidad de compilación es, en efecto, un sinónimo de archivo fuente. Los nombres pueden ser globales sólo dentro del archivo. Todos los contenidos de cualquier archivo son siempre parte de un paquete.
- *Entorno de Clase*. Los nombres dentro de una clase son accesibles a cualquier parte de la misma clase o de sus subclases si son declarados públicos (`public`) o protegidos (`protected`). Más aún, si los nombres son declarados como públicos, pueden ser accedidos por los usuarios de la clase, o si no se les especifica como públicos, protegidos o privados (`private`), pueden ser accedidos por otras clases dentro del mismo paquete.
- *Entorno Local*. Un enunciado compuesto (aquel enunciado delimitado por { y }, y que contiene una secuencia de otros enunciados) define un entorno local. Los cuerpos de métodos y enunciados de control son también entornos locales. Los entornos locales pueden ser anidados a cuantos niveles como sea necesario (por ejemplo, un entorno local puede ser declarado dentro de otro entorno local). Los nombres declarados en un entorno local son únicamente accesibles dentro de tal entorno o en cualquiera de sus entornos anidados. Más aun, los nombres son sólo utilizables desde el punto textual de declaración y hasta el fin del entorno, es decir, no puede usarse un nombre antes de que haya sido declarado en el texto (nótese la diferencia con el entorno de clase).

Los entornos son siempre anidados entre sí, siendo el entorno global el de más externo nivel. Un nombre declarado en un entorno anidado o contenido es accesible por el entorno anidado o contenido, a menos que sea "oculto" por otra declaración que use el mismo nombre.

2.6. Declaración de Variables

La sintaxis básica para declarar variables es:

tipo identificador;
ó
tipo identificador = *expresión*;

El primero representa una variable con inicialización por defecto, y el segundo es una variable inicializada explícitamente. En ciertos puntos del programa, la declaración de variables se precede por una o más de las palabras clave:

```
final, static, transient, volatile
```

Es posible declarar dos o más variables del mismo tipo en la misma declaración:

tipo identificador1, *identificador2*, ...;

Las variables dentro de la lista pueden a su vez ser inicializadas.

Existen en Java dos categorías de variables:

1. Variables de un tipo primitivo que directamente contienen una representación de un valor de tal tipo primitivo.
2. Variables de tipo de referencia que guardan una referencia a un objeto "conformado" con el nombre del tipo de la variable `null` (que es la referencia nula).

La declaración de una variable introduce tal variable al entorno actual, dándole un nombre y tipo. Todas las variables deben ser declaradas antes de ser usadas.

Además de las dos categoríasde variables, varios tipos de variables pueden ser identificadas basándose en cuándo son declaradas:

- Variables de Clase. Son aquellas declaradas como estáticas (`static`) dentro de una clase. Estas variables se comparten por todas los objetos como instancias de tal clase.
- Variables de Instancia. Son declaradas dentro de una clase. Cada objeto, como instancia de la clase, tiene su propia copia de cada variable de instancia declarada.
- Parámetros de métodos y constructores. Se declaran dentro de la lista de parámetros de un método o constructor.
- Parámetros de manejo de excepciones. Se declaran dentro de la lista de parámetros de la cláusula `catch`.
- Variables Locales. Se declaran dentro del entorno local, usualmente un enunciado compuesto, e incluso en el cuerpo de los métodos.

Una variable se declara mediante especificar su tipo y su nombre, donde el nombre es un identificador, siguiendo las reglas de construcción de identificadores. Por ejemplo, las siguientes variables son declaradas con una inicialización por defecto:

```
int j;
long d;
```

Además, en la situación adecuada, una variable puede ser declarada como `final`, `static`, `transient` o `volatile`, con la siguiente intención:

- `final`. La variable es considerada como constante, así que su valor no puede ser cambiado.
- `static`. La variable es una variable de clase, compartida por las instancias de la clase.
- `transient`. Una variable de instancia es declarada para no ser parte del estado persistente de los objetos (por ejemplo, si el estado del objeto es salvado, el valor de la variable transitoria no se incluye).
- `volatile`. Se necesita cuando una variable de instancia se usa por hebras de control (*threads*) para prevenir que el optimizador del compilador lo accese.

De todas estas, la palabra clave `final` es la más importante, ya que da a Java un mecanismo para declarar constantes. Variables de instancia, variables locales, parámetros de métodos y parámetros de manejo de excepciones pueden ser declarados como `final`.

Es posible declarar varias variables en la misma declaración, usando una lista separada por comas:

```
int a, b, c;
```

El nombre de una variable es un identificador. Por convención, las variables usualmente comienzan con una letra minúscula, y debe reflejar el propósito de la variable. Por lo tanto, nombres de variables como `position`, `weight` o `color` se prefieren sobre `p`, `w`, o `c`.

2.7. Inicialización de Variables

Los requerimientos de inicialización de variables varían dependiendo del tipo de variable.

2.7.1. Variables de Clase e Instancia

La práctica general es explícitamente inicializar estas variables. Sin embargo, para cumplir el requerimiento de que toda variable debe ser siempre inicializada, Java provee valores por defecto de los tipos primitivos y de referencia cuando se declaran sin inicialización explícita. El programador por lo tanto tiene la alternativa de confiar en los valores por defecto o proveer de una expresión de inicialización explícita. Se utilizan los siguientes valores por defecto:

- `byte: (byte) 0`
- `short: (short) 0`
- `int: 0`
- `long: 0l`
- `float: 0.0f`
- `double: 0.0d`
- `char: '\u0000'` (el caracter nulo)
- `boolean: false`
- tipos de referencia: `null`

Si una variable de referencia es inicializada a ser `null`, mantiene tal valor sin referenciar ningún objeto, hasta ser inicializada o definida.

Las variables de clase e instancia pueden ser explícitamente inicializadas en tres formas:

1. Mediante dar una expresión de inicialización seguida de un signo `=` después del nombre de la variable:

```
int i = 100;
char c = 'd';
float f = 10.345;
```

2. Por el uso de un inicializador de instancia, el cual es un enunciado compuesto colocado inmediatamente después de la declaración de la variable, y que contiene enunciados que inicializan la variable:

```
int i;
{i = 100 * 2 + 5}
```

3. Por asignación en el cuerpo de un constructor.

En el caso de una lista de variables en la declaración, cada variable de la lista puede explícitamente inicializarse usando expresiones de inicialización separadas:

```
int a = 10, b, c = 5;
```

2.7.2. Variables Parámetro

Estas variables son siempre inicializadas para ser una copia del argumento (el valor usado en el método correspondiente o llamada a constructor). No hay control del programador sobre este mecanismo. Nótese que son objetos que se pasan por referencia, de tal modo que el objeto referenciado es copiado, y no se trata del propio objeto.

2.7.3. Variables Parámetro de Manejo de Excepciones

Estas variables siempre se inicializan como copia del valor arrojado por una excepción.

2.7.4. Variables Locales

Todas las variables locales deben, ya sea directa o indirectamente, ser explícitamente inicializadas antes de ser usadas. Esto debe hacerse por una expresión de inicialización o inicializador:

```
{
        int i = 10;
        ...
}
```

o por una asignación hecha de la variable antes de ser usada en cualquier otra expresión:

```
{
        int i;
        ...     // donde no hay enunciados que utilicen i
        i = 10; // asigna un valor a i
        ...     // i puede ser ahora usada
}
```

El compilador de Java verifica que una inicialización o asignación apropiada haya sido hecha antes del uso de la variable, reportando un error si no se ha realizado de esta manera.

2.7.5. Variables Finales

Normalmente una instancia `final` o variable local se inicializan como parte de una declaración, como se muestra anteriormente. Sin embargo, una variable `final` vacía (*blank final*) se permite también, en la cual la inicialización es diferida. Antes de usar una variable `final` vacía, ésta debe ser inicializada por asignación, tras lo cual no podrá ser modificada.

El siguiente programa muestra algunas variables locales inicializadas y usadas. Nótese el uso de la variable `final` vacía:

```
import java.util.Date;
import java.util.Vector;

public class Var1 {
        public static void main(String [] args) {
                // Inicializacion de variables con expresiones
                int i = 1;
                String s = "Hola";
                Date d = new Date();
                final Vector v = new Vector(); // v es una constante
                int j; // j se declara pero no se inicializa aqui
                       // j no puede usarse aqui
                j = 4; // j se asigna aqui antes de ser usada
                       // j puede usarse aqui. Si no ha sido asignada
                       // el compilador reportara un error en esta linea

                System.out.println(j);

                final int q; // Declaracion de variable final vacia
                             // Aqui no debe usarse q aun
                q = 10;      // Inicializacion de q
                             // Aqui, el valor de q no puede modificarse
                             // por asignacion

                // q = 20;   // Error, si se quita el comentario

                // A continuacion, la declaracion e inicializacion de
                // tres variables en el mismo enunciado. Notese que
                // para inicializar valores float, tales valores
                // deben incluir f o F, o el compilador asumira que son
                // doubles, por lo que habra un error de tipos

                float a = 1.0f, b = 2.0f, c = 3.0f;
        }
```

```
        // Declaracion e inicializacion de una variable de clase
        static int m = 10;

        // Declaracion e inicializacion de una variable de instancia,
        // usando un enunciado inicializador
        int n;
        { n = m * 2 }
}
```

2.8. Arreglos de Variables

Los arreglos son contenedores de objetos y pueden almacenar un número fijo de elementos (o valores) que pueden ser accesados mediante indexado del propio arreglo. Una variable del arreglo mantiene una referencia al objeto arreglo que es creado por una expresión de inicialización del arreglo. Los corchetes se utilizan para denotar arreglos:

```
int [] x = ...;
```

donde se declara una variable `x` a ser una variable de tipo `array` de `int`.

Como en el caso de las variables simples, los arreglos pueden declararse e inicializarse en expresiones de la forma:

tipo [ ] *identificador*;
ó
tipo [ ] *identificador* = *expresión de inicialización del arreglo*;
ó
tipo identificador [ ];
ó
tipo identificador [ ] = *expresión de inicialización del arreglo*;

Arreglos multidimensionales pueden ser declarados mediante repetir pares de corchetes (`[ ]`) para la dimensión requerida.

Inicializadores de arreglos también son premitidos:

tipo identificador [ ] = **{** *lista de inicialización*};

Esto, en realidad, es la abreviación de:

tipo identificador [ ] = **new** *tipo* [ ] **{** *lista de inicialización*};

El tamaño del arreglo no se especifica en la declaración del tipo arreglo, pero se requiere cuando la variable es inicializada. El tamaño debe expresarse siempre mediante un valor (o variable) de tipo entero `int`:

```
char [] x = new char[10];
```

Esta expresión crea un arreglo de diez objetos de tipo `char`. Una vez creado, el tamaño del arreglo no puede cambiarse, pero la variable arreglo puede ser reasignada para referenciar un arreglo del mismo tipo, pero tamaño diferente. El ejemplo anterior muestra que declarar una variable arreglo es diferente de realmente crear una variable arreglo. Como las variables sencillas, las variables arreglo deben ser siempre inicializadas antes de ser usadas.

Pueden declararse arreglos de cualquier tipo (incluso, tipo arreglo), sobre los cuales el compilador de Java realiza una verificación de tipos para asegurarse que los valores del tipo correcto se almacenan en el arreglo:

```
String [] palabras = new String[100];
```

Después de que un arreglo se ha creado, es posible determinar el número de elementos que puede contener utilizando la expresión:

```
int longitud = palabras.length;
```

donde `length` es una variable de instancia que representa el tamaño o longitud del arreglo.

Arreglos multidimensionales pueden declararse mediante añadir conjuntos de corchetes. Por ejemplo, considérense las siguientes declaraciones e inicializaciones para arreglos de dos, tres y cuatro dimensiones:

```
int [][] dosD = new int [10][10];
// arreglo de 10 por 10

float [][][] tresD = new float [8][10][12];
// arreglo de 8 por 10 por 12

boolean [][][][] cuatroD = new boolean [4][8][12][16];
// arreglo de 4 por 8 por 12 por 16
```

Un arreglo multidimensional es efectivamente una arreglo de arreglos (y de arreglos, de arreglos, etc.), así que cada dimensión se representa por otro objeto arreglo.

El tamaño de un arreglo mutidimensional puede también determinarse por la variable `length`:

```
int longitud = dosD.length;
```

En este caso, se retorna el número de los elementos de la primera dimensión (el cual es de 10, utilizando la declaración anterior de `dosD`). La longitud de las demás dimensiones, que se representa por objetos arreglo, puede encontrarse usando una expresión de indexación para retornar un elemento del arreglo, y preguntando por su longitud:

```
int longitud = dosD[1].length; // longitud de la segunda
                               // dimension
```

Cuando se declaran arreglos multidimensionales, debe pensarse cuidadosamente en el tamaño total y la resultante cantidad de memoria usada por el arreglo. Es fácil declarar un arreglo que use una gran cantidad de memoria.

En los ejemplos anteriores se ha definido el tamaño del arreglo mediante valores enteros. De hecho, una expresión entera con valor puede ser usada para especificar el tamaño de un arreglo:

```
double [][] d = new double [renglones][columnas];
```

Los arreglos multidimensionales se almacenan en forma de "renglones". Para un arreglo bidimensional. Esto significa que la primera dimensión indica el número de renglones, y la segunda dimensión la longitud de cada renglón (o sea, el número de columnas). Para arreglos con más dimensiones, se aplica el mismo principio básico.

De la misma manera en que las variables de tipos primitivos pueden ser inicializadas directamente, también lo son los elementos de un arreglo. El siguiente ejemplo muestra la sintaxis necesaria:

```
int [] n = {1,2,3,4,5};
```

Una lista de inicialización se delimita por llaves ({ }), con un valor para cada elemento del arreglo separado por comas. Nótese que las llaves aquí no denotan un enunciado compuesto. El ejemplo anterior crea un arreglo de tamaño 5, determinado por la longitud de la lista de inicialización, e inicializa cada elemento siguiendo el orden descrito por los valores de inicialización. Cuando se usa una lista de inicialización o usando `new`, la creación del arreglo es implícita, por lo que no requiere ser hecha por el programador.

Los arreglos multidimensionales pueden ser inicializados mediante listas anidadas de inicialización:

```
int [][] m = { {1,2,3,4},{4,5,6,7} };
```

Esto crea un arreglo de 4 por 4 con el primer renglón inicializado por los valores 1, 2, 3, y 4, y el segundo renglón por 4, 5, 6 y 7. Si se da un conjunto incompleto de elementos:

```
int [][] m = { {1,2},{4,5,6,7} };
```

la inicialización del arreglo será aún válida, pero dará un arreglo de tamaños dispares, únicamente con las posiciones de los elementos inicializados a ser accesibles. El intento de accesar cualquier elemento en una posición no inicializada genera una excepción.

Basado en la sintaxis anterior, es posible hacer inicializaciones de arreglos anónimos (*anonymous arrays*), lo que permite la declaración e inicialización de arreglos fuera de una declaración de variables. Por ejemplo, el siguiente enunciado declara un arreglo anónimo y lo muestra utilizando un comando de impresión a pantalla:

```
System.out.println(new char [] {'h','o','l','a'});
```

Lo que hace que la palabra `hola` aparezca en la pantalla. En general, la declaración e inicialización de un arreglo anónimo puede aparecer en cualquier momento en que se hace una referencia a un arreglo del mismo tipo:

```
        int [] n;  // Inicializacion por defecto
        ...
        n = new int [] {5,6,7}; // Creacion de un arreglo anonimo y
                                // asignacion como referencia de n
```

Si un arreglo no es directamente inicializado, cada elemento del arreglo es inicializado automáticamente al valor por defecto dependiendo de su tipo.

El siguiente ejemplo provee de más ejemplos de declaración e inicialización de arreglos:

```
public class Arreglo1 {
    public static void main(String[] args) {
        int [] i = new int [5];  // Un arreglo de 5 elementos
        i = new int [10]; // Es posible asignar un nuevo
                          // arreglo a i
        // Inicializacion de arreglos de cadenas de caracteres
        String [] palabras = {"Hello","World"};

        // Inicializacion de arreglos 2D de valores double
        double [][] matrix = {{1.2, 4.7, 3.4 },
                              {0.6, -4.5, 9.2},
                              {10.9, 0.47, -0.98} };
    }
}
```

Finalmente, queda aún el tema de indexado en arreglos. Sin embargo, este tema se difiere para cubrirse en la sección de operadores.

2.9. Expresiones Primarias

Las expresiones primarias son bloques básicos para la construcción de programas, que permiten la búsqueda de valores que serán manipulados por otras expresiones. Las siguientes son expresiones primarias:

- Las palabras clave `this` y `super` (que serán descritas más adelante), así como el valor `null`
- Un valor literal
- Una expresión con paréntesis:

  (*expresion*)

- Una expresión de campo, usando '.':

  identificador
  ó
  expresión.*identificador*
  ó
  paquete.*identificador*

- Una expresión indexada de arreglo:

  término [*expresión entera*]

  en donde un término es cualquier expresión, excepto una expresión de alocación

- Una expresión de llamada de *método* (que será descrita más adelante)
- Una expresión de alocación

  new *tipo* (*lista de argumentos*)
  ó
  new *tipo* [*expresión entera*]

Las expresiones de campo permiten el acceso a variables campo o miembro de los objetos, clases y paquetes, y serán descritan también más adelante. Una expresión de campo puede usar un caracter '.', como en:

```
objeto.variable;
```

En este ejemplo, el lado izquierdo es esencialmente una referencia a `objeto`, y el caracter `'.'` se utiliza para seleccionar una `variable` con nombre perteneciente a tal objeto (como se define por su clase).

`null` evalúa la referencia nula y puede ser usado para probar la variable de un tipo referencia para ver si tiene el valor `null`:

```
if (variable == null ) ...
```

Las expresiones con paréntesis permiten la agrupación de expresiones juntas en la misma forma en que las expresiones matemáticas se agrupan. Más aun, esto da un control explícito del orden en que las sub-expresiones de una expresión son evaluadas. Por ejemplo, normalmente la expresión `a + b * c` podría ser evaluada por precedencia de operadores mediante hacer primero la multiplicación. Usando los paréntesis, es posible forzar la adición primero: `(a + b ) * c`. El valor de una expresión con paréntesis es la misma que el valor de la expresión dentro de ella, así como su tipo.

Las expresiones de indexación de arreglos se utilizan para localizar elementos de un arreglo usando la notación de corchetes. Cuando se usan en una expresión que evalúa una referencia a objeto arreglo (comúnmente, una variable de algún tipo de arreglo), el valor de retorno es una variable que puede ser asignada o buscarse su valor. Una expresión de indexado de arreglos típica tiene la forma:

```
a[1] = 10; // Asigna un valor a una variable indexada
x = a[2];  // Busca el valor de una variable indexada
```

Para evaluar una expresión de indexación, primero la expresión o identificador de la variable a la izquierda de un corchete de apertura (`[`) es evaluada para obtener la referencia del objeto arreglo. Si tal evaluación retorna un valor `null`, se genera una excepción `NullPointerException`. De otro modo, la expresión dentro de los corchetes, que debe evaluarse como de tipo `int`, se evalúa para obtener un índice entero. El valor del índice debe ser dentro del rango de cero a la longitud del arreglo menos uno (el indexado de arreglos comienza en cero, es decir, los arreglos tienen origen cero). Si el valor del índice se encuentra dentro del rango, la variable indexada (un elemento del arreglo) se retorna. Si esto no es así, se genera la excepción `ArrayIndexOutOfBoundsException`.

El indexado de un arreglo multidimensional sucede de la misma manera, mediante repetidamente evaluar las expresiones de indexación de izquierda a derecha. Por ejemplo:

```
int [][][] a = ...;
...
a[1][2][3] = 24;
```

La primera expresión de indexación `a[1]` arroja una expresión del tipo `int [][]` para la cual la segunda expresión puede ser evaluada. Esta, a la vez, arroja

una expresión de tipo `int []`, la cual retorna el valor requerido de la variable cuando la expresión de indexación del tercer arreglo es evaluado.

Las expresiones de alocación usan la palabra clave `new` para crear un objeto o arreglo. La palabra `new` es seguida por el tipo (ya sea un tipo referencia o primitivo) y opcionalmente una lista de parámetros en paréntesis (que pueden estar vacíos), o la creación de un arreglo utilizando corchetes.

2.10. Operadores

Los operadores con más alta precedencia dentro de una expresión son siempre aplicados primero. La siguiente lista muestra la precedencia de todos los operadores, indexado de arreglos, selección de campo (usando '.'), `cast` y `new`. La lista va de la más alta a la más baja precedencia. Operadores en el mismo renglón tienen la misma precedencia:

```
[] . () ++ --
++ -- + - ~ !
new cast
* / %
+ -
<< >> >>>
<  >  >=  <= instanceof
== !=
&
^
|
&&
||
?
= += -= *= /= %= >>= <<= >>>= &= ^= |=
```

2.10.1. Operadores de Incremento Posfijo

Java cuenta con dos operadores posfijos de incremento (`++`) y decremento (`--`). Dado que son posfijos, aparecen después del operando, incrementándolo o decrementándolo en una unidad. El operando debe tratarse de una variable o elemento de un arreglo. Ambos operandos pueden ser utilizados con tipos entero o de punto flotante. Por ejemplo:

```
byte i = 1;
float j = 10.4;
i++;   // incrementa i
j--;   // decrementa j
       // aqui, i == 2 y j == 11.4
```

Sin embargo, lo que no es obvio dentro de la explicación anterior es que el valor de la expresión de incremento o decremento es el valor original de la variable. El valor actualizado será disponible la siguiente vez que la variable es utilizada. Por ejemplo:

```
int i = 1;
int j = i++;  // j se asigna el valor de 1
int k = i;    // k se asigna el valor de 2
```

Esencialmente, estos dos operadores generan efectos colaterales por actualizar el valor de la variable tras retornar el valor original de la variable.

2.10.2. Operadores Unarios

Los operadores unarios son:

- incremento (`++`) y decremento (`--`) prefijo
- `+` y `-` unarios
- el operador complemento por bit ($\sim$)
- el operador complemento lógico (`!`)

Los operadores prefijos `++` y `--` pueden utilizarse para tipos enteros y de punto flotante, es decir, el operando debe ser una variable o un elemento de un arreglo de tipo entero o punto flotante. Como las versiones posfijas, estos dos operandos tienen efectos colaterales al incrementar o decrementar una variable en una unidad. Sin embargo, a diferencia de los operadores posfijos, el valor de una expresión prefija `++` o `--` se trata del valor actualizado de la variable. Por ejemplo:

```
int i = 1;
int j = ++i;  // j se asigna el valor de 2
int k = i;    // k se asigna el valor de 2
```

El operador unario `+` está disponible para tipos numéricos y retorna el valor de la expresión a la que se aplica sin realizar ninguna operación (similarmente a la operación matemática). Si el tipo del operando es `byte`, `short`, o `char`, al aplicar `+` arroja un tipo `int`, tras la apropiada conversión de tipos.

El operador unario `-` está también disponible para tipos numéricos, y su aplicación resulta en una negación aritmética. Para los tipos enteros, esto se realiza mediante primero convertir el valor del operando a su equivalente `int`, y restándolo entonces de cero. Nótese que, debido a la representación en complemento a dos, este operador no puede producir un valor positivo al negar un valor negativo entero más grande. Para tipos de punto flotante, el operador unario `-` cambia únicamente el signo del operando.

El operador complemento por bit $\sim$ puede aplicarse a tipos de datos enteros, convirtiéndolos en tipo `int` si es necesario. El valor entero es entonces tratado como una colección de bits, cambiando cada cero por un uno y cada uno por un cero. Por ejemplo:

Antes	`11110000110011001010101011111100`
Después	`00001111001100110101010100000011`

Este operador se utiliza para realizar operaciones "de bit" (*bitwise*) sobre números binarios. Es útil cuando se requiere operaciones a nivel de bit, como manipulaciones. Se utiliza como herramienta de programación de sistemas más que de programación de aplicaciones. Por ejemplo, el siguiente método utiliza $\sim$ y el operador de bit `&` para retornar un patrón de bits con los cuatro bits menos significativos (los cuatro últimos) forzados a ser cero:

```
int maskLastFour(int x){
        return x & ~0xf;
}
```

El operador complemento lógico `!` se aplica a valores booleanos y realiza una negación booleana: si el valor del operando es `true`, el resultado arroja un valor `false`; si es `false`, arroja un valor de `true`. Este operador es frecuentemente utilizado en expresiones booleanas asociadas con enunciados `if` y ciclos `while`, como por ejemplo:

```
boolean ready = false;
while(!ready) {
        ...
        if (...) ready = true;
        ...
}
```

Este tipo de programación puede mejorarse con enunciados de control de flujo explícitos, como lo muestra el siguiente ejemplo, el cual realiza la misma acción de control de flujo:

```
while(true){
        ...
        if (...) break;
        ...
}
```

2.10.3. El operador `new`

El operador `new` es utilizado para crear nuevos objetos a partir de una clase. El argumento de `new` debe tener un identificador de tipo. Si se especifica un identificador de clase, `new` puede ser seguido por una lista de parámetros que se utilizan por el constructor. Si un tipo arreglo es especificado, la dimensión o dimensiones del arreglo deben ser dadas. Como palabra reservada, `new` se discute en detalle en otros puntos del texto.

2.10.4. El operador `cast`

Un `cast` permite cambiar del tipo de una expresión, siempre y cuando el cambio no viole las reglas de compatibilidad de tipos. Estas reglas restringen las operaciones

`cast` posibles a aquellas que preserven la intención de los valores, previniendo de aquellas operaciones que modifiquen substancialmente el valor, como sería un cambio de `double` a `boolean`.

Una operación de `cast` válida se muestra en el siguiente ejemplo:

```
float f = 2.3;
int n = (int)f;
```

La operación de `cast` aparece en la segunda línea como `(int)f`. El operando es `f`, mientras que el operador prefijo consiste en un nombre de tipo entre paréntesis. Tal nombre provee del tipo al que el operando debe ser cambiado. El ejemplo utiliza un `cast` para asignar un valor `float` a una variable `int`. La asignación directa de un `float` a un `int` no es posible, pero el `cast` forza a permitirlo. Nótese, sin embargo, que la operación `cast` resulta en una pérdida de información, ya que todos los valores decimales del valor `float` se pierden: el `cast` provoca el truncamiento del valor. Por lo tanto, en el ejemplo anterior, `n` será inicializado al valor de 2.

Donde es posible, la validez de una operación `cast` es verificada durante la compilación (*compile-time*), resultando en un mensaje de error si no es válida. Sin embargo, algunos tipos de `cast` requieren ser verificados en tiempo de ejecución (*run-time*), ya que la información durante compilación resulta insuficiente para determinar si un `cast` es siempre una operación válida. Si durante tiempo de ejecución la verificación falla, se genera una excepción.

2.10.5. Operadores Artiméticos

Los operadores aritméticos son:

- Adición (`+`)
- Sustracción (`-`)
- Multiplicación (`*`)
- División (`/`)
- Residuo (`%`)

Todas estas son operaciones que requieren dos datos, realizando esencialmente las operaciones aritméticas obvias de adición, sustracción, multiplicación y división. El operador residuo (`%`) es menos familiar, pero simplemente retorna el residuo entero de una operación de división.

El aplicar estos operadores puede resultar en una sobrecapacidad (*overflow*) o una subcapacidad (*underflow*), lo que significa que los valores resultantes no pueden ser representados por el hardware. Si esto ocurre, los valores resultantes se retornan utilizando tantos bits como sean disponibles, y no arrojan ninguna excepción. Esto provoca que el valor parezca ser válido, pero no representa el resultado correcto.

Las operaciones de punto flotante (aquellas que involucran tipos `float` y `double`) son algo más complicadas, ya que pueden representar los valores mediante `NaN` (No-un-número o *not-a-number*), que se usa cuando no es posible representar un valor numérico.

2.10.6. Operador concatenación de cadenas de caracteres

El símbolo para el operador concatenación de cadenas de caracteres es `+` (aún otra sobrecarga de este símbolo). El operador es seleccionado cuando uno ó ambos operandos son de tipo `String`. Si solo un operando es de tipo `String`, entonces el otro operando es convertido a tipo `String`. Para los tipos primitivos, esto se lleva a cabo utilizando conversión automática. Para los tipos de referencia, la conversión se realiza mediante invocar la referencia del objeto al método `toString`. Todas las clases heredan el método `toString` de la clase `Object`, y generalmente se elaboran para retornar la representación en cadenas de caracteres apropiada para las instancias de la clase.

El operador concatenación se usa frecuentemente en enuncidados de salida:

```
System.out.println("El valor de i es: " + i);
```

Este enunciado funciona sin importar el tipo de la variable `i`.

2.10.7. Operadores de corrimiento

Los operadores de corrimiento son los siguientes:

- corrimiento a la izquierda (`<<`)
- corrimiento a la derecha (`>>`)
- corrimiento sin signo a la derecha (`>>>`)

Comúnmente, estos operadores de corrimiento se aplican a datos de tipo `int`, que pueden ser manipulados al considerarlos como conjuntos de bits individuales:

```
int a = 1000;
   ...
a << 5;  // a se recorre 5 bits a la izquierda

a >> 10; // a se recorre 10 bits a la derecha,
         // propagando su bit de signo

a >>> 2; // a se recorre 2 bits a la derecha,
         // sin propagar su bit de signo
```

2.10.8. Operadores relacionales

Los siguientes operadores relacionales se encuentran disponibles:

- igual a (`==`)
- no igual a (`!=`)
- menor que (`<`)
- mayor que (`>`)
- mayor o igual que (`>=`)
- menor o igual que (`<=`)
- objeto como instancia de un tipo (`instanceof`)

Todos estos operadores arrojan un valor `boolean`, dependiendo si se cumple o no la relación que representan. Normalmente, se utilizan dentro de los enunciados `if` y `switch`, o en los ciclos `while`, `do` y `for`, para controlar su ejecución (como se explica más adelante).

2.10.9. Operadores Lógicos a bit

Las tres operaciones lógicas se realizan a nivel de bit:

- AND lógico a nivel bit (`&`)
- OR lógico a nivel bit (`|`)
- XOR lógico a nivel bit (`^`)

Cada una de estas operaciones permiten evaluar las operaciones lógicas AND, OR o XOR considerando los operandos como conjuntos de bits individuales. Por ejemplo, el siguiente código imprime una dirección IP de 32 bits que se contiene en una variable `int` llamada `ip_address`, como 4 octetos decimales:

```
for (int i = 3; i >= 0; i--) {
    int octeto = (ip_address >>> (i*8)) & 0xFF;
    System.out.print(octeto);
    if (i != 0)
        System.out.print(".");
}
System.out.println();
```

2.10.10. AND y OR booleanos

Los operadores booleanos son:

- AND condicional booleano (`&&`)
- OR condicional booleano (||)

Estos operadores requieren dos operandos, y pueden ser aplicados a operandos del tipo `boolean`. Del mismo modo, retornan un resultado tipo `boolean`.

La diferencia crucial entre los operadores `&&` y ||, y los operadores `&` y |, recae en la forma en que evalúan sus operandos. `&&` y || primero evalúan su operador del lado izquierdo. Si el valor del operando es suficiente para determinar el valor de toda la expresión, la evaluación del lado derecho no se realiza. Esto significa que cuando una operación `&&` se aplica, el operando del lado derecho se evalúa sólo cuando el operando del lado izquierdo tiene el valor de `true`. Si el operando izquierdo tiene el valor `false`, el valor de la expresión entera debe ser `false`, y no es necesario evaluar el lado derecho para determinar el resultado. Una situación similar se aplica para la operación ||. Si el operando de la izquierda es `true`, entonces la expresión completa debe ser `true`, y el operando de la derecha no necesita ser evaluado.

2.10.11. Operador condicional

El operador condicional es único, ya que se trata de un operador ternario (toma tres argumentos). Esencialmente, se comporta como una versión comprimida del enunciado `if`, y permite la elección en la evaluación de dos expresiones. La estructura es como sigue:

expresiónBooleana **?** *expresiónTrue* **:** *expresiónFalse*

Existen tres operandos: el primero, delante del caracter `?`, es una expresión booleana. El segundo y tercero siguen después, y se separan entre sí por `':'`. Ambas representan expresiones del mismo tipo, o al menos de tipos compatibles.

Una expresión que usa este operador se evalúa como sigue: primero, la expresión booleana es evaluada. Si tiene el valor `true`, la expresión inmediatamente después de `?` se evalúa y su resultado se retorna como el resultado del operador. Si por el contrario, tiene el valor `false`, la expresión después de `:` se evalúa y retorna su resultado como el resultado del operador condicional.

2.10.12. Operadores de asignación

El operador básico de asignación es `=`. Éste es un operador binario (con dos argumentos), que permite asignar el valor del operando de la derecha a una variable o elemento de arreglo que representa el operando de la izquierda. La asignación está sujeta a las conversiones de tipo implícitas y a las reglas del `cast`.

Los otros operadores de asignación son todos binarios, y combinan la asignación con varias operaciones aritméticas o de bit:

```
+=, -=, *=, /=, %=, |=, &=, ^=, <<=, >>=, >>>=
```

Cada operador primero realiza la operación aritmética, lógica o de corrimiento indicada, y en seguida realiza la asignación a la variable correspondiente. Por ejemplo:

```
a += 100; // Significa lo mismo que a = a + 100
x *= 2;   // Significa lo mismo que x = x * 2
```

2.11. Enunciados

2.11.1. El enunciado `if`

if (*ExpresiónBooleana*) *enunciado*

ó

if (*ExpresiónBooleana*) *enunciado*
else *enunciado*

El enunciado `if` permite una selección en tiempo de ejecución del programa acerca de qué líneas de código se ejecutan o se omiten de la ejecución. En el caso de que la evaluación de la expresión booleana sea verdadera (`true`), el código que sigue se ejecuta, y en caso contrario (que se evalue como `false`), se omite. Por otro lado, el enunciado `if` puede extenderse mediante añadirle una parte llamada `else`, que corresponde a una secuencia de código que se ejecuta en caso de que la evaluación de la expresión booleana sea `false`.

El siguiente ejemplo es un programa completo que utiliza el enunciado `if`:

```
class IfTest1 {
    public static void main (String [] args) {
        int a = 1;
        int b = 2;
        if (a < b ) {
            System.out.println("a < b");
        }
        else {
            System.out.println("a >= b");
        }
    }
}
```

La condición de un enunciado `if` puede ser cualquier expresión booleana, y puede incluir llamadas a funciones.

```
class IfTest2 {
    public static boolean f (int x, int y) {
        return (x < y );
    }
    public static void main (String [] args) {
        int a = 1;
        int b = 2;
        if (f(a,b)) {
            System.out.println("f retorna true");
        }
        else {
```

```
            System.out.println("f retorna false");
        }
    }
}
```

2.11.2. El enunciado `switch`

switch (*Expresión***) {**
case *constante char/byte/short/int* **:** *secuencia de enunciados* **break;**
...
default :*secuencia de enunciados*
}

Además del enunciado `if` para la selección de código, existe otro enunciado llamado `switch` que permite escoger una entre varias opciones de secuencias de código a ser ejecutado. Su funcionamiento se basa en considerar una expresión que se va secuencialmente comparando con cada valor constante propuesto (o `case`) como resultado de su evaluación. Si el resultado es verdadero (`true`) se ejecuta el código correspondiente a esa opción. La evaluación se lleva a cabo en el orden en que los valores propuestos se escriben. Si se desea que el código correspondiente a un sólo un valor propuesto sea ejecutado, es necesario añadir un enunciado de control de flujo `break` para que la estructura `switch` no evalúe los demás valores propuestos. Al final, se añade una opción que recoge cualquier otra posibilidad de resultado de la evaluación de la expresión, bajo la etiqueta `default`.

El siguiente programa recibe un entero y da como salida el nombre del día correspondiente al entero, considerando el rango de 1 a 7, y asumiendo 1 como domingo:

```
class Switch1 {
      public static void main (String[] args) {
        KeyboardInput input = new KeyboardInput();
        System.out.print("Escriba un numero entre 1 y 7: ");
        int day = input.readInteger();
        switch (day) {
          case 1: System.out.println("Domingo"); break;
          case 2: System.out.println("Lunes"); break;
          case 3: System.out.println("Martes"); break;
          case 4: System.out.println("Miercoles"); break;
          case 5: System.out.println("Jueves"); break;
          case 6: System.out.println("Viernes"); break;
          case 7: System.out.println("Sabado"); break;
          default:System.out.println(day + " no es dia"); break;
        }
      }
}
```

Si un entero entre 1 y 7 se da como entrada, el nombre del correspondiente día de la semana se escribe en pantalla, utilizando el `switch` para seleccionarlo. Si el número es mayor que 7 o menor que 1, el mensaje por defecto (`default`) se despliega. Cualquiera que sea la entrada, solo el mensaje cuya etiqueta corresponda a la entrada se despliega.

El enunciado `switch` puede utilizar tipos `char`, `byte`, `short`, e `int`. El siguiente ejemplo funciona con `char`:

```
class Switch2 {
      public static void main(String [] args) {
        KeyboardInput input = new KeyboardInput();
        System.out.print("Escriba una vocal y presione retorno: ");
        char c = input.readCharacter();
        switch(c) {
          case'a' : System.out.println("Se escribio 'a'"); break;
          case'e' : System.out.println("Se escribio 'e'"); break;
          case'i' : System.out.println("Se escribio 'i'"); break;
          case'o' : System.out.println("Se escribio 'o'"); break;
          case'u' : System.out.println("Se escribio 'u'"); break;
          default: System.out.println("No es una vocal"); break;
        }
      }
}
```

2.12. Ciclos

2.12.1. El ciclo `while`

while (*expresión booleana*) *enunciado*

El ciclo `while` se utiliza para ejecutar repetidamente una secuencia de líneas de código, mientras la evaluación de una condición booleana se mantenga como verdadera (`true`). El siguiente programa ilustra el uso básico de los ciclos `while`, mostrando dos ciclos que usan la variable `n` para determinar cuándo terminar. El primer ciclo incrementa `n` y cuenta hacia arriba mientras que el segundo ciclo decrementa `n` y cuenta hacia abajo.

```
public class While1 {
    public static void main (String[] args) {
        int n = 0;
        while (n < 10) {
            System.out.println("Contando hacia arriba " + n);
            n++;
        }
        while (n > 0) {
            System.out.println("Contando hacia abajo " + n);
            n--;
        }
    }
}
```

2.12.2. El ciclo `do`

do *enunciado* **while (***Expresión booleana***) ;**

En ocasiones, es necesario que un código que potencialmente puede ser cíclico se ejecute al menos una sola vez, en cuyo caso el ciclo `do` puede utilizarse. El ciclo `do` permite que las líneas de código que encierra se ejecuten al menos una vez, antes de realizar la evaluación de la expresión booleana. Si tal evaluación resulta con un valor verdadero (`true`), entonces el ciclo se vuelve a repetir. En caso contrario, la ejecución continúa con la siguiente instrucción. El siguiente ejemplo muestra dos ciclos `do` típicos:

```
public class Do1 {
    public static void main (String[] args) {
        int n = 0;
        do {
            System.out.println("Contando hacia arriba " + n);
            n++;
        } while (n < 10 );
```

```
        do {
            System.out.println("Contando hacia abajo " + n);
            n--;
        } while (n > 0);
    }
}
```

2.12.3. El ciclo `for`

for (*Expresión inicial*;*Expresión booleana*;*Expresión de actualización*) *enunciado*

El ciclo `for` permite la repetición controlada de una secuencia de líneas de código. La repetición se controla mediante una serie de expresiones: la expresión inicial debe dar como resultado un valor con el cual el ciclo `for` comienza su iteración; la expresión booleana determina la condición de terminación del ciclo, es decir, la ejecución continúa en el ciclo si la evaluación de la expresión booleana es verdadera (`true`) y termina si es falsa (`false`); la expresión de actualización permite cambiar el valor dado por la expresión inicial, idealmente hasta que la expresión booleana se evalúa como falsa.

El primer ejemplo es equivalente a los ejemplos para los ciclos `while` y `do`:

```
public class For1 {
    public static void main (String[] args){
        for (int i = 0; i < 10; i++) {
            System.out.println("Contando hacia arriba " + i);
        }
        for (int i = 10; i > 0; i--) {
            System.out.println(Contando hacia abajo " + i);
        }
    }
}
```

Un uso típico del ciclo `for` es en conjunción con el indexado de un arreglo (o vector). La variable del ciclo es usada para indexar el arreglo:

```
public class For2 {
    public static void main (String[] args) {
        int[] x = new int[10];
        for (int i = 0; i < 10; i++) {
            x[i] = i;
        }
        for (int i = 0; i < 10; i++) {
```

```
            System.out.println(x[i]);
        }
    }
}
```

Finalmente, el siguiente ejemplo muestra un programa que toma una cadena dada como argumento del programa, e imprime una versión convirtiendo los caracteres de mayúsculas a minúsculas:

```
public class For 3 {
    public static void main (String[] args) {
        if (args.length == 0 ){
            System.out.println("No hay cadena que transformar");
        }
        else {
            for (int i = 0; i < args.length; i++ ) {
                String s = args[i];
                char[] result = new char[s.length()];
                for (int j = 0; j < s.length(); j++) {
                    result[j] = Character.toLowerCase(s.charAt(j));
                }
                System.out.println(result);
            }
        }
    }
}
```

2.13. Control de ejecución

2.13.1. `break`

break;
ó
break *etiqueta*;

El enunciado `break` puede ocurrir en cualquier parte dentro de los enunciados `switch`, `for`, `while` o `do`, y causa que la ejecución salte al enunciado siguiente a partir del enunciado más interno donde se aplica. Intentar utilizar el enunciado `break` fuera de cualquiera de los enunciados mencionados causa un error en tiempo de compilación.

El siguiente ejemplo muestra el uso básico de `break` para terminar un ciclo cuando una variable tiene cierto valor. En este caso, el ciclo continúa pidiendo al usuario que escriba un dígito hasta que se da como entrada un `'5'`.

```
public class Break1 {
    public static void main(String[] args){
        KeyboardInput in = new KeyboardInput();
        while (true) {
            System.out.print("Escriba un digito: ");
            int n = in.readInteger();
            if (n == 5) {
                System.out.println("Escribio un 5. Termina");
                break;
            }
        }
    }
}
```

La expresión boolena del ciclo `while` tiene el valor de `true`, de tal modo que tal ciclo potencialmente es infinito. Usando el enunciado `break` se provee de una forma conveniente de salir de un ciclo sin introducir una variable booleana.

2.13.2. `continue`

continue;
ó
continue *etiqueta*;

El enunciado `continue` puede ocurrir dentro de los enunciados `switch`, `for`, `while` o `do`, y causa que la ejecución salte al final del cuerpo del ciclo, listo para iniciar la siguiente iteración. Un intento de utilizar `continue` fuera de los enunciados provocará un error en tiempo de compilación.

El siguiente ejemplo usa `continue` para mostrar que parte del cuerpo de un ciclo puede saltarse fácilmente si es necesario.

```
public class Continue1{
    public static void main (String[] args){
        KeyboardInput in = new KeyboardInput();
        for (int i = 0; i < 10; i++) {
            System.out.print("Escriba un digito: ");
            int n = in.readInteger();
            if (n == 0) {
                System.out.println("Division entre cero");
                continue;
            }
            System.out.println("100/" + n + " = " + 100.0/n);
        }
    }
}
```

Como dividir entre cero no puede producir un resultado válido, el enunciado `continue` se utiliza para saltar el enunciado que contiene la división (si la división entre cero continuara, el resultado sería el valor que representa infinito). Nótese cómo se ha utilizado la constante `double 100.0` (en lugar de la constante `int 100`) para forzar el uso de artimética flotante en la división.

2.13.3. `return`

return;
ó
return *expresión*;

Un enunciado `return` puede aparecer en cualquier parte del cuerpo de un método y en múltiples lugares. Un método no-vacío debe contener al menos un enunciado `return`. El siguiente ejemplo incluye un método `min` que retorna el mínimo de dos argumentos:

```
public class Return1 {
    private static int min (int a, int b){
        if (a < b) return a;
        else return b;
    }
    public static void main( String[] args) {
        System.out.println("El menor de 5 y 8 es: " + min(5,8));
    }
}
```

Capítulo 3

Programación Orientada a Objetos en Java

3.1. Clases

3.1.1. Declaración de clases

La declaración de una clase introduce un nuevo tipo de referencia del mismo nombre. Una clase describe la estructura y comportamiento de sus instancias u objetos en términos de variables de instancia y métodos. La accesibilidad de las variables y métodos puede ser explícitamente controlada, permitiendo a la clase actuar como una unidad de encapsulación.

class *identificador***{**
declaraciones del constructor
declaraciones de métodos
declaración de métodos estáticos
declaración de variables de instancia
declaraciones de variables estáticas
}

Una clase se introduce por la palabra clave `class`, la cual se sigue de un identificador que da nombre a la clase. Por convención, el nombre de una clase siempre comienza con mayúscula. El cuerpo de la clase se delimita entre llaves, y puede contener una serie de declaraciones de variables, métodos, clases o interfaces. La palabra clave `class` puede precederse de los modificadores, como se muestra en las secciones subsecuentes.

El nombre de la clase introduce un nuevo nombre de tipo de referencia, permitiendo la declaración de variables de ese tipo. Un tipo de variable de referencia mantiene una referencia a un objeto o instancia de la clase. Un objeto se crea usando

el operador `new` en una expresión de alojamiento en memoria, como por ejemplo:

```
Stack s = new Stack();
```

Una clase actúa como una unidad de encapsulación y crea un entorno de clase. Todas las clases son declaradas dentro de un paquete (*package*), el cual puede ser el paquete anónimo por defecto (*default anonymous package*). Además, normalmente una clase está contenida en un archivo fuente o unidad de compilación, que es parte de un paquete. Ambos, el paquete y el archivo, proveen entornos. En general, a menos que esté anidada, cada clase se declara en un archivo fuente por separado, pero es posible también tener varias declaraciones de clases en el mismo archivo.

El nombre del archivo fuente que contiene la declaración de una clase debe ser el mismo que el nombre de la clase que contiene, con la extensión `.java`. Por ejemplo, la clase `Stack` debe ser almacenada en un archivo con el nombre `Stack.java`. Si hay varias clases dentro del mismo archivo fuente, el archivo debe nombrarse a partir de la única clase pública que contiene. Si esto no se hace correctamente, el archivo fuente no podrá ser compilado.

El entorno de clase permite que cualquier expresión dentro de la declaración de una clase pueda usar cualquier variable de instancia o método declarado en la clase. El orden de las declaraciones no es importante, a diferencia del orden de los entornos. Una consecuencia significativa del entorno de clase es que las variables y métodos de cualquier objeto de una clase son accesibles a cualquier método de la propia clase (véase el ejemplo siguiente). Esto significa que aun cuando los objetos son en general fuertemente encapsulados, dentro de su clase la encapsulación depende de su entorno de clase.

El siguiente ejemplo ilustra la declaración de una clase, apuntando los efectos del entorno de clase:

```
class ClassScope {
    // Una variable privada solo accesible dentro del entorno de
    // la clase
    private int n = 1;

    // El siguiente metodo ilustra que las variables privadas de
    // cualquier objeto de la clase ClassScope pueden ser accesadas
    // por un metodo de ClassScope
    public void alter(final ClassScope c) {
        // c es le parametro, no el objeto que llama al metodo, pero
        // la variable n puede ser aun accesada.
        c.n = 123;
    }

    // El siguiente metodo despliega el valor de un objeto usando
    // el valor de la variable privada n
```

```
    public String toString() {
        return new Integer(n).toString();
    }
}

// La siguiente clase prueba el comportamiento de la clase ClassScope

public class ScopeTest {
    public static void main (String [] args) {
        // Es correcto crear objetos ClassScope y usar sus metodos
        // publicos
        ClassScope a = new ClassScope();
        ClassScope b = new ClassScope();

        // Notese que la variable privada n perteneciente al objeto
        // referenciado por b puede ser modificada por un metodo
        // llamado por un objeto de la misma clase referenciada
        // por a
        a.alter(b);

        System.out.println("Objeto a: " + a);
        System.out.println("Objeto b: " + b);

        // Sin embargo, no es posible romper la encapsulacion de
        // un objeto ClassScope desde aqui. El no comentar
        // la siguiente linea causaria un error de compilacion.
        // Esto es cierto aun cuando se coloque esta linea en
        // cualquier metodo a~adido a esta clase
        // a.n = 123;
    }
}
```

3.1.2. Público, Privado y Protegido

Cualquier declaración dentro del entorno de una clase puede precederse de una de las siguientes palabras clave (conocidas también como modificadores de acceso):

```
public, protected, private
```

El uso de los modificadores de acceso es opcional, y puede ser omitido. Las palabras clave tienen el siguiente significado:

- `public` hace que una declaración sea accesible por cualquier clase.
- `protected` hace una declaración accesible por cualquier subclase de la clase que se declara, o a cualquier clase dentro del mismo paquete (nótese que esta

semántica difiere de aquélla utilizada en C++. Sin embargo, parece ser que C++ está en lo correcto, mientras que Java parece tenerlo errado).

- `private` hace una declaración accesible sólo dentro de la clase en que se declara.

Si no se provee ninguna de estas tres palabras clave, se dice que la declaración tiene accesibilidad por defecto (*default accesibility*), lo que significa que es accesible por cualquier otra clase dentro del mismo paquete. Nótese, sin embargo, que tal declaración no es accesible a cualquier subclase dentro de un paquete diferente, por lo que se concluye que la accesibilidad por defecto se controla estrictamente por el entorno de paquete.

Como conclusión, las siguientes declaraciones (por ejemplo) pueden aparecer dentro de una clase:

```
public void metodo1() {...}
private int metodo2() {...}
protected int x = 1;
private Stack s;
float metodo3() {...}
String t = "Hola";
```

3.1.3. Variables de Instancia

Las variables de instancia deben ser escogidas de tal forma que representen el estado de los objetos de una clase. Pueden ser de cualquier tipo, incluyendo aquél de la clase dentro de la cual se declaren (de esta forma, los objetos de la clase pueden mantener referencias a otros objetos de la misma clase). Por convención, una variable de instancia comienza con una letra minúscula.

Las reglas para la inicialización requieren que las variables de instancia sean inicializadas cuando se declaran, o se hace una inicialización por omisión. También, es necesario que el programador decida si se trata de una variable pública, privada, protegida o de acceso por defecto.

Una variable de instancia puede ser declarada como `final`, lo que significa que es una constante y no puede ser cambiada por asignación. Esto resulta muy útil si la variable debe ser pública y debe también siempre ser usada sin cambio en el valor inicial de la variable.

El siguiente programa declara varias variables de instancia:

```
class Class1{
    public String hello = "hello";
    public final String world = "world";
    protected int count = 0;
    private float length = 2.34f;
```

```
    long size = 12356L;
}
```

En este caso la clase sólo consiste de declaraciones de variables de instancia. Esto es legal, pero normalmente sólo se usa si sus instancias u objetos se utilizan como simples estructuras de datos que son parte de la infraestructura de otra clase.

3.1.4. Variables Estáticas o de clase

Una variable estática pertenece a una clase y no es parte del estado de los objetos individuales de la misma. Sólo existe una copia de cada variable estática. Las variables estáticas son llamadas frecuentemente variables de clase.

Las variables de clase tienen varios usos distintos:

- Como variables compartidas por todas las intancias de la clase. Como tales, pueden describirse como "globales" dentro del entorno de la clase, y son típicamente declaradas como privadas.
- Como variables accesibles a todos los clientes de la clase. Así, son efectivamente globales para un programa entero, y necesitarían ser declaradas como públicas. Tales variables, al poderse actualizar por cualquier trozo de código, representan un gran peligro y por lo tanto son extremadamente raras.
- Como constantes declaradas como `final`. Típicamente tales constantes son hechas públicas para ser usadas en cualquier sitio dentro de un programa. Muchas bibliotecas de Java hacen uso de tales variables declaradas como `public static final`. Por convención sus nombres se escriben con letras mayúsculas.

Una variable estática puede ser accesada por cualquier método o cualquier método estático del mismo entorno de clase simplemente usando su nombre. Los clientes fuera del entorno de la clase necesitan usar una expresión de campo usando el nombre de la clase, como por ejemplo:

```
Integer.MAX_VALUE;
```

que accesa a la variable `MAX_VALUE` declarada en la clase `Integer`.

La siguiente clase declara y usa algunas variables estáticas:

```
import java.util.Vector;

class Static1 {
    // Constantes accesibles publicamente
    public final static int UP = 1;
    public final static int DOWN = 2;

    // Variable de clase publicamente accesible que puede ser
```

```
    // modificada por asignacion
    // NO SE HAGA ESTO EN UN PROGRAMA REAL,
    // A MENOS QUE *REALMENTE* SEA NECESARIO
    public static String s = "default";

    // Variables de clase privadas
    private static float f = 3.141f;
    private static Vector v = new Vector();

    // Variables estaticas que pueden ser accesadas desde un metodo
    // No es posible decir realmente si son variables de clase o de
    // instancia por su uso
    public void test() {
        int i = UP;
        s = "hello";
        v.addElement(s);
    }

    public static void main (String args[]) {
        // Las variables estaticas pueden accederse desde un
        // metodo estatico
        s = "hello";
        v.addElement(s);
        Static1 s1 = new Static1();
        s1.test();
    }
}
```

Este código muestra que las variables estáticas pueden ser directamente accesadas por ambos tipos de métodos, normales y estáticos.

3.1.5. Clases de alto nivel

Una clase de alto nivel es una clase ordinaria declarada al nivel del entorno de paquete (*package scope level*), el "más" alto nivel en el que una clase puede ser declarada, y por lo tanto, se le llama clase de alto nivel. Las clases de alto nivel pueden opcionalmente ser declaradas como `public`, `abstract` o `final`, con los siguientes significados:

- `public`. Una clase pública es globalmente accesible, y puede ser utilizada por cualquier otra clase. Para ser utilizada por una clase en otro paquete, es necesario declarar un enunciado de importación (*import statement*) en el archivo fuente conteniendo la clase, a menos que la clase pertenezca al paquete por defecto. Un archivo fuente único puede solo tener una clase o interfaz pública.

- `abstract`. Una clase abstracta no puede tener intancias u objetos, y es diseñada para que se herede de ella.
- `final`. Una clase final no puede ser heredada, es decir, no puede tener subclases

Si una clase de alto nivel no es declarada como pública, solo puede ser accesada por las otras clases dentro del mismo paquete. De otra manera, las clases de alto nivel se comportan tal y como se ha descrito.

3.1.6. Clases anidadas

Una clase anidada o interna se declara dentro de un entorno anidado en otra clase. Hay diferentes tipos de clases anidadas: clases anidadas de alto nivel, clases miembro, clases locales, y clases anónimas. Las clases anidadas no pueden ser declaradas usando los modificadores `native`, `synchornized`, `transient` o `volatile`.

Clases anidadas de alto nivel

Una clase anidada de alto nivel es una clase estándar declarada dentro del entorno de una clase de alto nivel. Esto permite que el nombre de la clase sea sujeto a las reglas de entorno y encapsulación de clase. En particular, una clase anidada puede ser declarada como privada, de tal modo que sólo es accesible dentro de la clase en que se le declara.

Las clases anidadas de alto nivel pueden ser decladas como públicas, requiriendo que su nombre sea cualificado por el nombre de la clase donde están anidadas, y si son usadas por otras clases:

```
ClaseExterna.ClaseInterna c = new ClaseExterna.ClaseInterna();
```

Esto puede ser útil si el entorno del nombre de la clase anidada necesita ser restringido. Sin embargo, de manera más común, las clases anidadas de alto nivel son declaradas protegidas o privadas, y actúan como bloques de construcción de las estructuras de datos internas de la clase donde están anidadas. Por ejemplo, una clase de listas ligadas puede tener una estructura interna consistente en una cadena de objetos nodo. Cada nodo podría ser una instancia de una clase anidada de alto nivel. La clase se hace privada ya que no hay necesidad de que los clientes de la lista sepan acerca de ella o intenten usarla; simplemente, funciona como una pieza de la estructura interna.

Una clase anidada de alto nivel puede accesar variables estáticas declaradas en su clase residente aun cuando sean declaradas como privadas, pero no tiene acceso a las variables de instancia. Clases anidadas de alto nivel pueden contener aun más clases anidadas de alto nivel.

El siguiente ejemplo usa una clase anidada de alto nivel llamada `CountedString`. Los objetos de la clase asocian una cuenta con una cadena de caracteres, de tal modo que la clase `StringCounter` puede usarlas para contar el número de veces que una cadena de caracteres se añade usando el método `addString`. Nótese que las variables de instancia de `CountedString` no son privadas y pueden ser directamente accesadas por la clase residente. Esta decisión se hace debido a que la clase anidada se usa como bloque de construcción, y no requiere una fuerte encapsulación.

```
import java.util.Vector;

class StringCounter {

    private static class CountedString {
        public CountedString(String s) {
            item = s;
        }
        String item;
        int count = 1;
    }

    // Si el argumento String ya existe, incrementa el contador.
    // De otra forma, a~ade un nuevo objeto CountedString con
    // valor count = 1
    public void addString(String s) {
        CountedString tmp = searchFor(s);
        if (tmp == null) {
            items.addElement(new CountedString(s));
        }
        else {
            tmp.count++;
        }
    }

    // Retorna el numero de veces que una cadena ha sido a~adida
    public int getCount(String s) {
        CountedString tmp = searchFor(s);
        return tmp == null ? 0 : tmp.count;
    }

    // Metodo auxiliar privado
    // Si una cadena ha sido a~adida, retorna su objeto
    // CountedString. De otra manera, retorna null
    private CountedString searchFor(String s) {
        for (int i = 0; i < items.size(); i++) {
            CountedString c = (CountedString)items.elemtAt(i);
```

```
            if ((c.item).equals(s)) return c;
        }
        return null;
    }

    public static void main(String[] args) {
        StringCounter c3 = new StringCounter();
        c3.addString("hello");
        c3.addString("world");
        c3.addString("hello");
        c3.addString("world");
        c3.addString("hello");
        c3.addString("world");
        c3.addString("hello");
        c3.addString("world");
        c3.addString("hello");
        c3.addString("world");

        System.out.println("hello ha sido a~adida " +
            c3.getCount("hello") + " veces");

        System.out.println("world ha sido a~adida " +
            c3.getCount("hello") + " veces");
    }

    private Vector items = new Vector();
}
```

Clases miembro

Una clase miembro es una variedad de clase anidada o interna que no es declarada como estática. Un objeto de una clase miembro es directamente asociado con el objeto de la clase residente que lo ha creado, y automáticamente tiene referencia implícita a él. Como resultado, el objeto de la clase miembro tiene acceso directo a las variables de instancia del objeto de la clase residente.

Las clases miembro comparten las características estándar de las clases anidadas, pero no pueden ser declaradas estáticas, ni pueden declarar variables estáticas, métodos, o clases anidadas de alto nivel. Una clase miembro no puede tener el mismo nombre de ninguna de las clases residentes o de paquetes (lo cual es diferente de las reglas acerca de los nombres de variables y métodos).

Las palabras clave `this`, `new`, y `super` tienen una sintaxis extendida para usarse en clases miembro:

```
nombreDeLaClase.this
```

```
referenciaAObjeto.new
referenciaAObjeto.super
```

Un objeto de una clase miembro (u objeto miembro) tiene una referencia implícita al objeto de la clase residente que lo ha creado, el cual es automáticamente añadido por el compilador de Java. Esto permite el acceso al estado del objeto de la clase residente. El acceso en la dirección contraria (de la clase residente a la clase miembro) respeta la encapsulación del objeto de la clase miembro, de tal modo que sus variables y métodos privados no pueden ser accesados directamente.

Los objetos miembro son típicamente usados para crear estructuras de datos, cuyos objetos necesitan saber qué objeto los contiene, y ayudan a accesar tales estructuras de datos. El ejemplo principal de esto son los objetos de enumeración, los cuales iteran a través de estructuras de datos privadas.

Para dar soporte a las clases miembro se provee de muchos otros tipos de expresiones extra. La primera es una extensión a las expresiones con `this`. Una expresión como:

```
x = this.yes;
```

será sólo válida si `yes` es una variable de instancia declarada por la clase miembro, y no si `yes` es declarada por la clase residente. Si `yes` pertenece a la clase residente, entonces tiene que ser accesada usando una expresión como:

```
x = TestClass.this.yes;
```

donde `TestClass` es el nombre de la clase residente.

Las clases anidadas pueden declararse formando diferentes niveles, y a cualquier profundidad, pudiendo utilizar el mecanismo de `this` para la anidación. Por ejemplo, la clase `C` es anidada en `B`, la cual es anidada en la clase `A`:

```
class A {
    class B {
        class C {
            ...
            int c;
            void f() {
                A.this.a; //Variable declarada en la clase A
                B.this.b; //Variable declarada en la clase B
                this.c;   //Variable declarada en la clase C
            }
        }
        ...
        int b;
```

```
    }
    ...
    int a;
}
```

Esto permite que las variables declaradas en cada clase residente puedan ser accesadas mediante nombrar la clase en una expresión `this`. Nótese que una expresión como:

```
A.B.C.this.a;
```

no es válida, dado que la clase residente debe ser nombrada directamente.

La siguiente expresión adicional involucra al operador `new`. Los objetos de las clases miembro pueden sólo ser creados si tienen acceso a un objeto de la clase residente. Esto ocurre por defecto si el objeto de la clase miembro se crea por un método de instancia perteneciente a su clase residente. De otra forma, es posible especificar un objeto de la clase residente usando el operador `new` de la siguiente forma:

```
B b = a.new B(); // Notese la sintaxis referenciaAObjeto.new
```

donde el operador `new` se precede por la referencia al objeto, seguido por el operador punto. Esto significa crear un nuevo objeto miembro `B` usando el objeto referenciado por la variable `a` (la cual debe ser una instancia de la clase residente correcta). El nuevo objeto miembro tendrá entonces acceso al estado del objeto referenciado por `a`.

La sintaxis de esta forma de expresión `new` puede ser sorpresiva al principio. Si una clase `C` está anidada en una clase `B`, la cual está anidada en una clase `A`, entonces es posible escribir el siguiente enunciado válido:

```
A.B.C c = b.new C(); // donde b es referencia a un objeto de B
```

Esto significa declarar una variable llamada `c` de tipo `C` la cual está anidada en `B`, que a la vez está anidada en `A` (nótese la parte `A.B.C`). Entonces crea un nuevo objeto `C` dada una referencia de un objeto de tipo `B`. Nótese que se usa una expresión `b.new C()`, y no algo como `b.new A.B.C()`. Esto se debe a que la clase de un objeto a ser creado se da relativo a la clase del objeto residente. La sección de ejemplos ilustra más adelante esta sintaxis.

La tercera y final adición a los tipos de expresiones se refiere a `super`, y se cubre en una sección posterior.

Una consecuencia del acceso de una clase miembro a su clase residente es que las reglas de entorno se vuelven más complicadas, ya que ahora hay dos formas de

considerar un nombre: *(a)* usando la jerarquía de herencia, y *(b)* usando la jerarquía de contención de objetos residentes.

El siguiente ejemplo muestra el uso básico de las clases y objetos miembro:

```
class Class5 {
    private class Member {
        // Este metodo muestra que las variables de instancia
        // privadas del objeto de la clase residente pueden ser
        // accesadas
        public void test() {
            i += 10;
            System.out.println(i);
            System.out.println(s);
        }
    }

    // Este metodo crea un objeto de la clase Member el cual
    // tiene acceso al estado privado del objeto que lo crea
    public void test() {
        Member n = new Member();
        n.test();
    }

    public static void main (String[] args) {
        Class5 c5 = new Class5();
        c5.test();
    }

    private int i = 10;
    private String s = "hello";
}
```

Clases Locales

Una clase local es aquélla declarada dentro del entorno de un enunciado compuesto, es decir, es local al cuerpo del método o inicializador de instancias en que se encuentra, como si fuera una variable local. Con ciertas restricciones, una clase local es también una clase miembro, ya que es creada dentro del objeto de una clase residente. Así como cualquier clase miembro tiene acceso a datos miembros en cualquier clase residente, una clase local tiene acceso a cualquier variable de parámetro o local (sujeto al orden de declaración) que se declaran en el mismo entorno, siempre y cuando tengan el modificador `final`.

Si una referencia al objeto de una clase local permanece después de que su entorno de creación ha sido terminado (por ejemplo, como referencia de retorno de

un método), entonces el objeto local permanece en existencia, manteniendo acceso a los parámetros finales o variables locales finales de su entorno de creación.

El nombre de una clase local no puede ser el mismo de su clase o paquete residente. Las clases locales no pueden incluir variables, métodos o clases estáticas. Además, no pueden ser declaradas como públicas, protegidas, privadas o estáticas. De hecho, una clase local puede usar la sintaxis extendida que se presenta para las clases miembro respecto a la referencia `this`, pero no para `new` y `super`.

Las clases locales son esencialmente clases miembro declaradas en un entorno local y, como resultado, están sujetas a reglas y restricciones adicionales. La nueva característica más importante de las clases locales es su habilidad de accesar variables y parámetros finales en su entorno de declaración. Esto da una limitada capacidad a los objetos de las clases locales de retener el estado del entorno local después de haber salido del entorno.

Típicamente, una clase local es declarada en un método como implementando un tipo interfaz, y cuando un objeto de la clase se retorna desde un método. Esto permite que el entorno de declaración de la clase sea limitado a un área de acción mínima, pero aún es posible generar objetos útiles.

El siguiente ejemplo muestra cómo las clases locales pueden accesar variables de su entorno residente y de su objeto residente:

```
class Class8 {
    public void f (final String h, String w) {
        int j = 20;
        final int k = 30;

        class Local {
            public void test() {
                System.out.println(h); // h es local
                // System.out.println(w); ERROR!
                // No es posible imprimir w, dado que
                // w no es final

                // System.out.println(j); ERROR!
                // Tampoco puede imprimirse j, dado que
                // no es final

                System.out.println(k); // k es final

                // System.out.println(i); ERROR!
                // i aun no ha sido declarada

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

```
                // Como una clase miembro, las variables de
                // instancia del objeto residente pueden ser
                // accesadas. No necesitan ser finales.
            }
        }

        Local l = new Local();
        l.test();

        final int i = 10;
    }

    public static void main (String[] args) {
        Class8 c8 = new Class8();
        c8.f("hello","world");
    }

    private String name = "Class8";
}
```

Clases Anónimas

Una clase anónima es una clase local que no tiene nombre. Se declara como parte de una expresión `new`, y debe ser o una subclase o implementar una interfaz:

new *NombredelaClase*(*Listadeargumentos*) { *Cuerpo de la clase*}
new *NombredelaInterfaz*( ) { *Cuerpo de la clase*}

NombredelaClase es el nombre de la superclase de la clase anónima, mientras que *NombredelaInterfaz* es el nombre de la interfaz de la cual la clase anónima debe conformarse. La lista de argumentos es una lista de parámetros que es utilizada para llamar al constructor coincidente de la superclase.

El cuerpo de la clase puede definir métodos, pero no puede definir constructores (dado que no pueden ser nombrados). Las restricciones impuestas a las clases locales también son válidas para las clases anónimas. Al igual que las clases locales, las clases anónimas tienen las propiedades básicas de las clases miembro, y sus objetos tienen una referencia implícita al objeto donde son creados.

El principal uso de las clases anónimas es directamente definir un objeto usando una extensión de la sintaxis para `new`. Esto permite que una sola instancia de un tipo específico de objeto sea creada exactamente donde es necesaria sin declarar una clase completa para ella. La clase anónima debe, sin embargo, explícitamente ser una subclase o implementar una interfaz. Si implementa una interfaz, la clase será una subclase de `Object`.

Cuando una clase anónima se declara como subclase, la intención es que los métodos heredados sean sobreescritos (*overriden*) y especializados. Hay pocas razones para añadir otros métodos públicos, dado que no pueden ser llamados. Sin embargo, se pueden añadir métodos privados.

Como una clase anónima no tiene constructores, depende de la inicialización directa de variables de instancia y de inicializadores de instancia para inicializar sus variables de instancia.

El siguiente programa ilustra el uso de las clases anónimas:

```
interface Thing {
    void test(final String s);
}

class Subclass {
    public void doSomething() {
        System.out.println("Do something");
    }
}

class Class 10 {
    // Retorna un objeto creado usando una clase anonima que
    // implementa la interface
    public Thing i1() {
        Thing t = new Thing() {
            public void test (final String s) {
                System.out.println(s);
            }
        }; // Notese el punto y coma aqui
        return t;
    }

    // Una version mas corta del ultimo metodo, eliminado la
    // variable local

    public Thing i2() {
        return new Thing() {
            public void test (final String s) {
                System.out.println(s);
            }
        }; // Notese el punto y coma aqui
    }

    // Retorna un objeto creado con una clase anonima que es una
    // subclase de una clase
```

```
    public Subclass f1() {
        Subclass t = new Subclass() {
            public void doSomething() {
                something();
            }
            private void something() {
                System.out.println(name);
            }
            String name = "Anonymous 1";
        };
        return t;
    }

    // Una version mas simple de f1
    public Subclass f2() {
        return new Subclass() {
            public void doSomething() {
                something();
            }
            private void something() {
                System.out.println(name);
            }
            String name = "Anonymous 2";
        };
    }

    public static void main(String[] args) {
        Class10 c10 = new Class10();
        Thing t1 = c10.i1();
        t1.test("hello");
        Thing t2 = c10.i2();
        t2.test("world");
        Subclass t3 = c10.f1();
        t3.doSomething();
        Subclass t4 = c10.f2();
        t4.doSomething();
    }
}
```

3.1.7. Clases finales

La declaración de una clase final se precede por el modificador `final`. Una clase final no puede tener subclases. Sin embargo, normalmente una clase debe poder tener subclases si su diseñador lo considera así, y ha verificado que esto

tenga sentido. Si hay alguna duda o no se desea tener subclases, la clase debe ser declarada como `final`, permitiendo al compilador verificar que la generación de subclases no sea hecha. Una consecuencia de declarar una clase final es que el compilador no puede optimizar su código. En particular, no es posible redefinir métodos de subclases, así que es factible realizar un ligado estático en lugar de un ligado dinámico, que es más costoso.

El siguiente código es una descripción de una clase final simple:

```
public final class Example {
    // Declaraciones de metodos y variables usuales
}

// ERROR. No puede declararse la siguiente clase
class Subclass extends Example {
    ...
}
```

3.1.8. Clases abstractas

Una clase abstracta (*abstract class*) es un contenedor para declarar métodos y variables compartidos para usarse por subclases y para declarar una interfaz común de métodos y variables accesibles. La declaración de una clase abstracta incluye el modificador `abstract` antes de la palabra clave `class`.

Una clase declarada como `abstract` puede incluir cualquier variable estándar y declaración de métodos, pero no puede ser usada en una expresión. Puede también incluir declaraciones de métodos abstractos. La clase define un tipo, así que las variables del tipo pueden ser declaradas y pueden mantener referencias a objetos de una subclase.

Una subclase de una clase abstracta puede ser también abstracta. Sin embargo, en general es una clase concreta (es decir, diseñada para tener instancias u objetos).

3.2. Métodos

3.2.1. Declaración de Métodos

La declaración básica de un método es de la siguiente forma:

modificadores tipo nombre(*lista de parámetros*) **{**
Secuencia de enunciados
}

Los modificadores son opcionalmente uno, pudiendo ser `public`, `private` o `protected`. Además, puede añadírseles los modificadores `abstract`, `final`, `native` y `synchronized`. El cuerpo del método es un enunciado compuesto delimitado por llaves, que contiene una secuencia de enunciados.

Un método puede arrojar excepciones, que se declaran mediante una cláusula `throws`:

modificadores tipo nombre(*listadeparámetros*) **throws** *listadenombresdetipos***{**
Secuencia de enunciados
}

La lista de nombres de tipos que sigue a la palabra clave `throws` es una lista separada por comas de uno o más nombres de los tipos de excepciones que pueden ser arrojadas.

Esto es también el caso para los métodos declarados como `abstract`:

modificadores **abstract** *tipo nombre*(*listadeparámetros*) **;**
modificadores **abstract** *tipo nombre* (*listadeparámetros*) **throws***nombredetipo***;**

Si un método retorna un tipo arreglo, entonces la siguiente variación de la sintaxis puede ser utilizada:

modificadores tipo nombre(*listadeparámetros*) []**{**
Secuencia de enunciados
}

con un par de corchetes tras la lista de parámetros. Aun cuando lo anterior representa una sintaxis legal, es mucho más común colocar los corchetes donde realmente pertenecen, siguiendo el nombre del tipo de retorno:

modificadores tipo[]*nombre*(*listadeparámetros*) **{**
Secuencia de enunciados
}

Los arreglos con más de una dimensión simplemente requieren conjuntos extra de corchetes, exactamente como en una declaración de variables.

Los métodos son procedimientos llamados o invocados por un objeto específico utilizando una llamada de campo o una expresión de llamada a método. Hay dos tipos de métodos: aquéllos que no retornan valor (`void`) y aquéllos que son de retorno de valor (*value-returning*).

Los métodos de retorno vacío (o métodos `void`) se declaran con tipo de retorno `void`. El cuerpo de un método `void` simplemente realiza un procesamiento, que tendrá el efecto colateral de cambiar el estado del objeto para el cual fue invocado, y termina sin explícitamente retornar un valor, aun cuando puede usarse el enunciado `return` sin argumento:

```
void f(){
    a = 4; // Supongase que a es una variable de instancia
    return;
}
```

Los métodos con retorno de valor (o métodos *no-void*) se declaran con un tipo de valor de retorno diferente a `void`. El cuerpo del método debe contener al menos un enunciado `return`, retornando un valor del tipo que coincida con o pueda ser convertido al tipo especificado de retorno:

```
int f(){
    return a * 10; // Supongase que a es una variable accesible
}
```

Como un valor debe ser retornado por un método con retorno de valor, el compilador verifica que todas las salidas posibles de un método resultan en la ejecución de un enunciado `return`. En el siguiente ejemplo este no es el caso, ya que cuando `x` tiene un valor mayor que 9 el control pasa al final del cuerpo del método sin alcanzar el enunciado `return`. Como resultado, el compilador no aceptará el método hasta que otro enunciado `return` sea añadido:

```
int f(){
    // Cuidado: esto no compilara
    if (x < 10) // x es una variable de instancia
        return value;
    // ERROR: debe haber un enunciado return aqui
}
```

Los métodos pueden ser sobrecargados (*overloaded*), lo que significa que dos o más métodos en la misma clase pueden tener el mismo nombre, siempre y cuando tengan una lista de parámetros diferente. Por ejemplo:

```
int f(int a) {...}
char f(float g) {...}
String f(String a, String b) {...}
double f (int a, int b, int c, int d) {...}
```

Todos estos métodos se llaman `f`, pero pueden distinguirse por su diferente lista de parámetros (en una llamada a un método, el número y tipos de los argumentos son utilizados para determinar cuál método debe usarse). Los tipos de retorno no se toman en cuenta, y pueden diferir como se muestra aquí.

El siguiente programa consiste de una clase que declara varios métodos:

```
class Class1 {
    // Asigna todos los elementos de un arreglo con un valor dado
    public int [] fill(int [] array, int value){
        for (int i = 0; i < array.length; i++) {
            array[i] = value;
        }
        return array;
    }

    // Notese el parametro con modificador "final"
    public int f(final int i) {
        if (i > 0) {
            return i;
            // El intentar una asignacion a i genera un error
            // i = 0;  // Error!
        }
        return 0;
    }

    // El siguiente metodo solo puede ser utilizado por una
    // hebra de control (thread) en un momento dado, para un
    // objeto en particular
    public synchronized void assignString(final String s) {
        name = s;
    }

    public static void main(String []args) {
        Class1 c = new Class1();
        int[] array = new int[10];

        array = c.fill(array, 5);
        c.assignString("world");
        int n = c.f(-1);
    }

    private String name = "hello";
}
```

El siguiente ejemplo muestra una serie de métodos sobrecargados `max`, que retornan el máximo de entre dos valores. Cada método trabaja con tipos primitivos diferentes.

```
class Class2{
    public byte max(final byte a, final byte b){
        return a > b ? a : b;
    }
    public short max(final short a, final short b){
        return a > b ? a : b;
    }
    public int max(final int a, final int b){
        return a > b ? a : b;
    }
    public long max(final long a, final long b){
        return a > b ? a : b;
    }
    public float max(final float a, final float b){
        return a > b ? a : b;
    }
    public double max(final double a, final double b){
        return a > b ? a : b;
    }
    public char max(final char a, final char b){
        return a > b ? a : b;
    }

    public static void main(String[] args) {
        Class2 maxObject = new Class2();

        byte a = maxObject.max((byte)3, (byte)5);
        short b = maxObject.max((short)3, (short)5);
        int c = maxObject.max(3,5);
        long d = maxObject.max(3l,5l);
        float e = maxObject.max(3.4f,5.6f);
        double f = maxObject.max(3.4,5.6);
        char g = maxObject.max('a','z');
    }
}
```

Nótese que cada método `max` tiene diferentes tipos de parámetros y también que retorna un tipo diferente de resultado. Esta clase sólo declara métodos, y efectivamente, conjunta un conjunto de funciones (métodos con retorno de valor). Este estilo de clase se utiliza para crear objetos-función (*function objetcs*), los cuales permiten a los métodos ser tratados como objetos, así como ser almacenados y pasados como parámetros.

3.2.2. Métodos Estáticos o de clase

Un método estático o de clase pertenece a una clase, y no es parte de la implementación de objetos o instancias individuales. Los métodos estáticos pueden ser llamados directamente en lugar de invocarse mediante un objeto.

La declaración de un método estático es similar a la declaración de un método estándar. La declaración básica de un método estático tiene la siguiente forma:

modificadores **static** *tipo nombre*(*listadeparámetros)*{
Secuencia de enunciados
}

El cuerpo del método es un enunciado compuesto delimitado por llaves, y conteniendo una secuencia de enunciados. Los modificadores son opcionalmente `public`, `private` o `protected` (como se han descrito anteriormente), y cero o más palabras clave como `final`, `native`, o `synchronized`. Los métodos estáticos no pueden ser declarados como `abstract`, y en realidad puede considerárseles más bien como de tipo `final`, aun cuando no son explícitamente especificados como tales.

Los métodos estáticos pueden arrojar excepciones, y por lo tanto, pueden ser declarados usando la cláusula `throws`:

modificadores **static***tipo nombre*(*listadeparámetros*) } **throws** *nombresdetipo*{
Secuencia de enunciados
}

La lista de nombres de tipo que sigue a la palabra clave `throws` es una lista separada por comas de uno o más nombres de tipos de excepciones que pueden ser arrojadas.

Los métodos estáticos están sujetos por mucho a las mismas reglas que los métodos estándar, con las siguientes excepciones:

- Un método estático pertenece a una clase, y no a sus objetos o instancias.
- Un método estático puede ser llamado tanto directamente como por un objeto de la misma clase.
- Un método estático no puede accesar ningún variable o método de instancia (dado que no pertenece a ninguna instancia), pero puede accesar cualquier variable o método estático.
- La palabra clave `this` no puede ser usada sobre métodos estáticos.

A pesar de estas restricciones, un método estático puede ser declarado y utilizado como sea requerido.

Hay un uso especial de los métodos estáticos en la forma del `static main`. Cuando una clase declara un método `public static main`, éste provee de un punto

de inicio para la ejecución de un programa utilizando esa clase. Todos los ejemplos de programas vistos hasta ahora incluyen tal método. Este método se hace cargo de establecer el estado inicial del programa, creando el conjunto inicial de objetos, y haciendo las llamadas a los métodos apropiados para continuar la ejecución.

Cualquier clase puede tener un método `main`. Cuando un programa se ejecuta usando `java`, el nombre de la clase que contiene el método `main` a ser ejecutado se da como sigue:

```
java MyClass
```

donde, en este caso, el primer método a ser ejecutado es el `main` de la clase `MyClass`.

Un método estático `main` toma un argumento de tipo arreglo de `String`, llamado `args`, y contiene cualquier argumento del comando de línea.

La siguiente clase declara y usa métodos estáticos:

```
class Static2 {
    public static void g(final int i) {
        System.out.println("En static void g(i = " + i + ")");
    }

    public static void f(){
        System.out.println("En static void f()");
        g(10);
        // No es posible llamar aqui un metodo sin objeto
        // test(); // ERROR!
    }

    // Un metodo de instancia llamando a un metodo de clase
    public void test(){
        g(5);
    }

    public static void main(String args[]){
        f();
        g(10);
        // Crear una instancia de Static2
        Static2 s2 = new Static2();
        s2.f();  // Notese, esto se puede hacer!
        s2.test();
    }
}
```

Nótese que en la línea `s2.f();`, un método estático es llamado como un método de instancia. Parece indistinguible de un método de instancia, pero sin embargo, es una llamada a método estático.

3.2.3. Constructores

Un *constructor* se declara como cualquier método, contando con una lista de parámetros. Sin embargo, la declaración de un constructor no tiene tipo de retorno, y su nombre debe coincidir con el nombre de la clase dentro de la cual es declarado. El constructor tiene la siguiente forma:

nombredelconstructor (*listadeparámetros*) **{**
Secuencia de enunciados
}

Un constructor puede ser declarado opcionalmente como `public`, `protected` o `private`. No se permite ningún otro modificador.

Un constructor también puede arrojar excepciones, utilizando la cláusula `throws`:

nombredelconstructor (*listadeparámetros*) **throws** *listadenombresdetipos* **{**
Secuencia de enunciados
}

En cualquier momento que un objeto es creado, un método constructor es llamado automáticamente. Esto se garantiza de siempre suceder y no puede evitarse, sin importar cómo las variables de instancia sean inicializadas. Una clase puede declarar uno o más métodos constructores, permitiéndose la sobrecarga de constructores siempre y cuando los métodos constructores difieran en su lista de parámetros.

Un constructor con una lista de parámetros vacía se conoce como *constructor por omisión* (*default constructor*). Si una clase no tiene explícitamente declarado ningún constructor, el compilador de Java automáticamente creará un constructor por omisión público (con lista de parámetros vacía). Si una clase declara explícitamente cualquier constructor, con cualquier lista de parámetros, entonces el compilador no generará automáticamente un constructor por omisión.

Un objeto se crea mediante una expresión de alojamiento, usando la palabra clave `new`, como por ejemplo:

```
Vector v = new Vector();
Vector w = new Vector(10);
```

La palabra clave `new` se sigue por el nombre de una clase y opcionalmente por una lista de parámetros actuales. La lista de parámetros actuales se hace coincidir con aquellas provistas por los constructores de la clase, a fin de determinar cuál

constructor llamar. Una lista vacía de parámetros coincide con el constructor por omisión. Siempre y cuando se encuentre una coincidencia, el constructor será llamado. Antes de que el cuerpo del constructor sea ejecutado, el constructor de cualquier superclase será ejecutado, seguido de cualquier inicialización de variables de instancia declaradas en la clase. Esto permite que el cuerpo del constructor sea escrito con el conocimiento de que toda otra inicialización será realizada antes. La buena práctica dicta que el cuerpo del constructor sólo realice inicializaciones relevantes a su clase, y no intente inicializar variables de sus superclases.

Como las variables de instancia y de clase pueden ser inicializadas por expresiones o bloques de inicialización, puede no haber necesidad de explícitamente declarar ningún constructor, y la clase puede simplemente depender del constructor por omisión generado por el compilador de Java. Sin embargo, los constructores son la única forma general y efectiva de parametrizar la creación de objetos, de tal modo que la creación e inicialización pueda ser ajustada para cada situación en particular. Por ejemplo, la clase `String` del paquete `java.lang` provee de los siguientes constructores. Cada uno crea un objeto de tipo `String` a partir de un conjunto diferente de argumentos:

```
String()
String(byte[])
String(byte[],int)
String(byte[],int,int)
String(byte[],int,int)
String(byte[],int,String)
String(byte[],String)
String(char[])
String(char[],int,int)
String(String)
String(StringBuffer)
```

Los constructores públicos, incluyendo el constructor por omisión generado por el compilador, permiten a las instacias de la clase ser creadas en cualquier parte del programa. La mayoría de las clases de alto nivel, si no todas, incluyen un constructor público.

Los constructores pueden ser tanto protegidos como privados, dando una forma de limitar el acceso a algunos o todos los constructores, y por lo tanto, controlando cuáles otras partes de un programa pueden crear instancias de una clase. En particular, si todos los constructores se hacen privados, entonces sólo los métodos de la misma clase pueden crear objetos de la clase. Esto puede en realidad ser muy útil; algunas clases pueden declarar únicamente métodos estáticos, y no necesitar instancias, o bien puede requerirse que una clase no permita a sus clientes crear objetos utilizando `new`.

Cuando un objeto deja de referenciarse, es destruido por el recolector de basura (*garbage collector*) que es parte del sistema de ejecución de Java. Un método llamado `finalize` puede ser definido para ser llamado justo antes de la destrucción.

El siguiente ejemplo tiene tres constructores sobrecargados, cada uno de los cuales inicializa las variables de instancia en una forma específica. Todos son públicos, así que cualquiera de ellos puede ser usado por el código de sus clientes:

```
import java.util.vector;

class Constructor1 {

    // Constructor por Omision
    public Constructor1 (){
        // La asignacion de esta variable no es realmente
        // necesaria, pues la inicializacion por defecto
        // es correcta
        v = new Vector();
    }

    // Crear un vector con un tama~o dado
    public Constructor1(int size) {
        initialSize = size;
        v = new Vector(size);
    }

    // Crear un vector dado su tama~o e inicializar todos sus
    // elementos con un argumento String
    public Constructor1 (int size, String val) {
        initialSize = size;
        v = new Vector(size);
        for (int i = 0; i < size; i++){
            v.addElement(val);
        }
    }

    public static void main(String[] args){
        // Crear un objeto con el constructor por omision
        Constructor1 c1 = new Constructor1();

        // Otros constructores
        Constructor1 c2 = new Constructor1(10);
        Constructor1 c3 = new Constructor1(10,"Hello");
    }
```

```
    // Variables de instancia que seran inicializadas por defecto
    // a menos que sean asignadas por un constructor
    private int initialSize;
    private Vector v;
}
```

3.2.4. Inicializadores Estáticos

Un inicializador estático (*static initializer*) es una secuencia de enunciados que se ejecutan cuando se carga una clase. Puede ser utilizado para inicializar variables estáticas y realizar otra inicialización necesaria más bien por una clase que por los objetos de la clase.

La declaración de una clase puede contener uno o más inicializadores, consistiendo en un enunciado compuesto que se precede por la palabra clave `static`:

static { *Secuencia de enunciados*}

Sólo las variables estáticas declaradas antes del inicializador se encuentran en el entorno dentro del enunciado compuesto.

Cuando un programa se ejecuta, las clases se cargan en memoria como se vayan necesitando (en lugar de tenerlas todas antes de la ejecución). Cargar una clase significa buscar el archivo `.class`, que contiene el código de los métodos y hacer todas las inicializaciones estáticas. Las variables estáticas y los inicializadores estáticos se inicializan o ejecutan a fin de que sean escritos dentro de la declaración de la clase. Considérese el siguiente ejemplo:

```
class Example {
    ...
    static int i = 10;

    static int j;
    static { j = i * 10; }

    static String h = "hello";
    static { h = h + "world" ; }
    ...
}
```

La variable `i` es inicializada al principio, seguida de una inicialización por defecto de `j` a cero (dado que no hay expresión de inicialización directa). Entonces, a `j` se le asigna un valor en el inicializador estático, seguido de la inicialización de `h`, y finalmente, la evaluación de otro inicializador estático que asigna un nuevo valor a `h`. Nótese que un inicializador estático no puede referirse a variables estáticas declaradas después de sí mismo, así que el sigueinte programa generaría un error:

```
class Example {
    ...
    static { j = i * 10; } // Error: referencia ilegal

    static int i = 10;
    static int j;
    ...
}
```

3.2.5. `this`

`this` es una variable `final` que se declara automáticamente en los constructores, métodos de instancia e inicializadores de instancias, y se inicializa para mantener una referencia al objeto creado por el constructor, método o inicializador que lo invoca. `this` es de tipo referencia al objeto.

`this` puede ser usado como cualquier otra variable de tipo referencia a clase, sin olvidar que no se permite su asignación dado su atributo `final`. Esto significa que dentro de un método es posible escribir enunciados usando el operador punto como:

```
{
    ...
    this.x = 10;
    this.f();
    ...
}
```

donde `x` es una variable de instancia y `f` es un método, ambos declarados dentro de la misma clase. Esto tiene exactamente el mismo resultado que simplemente escribir en forma más familiar:

```
{
    ...
    x = 10;
    f();
    ...
}
```

Es decir, estos enunciados son en realidad una abreviatura de escribir una versión más larga.

Algunas veces usar la forma larga puede ser útil. Por ejemplo, cuando el nombre de una variable de instancia queda oculto por el nombre de un parámetro dentro de un entorno anidado:

```
class Example {
    ...
    void f (int x) { // el parametro x oculta la variable de instancia
        this.x = x;  // this.x es usado para tener acceso a
                     // la variable de instancia oculta
    }


    ...
    private int x = 10;
}
```

Otro uso común de `this` es pasar la referencia del objeto actual a un método de otra clase:

```
{
    ...
    A a = new A();
    ...
    a.f(this) ; // Pasa this al metodo f de la clase A
    ...
}
```

`this` es también importante cuando se usa en conjunción con los constructores, ya que permite que un constructor llame a otro. Una expresión formada por `this` seguido de una lista de parámetros en paréntesis llamará al constructor con la lista de parámetros coincidente. Por ejemplo, si una clase tiene dos constructores, uno por omisión y otro tomando un argumento `int`, entonces lo siguiente es posible:

```
public Test() {
    this(10); // Llama al constructor con int
}
```

Aquí, la expresión `this(10)` resulta en una llamada al constructor tomando un `int` como argumento. Nótese que la expresión `this` debe aparecer como el primer enunciado en el cuerpo de un constructor, y que tal llamada a un constructor sólo puede hacerse desde otro constructor, no desde un método de instancia o de clase.

Las llamadas a constructores usando `this` no pueden ser recursivas. Por ejemplo, el siguiente constructor no compilará:

```
public Test(int x) {
    this(10); // Error: Llamada recursiva al mismo constructor
}
```

3.2.6. Redefinición de Métodos

Una subclase puede redefinir un método heredado mediante proveer una nueva definición del método que tenga el mismo nombre, el mismo número y tipo de parámetros, y el mismo tipo de resultado de retorno que el método heredado. El método heredado se oculta al entorno de la subclase. Cuando el método es invocado por un objeto de la subclase, el método redefinido es ejecutado usando ligado dinámico.

Un método redefinido es declarado en una subclase como cualquier otro método, pero debe tener el mismo nombre, el mismo número de parámetros del mismo tipo, y el mismo tipo de retorno que el método de la superclase.

Los métodos privados no pueden ser redefinidos, así que que un método coincidente de la subclase se considera completamente separado. Por otro lado, el acceso de un método redefinido público (`public`), protegido (`protected`) o por defecto (si no hay modificador) debe tener el mismo acceso que el método de la superclase, o hacerse más accesible (por ejemplo, de `protected` a `public`). Un método redefinido no puede hacerse menos accesible.

Los métodos estáticos no pueden ser redefinidos. Los métodos de instancia no pueden redefinirse por un método estático.

La redefinición de métodos depende del ligado dinámico (*dynamic binding*), de tal modo que el tipo del objeto para el cual un método es invocado determina cuál versión de un método redefinido se llama. Así, en una expresión como:

```
x.f(); // donde f es un metodo redefinido
```

la versión de `f` que se llama depende de la clase del objeto que se referencía por `x` cuando la llamada al método es evaluada en tiempo de ejecución, y no en el tipo de la variable `x`. La mayor consecuencia de esto es que una variable de tipo de una superclase puede asignarse a una referencia a un objeto de la subclase, pero cuando se llama a los métodos, el ligado dinámico asegura que los métodos redefinidos de la subclase sean llamados. Por tanto, la misma expresión evaluada a diferentes momentos durante la ejecución de un programa puede resultar en llamados a diferentes métodos. Los métodos estáticos no pueden usar ligado dinámico.

Es importante recordar que el nombre, los tipos de los parámetros y el tipo de retorno (comúnmente llamado *method signature* o "signatura del método") de un método redefinido debe coincidir exactamente. Si no, entonces ocurre la sobrecarga de métodos, en la que el método tiene más de un significado entre las dos clases.

La regla que la accesibilidad de un método redefinido debe ser igual o incrementarse es necesaria para preservar la substitutibilidad. Si el objeto de una subclase debe ser usado donde un tipo de la superclase ha sido especificado, entonces debe soportar al menos la misma interfaz pública (o protegida, o por defecto). Remover un método de la interfaz prevendría la substitución.

El uso del ligado dinámico es parte esencial de la programación Orientada a Objetos, pero puede hacer difícil averiguar cuál método será llamado al leer el código fuente. Dada una expresión como `x.f()` o sólo `f()` (que realmente es `this.f()`), el programador debe primero determinar el tipo de `x` o `this`, y entonces inspeccionar la clase correspondiente, superclases y subclases, para averiguar qué clase de método es `f`, y si es redefinido. El programador debe también determinar los tipos de objetos que pueden ser referenciados por `x`. Esto puede ser tardado y confuso, pero considerar esto es importante. Es claro que un buen ambiente de desarrollo puede dar soporte para esta búsqueda, pero esto no significa que el comentario no deba hacerse.

El siguiente programa ilustra una variedad de métodos redefinidos y el uso del ligado dinámico:

```
class Superclass {
    // Metodo que puede ser redefinido
    public void f(int x){
        System.out.println("Superclass f: " + x);
        // Siempre llama al metodo g declarado en esta clase,
        // ya que g es privado
        g();
    }

    // No se puede redefinir el siguiente metodo
    private void g(){
        System.out.println("Superclass g");
    }

    // Metodo que puede ser redefinido
    protected void h(){
        System.out.println("Superclass h");
    }

    public void k(){
        System.out.println(Superclass k");
        // Siempre llama a Superclass g, ya que g no puede ser
        // redefinido
        g();
        // Llama a h dependiendo del tipo de objeto (el tipo de
        // this) usando ligado dinamico
        h();
        // Siempre llama a s en esta clase, ya que s es
        // estatico
        s();
    }
```

```
    public static void s() {
        System.out.println("Superclass static s");
    }
}

class Subclass extends Superclass {
    // Metodo redefinido. Debe ser publico
    public void f(int x) {
        System.out.println("Subclass f: " + x);
        // Llama a g en esta clase
        g();
    }

    // Nueva version de g, que no redefine la version de
    // Superclass
    private void g(){
        System.out.println("Subclass g");
    }

    // Metodo redefinido del heredado h con acceso incrementado
    // El hacer el acceso de este metodo privado o por defecto
    // generaria un error
    public void h(){
        System.out.println("Subclass h");
    }

    public static void s() {
        System.out.println("Subclass static s");
    }
}

class Override {
    public static void main (String[] args) {
        Superclass superclass = new Superclass();
        // Llama a la version de f en Superclass
        superclass.f(1);
        superclass.h();
        superclass.k();
        superclass.s();

        Subclass subclass = new Subclass();
        // Llama a la version redefinida de f y h en Subclass
        subclass.f(2);
        subclass.h();
```

```
        // Llama a la version de k en Superclass, ya que no
        // esta redefinida
        subclass.k();

        // Llama a la version de s en Subclass
        subclass.s();

        // Ahora, pone una variable de tipo Superclass a una
        // referencia a objeto de tipo Subclass
        superclass = subclass;

        // Llama a las versiones de f y h en Subclass
        superclass.f(3);
        superclass.h();

        // Llama a k para un objeto de Subclass
        superclass.k();

        // Llama a la version de s en Superclass, ya que este
        // metodo es estatico y no puede ser dinamicamente
        // ligado, por lo que el metodo depende el tipo de la
        // referencia
        superclass.s();
    }
}
```

3.2.7. Métodos Finales

Un método final de instancia no puede ser redefinido (pero aún puede ser sobrecargado). Un método final estático no puede ser re-declarado en una subclase. Una declaración de método final incluye al modificador `final`.

Los métodos finales previenen de que un método que tiene el mismo nombre y los mismos tipos de parámetros sea declarado en una subclase (el tipo del resultado es ignorado). Esto toma en cuenta tanto variables estáticas como de instancia. Sin embargo, el modificador `final` no previene que los métodos sean sobrecargados en una subclase.

Al igual que las clases estáticas, los métodos estáticos potencialemente permiten al compilador hacer optimizaciones en las llamadas a métodos.

3.2.8. Expresiones de invocación a Métodos

Una expresión de invocación o llamada a un método determina cuál método es llamado en base al nombre del método y sus parámetros. Una serie de reglas se usan

para determinar exactamente cuál método es invocado. Un método (que puede ser estático) se invoca mediante una expresión que consiste en una identificación de método seguida de una lista de parámetros actuales o argumentos:

identificadordemétodo(*lista de argumentos*);

La lista de argumentos puede estar vacía, pero el paréntesis debe estar presente.

El método a ser invocado puede ser identificado de varias formas:

- Usando el nombre del método directamente, si el método está declarado en el entorno de la misma clase o superclase:

  identificador (*lista de argumentos*);

- Usando una expresión de campo (el operador punto) con la referencia de un objeto:

  expresiónprimaria.*identificador*(*lista de argumentos*);

  donde la expresión primaria antes del punto debe evaluarse como referencia a objeto que puede entonces ser usada para seleccionar el nombre del método a ser invocado. Tal nombre debe ser accesible, lo cual generalmente significa que debe ser público, protegido, o declarado en la clase actual. El método puede ser estático.

- Usando una expresión de campo con el nombre de una clase o interfaz:

  nombredeclaseointerfaz.*identificador*(*lista de argumentos*);

  Esto permite invocar métodos estáticos o de instancia mediante especificar cuál clase o interfaz debe ser usada para hallar el método.

A fin de invocar un método, se usa un proceso llamado "vigía de método" (*method lookup*). Este es usado para determinar exactamente cuál método llamar, y que la llamada no sea ambigua. En algunos casos, un proceso vigía de método completo puede ser producido por el compilador, permitiendo exactamente determinar cuál método llamar en tiempo de compilación. Esto es lo que se llama "ligado estático" (*static binding*): la invocación del método se liga o asocia con un método durante la compilación.

En otros casos, sin embargo, no es posible para el compilador hacer el proceso vigía completo durante la compilación, ya que la información necesaria y completa no está disponible sino hasta tiempo de ejecución, es decir, en el momento en que el método es invocado. Para superar este problema, se utiliza el "ligado dinámico" (*dynamic binding*), donde el ligado se completa justo antes de que la llamada al método tenga lugar, y el tipo exacto de objeto del que se invoca el método es

conocido. El ligado dinámico es particularmente asociado con los tipos de referencia y herencia, ya que una referencia de un tipo superclase puede referirse a un objeto de cualquiera de un número de tipos de las subclases.

Para entender cómo funciona un vigía de método, la primera cuestión a considerar son los métodos sobrecargados, sin considerar los métodos heredados. Si una llamada se hace a una colección de métodos sobrecargados, debe ser posible seleccionar al menos uno para ser llamado. Esto se hace mediante comparar los tipos de los argumentos (parámetros actuales) con los tipos de los parámetros formales declarados para el método. Declaraciones donde no hay coincidencia entre tipos de parámetros actuales y formales se eliminan de la búsqueda. Si queda un solo método tras la eliminación, la invocación ya no es ambigua, y la expresión de llamada es válida. Si queda más de un método, esto significa un error.

Desafortunadamente, determinar si el tipo de un argumento y un parámetro son coincidentes es complicado debido a los varios tipos de conversión implícita, conocidas como compatibilidades por asignación. Por ejemplo, un método que toma un argumento `double` puede ser invocado con un argumento `int`, ya que el tipo `int` es compatible por asignación con `double`. Dado esto, considérese qué hacen las invocaciones a `f` en el siguiente ejemplo:

```
class Example {
    ...
    public void f(int x) { ...}
    public void f(double x) { ... }
    ...
    public void test(){
        f(10);    // Argumento int
        f(1.1);    // Argumento double
    }
}
```

Potencialmente, cuando `f` es llamada con un argumento `int`, cualquier método puede ser llamado. En tales casos, se hace un intento por encontrar la coincidencia "más significativa". La llamada `f(10)` es una coincidencia directa para `f(int x)`, así que éste es el método invocado. Invocar el método `f(double x)` requeriría una conversión implícita, y por lo tanto, la invocación es menos específica. La llamada `f(1.1)` no implica propiamente un problema, ya que es la única coincidencia dado que `double` no puede ser convertido a `int` automáticamente.

En seguida, considérese el siguiente caso:

```
class Example {
    public void f(int x, double y) { ... }
    public void f(double x, int y) { ... }
    ...
```

```
    public void test() {
        f(1.0,1);
        f(1,1);
        f(1,1.0);
    }
}
```

En este ejemplo, la primera llamada `f(1.0,1)` es válida ya que `f(int x, double y)` será rechazada debido a que `double` no es compatible por asignación con `int`, dejando sólo el otro método que coincide exactamente. La llamada `f(1,1.0)` es también válida por razones similares, resultando en una llamada a `f(int x, double y)`. Sin embargo, la llamada `f(1,1)` es inválida ya que cualquier versión de `f` puede ser llamada, y ambas requieren una conversión implícita. Ningún método resulta ser específico con respecto a la llamada. Al compilar la clase anterior, la llamada `f(1,1)` tendría que ser removida, o uno de los dos parámetros explícitamente propuesto como `double`.

Nótese que es muy posible declarar un conjunto de métodos sobrecargados que son potencialmente ambiguos respecto a las invocaciones de métodos. El compilador de Java se quejará sólo si encuentra una expresión de llamada ambigua. Si no, el código compilará sin notificación de ambigüedad potencial.

Los ejemplos anteriores usan los tipos primitivos `int` y `double` explotando el hecho de que un `int` es compatible por asignación con un `double`. Un razonamiento análogo se aplica para el uso de cualquier tipo y sus compatibles por asignación. En particular, no se olvide que al tratar con referencias, un subtipo es compatible por asignación con su supertipo.

Invocar un método sobrecargado es una extensión al comportamiento de llamar un método no-sobrecargado (un método no-sobrecargado puede ser considerado como un método sobrecargado con una sola posibilidad de elección). Las reglas para compilar una invocación a un método donde no hay métodos heredados involucrados pueden resumirse como sigue:

1. Localizar todos los métodos con el nombre requerido que se encuentran en el entorno, incluyendo tanto métodos estáticos como de instancia. Si no se encuentra un método, se reporta un error.

2. Hágase coincidir los tipos de los parámetros actuales con los tipos de los parámetros formales, eliminando todos aquellos métodos que difieren en uno o más tipos de parámetros, y en donde no es posible una conversión automática de tipos. Si sólo un método coincide, la invocación es válida y el método es seleccionado. Si más de un método coincide, o ningún método coincide, reporta un error.

3. Tómese los métodos remanentes y considere las conversiones automáticas para cada parámetro. Si un método puede ser invocado con un menor número

de conversiones que otro, aquél con mayor número de conversiones es eliminado. Tras la eliminación, si un método permanece como el más específico, la invocación es válida y el método es seleccionado. Si dos o más métodos permanecen entonces la invocación es ambigua, y se reporta un error.

La herencia introduce una complicación extra a la invocación de métodos, con respecto a los métodos heredados pero sobrecargados y los métodos redefinidos. Si una subclase declara uno o más métodos con el mismo nombre de uno o más métodos de la superclase, entonces, a menos que ocurra una redefinición, todos los métodos sobrecargarán al mismo nombre del método. Esto significa que al intentar resolver cuál método sobrecargado llamar debe tomarse en cuenta los métodos heredados y los métodos de la subclase. Más aún, la sobrecarga del nombre de un método en las subclases que aún no han sido escritas debe tomarse en cuenta también.

El proceso vigía de método se modifica para lidiar con estas cuestiones mediante cambiar la definición de lo que significa "método más específico" al tratar con métodos heredados o sobrecargados, y mediante depender del ligado dinámico.

Dada una expresión para invocar un método que resulte en la identificación de dos métodos potenciales que sobrecargan al mismo nombre, el método más específico es aquél que:

- Es declarado en la misma clase o en una subclase de la clase donde el otro es declarado, y
- Tiene cada parámetro como compatible por asignación con los parámetros correspondientes del otro método.

Para ilustrar esto, considere el siguiente ejemplo:

```
class A {
    void f(double d){ ... }
}

class B extends A {
    void f(int i) { ... }
}

...

B b = new B();
b.f(1); // Que metodo se llama?
```

El método `f(int)` en la clase `B` sobrecarga al método heredado `f(double)` en la clase `A`. La llamada de `b.f(1)` podría potencialmente invocar cualquier método. Sin embargo, por las reglas anteriores, `f(int)` resulta más específico, ya que se encuentra declarado en la subclase. Por lo tanto, `f()` declarado en la clase `B` es

seleccionado, lo que no es claramente notorio solamente al inspeccionar el código. Sin embargo, si el ejemplo se cambia como sigue, la llamada resultará ahora ambigua:

```
class A {
    void f(int i){ ... } // Notese el cambio de tipo
}

class B extends A {
    void f(double d) { ... } // Notese el cambio de tipo
}

...

B b = new B();
b.f(1); // Ambiguo
```

Aun cuando podría parecer que `f(int)` debiera coincidir, falla respecto a la regla 1, mientras que `f(double)` falla en la regla 2, y ningún método se puede considerar como más específico. Resulta todo en un mensaje de error:

```
Reference to f is ambiguous. It is defined in void f(double) and
void f(int).
```

La aplicación de la regla del más específico puede revisarse de nuevo considerando un método con dos argumentos:

```
class A {
    void g(double d, int i) { ... }
}

class B extends A {
    void g(int i, double d) { ... }
}

...

B b = new B();
b.g(1,1.0); // OK
b.g(1.0,1); // OK
b.g(1,1);    // Ambiguo
```

En este ejemplo, las dos primeras llamadas a `g` no representan un problema, ya que el único método sobrecargado en cada caso puede hacerse coincidir, y no es necesario utilizar la regla del más específico (por ejemplo, `b.g(1,1.0)` no coincide con `g (double, int)`, ya que `double` no es compatible por asignación con `int`). La tercera invocación, sin embargo, es ambigua dado que ambas declaraciones de `g`

podrían coincidir con ella, pero de acuerdo con la regla, ninguna resulta la más específica. Intercambiar las declaraciones de los dos métodos no hace mayor diferencia en este caso.

Los mismos principios se aplican cuando los argumentos son de tipos referencia:

```
class X { ... }
class Y extends X { ... }

class A {
    void h(X x) { ... }
}

class B extends A {
    void h(Y y) { ... }
}

...

B b = new B();
X x = new X();
Y y = new Y();

b.h(x);    // OK
b.h(y);    // OK
```

Aquí ambas llamadas al método `h` son válidas. Con la primera, sólo hay un método que puede coincidir: `h(X)` en la clase `A`. Con el segundo, ambos métodos `h` podrían coincidir, pero la regla dice que `h(Y)` en la clase `B` resulta el más específico, y es el que termina siendo llamado.

Si las declaraciones se intercambiaran:

```
class X { ... }
class Y extends X { ... }

class A {
    void h(Y y) { ... }
}

class B extends A {
    void h(X x) { ... }
}

...
```

```
B b = new B();
X x = new X();
Y y = new Y();

b.h(x);    // OK
b.h(y);    // Ambiguo
```

la segunda llamada a `h` se vuelve ambigua, ya que ambos métodos podrían coincidir, pero ninguno resulta el más específico.

Una vez que la sobrecarga de métodos ha sido resuelta, se debe considerar la redefinición de métodos. Si el método seleccionado puede ser redefinido, entonces la llamada al método podría necesitar ser dinámica. Dado que ningún método no-privado o no-estático puede ser redefinido (a menos que la clase o el método sea final), entonces todas las llamadas deben ser dinámicas. No debe olvidarse que subclases redefiniendo un método pueden ser añadidas al programa en cualquier momento sin recompilar las superclases, así que el ligado dinámico debe lidiar con objetos de clases que podrían sólo existir en el futuro.

El ligado dinámico trabaja utilizando código para hallar la clase del objeto por el que un método es invocado, verificando que la clase declara un método redefinido con el nombre, tipo de parámetros y tipo de retorno correctos, y entonces invocando al método. Aun cuando esto suena lento y costoso para ejecutarse, la información para hacer la coincidencia está codificada para reducir el tiempo extra al mínimo. Los archivos `.class` contienen la información, así que la verificación puede hacerse una vez que una clase es cargada.

En resumen, el proceso general para compilar la invocación o llamada a un método es como sigue:

1. Determinar dónde comenzar la búsqueda de un método, mediante encontrar el tipo de expresión denotando el tipo del objeto para el cual el método es invocado, y usar la clase correspondiente.
2. Localizar todos los métodos (de instancia o estáticos) que son accesibles (es decir, `private`, `protected`, `public`, etc. que sean apropiados) con el nombre que coincida con el método invocado. Esto implica una búsqueda en el entorno local y los entornos anidados (si los hay), el entorno de la clase y de las superclases. Si no se encuentra una declaración se reporta un error.
3. Eliminar todos los métodos cuyos parámetros formales, en número o tipo, no coinciden con los parámetros actuales de la invocación. Si sólo hay un método tras la eliminación, se continúa con el paso 5. Si no queda ningún método, se reporta un error.
4. Aplicar la regla del más específico y determinar si un método es más específico que el resto. Si queda sólo un método, se sigue con el paso 5. Si no queda ningún método, se reporta un error.

5. Determinar si la invocación al método necesita ser dinámica, y si así es, generar una llamada dinámica a método, y en caso contrario, una llamada estática.

3.2.9. Búsqueda de nombre modificado para miembros de una clase

Los objetos de clases miembro, locales y anónimas están contenidas dentro de un objeto u objetos (si las clases se encuentran profundamente anidadas) de clases residentes. Esto crea una "jerarquía de contención" (*containment herarchy*) de objetos que es diferente de la jerarquía de herencia entre clases. La jerarquía de contención representa otro conjunto de entornos, donde es necesario buscar cuando la búsqueda de una variable o método toma lugar.

La sintaxis para las clases miembro, locales o anónimas se encuentra descrita en otro punto de estas notas. Sin embargo, nótese que la sintaxis extendida para el uso de `this` puede usarse para accesar explícitamente variables y métodos en los objetos contenedores:

nombredeclase.**this.***nombredevariable*
nombredeclase.**this.***nombredemétodo*

Como hay dos distintas jerarquías en que buscar por nombres de variables y métodos, hay un problema potencial si el nombre se encuentra en ambas. La jerarquía de herencia se busca primero, seguida de la jerarquía de contención. Si el mismo nombre con el mismo tipo se encuentra en ambas jerarquías, ambas declaraciones son accesibles, y se reporta un error. Estos conflictos pueden resolverse utilizando la sintaxis extendida de `this` para excplícitamente nombrar la clase que declara el nombre a ser usado. Por ejemplo, si ambas jerarquías declaran una variable accesible `x` dentro del entorno, entonces una expresión como:

```
A.this.x
```

puede ser usada para acceder a `x`, donde `A` es el nombre de la clase que declara la variable que necesita ser accesada.

El siguiente programa ilustra el uso de clases miembro y herencia. Nótese que la clase miembro es también una subclase de una clase no-miembro, así que existen dos jerarquías de herencia, en una de las cuales la clase miembro pertenece, y la otra en que es parte la clase contenedora. El objeto de la clase miembro también será parte de una jerarquía de contención definida por la clase contenedora y su superclase. Nótese el comentario indicando el conflicto que ocurre entre la jerarquías de herencia de la clase miembro y la jerarquía de contención.

```
class A {
    public void f() {
        System.out.println(name + " f");
```

```
    }
    protected static String name = "A";
}

class Superclass {
    public void name(){
        System.out.println(name);
    }
    protected static String name = "Superclass";
    protected String vname = "Superclass instance variable";
}

class Subclass extends Superclass {
    private class Member extends A {
        // Las clases miembro pueden heredar
        public void show() {
            // Esta es una variable de instancia
            // Su uso esta bien, ya que la variable de
            // instancia name oculta cualquier otra variable
            // heredada o contenida
            System.out.println(name);

            // Las variables pueden ser explicitamente usadas
            // en cada entorno usando la sintaxis this

            System.out.println(Subclass.this.name);
            System.out.println(Superclass.this.name);
            System.out.println(A.this.name);
            System.out.println(Subclass.this.vname);
            System.out.println(Superclass.this.vname);

            // No hay conflicto aqui, ya que solo la jerarquia
            // de contencion tiene vname
            System.out.println(vname);

            // Los metodos del objeto contenedor pueden ser
            // llamados
            Subclass.this.name();
            Superclass.this.name();

            // Si this es omitido, entonces los metodos en el
            // objeto contenedor todavia pueden ser accesados.
            // Ya que no se hereda el metodo name, tampoco hay
            // conflicto
            name();
```

```
            // Lo siguiente es ambiguo, dado el conflicto
            // entre name por herencia y por contencion
            // f(); // ERROR!
            A.this.f();
            Subclass.this.f();
        }

        public void test() {
            System.out.println(name + " test");
        }

        private String name = "Member";
    }

    public void name() {
        System.out.println(name);
    }

    public void test() {
        Member m = new Member();
        m.show();
    }

    public void f() {
        System.out.println("Subclass:f");
    }

    protected static String name = "Subclass";
    protected String vname = "Subclass intance veriable";
}

class Lookup1 {
    public static void main(String[] args) {
        Subclass subclass = new Subclass();
        subclass.test();
    }
}
```

3.2.10. Métodos abstractos

Un método puede ser declarado como abstracto, de tal modo que pueda ser redefinido por las subclases. Un método abstracto no tiene cuerpo. Su declaración termina con punto y coma, y no un enunciado compuesto:

modificadores **abstract** *tipo nombredelmetodo*(*lista de parámetros*);

La lista de parámetros es una secuencia de cero o más parámetros separados por comas. Cada parámetro consiste en un nombre de tipo seguido por un identificador (por ejemplo, `int x`). El identificador debe estar presente aun cuando no sea utilizado. Un método que redefine a un método abstracto no requiere utilizar los mismos nombres de identificadores.

Una clase que declara uno o más métodos abstractos debe ser declarada como clase abstracta. Los métodos privados y estáticos no pueden ser abstractos, ya que no hay manera de redefinirlos.

Los métodos abstractos permiten que una clase abstracta especifique la interfaz completa de un método público o protegido, aun cuando no puede proveer un cuerpo para el método. Las reglas del lenguaje aseguran que las subclases concretas deben proveer de un implementación completa para cada método abstracto heredado. Las subclases abstractas de una clase abstracta pueden escoger entre implementar un método abstracto heredado o añadir métodos abstractos adicionales.

3.2.11. Métodos heredados de la clase `Object`

La clase `Object` es la base de toda la jerarquía de herencia en Java, y declara implementaciones por omisión de un pequeño número de métodos que pueden ser redefinidos por sus subclases, es decir, toda clase que se implementa en Java. Para que los objetos de las subclases tengan un comportamiento correcto, el programador debe asegurarse que la implementación por omisión sea adecuada, o proveer una versión redefinida mejor.

La clase `Object` declara los siguientes métodos que pueden ser redefinidos por las subclases:

```
public boolean equals (Object obj);
public String toString ();
public final native int hashCode();
protected native Object clone();
protected void finalize();
```

Estos métodos también soportan arreglos.

De los cinco métodos que pueden ser redefinidos, tres son públicos y pueden ser invocados por cualquier objeto o instancia de cualquier clase. Los otros dos son protegidos, y necesitan ser redefinidos como métodos públicos para que puedan ser invocados en la clase que los declara.

`boolean equals(Object obj)`

El método `equals` es usado para comparar el objeto que lo llama con el objeto en su argumento. La implementación por defecto retorna `true` si los dos objetos son en realidad el mismo objeto (usando el operador `==`). Una subclase debe redefinir este método a fin de comparar los valores de los dos objetos, en general mediante sistemáticamente comparar uno a uno los valores de sus variables de instancia.

El método `equals` de una subclase debe enfocarse en comparar el estado añadido por la clase, e invocar `super.equals()` para obtener el estado de su superclase comparado. Nótese, sin embargo, que una subclase directa de la clase `Object` no requiere invocar al método `Object.equals()`.

El programador necesita decidir cómo comparar dos objetos de la misma clase. Puede ser suficiente tan solo comparar todas las variables de instancia (especialmente, si todas ellas son de tipo primitivo). Sin embargo, las variables de referencia a otros objetos requieren que cada uno de los objetos a los que se hace referencia sea comparado completamente, dependiendo que en sus clases haya implementada una versión de `equals` apropiada. Por tanto, una comparación completa puede resultar en toda una serie de invocaciones y ejecuciones de métodos `equals` de un número de clases. El programador debe tomar en cuenta que esta operación puede consumir tiempo operacional de ejecución.

El método `equals` implementa una relación de equivalencia. La documentación de JDK define un conjunto riguroso de reglas para lo que significa igualdad, las cuales el programador debe considerar cuando redefine `equals`:

- Es reflexiva: para cualquier valor de referencia `x`, `x.equals(x)` debe retornar `true`.
- Es simétrica: para cualesquiera valores de referencia `x` y `y`, `x.equals(y)` debe retornar `true` si y solo si `y.equals(x)` retorna `true`.
- Es transitiva: para cualquier valores de referencia `x`, `y` y `z`, si `x.equals(y)` retorna `true`, y `y.equals(z)` retorna `true`, entonces `x.equals(z)` debe retornar `true`.
- Es consistente: para cualesquiera valores de referencia `x` y `y`, múltiples invocaciones de `x.equals(y)` consitentemente retornan `true` o `false`. Para cualquier valor de referencia `x`, `x.equals(null)` debe retornar siempre `false`.

El método `equals` para la clase `Object` implementa la relación de equivalencia más discriminante posible entre objetos, es decir, que para dos referencias `x` y `y`, este método retorna `true` si y solo si `x` y `y` se refieren al mismo objeto (`x == y` tiene el valor de `true`).

Es importante implementar `equals` correctamente si los objetos instancia de una clase son usados por otras clases o métodos que dependen de la relación de igualdad.

`String toString()`

Este método es utilizado para obtener una cadena de caracteres representando el valor del objeto para el cual se invoca. Por defecto, el método de la clase `Object` retorna una cadena como:

```
ClassName@1cc7a0
```

Este es el nombre de la clase del objeto, seguido por una `@`, que se sigue por el valor hexadecimal del código `hash` del objeto (véase la siguiente sección). Para generar una cadena más útil, una subclase puede redefinir este método y retornar cualquier cadena que sea una representación razonable del valor del objeto.

El método `toString` es ampliamente usado por otros métodos para convertir de tipo referencia de un objeto al tipo `String`, notablemente cuando se utilizan "flujos" (*streams*), así que normalmente debe ser redefinido para hacer algo útil.

`int hashCode()`

Un código *hash* es un valor entero único que representa el valor de un objeto, de tal modo que el valor del objeto completo puede ser relacionado con un entero. Cada valor distinto que un objeto puede representar debe tener un código *hash* relativamente único (ya que el número de valores es frecuentemente mayor que lo que se puede representar mediante un `int`). Los códigos *hash* se usan como valores clave en una tabla *hash* (como la que se implementa en la clase `HashTable` en `java.util`).

La versión por defecto de `hashCode` intentará generar un código *hash* para cualquier objeto, pero puede terminar generando diferentes códigos *hash* para objetos diferentes representando el mismo valor. Si tal es el caso, entonces `hashCode` debe ser redefinido para implementar una nueva función *hash* que genere códigos *hash* correctos. La documentación de JDK establece que el contrato general para `hashCode` es:

"En cualquier momento que se invoca el mismo objeto más de una vez durante la ejecución de una aplicación Java, el método ***`hashCode`*** *debe consistentemente retornar el mismo valor* ***`int`****. Este valor* ***`int`*** *no requiere permanecer consistente de una ejecución a otra. Si dos objetos son iguales de acuerdo con el método* ***`equals`****, entonces llamar al método* ***`hashCode`*** *para cada uno de estos objetos debe producir el mismo resultado* ***`int`****".*

En la práctica, sólo aquellos objetos que pueden ser almacenados en una tabla *hash* requieren del método `hashCode`. Los programadores generalmente dependen de la implementación por defecto de `hashCode`, en lugar de implementar una nueva versión que podría significar una dura labor.

`Object clone()`

El método `clone` crea una copia de un objeto. Por defecto, sólo el objeto es copiado, es decir, se copian los valores de sus variables primitivas. Sin embargo, los valores de tipos referencia contenidos en el objeto no se copian. Esto se conoce como "copia superficial" (*shallow copy*).

La debilidad potencial del método `clone` por defecto es que su uso puede resultar en dos copias de un objeto, ambas referenciando o compartiendo una colección de otros objetos. Si el estado de alguno de los objetos referenciados cambia, entonces efectivamente también cambia el estado de todos los objetos que le hacen referencia. Para sobrellevar esto, `clone` puede ser redefinido para relizar una "copia profunda" (*deep copy*), en donde tanto el objeto a copiar como los objetos a los que hace referencia son copiados. La redefinición de `clone` también necesita ser hecha pública, si se espera utilizarse en general.

Una copia profunda completa depende de `clone` o una operación de copia similar, que requiere ser implementada correctamente por cada clase de los objetos a los que se hace referencia. Como es el caso con `equals`, invocar a `clone` puede resultar en una larga secuencia de métodos invocados, los cuales pueden usar mucho del tiempo de ejecución de un programa. Como resultado, es importante definir cuidadosamente qué significa copiar un objeto y qué realmente debe hacerse.

Un método `clone` debe ser responsable de clonar el estado (es decir, el valor de las variables) declaradas en su clase, y debe invocar a `super.clone()` para copiar el estado de la superclase, a menos que tal clase sea `Object`, en cuyo caso la invocación al método `clone` por defecto puede no ser necesaria.

Si un objeto no puede ser clonado, la excepción `CloneNotSupportedException` puede ser arrojada, por lo que el método debe ser declarado con una cláusula `throws`. Ciertas clases no pueden redefinir `clone`, a favor de proveer otros métodos para copiar o utilizar constructores (por ejemplo, la clase `String` de la biblioteca de Java).

`void finalize()`

El método `finalize` es llamado automáticamente por el recolector de basura cuando un objeto no es referenciado y puede ser desechado. La versión por defecto en la clase `Object` es un método con cuerpo vacío. El recolector de basura puede ejecutarse en cualquier momento, por lo que no hay una forma confiable de determinar cuando será invocado `finalize` para un objeto en particular. Típicamente, sin embargo, el recolector de basura es normalmente invocado cuando la memoria disponible para el programa en ejecución disminuye.

`finalize` sólo necesita ser redefinido si un objeto tiene un estado que no puede ser correctamente manipulado simplemente con desecharlo. Por ejemplo, cuando

sus datos deban ser escritos a un archivo antes de desecharlo o podrían perderse, o una conexión de red que debe ser propiamente cerrada.

Un método `finalize` debe ser responsable de finalizar los objetos de la clase en la que se encuentra declarado, y debe invocar `super.finalize()` para coordinarse con su superclase. Un método `finalize` puede ser declarado para arrojar una excepción si un error ocurre. Si una excepción se arroja, será atrapada por el recolector de basura e ignorada, dejando al programa proseguir su ejecución.

Para mayor información acerca de estos métodos pertenecientes a la clase `Object`, refiérase a la documentación estándar de JDK.

3.3. Herencia

Una subclase hereda de una superclase mediante extenderla. La subclase toma todas las declaraciones de variables y métodos de la superclase (aún cuando no todos puedan ser accesibles) y puede añadir nuevas variables o métodos, o sobrecargar (*override*) los existentes.

Una subclase hereda de una superclase usando la palabra clave `extends`:

class *nombredelasubclase* **extends** *nombredelasuperclase* **{**
declaraciones de variables y métodos
}

El cuerpo de una clase conteniendo declaraciones de variables y métodos es tal y como se ha descrito anteriormente.

La herencia se aplica a las clases estándar de alto nivel, las clases anidadas de alto nivel, las clases miembro, las clases locales y las clases anónimas.

Una clase puede heredar de cualquier otra clase que no es `final`. Los objetos de la subclase contienen todas las variables de instancia declaradas por la superclase. También, todos los métodos declarados por la superclase pueden ser llamados desde el objeto de la subclase. El hecho que la subclase tiene copias de las variables de instancia de la superclase hace que esto último sea razonable y permisible. Nótese, sin embargo, que las reglas de accesibilidad son respetadas, así que las variables y métodos privados no son accesibles a los métodos e inicializadores de la subclase. Una clase puede contener partes que algunos de sus propios métodos no pueden accesar.

Crear subclases puede ser repetido tantas veces, o hasta tantos niveles, como se desee; una subclase puede tener a la vez otra subclase, y ésta otra, etc. Esto es a lo que se refiere como una "cadena de herencia". Sin embargo, es una buena práctica limitar el número de niveles a menos de cinco.

Una clase puede sólo tener una superclase, pero puede tener tantas subclases como sea necesario. Una colección de clases en una relación de herencia es referida como una jerarquía de clases (*class hierarchy*). Debido a la regla de "una sola superclase", esta jerarquía forma un árbol.

Todas las clases heredan directa o indirectamente de la clase `Object` (es decir, todas son inmediatamente subclases de `Object` o se encuentran en un nivel más abajo en una cadena jerárquica que comienza con `Object`). Esto es cierto aun cuando una clase no haya sido explícitamente declarada como una subclase de `Object` usando la palabra clave `extends`.

La relación de herencia entre clases también se aplica a los tipos que las clases definen, de tal modo que es posible hablar de "supertipos" y "subtipos". Esto

permite compatibilidad para la asignación o conversión entre tipos de referencia, así que es posible asignar una referencia a un objeto de un subtipo a una variable cuyo tipo es el supertipo. Esto también funciona para el paso de parámetros y el retorno de resultados de un método.

La herencia tiene el efecto de incrementar el número de entornos que necesitan ser revisados para verificar que una variable o método está en un entorno, y si es accesible.

Para las variables, la revisión procede mediante checar la declaración de los identificadores de las variables en varios entornos. Tanto las declaraciones estáticas como de instancia son verificadas. El orden de la búsqueda es como sigue, y termina cuando el identificador es hallado:

1. Verificar el entorno local y cualquier entorno anidado.
2. Verificar el entorno de clase.
3. Verificar cada entorno de superclases, a lo largo de la cadena de herencia.

Si en algún otro entorno se encuentra una declaración del identificador de una variable, se reporta un error. Si no se encuentra una declaración en todos los entornos, entonces también se reporta un error de declaración de la variable. Si varias variables en diferentes entornos son declaradas con el mismo identificador, entonces la primera declaración encontrada es utilizada, es decir, aquella en el entorno más cercano. Las declaraciones en entornos anidados se dice están ocultas (*hidden* o *shadowed*) del entorno anidado.

Si el nombre de la variable es parte de una expresión de campo con un nombre de tipo del lado izquierdo (por ejemplo, `Math.PI`, que es una variable estática `double` en la clase `Math` del paquete `java.lang` de la biblioteca de Java, cuyo valor es aproximadamente π), entonces la búsqueda comienza en el entorno del nombre de la clase.

El siguiente ejemplo muestra un uso muy simple de la herencia, e ilustra qué métodos de ambas subclase y superclase pueden ser invocados por los objetos de la subclase. También, nótese que la referencia de la subclase es asignada a una variable del tipo de la superclase.

```
class Superclass {
    public void supermethod() {
        System.out.println("Superclass");
    }
}

// Esta clase hereda supermethod() de Superclass, asi que puede
// invocarlo a partir de objetos de la subclase
```

```
class Subclass extends Superclass {
    public void submethod(){
        System.out.println("Subclass");
    }
}

class Inherit1 {
    public static void main(String[] args) {
        // Crea un objeto Superclass y llama al metodo
        // supermethod()

        Superclass superclass = new Superclass();
        superclass.supermethod();

        // Crea un objeto Subclass y llama a ambos metodos
        // supermethod() y submethod()

        Subclass subclass = new Subclass();
        subclass.supermethod();
        subclass.submethod();

        // La asignacion en valida, pero no a la inversa
        superclass = subclass;
    }
}
```

3.3.1. Palabras Clave `private`, `protected`, y Herencia

Las variables y los métodos pueden ser declarados con el modificador `public`, `protected` o `private`. Lo importante para el programador es si declarar métodos o variables como `public`, `protected`, `private` o sin modificador de acceso. Todas estas declaraciones tienen ventajas y desventajas. Los métodos y las variables privados no son accesibles a una subclase, aun cuando son parte de la infraestructura de los objetos de la subclase. Los métodos y las variables protegidos son accesibles a una subclase, y proveen una forma de abrir la encapsulación dada por la clase en una forma controlada.

- La declaración `private` ofrece una garantía de que ninguna otra clase, incluyendo las subclases, pueden acceder al método o variable. Esto evita cualquier dependencia directa en los detalles de las declaraciones. Por tanto, si se realizan cambios a los métodos o variables privados en una clase, entonces ninguna otra clase necesita ser editada. Sin embargo, esto puede requerir la adición de métodos de acceso a la superclase para accesar indirectamente sus métodos o estado privado.

- La declaración `protected` permite que una subclase (tal vez de un paquete diferente) accese directa y eficientemente a su superclase cuando pueda ser útil para las variables de instancia de la subclase y para los métodos auxiliares que no deban ser públicos. Esto permite una forma controlada de compartir. Sin embargo, esto puede llevar a que en la subclase se utilice errónamente las características heredadas y volverse innecesariamente dependiente de ellas.
- La declaración `public` hace accesible a la subclase y a todo lo demás. Esto es esencial si la clase tiene instancias que proveen de un servicio útil, pero puede ser peligroso si mucho se hace público, especialmente en el caso de variables de instancia.
- El acceso por omisión es igual al acceso público si una subclase se encuentra en el mismo paquete, e igual al acceso privado si no.

Como una regla general, las declaraciones deben ser públicas si clientes no relacionados deben usarlas, o de otra manera deberían hacerse privadas por omisión a menos que sea claro que las subclases necesitan acceso directo, en cuyo caso podrían ser protegidas. Como una superclase puede ser escrita antes que una subclase haya sido pensada, las declaraciones protegidas requieren una planeación cuidadosa.

Un punto a recordar acerca de las declaraciones protegidas es que en Java éstas son accesibles tanto a subclases como a clases en el mismo paquete. Esto hace que el uso de declaraciones protegidas sea algo arriesgado, ya que hace que varias clases tengan acceso a un elemento protegido, pero previendo cierta flexibilidad a la apertura de la encapsulación.

El siguiente ejemplo ilustra el uso de variables y métodos protegidos y privados.

```
class Superclass {
    // Metodo que sera heredado pero no accesible a una subclase
    private void f() {
        System.out.println("f");
        // Puede llamarse desde aqui a g y h
        // g();
        // h();
    }

    // Metodo que sera heredado y es accesible a una subclase y a
    // otras clases dentro del mismo paquete
    protected void g(){
        System.out.println("g");
        // Puede llamarse desde aqui a f y h
        // f();
        // h();
    }
```

```
    // Metodo compartido que no es sobreescrito en la subclase
    public void h() {
        System.out.println("Shared");
        // Puede llamarse desde aqui a f y g
        // f();
        // g();
    }
    public int i = 5;
    protected int j = 10;
    private int k = 20;
}

class Subclass extends Superclass {
    public void test() {
        i = 10; // Bien. Variable publica heredada
        j = 20; // Bien. Variable protegida heredada
        // k = 30;  // ERROR. No se puede acceder una variable
                    // privada
        // f(); // ERROR. f es privado en la superclase
        g();
        h();
    }
}

class Inherit2{
    public static void main(String[] args) {
        Superclass superclass = new Superclass();
        // superclass.f(); // ERROR. El metodo f es privado

        // Se puede llamar a g ya que aun cuando el metodo es
        // protegido, esta clase esta en el mismo paquete
        superclass.g();
        superclass.h();

        Subclass subclass = new Subclass();
        subclass.test();
        // subclass.f();  // ERROR. El metodo f es heredado
                          // pero privado

        // Puede llamarse a g ya que aunque el metodo es
        // protegido esta clase esta en el mismo paquete
        subclass.g();
        subclass.h();
    }
}
```

3.3.2. Constructores y Herencia

En Java, todas las clases existen como parte de una jerarquía de herencia. Cuando un objeto es creado, un constructor debe ser invocado por cada superclase y subclase de la jerarquía. Esto se refuerza por las reglas propias del lenguaje y el compilador.

La palabra clave `super` puede ser usada para explícitamente llamar al constructor de una superclase:

super(*lista de argumentos***);**

La lista de argumentos es usada para seleccionar un constructor en particular. Puede ser vacía, en cuyo caso el constructor por omisión de la superclase es invocado.

Los constructores no son heredados, y no pueden ser redefinidos. Su sobrecarga está restringida al conjunto de constructores declarados en una clase.

El uso básico de constructores ha sido introducido anteriormente, y esta sección completa tal descripción. Debe notarse que, aparte del constructor de la clase `Object`, la cual no tiene superclase, todos los constructores necesitan invocar un constructor de una superclase. De hecho, la clase `Object` depende de un constructor por omisión generado por el compilador, en lugar de declarar uno explícitamente. Es interesante que la documentación de JDK (y otras fuentes) muestran a `Object` como si tuviera un constructor por omisión explícito. En caso de duda, inspecciónese el código fuente de la clase `Object`.

Cuando se crea un objeto de una subclase, la siguiente secuencia de eventos toma lugar:

1. El objeto es alojado en memoria y el constructor del objeto de la subclase es invocado.
2. Antes de que cualquier enunciado del cuerpo del constructor sea ejecutado, un constructor de la superclase es invocado o si el primer enunciado utiliza `this` (véase a continuación) entonces otro constructor en la subclase es invocado, lo que repite este paso.
3. Cuando el constructor de la superclase (o `this`) retorna la llamada, las variables de instancia son inicializadas mediante un bloque inicializador de instancias que se ejecuta en el orden en que las variables de instancia están escritas.
4. El cuerpo del constructor es ejecutado.
5. La invocación del constructor retorna un objeto inicializado.

Por tanto, crear un objeto resulta en una serie de llamadas a constructores e inicilizaciones. Esto sucede automáticamente y no puede ser detenido. El programador puede, sin embargo, controlar cuál método de la superclase es invocado por

el constructor utilizando la palabra clave `super` seguido por una lista de argumentos. Si se usa tal enunciado, éste debe aparecer como el primer enunciado dentro del cuerpo del constructor:

```
public Example(int a, String b) {
    super(a, 1, b); // Llamada al constructor de la superclase,
                    // que toma int, int, String como argumentos
}
```

Cualquier constructor de la superclase puede ser invocado (incluyendo al constructor por omisión). No importa si el número de parámetros es diferente al dado por el constructor de la subclase. Es claro que el constructor apropiado de la superclase no debe ser privado.

Si `super` no es usado explícitamente, entonces una llamada implícita a `super()` se inserta por el compilador como el primer enunciado del cuerpo del constructor, a menos que una llamada explícita se haga a otro constructor en la misma clase usando `this`. Por tanto, todos los constructores (excepto el de `Object`) que no usan explícitamente `super` o `this` realmente consisten en lo siguiente:

```
public A() {
    super();
    ...
    // el resto del cuerpo del constructor
}
```

Capítulo 4

Tópicos Especiales en Java

4.1. Paquetes

Un paquete permite que una colección de clases sea agrupada en una sola unidad y bajo un mismo nombre que también actúa como un entorno. Hay dos partes de la sintaxis de paquete: una para declarar los paquetes en sí y otra para importar clases de otros paquetes.

Un paquete se declara usando un enunciado de paquete (*package statement*) que debe aparecer antes que ningún otro enunciado en un archivo fuente (también conocido como unidad de compilación). Tiene la forma:

package *nombredelpaquete*;

El nombre del paquete es un identificador o una serie de identificadores separados por un punto (como por ejemplo `java.awt.event`).

Las clases se importan dentro de un paquete desde otro paquete usando un enunciado de importación. Estos vienen en dos variedades:

import *nombredelpaquete.nombredelaclase*;
ó
import *nombredelpaquete.**;

La primera forma importa una clase específica nombrada, mientras que la segunda forma es una abreviación conveniente que permite que todas las clases de un solo paquete sean importadas de una vez.

Para entender cómo los paquetes son usados es necesario darse cuenta de que los nombres de los paquetes mapean a nombres de directorios dentro del almacenamiento de archivos locales (esto podría extenderse en el futuro a sitios de Internet). Cada directorio contiene todos los archivos `.class` para las clases dadas en el paquete. Las herramientas como el compilador de Java localizan estos directorios en forma

relativa a series de puntos fijos en el almacenamiento de archivos, que se almacenan como rutas de archivos en una variable de ambiente llamada `CLASSPATH` (diferentes ambientes de desarrollo pueden proveer mecanismos alternativos para las variables de ambiente).

El mapeo del nombre de un paquete a un directorio específico se hace mediante primero tomar cada componente del nombre del paquete y mapearlo a una ruta relativa apropiada para el almacenamiento de archivos en la máquina local. Por ejemplo, un paquete llamado:

```
adts.containers.sorted
```

se mapearía a la ruta en Windows:

```
\adts\containers\sorted
```

o a la ruta en UNIX:

```
/adts/containers/sorted
```

La ruta generada es entonces añadida como entrada a la variable de ambiente `CLASSPATH` para generar una ruta completa. La ruta completa es usada para buscar en cada directorio por clases. Por ejemplo, cuando se usa Windows y la variable `CLASSPATH` contiene:

```
.;D:\myclasses
```

se generan los siguientes nombres de directorios para el paquete antes mencionado:

```
.\adts\containers\sorted
D:\myclasses\adts\containers\sorted
C:\jdk1.1.3\lib\adts\containers\sorted
```

suponiendo que Java ha sido instalado en `C:\jdk1.1.3`.

Si se trata de un sistema UNIX, entonces para el `CLASSPATH`:

```
.:$HOME/lib/Java
```

resulta los siguientes nombres de directorios generados para el paquete antes mencionado:

```
./adts/containers/sorted
$HOME/lib/Java/adts/containers/sorted
/opt/jdk1.1.3/lib/adts/containers/sorted
```

esto suponiendo que Java ha sido instalado en el directorio `/opt/jdk1.1.3`.

El archivo con el mismo nombre de la clase que se busca debe encontrarse en uno de esto directorios. Si un archivo `.class` es encontrado, entonces se utiliza. Si un archivo `.class` no es hallado, pero se encuentra un archivo `.java`, éste es compilado para crear una archivo `.class`, que puede entonces ser usado.

Nótese que la sintaxis de la variable de ambiente `CLASSPATH` sigue la sintaxis del sistema operativo: bajo Windows, el separador de ruta es un punto y coma (ya que los dos puntos se usan como parte de la especificación de la unidad de disco), mientras que en UNIX se utilizan como separadores de ruta los dos puntos.

Es también posible especificar el nombre de un archivo `.zip` (un archivo conteniendo un número de archivos en forma comprimida) en el `CLASSPATH`, así como si se tratara de directorios conteniendo archivos `.class`. Si este es el caso, entonces se buscan en el archivo `.zip` los archivos `.class` de forma análoga a la forma en que los directorios se buscan para hallar archivos `.class`.

Cuando una clase es declarada para ser parte de un paquete usando el enunciado `package`, se espera que el archivo `.class` coincidente se localice en el directorio correspondiente. Durante el desarrollo de la clase, el archivo `.java` es generalmente localizado en el mismo directorio por conveniencia. Los nombres de los paquetes pueden tener tantos subdirectorios como se desee, pero mientras sean más, más profunda será la estructura del directorio.

Los enunciados de importación (*import statements*) especifican ya sea una sola clase, o todas las clases del paquete. El mismo mapeo de nombres de paquetes a directorios es utilizada para localizar la clase o clases que se importan. Si el mapeo falla, la compilación falla también.

Si cualquier clase de otro paquete es usada en el paquete actual, debe ser importada. Para todas las clases, incluyendo las clases de las bibliotecas de Java, ésto debe hacerse explícitamente, excepto para las clases en `java.lang`, las cuales siempre se importan en forma implícita.

Para importar de otro paquete, una clase debe ser declarada pública (recuérdese que sólo una clase en el paquete puede ser pública). Las clases no-públicas son locales a su paquete, dando un grado de control sobre la visibilidad de las clases.

Las clases en un paquete se declaran en una serie de archivos fuente, así que un paquete se deriva a la vez de una colección de archivos fuente. Una clase puede añadirse a un paquete mediante simplemente incluir el enunciado `package` al principio de la clase. Nótese, sin embargo, que una clase (y por lo tanto un archivo) puede sólo ser parte de un paquete único, así que sólo puede haber un enunciado `package` en un archivo fuente.

Los enunciados `package` se tratan en forma bastante casual, de tal modo que una clase puede ser declarada para ser parte de cualquier paquete que el programador desee, hasta dentro de los paquetes de las bibliotecas de Java. La única restricción en esto es que el directorio coincidente con el nombre del paquete debe ser accesible y escribible, de tal modo que el archivo `.class` pueda colocarse ahí.

Si una clase no es declarada dentro de un paquete (como son todos los ejemplos de programas vistos hasta ahora), entonces pertenece a un paquete por omisión sin nombre, así que todas las clases realmente pertenecen a un paquete. Para propósitos de aprendizaje y prueba de un ejemplo, esto resulta muy conveniente, ya que todos los archivos fuente de un programa pueden estar contenidos en el directorio actual de trabajo, sin causar ningún conflicto con el proceso de mapeo de paquetes a directorios. Tan pronto como el programador comienza a usar paquetes, sin embargo, la estructura apropiada de directorios debe colocarse.

4.2. Interfaces

La declaración de interfaces permite la especificación de un tipo de referencia sin proveer una implementación en la forma en que una clase lo hace. Esto provee de un mecanismo para declarar tipos que son distintos de las clases, lo que da una extensión importante a la forma en que los objetos y la herencia se usan en Java.

Las interfaces explotan el concepto de "conformación de tipos" (*type conformance*). Un tipo puede ser especificado por un nombre y un conjunto de métodos (cada uno de los cuales tiene un nombre, un conjunto de tipos de parámetros y un tipo de retorno). Un tipo puede "conformarse" con otro tipo si especifica el mismo conjunto de métodos (o posiblemente más), y cada método tiene el mismo nombre, tipos de parámetros y tipo de retorno. Más aún, los dos tipos no tiene que estar relacionados por herencia, dando una mayor libertad a qué tipos pueden conformarse con otros tipos.

Para ilustrar la idea básica, si `Type1` se especifica como:

```
void f();
int g(String s);
```

y `Type2` se especifica como:

```
void f();
int g(String s);
double h(int x);
```

entonces se dice que `Type2` se conforma con `Type1`, ya que tiene todos los métodos que `Type1` posee. Esto se escribe frecuentemente como una relación de la forma:

`Type1` $\leq$ `Type2`

Si `Type2` se conforma con `Type1`, entonces un valor de `Type2` puede ser substituido por un valor de `Type1`, ya que soporta todos los métodos que un valor `Type1` tiene.

La idea de substitutibilidad debe ser familiar a partir de observar las propiedades de las clases y la herencia. La característica crucial que las interfaces añaden es que son definidas fuera de la jerarquía de herencia (no se olvide que todas las clases heredan de la clase `Object`, lo que hace que todas pertenezcan a la misma única jerarquía de clases). Una vez definida, una interfaz puede ser "implementada" por una clase que crea un nuevo tipo que se conforma al tipo de la interfaz. Más aún, cualquier clase puede implementar la interfaz, sin importar su disposición en la jerarquía de clases. La figura 4.1 ilustra la idea básica.

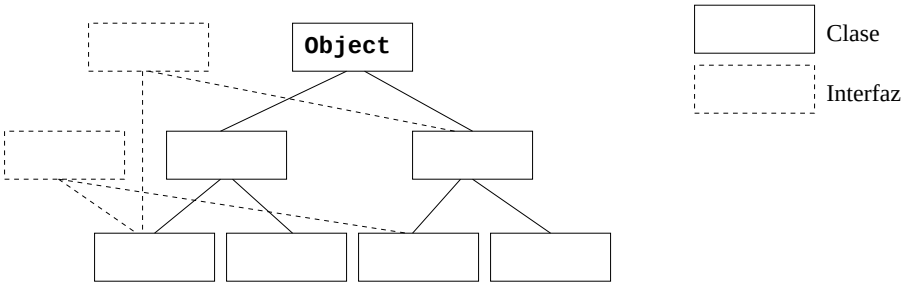

Figura 4.1: Clases e Interfaces

Es entonces posible declarar una variable del tipo de la interfaz y tener tal variable como referencia a cualquier objeto de cualquier clase que implemente la interfaz. El objeto se dice que se conforma a la interfaz, o se conforma al tipo. Más aún, una clase puede implementar varias interfaces de tal modo que los objetos de tal clase pueden ser usados en cualquier momento en que los tipos de sus interfaces sean especificados.

Esto, en efecto, provee de una conexión que permite a una clase especificar si sus objetos pueden ser usados cuando una interfaz coincidente se provea, y puede hacerlo sin necesidad de que la clase se encuentre en un sitio particular dentro de la jerarquía de herencia. Esto tiene importantes consecuencias para el código del programa, ya que puede ser escrito para usar un tipo interfaz, en el conocimiento de que una clase sólo tiene que implementar la interfaz para permitir que sus objetos sean usados. Si el mismo código se escribe usando una clase, entonces sólo los objetos de la clase correspondiente y sus subclases pueden ser usados. Esto forzaría a la clase de cualquier otro objeto que pudiera ser utilizado en el futuro a ser una subclase de la clase específica.

Una cuestión común en el diseño que las interfaces también atienden es la falta de herencia múltiple entre clases. Una clase puede tener sólo una superclase, pero puede requerir heredar de dos o más clases repartidas en la jerarquía de herencia. Esto permitiría a los objetos de la clase substituir cualesquiera objetos de diferentes superclases que les fueran especificados. Añadir interfaces resuelve el problema de substitutibilidad, ya que una clase puede implementar varias interfaces. La herencia de métodos y variables se resolvería mediante cambiar las relaciones de herencia entre clases por asociaciones (lo que no es ideal, pero resulta práctico).

Las bibliotecas de clases de Java hacen un gran uso de las interfaces para especificar un conjunto de métodos que una clase debe proveer para que sus objetos sean utilizados en ciertas situaciones. Un ejemplo es la interfaz `Enumeration`, la cual especifica un tipo y un conjunto de métodos para iterar a través de una estructura de datos. Diferentes estructuras de datos requieren diferentes clases que provean los

iteradores, pero todas las clases pueden implementar la misma interfaz. Los clientes de los iteradores deben entonces sólo tener que usar una interfaz de `Enumeration`, sin la necesidad de saber cómo un iterador particular funciona, o cuál es su clase real.

Un punto final acerca de la conformación de tipos en Java que vale la pena mencionar es que ésta se basa únicamente en la coincidencia sintáctica de nombres de identificadores y nombres de tipos. No especifica el comportamiento de los métodos de la interfaz ni verifica si la implementación de esos métodos hace lo correcto. Eso se deja para que el programador lo verifique.

4.2.1. Declaración de Interfaces

Una declaración de interfaz requiere el uso de la palabra clave `interface`, especificando un tipo de referencia, consistente en un nombre de tipo, un conjunto de declaraciones de métodos abstractos y un conjunto de variables finales estáticas (constantes).

Una interfaz se declara usando la siguiente sintaxis:

modificadordeinterfaz **interface** *identificador***{**
declaración de métodos de la interfaz
declaración de variables de la interfaz
}

El modificador opcional de la interfaz permite que una interfaz sea declarada como pública (`public`). El modificadora `abstract` es permitido, pero es obsoleto y debe evitar usarse.

Las variables de la interfaz pueden ser declaradas usando la declaración estándar de variables estáticas, y deben ser inicializadas. Los únicos modificadores permitidos son `public`, `static` y `private`, pero son redundantes y no deben ser utilizados. Una variable de interfaz puede ser inicializada por una expresión usando otra variable de interfaz, siempre y cuando esta última haya sido declarada textualmente antes de la inicialización.

Los métodos de interfaz son siempre abstractos, y son declarados de la misma forma que los métodos abstractos de las clases: sin cuerpo del método. Los únicos modificadores permitidos son `public` y `abstract`, pero son también redundantes, y no deben ser usados.

Una interfaz puede ser extendida por una o más interfaces (algo parecido a la herencia), usando la palabra clave `extends`.

modificadordeinterfaz **interface** *identificador*
extends *listadenombresdeinterfaces* **{**
declaración de métodos de la interface
declaración de variables de la interfaz
}

La lista de nombres de interfaces puede ser un solo nombre de interfaz o una lista separada por comas de nombres de interfaces. Una interfaz no puede extender una clase.

Una interfaz puede ser anidada dentro de una clase de alto nivel o de otra interfaz, e implícitamente será tratada como estática, aun cuando puede ser declarada como `public`, `private` o `protected` como las declaraciones de clases.

Una interfaz consiste de declaraciones de una secuencia de métodos abstractos y variables estáticas y finales, como por ejemplo:

```
interface X {
    int f(String s);
    boolean g(double d, long l);

    int SIZE = 10;
    String word = "hello";
}
```

Cualquier método declarado será implícitamente considerado como abstracto, por lo que no es necesario explícitamente ser declarado como tal. Aun cuando algunas versiones de Java todavía permiten que aparezca la palabra clave `abstract`, este no será el caso en versiones futuras, y debe ser omitida. Nótese que como los métodos abstractos de las clases, los parámetros deben ser nombrados aun cuando solo su tipo es necesario.

Cualquier variable declarada es implícitamente constante y estática (como las variables de clase). Las variables pueden ser declaradas usando los modificadores `static` y `final`, pero como los métodos, estos modificadores no deben ser utilizados.

Una interfaz puede estar vacía, de tal modo que solo declara el nombre de un tipo:

```
interface Empty {
}
```

Una interfaz se declara en un archivo fuente (o unidad de compilación) como una clase. Puede ser declarada pública, en cuyo caso puede ser accesada por las clases en otros paquetes. Si una interfaz es declarada pública, debe ser la única

interfaz pública (o clase pública) en el mismo archivo fuente. Si no es pública, una interfaz sólo podrá ser usada dentro del mismo paquete donde está declarada.

Si una interfaz es declarada pública, entonces todos los métodos y variables que declara son también públicos. De otra manera, todos ellos tienen un acceso por defecto, y sólo serán accesibles dentro del mismo paquete. Este comportamiento es diferente al de las clases, así que vale la pena mencionarlo.

Una interfaz puede extender o heredar de una o mas interfaces (nótese la diferencia con las clases):

```
interface C extends A, B { // Extiende las interfaces A y B
    // declaraciones
}
```

Todos los nombres declarados en todas las interfaces se vuelven parte de la interfaz que hereda, pero los nombres de las variables pueden ocultarse mediante re-declaraciones de la interfaz que hereda. Las expresiones de campo pueden ser utilizadas para acceder los nombres de las variables ocultas, pero sólamente dentro del mismo paquete, como por ejemplo:

```
A.x;    // acceso a la variable x de la interfaz A
```

Esto no incluye a los métodos, ya que éstos son declarados únicamente para ser redefinidos, y no tienen cuerpo de método.

Una interfaz puede ser anidada dentro de una clase de alto nivel, lo cual permite que el entorno de su nombre sea controlado por su clase anfitriona. Una interfaz anidada es parte de su clase anfitriona, y es siempre tratada como estática, por lo que no es necesario que sea declarada como tal (una interfaz no-estática no tiene caso, ya que las interfaces no son parte de los objetos).

Una interfaz puede también ser anidada dentro de otra interface, como se mostrará en un ejemplo posterior. No es claro si esto es realmente necesario o no, pero puede hacerse.

4.2.2. `implements`

La palabra clave `implements` permite a una clase implementar (o mas bien, conformarse a) una o más interfaces.

El nombre de la clase en una declaración de clase se sigue de la palabra clave `implements` y una lista de uno o mas nombres de interfaces separados por comas:

modificadores **class** *nombredeclase* **implements** *listadenombresdeinterfaces* **{**
declaraciones
}

El resto de la declaración de la clase se hace en forma normal. Si la clase no redefine todos los métodos declarados en la interfaz o interfaces a implementar, entonces la clase debe ser declarada como abstracta.

Una clase puede implementar cualquier número de interfaces, y también extender una clase al mismo tiempo. Toda variable declarada en la interfaz se convierte en una variable estática de la clase. Cualquier método declarado en la interfaz debe ya sea ser redefinido o la clase que implementa debe ser abstracta. Las subclases de la clase deben ser también abstractas o redefinir el resto de los métodos.

Cuando se implementan dos o más interfaces puede haber problemas con las variables con un nombre igual que aparecen en varias interfaces, en cuyo caso cualquier intento de utilizarlas directamente resultará ambiguo. Esto puede resolverse utilizando expresiones de campo:

```
X.y; // Use la variable estatica y de la interfaz X
```

Nótese, sin embargo, que cuando las interfaces son heredadas por otras interfaces hay limitaciones para acceder a las variables de interfaz por parte de clases en un paquete diferente al de la interfaz (véase el ejemplo más abajo).

No hay ambigüedad con los métodos, ya que las interfaces solo contienen declaraciones de métodos abstractos. Sin embargo, si dos interfaces declaran el mismo método, con la misma lista de argumentos y el mismo tipo de retorno, se espera que el método sea redefinido de forma diferente. Por tanto, el método en la clase que implementa a las interfaces necesitará implementar ambas redefiniciones en el mismo método.

Un método declarado en una interfaz pública debe ser público en la clase que lo implementa, ya que la accesibilidad no puede decrementarse.

Si un conjunto complicado de declaraciones de interfaces heredadas se implementa por una clase, es posible implementar la misma interfaz por dos o más diferentes rutas, como se muestra en la figura 4.2. Esto es permitido, y no introduce ningún problema. Sólo una copia existe de cualquier variable de interfaz que es heredada por cualquiera de las dos o más rutas.

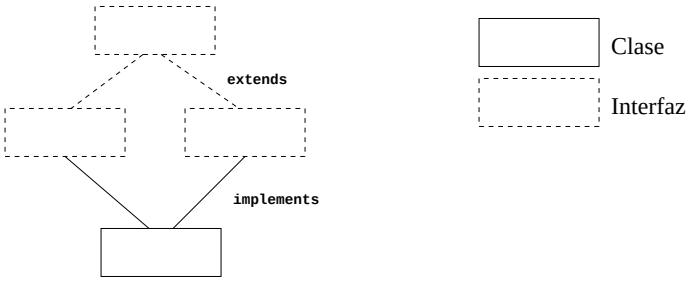

Figura 4.2: Interfaz implementada a través de dos rutas

Los siguientes archivos fuente usan las interfaces declaradas en el paquete `ITest`. El primero declara una clase en el mismo paquete, el cual tiene acceso a las variables de interfaz ocultas.

```
package ITest;

// Esta clase se encuentra en el mismo paquete como
// todas las declaraciones de interfaces

public class Thing implements X {
    public void f(int y) {
        System.out.println("Thing:f " + y);

        // La variable name esta declarada en una interfaz en el
        // mismo paquete, por lo que es accesible pero ambigua a menos
        // que se use en una expresion de campo nombrando a la
        // interfaz que declara a la variable

        System.out.println(X.name);
        System.out.println(Y.name);

        // El compilador puede generar mensajes de advertencia acerca
        // de esto dos ultimos enunciados

        // La siguiente linea es ambigua
        // System.out.println(name);  // ERROR
    }

    // Debe de redefinirse g y h

    public void g() {
        System.out.println("Thing:g ");
```

```
    }

    public int h(int x) {
        System.out.println("Thing:h " + x);
        return x * 4;
    }
}
```

El siguiente archivo fuente es un cliente del paquete `ITest`, y también contiene al método `main`.

```
import ITest.*;

class A implements X {

    // Puede tenerse una interfaz anidada
    private interface Local { }

    // Implementacion de la variable de interface
    public void f(int x) {
        System.out.println("A:f " + x);

        // Uso de las variables de interface
        System.out.println(f);
        System.out.println(X.f); // Tambien se permite
        System.out.println(m);
        System.out.println(n);
        System.out.println(p);

        // name es declarada en una interfaz heredada no-publica, y
        // no puede ser accesada fuera del mismo paquete

        // System.out.println(name);  // ERROR
        // System.out.println(X.name);  // ERROR

        // Las variables de interfaz son finales, asi que no puede
        // hacerse la siguiente asignacion

        // SIZE = 5; // ERROR


        // Esta clase miembro implementa una interfaz anidada
        // declarada en la interfaz Example. Porque se querria hacer
        // lo siguiente, es otra cosa
```

```
        class NestedClass implements Example.Nested {
            public void z() {
                System.out.println("NestedClass:z");
            }
        }

        NestedClass nest = new NestedClass();
        nest.z();
    }

    // Debe redefinirse g y h

    public void g() {
        System.out.println("A:g ");
    }

    public int h(int x) {
        System.out.println("A:h " + x);
        return x * 2;
    }
}

class InterfaceTest {
    public static void main (String[] args) {
        A a = new A();
        a.f(10);

        Thing t = new Thing();
        t.f(20);

        // No es posible accesar una interfaz privada anidada
        // A.Local l;  // ERROR
    }
}
```

4.3. Excepciones

Una excepción ocurre cuando algo falla durante la ejecución de un programa, es decir, un evento excepcional ocurre que no permite al programa proceder normalmente. Sin un manejo explícito, un programa en el cual ocurre una excepción termina abruptamente, perdiendo el estado de las variables, incluyendo todos los datos que no han sido guardados en un archivo. Esto es parecido a la situación de un documento en el que se ha trabajado por horas y en el último minuto desaparece sin rastro.

El mecanismo de manejo de excepciones de Java permite que las excepciones sean atrapadas (*caught*) y atendidas, idealmente mediante corregir el problema que causó la excepción y permitiendo que el programa continúe sin pérdida de datos.

4.3.1. Clases de Excepción

Las clases de excepción permiten que las excepciones sean representadas como objetos. Un objeto excepción almacena la información acerca de la causa de la excepción y puede pasarse como un valor a un manejador de excepciones que se implementa en un bloque `catch`.

El conjunto estándar completo de clases de excepción definida por las bibliotecas de Java se describe en la documentación de JDK API (véase la página índice para el paquete `java.lang`), y no se presenta aquí.

Todas las clases de excepción deben ser directa o indirectamente subclases de la clase `Throwable`, la cual define una interfaz con un método por omisión de una clase de excepción. `Throwable` también define una variable privada `String` que puede ser utilizada para almacenar un mensaje describiendo la causa de una excepción. Esta variable puede ser inicializada mediante un constructor, y accesarse vía un método público.

El método público de la interfaz de `Throwable` es como sigue:

```
// Constructor por omision
Throwable();

// Constructor con un argumento String describiendo el error
// La subclase definida por el programador debe proveer siempre
// una version de este constructor
Throwable(String);

// Metodo que retorna el String describiendo el error
String getMessage();
```

```
// Metodo que retorna una version de String en el lenguaje local
// Este metodo debe ser redefinido por subclases, ya que el
// comportamiento por defecto es retornar el String de error
String getLocalizedMessage();

// Version redefinida del metodo toString() de la clase Object,
// que retorna informacion sobre la excepcion incluyendo el
// String de error. Puede ser redefinido, pero la version
// incluida en Throwable es bastante util
String toString();

// Los siguientes metodos despliegan un rastro de stack (las
// llamadas a  metodos en progreso cuando ocurre la excepcion
void printStackTrace(); // Envia al stream estandar de error
void printStackTrace(PrintStream); // Envia de PrintStream
void printStackTrace(PrintWriter); // Envia a un print writer

// Ajusta el rastro en el stack si la excepcion vuelve a lanzarse
Throwable fillInStackTrace();
```

Las subclases pueden añadir tantas variables y métodos como sea necesario para almacenar más información acerca de la excepción que se representa por sus instancias.

Una clase de excepción, a pesar de su nombre, es una clase ordinaria definida en forma normal. Todas las clases de excepción son, ya sea directa o indirectamente, subclases de la clase `Throwable` de la biblioteca de clases de Java, que se define en el paquete `java.lang`. El mismo paquete define toda una familia de subclases de `Throwable`, que son usadas por los paquetes de la biblioteca. El programador puede ya sea usar estas clases directamente para representar excepciones o, más comúnmente, crear subclases para representar y definir sus excepciones.

La figura 4.3 muestra la jerarquía básica de clases de excepción. La biblioteca de clases de excepción se dividen en dos categorías basadas en las bibliotecas de clases `Error`, `Exception` y `RuntimeException`:

- `Error`, `RuntimeException` y sus subclases representan serios errores causados en general por la Máquina Virtual de Java (*Java Virtual Machine*, o JVM), la clase cargador (*loader*) o el ambiente de ejecución (*runtime environment*). Estas excepciones son en su mayoría no-recobrables, y causan que el programa termine o falle de una forma irrecuperable. Las excepciones de error indican una falla en el sistema fuera del control del programa. Un programa en Java no necesita explícitamente tratar con estas excepciones, dado que la terminación abrupta es aceptable.

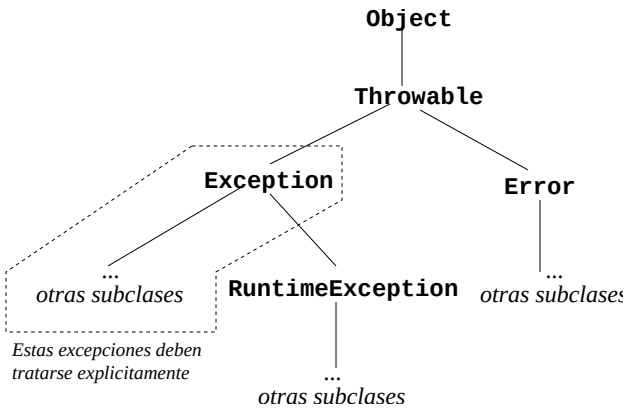

Figura 4.3: Jerarquía de Clases de `Throwable`

- Las subclases de `Exception`, excepto `RuntimeException` y sus subclases, representan errores causados por omisiones y fallas lógicas iniciadas por el programador. Una excepción representada por una de estas clases debe ser explícitamente tratada por el programa en Java. Las excepciones definidas por el programador son casi siempre una subclase de `Exception`.

En la mayoría de los casos, una clase de excepción definida por el programador es directamente una subclase de `Exception`. Las otras clases de excepción no son, en general, subclases (aún cuando podrían serlo). Una clase de excepción definida por el programador generalmente define un constructor por omisión incluyendo la llamada a `super` con un mensaje apropiado tipo `String`. Un constructor que toma un argumento `String` y que llama a `super` para pasar el argumento a `Throwable` también aparece frecuentemente. En muchos casos, la subclase no necesita añadir ningún nuevo método o variable, pero está en libertad de hacerlo si puede usarlos para proveer de información extra acerca de la causa de la excepción.

El siguiente ejemplo muestra una clase de excepción básica definida por el programador. Frecuentemente, las clases de excepción definidas por el programador no son más complicadas que:

```
class UserException extends Exception {
    public UserException() {
        super("Un mensaje adecuado");
    }

    public UserException(String msg) {
        super(msg);
    }
}
```

4.3.2. `try`, `catch` y `finally`

Las palabras clave `try` y `catch` proveen de un núcleo sintáctico para manejar excepciones. La palabra clave `finally` provee de un mecanismo para garantizar que un bloque de código se ejecuta sin importar que otra cosa suceda.

Un bloque `try` consiste de lo siguiente:

try {
secuencia de enunciados
}

Este puede ser seguido por cero o más bloques `catch`:

catch (*parámetro*) **{**
secuencia de enunciados
}

Siguiendo a la palabra `catch` hay una declaración de parámetro que especifica el tipo de objeto excepción que cada bloque `catch` atrapa.

Un bloque opcional `finally` puede seguir a cualquier bloque `catch`:

finally {
secuencia de enunciados
}

El bloque `try` debe ser seguido de al menos un bloque `catch` o un bloque `finally`. Un bloque básico `try-catch` puede conjuntarse como sigue:

```
try {
    f(); // llamada a un metodo que puede arrojar una excepcion
}

catch (Exception e) {
    System.out.println("Llamada a f() ha fallado");
    // imprime el mensaje de la excepcion
    System.out.println(e.getMessage());
}
```

El bloque `try` intenta ejecutar una llamada al método `f()`. Si una excepción ocurre mientras se ejecuta `f()`, entonces se crea y arroja (*throw*) un objeto excepción. El bloque `catch` que sigue tiene entonces la oportunidad de atrapar (*catch*) la excepción arrojada, si el tipo del objeto excepción es compatible por asignación con el tipo especificado como parámetro del bloque `catch`. Si la excepción es atrapada, los enunciados en el bloque `catch` respectivo se ejecutan, y la ejecución prosigue con el enunciado siguiente al final del bloque `try-catch`. Si no ocurre ninguna excepción, se salta el bloque `catch` y procede la ejecución con el siguiente enunciado

al final del bloque `try-catch`. Por lo tanto, lo que sea que ocurra, el programa tiene una oportunidad de seguir ejecutándose normalmente.

Hacer corresponder un objeto excepción con el parámetro de un bloque `catch` es similar a una llamada a método. Si el tipo del objeto es compatible por asignación, el parámetro de la excepción se inicializa para mantener una referencia al objeto excepción, permitiendo que el objeto sea accesado dentro del entorno del bloque `catch`. En el ejemplo anterior el parámetro es de tipo `Exception` o cualquiera de sus subclases.

Un bloque `try` puede ser seguido de varios bloques `catch`, cada uno tratando una excepción diferente:

```
try {
    f(); // llamada al metodo que puede arrojar una excepcion
}

catch(UserException e) {
    System.out.println("Llamada al metodo f ha fallado");
    System.out.println(e.getMessage());
}

catch(Exception e) {
    System.out.println("Llamada al metodo f ha fallado");
    System.out.println(e.getMessage());
}
```

Cuando una excepción es arrojada, el tipo del objeto se revisa contra el parámetro de cada bloque `catch` en el orden en que están escritos. El primer bloque que coincida se selecciona y ejecuta, y el resto son ignorados. En el ejemplo, esto significa que el objeto excepción se compara primero para ver si es instancia de `UserException` o alguna de sus subclases. Si es así, entonces el bloque `catch` correspondiente es ejecutado, o si no, se verifica la coincidencia con el siguiente bloque `catch`.

Nótese que el segundo bloque es más general, y atrapará más tipos de excepciones que el primero. Si el orden de los bloque `catch` se invierte, se da que:

```
try {
    f(); // llamada al metodo que puede arrojar una excepcion
}

catch(Exception e) {
    System.out.println("Llamada al metodo f ha fallado");
    System.out.println(e.getMessage());
}
```

```
catch(UserException e) {
    System.out.println("Llamada al metodo f ha fallado");
    System.out.println(e.getMessage());
}
```

lo que provoca que el segundo bloque no pueda ser alcanzado nunca, ya que el primero siempre será exitoso (`UserException` es compatible por asignación con `Exception`). El compilador de Java detecta esta situación y reporta un error.

Un bloque `finally` puede seguir al bloque `try-catch`:

```
try {
    f(); // llamada al metodo que puede arrojar una excepcion
}

catch(Exception e) {
    System.out.println("Llamada al metodo f ha fallado");
    System.out.println(e.getMessage());
}

finally {
    g(); // Llama siempre a g, no importa lo que pase
}
```

Un bloque `finally` siempre se ejecuta, sin importar qué pase con el bloque `try`, aún si se arroja otra excepción o se ejecuta un enunciado `return`. Esto provee de un lugar para poner enunciados cuya ejecución está garantizada.

Normalmente, el programador desea utilizar un bloque `catch` para recobrar el estado del programa, y dejarlo proseguir sin terminar. El bloque `catch` puede usar cualquier variable y llamar a cualquier método en el entorno, y puede desplegar un mensaje informando al usuario que ha habido un problema. Si otra excepción ocurre dentro de un bloque `catch`, no será atrapada en el mismo bloque `try-catch`, y tendería a propagarse.

Cuando un objeto excepción es atrapado por un bloque `catch`, se tiene la opción de llamar a uno de los métodos de trazado de la pila del objeto, heredado de `Throwable`, así como de usar el método `getMessage()` para obtener un mensaje. Esto muestra una cadena de llamadas a métodos activos que se han hecho para alcanzar el punto actual en el programa. Esta información puede ser usada para revisar el programa.

Una traza de la pila se ve como sigue:

```
java.lang.NumberFormatException: abc
at java.lang.Integer.parseInt(Integer.java:229)
at java.lang.Integer.parseInt(Integer.java:276)
```

```
    at Exception1.convert(Exception1.java:14)
    at Exception1.main(Exception1.java:29)
```

La primera línea es el nombre de la clase de excepción, seguido por el mensaje de tipo `String`. Las siguientes líneas muestran cada llamada activa a método hasta el método `main` (o un método `Thread run`, si la excepción fue arrojada por una hebra de control separada).

El siguiente ejemplo muestra el típico bloque `try-catch` cuando se convierte de un `String` representando un entero a su equivalente de tipo `int`. Esto llama a un método de la biblioteca que arroja una excepción si la conversión no puede hacerse.

```
class Exception1{
    public int convert(String s) {
    int result = 0;
    try {
        // Se usa un metodo de la biblioteca de clases Integer
        // para convertir un String representando un entero a
        // un valor int. parseInt arroja una exception si el
        // valor String no es convertible a un int.
        result = Integer.parseInt(s);
    }

    catch(NumberFormatException e) {
        System.out.println("Falla en conversion de String: + e");
        e.printStackTrace();
        return result;
    }

    public static void main (String[] args) {
        Exception1 e1 = new Exception1();
        e1.convert("123");
        e1.convert("abc");
    }
}
```

El segundo ejemplo muestra el uso de múltiples bloques `catch` y un bloque `finally`.

```
class MyException extends Exception {
    MyException(String s) {
        super(s);
    }
}

class Exception2 {
```

```
// Este metodo deliberadamente arroja una excepcion seleccionada por el
// argumento. Notese que NullPointerException e InterruptedException son
// clases de excepciones de la biblioteca. InterruptedException es una
// subclase directa de Exception, mientras que NullPointerException es una
// subclase de RuntimeException, la cual es subclase de Exception.
public void g(int x) throws Exception {
    switch(x) {
        case1 :
            throw new MyException("Metodo g ha fallado");
        case2 :
            throw new NullPointerException("Metodo g ha fallado");
        case3 :
            throw new InterruptedException("Metodo g ha fallado");
        default :
            throw new Exception("Metodo g ha fallado");
    }
}

public void f(int x) {
    try {
        if (x < 5) {
            g(x);
        }
        System.out.println("Pasa llamada a g sin excepcion");
        return;
    }

    catch (MyException e) {
        System.out.println("MyException atrapado en metodo f: ");
        System.out.println(e.getMesssage());
    }

    catch (Exception e) {
        System.out.println("Exception atrapado en metodo f: ");
        System.out.println(e.getMessage());
    }

    finally {
        System.out.println("Se ha ejecutado f");
    }
}

public static void main(String[] args) {
    Exception2 e2 = new Exception2();
    e2.f(1);
```

```
        e2.f(2);
        e2.f(3);
        e2.f(4);
        e2.f(5);
    }
}
```

4.3.3. Propagación de Excepciones

Si una excepción no es atrapada dentro de un método, se propaga ascendentemente sobre la cadena de llamadas a métodos activos hasta hallar un bloque `catch` que lo maneje. Las excepciones no pueden ser olvidadas.

Las excepciones de las clases `Error`, `RuntimeException`, o de sus subclases no tienen por que ser explícitamente manejadas y, si se arrojan, se propagan a través de la cadena de llamadas a métodos activos. No son necesarios los bloques `try-catch` o declaraciones `throws`, aún cuando un método puede atrapar la excepcion si se requiere.

Todas las demás excepciones deben siempre ser explícitamente tratadas usando bloques `try-catch` y declaraciones `throws`. Cuando se usan estas excepciones, un método puede retornar sin que la excepción se atrape si el método mismo tiene una declaración `throws` y aún si no tiene ningún bloque `catch`, o que ninguno de los bloques `catch` siguientes en el bloque `try-catch` pueda atrapar la excepción. La excepción no atrapada se propaga de regreso al método original. Si hay un bloque `finally` presente, éste será ejecutado antes de que la excepción se propague.

Cuando una excepción se propaga, cualquier bloque `catch` correspondiente a un bloque `try` en el método que lo invoca tiene una oportunidad de atrapar la excepción. Si uno de ellos efectivamente la atrapa, entonces el bloque `catch` correspondiente se ejecuta, permitiendo que la ejecución continúe con el enunciado siguiente al bloque `try-catch`, o con el enunciado siguiente del bloque `finally` que se adecúe. De otra manera, o si no hay bloques `catch`, la excepción se propaga de regreso hacia el método invocado anteriormente. Como parte de la propagación, un método sale del entorno, lo que provoca que todas las variables locales sean destruidas.

Cuando cualquier excepción (de cualquier categoría) alcanza la cima de la cadena de métodos activos (es decir, el método `main` o el método `Thread run`), un manejador por omisión lo atrapa y finaliza la hebra de control actual. Si tal hebra de control es la única, o es la hebra que ejecuta el `main`, el programa termina. Si el `main` y otras hebras de control se encuentran aún activas entonces el programa puede continuar su ejecución, pero se encuentra bajo un serio riesgo.

4.3.4. Declaración `throws`

Se requiere hacer una declaración `throws` para cualquier método que arroje o propague una excepción que deba ser tratada explícitamente (por ejemplo, una subclase de `Exception` excepto `RuntimeException` o alguna de sus subclases). La declaración enuncia el tipo o tipos de excepción que pueden ser arrojadas por el método.

Cualquier método estático, método de instancia, método abstracto o declaración de constructor puede incluir una declaración `throws` en seguida de la lista de parámetros y antes del cuerpo del método:

modificadores Nombredetipo NombredelMetodo(*lista de parametros*)
throws *lista de nombres de tipos*{
secuencia de enunciados
}

La lista de nombres de tipos especifica uno o más nombres de clases o interfaces separados por comas. El tipo de toda excepción que puede ser arrojada o propagada por el método deben ser incluidos. Las excepciones que se atrapan en el cuerpo del método no necesitan ser listadas.

Cuando se trabaja con excepciones que deben ser tratadas explícitamente, un método debe ya sea atrapar la excepción o propagarla. La declaración `throws` establece cuáles tipos de excepciones serán propagadas.

Si un método invoca a otro método con una declaración `throws`, se da alguno de los siguientes casos:

- El método es invocado desde un bloque `try` que tiene bloques `catch` capaces de atrapar sólo algunas de las excepciones que pueden ser arrojadas. El método que invoca no requiere ninguna declaración `throws` respecto a la llamada (pero puede requerir una declaración de otras invocaciones).
- El método se invoca de un bloque `try` con bloques `catch` que pueden atrapar sólo algunas de las excepciones que pueden ser arrojadas. Los demás tipos de excepciones pueden ser propagados y deben aparecer en la declaración `throws` del método que invoca.
- El método es invocado fuera de un bloque `try` y todos los tipos de excepciones deben aparecer en el método que invoca en una declaración `throws`.

Las excepciones de las clases `Error`, `RuntimeException` y sus subclases no necesitan ser declarados en `throws`. Cualquier tipo especificado en una declaración `throws` debe ser de tipo `Throwable` o de sus subclases.

Los constructores siguen las reglas anteriores pero las expresiones `new`, que crean objetos de la clase del constructor, no tienen que aparecer en el bloque `try`. Sin

embargo, si las excepciones arrojadas por un constructor no son atrapadas, pueden provocar la terminación de la hebra actual, o posiblemente del programa.

El siguiente programa incluye declaraciones `throws` y varios ejemplos de propagación de excpeciones:

```
// Se declaran dos subclases de la clase Exception para proveer
// de clases definidas por el usuario. Estas excepciones deben
// tratarse explicitamente

class MyException extends Exception {
    MyException(String s) {
        super(s);
    }
}

class YourException extends Exception {
    YourException(String s) {
        super(s);
    }
}

class Exception3 {
    // Un constructor que arroja una excepcion
    public Exception3(int x) throws MyException {
        throw new MyException("Falla de constructor");
    }

    // Un metodo que puede arrojar dos tipos diferentes
    // de excepciones, requiriendo que la declaracion throws
    // liste ambas excepciones

    public void h(int x) throws YourException, MyException {
        if (x == 1) {
            throw new YourException("Arrojada en h");
        }
        else {
            throw new MyException("Arrojada en h");
        }
    }
```

```
// El siguiente metodo llama a h y puede tratar cualquier
// excepcion de tipo MyException que h pueda arrojar.
// Sin embargo, no puede atrapar excepciones del tipo
//  YourException, propagandose en forma ascendente,
//  y requiriendo una declaracion

public void g(int x) throws YourException {
    try {
        h(x);
    }

    catch (MyException e) {
        System.out.println("Excepcion atrapada en g");
        System.out.println(e.getMessage());
    }

    finally {
        System.out.println("finally en g");
    }
}

// Este metodo trata cualquier excepcion arrojada por la
// llamada a g y necesita una declaracion throws

public void f(int x) {
    try {
        g(x);
    }

    catch(Exception e) {
        System.out.println("Excepcion atrapada en f");
        System.out.println(e.getMessage());
    }
}

// Los metodos r, s y t siguientes muestran que una excepcion o tipo
// Error no necesitan ser tratados exclicitamente, asi que no se
// provee ninguna declaracion throws

public void r() {
    throw new Error("Error deliberado");
}
public void s() {
    r();
}
```

```
    // Una excepcion de tipo Error puede ser aun atrapada usando un
    // bloque try-catch

    public void t () {
        try {
            s();
        }

        catch (Error e) {
            System.out.println("Excepcion atrapada en t");
            System.out.println(e.getMessage());
        }
    }

    public static void main(String [] args) throws Exception {
        // Prueba del tratamiento de excepciones

        Exception3 e3 = new Exception3();
        e3.f(1);
        e3.f(2);
        e3.t();

        // El constructor tomando un int como argumento genera una
        // excepcion que puede ser atrapada

        try {
            Exception3 e3b = new Exception3(1);
        }

        catch (Exception e){
            System.out,println(e.getMessage());
        }

        // Sin embargo, la expresion new no necesita aparecer en un
        // bloque try. Las excepciones arrojadas por el constructor
        // pueden causar que el programa termine prematuramente

        Exception3 e3c = new Exception3();
    }
}
```

4.3.5. El enunciado `throw`

Un enunciado `throw` permite que una excepción sea arrojada. La palabra clave `throw` es seguida de una expresión del tipo `Throwable`, o uno de sus subtipos.

throw *ExpresiónThrowable*;

El enunciado `throw` arroja una excpeción que sigue las reglas presentadas en las secciones anteriores. Normalmente, se espera que la excepción se propague en forma ascedente al método invocado, de tal modo que el método que contiene el `throw` tiene una declaración `throws`. Las excepciones pueden ser arrojadas dentro de un bloque `try` y atrapadas dentro del mismo método, si es necesario.

Una excepción se crea usando una expresión `new` como parte del enunciado `throw`, como por ejemplo:

```
throw new Exception("Algo ha fallado");
```

Un enunciado `throw` puede también ser usado para re-arrojar una excepción que ya había sido atrapada. Esto permite que más de un método dentro de la cadena de métodos activos pueda tratar con la excepción, tal vez cada uno haciendo su parte para restaurar el estado del programa para que pueda continuar en forma segura después de que la excepción ha sido tratada.

Una excepción puede ser re-arrojada directamente desde un bloque `catch`:

```
try {
    f();
}

catch (Exception e) {
    throw e; // excepcion re-arrojada
}
```

El principal uso de `throw` es dentro de los métodos en bibliotecas y para dar soporte a las clases al notificar a sus clientes que la llamada a un método ha fallado, típicamente cuando un objeto de una clase en la biblioteca ha sido usado erróneamente por el cliente (por ejemplo, si el cliente trata de sacar un valor de una pila vacía). La presencia del enunciado `throw` en el método de la biblioteca forza al programador a considerar la posibilidad de llamar al método que potencialmente puede fallar mediante un bloque `try-catch`.

El siguiente ejemplo (y el de la sección anterior) muestra el uso de los enunciados `throw`.

```
class Exception4 {
    // Arroja una excepcion y la atrapa en el mismo metodo.
    // Esto puede no ser muy util
    public void f() {
        try {
            throw new Exception("Arrojada en f");
        }

        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    // El siguiente metodo arroja una excepcion, por lo que
    // debe incluir una declaracion throws

    public void g() throws Exception {
        throw new Exception("Arrojada en g");
    }

    public void h() throws Exception {
        try {
            g();
        }

        catch (Exception e) {
            // Imprime una traza para ver la cadena de metodos
            // activos
            e.printStackTrace();

            // Reinicia el trazo, para que comience desde aqui
            e.fillInStackTrace();

            // Re-arroja la excepcion para permitir que otro metodo
            // en la cadena de metodos activos la trate
            throw e;
        }
    }
```

```
    // Atrapa la excepcion para prevenir mayor propagacion
    public void k() {
        try {
            h();
        }

        catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Algunos metodos extra para hacer mas profunda la traza

    public void x() {
        k();
    }

    public void y() {
        x();
    }

    public static void main(String[] args) {
        Exception4 e4 = new Exception4();
        e4.f();
        e4.y();
    }
}
```

La salida de este programa (considerando que la salida de error estándar es visible para el método `printStackTrace`) es:

```
Arrojada en f
java.lang.Exception: Arrojada en g
at Exception4.g(Exception4 java 23)
at Exception4.h(Exception4 java 10)
at Exception4.k(Exception4 java 47)
at Exception4.x(Exception4 java 58)
at Exception4.y(Exception4 java 63)
at Exception4.main(Exception4 java 70)
java.lang.Exception: Arrojada en g
at Exception4.h(Exception4 java 38)
at Exception4.k(Exception4 java 47)
at Exception4.x(Exception4 java 58)
at Exception4.y(Exception4 java 63)
at Exception4.main(Exception4 java 70)
```

El método `printStackTrace` es declarado dos veces, así que hay dos trazas de la pila comenzando con la línea `java.lang.Exception: Arrojada en g`. Cada traza muestra la cadena de métodos activos hasta el `main`. Nótese que la primera traza comienza a partir del método `g`, mientras que la segunda comienza desde el método `h`, como resultado de la llamada al método `fillInStackTrace`, el cual reinicia la traza en el método actual.

4.4. Hebras de Control (*Threads*)

Una hebra o hilo (*thread*) es un flujo de control en un programa, generalmente caracterizado como una secuencia de invocaciones a métodos que se hacen cuando la hebra se ejecuta. Todos los ejemplos anteriores han supuesto hasta ahora una sola hebra que comienza su ejecución con el método `static main`. Sin embargo, un programa puede usar múltiples hebras de control mediante tener una sola hebra que genere nuevas hebras. Esto significa que en cualquier momento puede haber un número de secuencias de invocaciones a métodos en ejecución.

La ejecución de hebras es entremezclada (*interleaved*), de tal modo que puede parecer que varias hebras se ejecutan simultáneamente, lo que permite que un programa realice varias cosas a la vez. Esto no es propiamente procesamiento paralelo, ya que sólo una hebra se encuentra activa en un momento dado y la Máquina Virtual de Java únicamente tiene acceso a un solo procesador. Sin embargo, la hebra activa puede ser intercambiada por otra, de tal modo que, de un momento a otro, diferentes hebras se encuentran activas creando la ilusión de "procesamiento paralelo". A esto se le conoce frecuentemente como procesamiento concurrente (*concurrent processing*) o multitarea (*multitasking*). En un procesamiento paralelo real, dos o más hebras se encuentran ejecutándose simultáneamente, lo que requiere de dos o más procesadores físicos en una máquina.

Para ayudar a administrar múltiples hebras y decidir cuál debe estar activa, las hebras tienen una prioridad. Una hebra con mayor prioridad tiene precedencia sobre otra hebra de menor prioridad, y por lo tanto, tiene la oportunidad de ejecutarse primero. Cuando una hebra se detiene o se suspende (a lo que a veces se le llama "dormirse"), otra hebra tiene la oportunidad de ejecutarse. La selección y administración de hebras se maneja por un proceso planificador (*scheduler*), el cual mantiene una cola de hebras, cada una con su prioridad, y determina cuál hebra debe ejecutarse en un momento y cuál debe ejecutarse después, partiendo de un conjunto de reglas. La regla principal es, como se menciona anteriormente, escoger la hebra con mayor prioridad. Pero es posible que haya más de una hebra con esta misma prioridad, en cuyo caso se escoge aquélla que se encuentre al frente de la cola.

Algunas versiones de la Máquina Virtual de Java entremezclan automáticamente hebras con la misma prioridad (usando mecanismos como el *round robin* global) de tal modo que cada hebra recibe una parte casi igual en la división del tiempo de ejecución. Sin embargo, otras versiones no hacen esto, lo que significa que hebras con la misma prioridad pueden no tener la misma oportunidad de ejecutarse. Esto, desafortunadamente, implica que un programa que usa ciertos aspectos de hebras puede trabajar propiamente en un sistema pero no en otro. Conforme las nuevas versiones de la Máquina Virtual de Java mejoren, esto pudiera remediarse en el futuro.

Una consecuencia del mecanismo de priorización y planificación es que las hebras con baja prioridad podrían no tener la oportunidad de ejecutarse nunca, ya que siempre es posible que haya una hebra con mayor prioridad esperando en la cola. El programador debe estar consciente de esto, y tomar las precauciones para asegurar que todas las hebras tengan la oportunidad de ejecutarse tan frecuentemente como sea posible para realizar su labor.

Otra característica de la priorización de hebras es que una hebra en ejecución puede ser interrumpida (*pre-empted*) por otra hebra de mayor prioridad que comienza o regresa de su estado suspendido ("despierta") e intenta ejecutarse. Esto provee de un mecanismo importante para la respuesta rápida a eventos que causen el despertar de la hebra con mayor prioridad, pero también requiere que el programador planee en forma cuidadosa qué hebra tiene qué prioridad.

Todos los programas en Java, ya sean aplicaciones o *applets*, son en realidad multihebra, aun cuando el código que el programador produce tenga una sola hebra. Esto se debe a que la Máquina Virtual de Java usa hebras para proveer y admnistrar varias actividades internas de mantenimiento, como por ejemplo, la recolección de basura (*garbage collection*). La hebra que ejecuta el método `main` de un programa es tan solo otra hebra generada por la Máquina Virtual de Java. El conjunto completo de las hebras que un programa usa puede ser visto utilizando el depurador (*debugger*) de JDK, `jdb`, o su equivalente en ambientes comerciales.

Un programa se mantiene en ejecución mientras que su hebra original o cualquiera de las hebras que genere estén en existencia, aun cuando ninguna de ellas pueda ser planificada para ejecutarse porque todas se encuentren bloqueadas o suspendidas por alguna razón. Cuando todas las hebras en un programa han terminado, también lo hace el programa; el estado de las hebras "internas" de la Máquina Virtual de Java para propósitos de mantenimiento del sistema de ejecución no se toman en cuenta para esto.

Las hebras son útiles para muchos propósitos incluyendo el soporte de técnicas de programación concurrente. Un buen número de bibliotecas de clases de Java usan hebras de control, típicamente para permitir actividades lentas, poco frecuentes o inconexas, que se realizan sin atravesarse en el flujo principal del programa. Por ejemplo, la biblioteca `AWT` usa hebras para permitir a la interfaz gráfica (*GUI*) de un programa el ser manipulada o actualizada mientras que el programa continúa con su labor principal.

Las hebras también proveen de soluciones a problemas que de otra manera sería muy difícil o imposible de programar. Esto incluye el manejo de eventos múltiples asíncronos, como leer datos de diversas fuentes en una red, sin que el programa principal se detenga esperando a que lleguen los datos.

4.4.1. La clase `Thread`

Las hebras de control se representan en Java como instancias de la clase `Thread`, que se encuentra en el paquete de bibliotecas `java.lang`. La clase `Thread` provee de una variedad de métodos para crear y manipular hebras. La información de la clase `Thread` puede encontrarse en la sección `java.lang` de la documentación de JDK.

Todas las hebras tienen un nombre, el cual si no es explícitamente dado, se genera automáticamente por omisión. Cada hebra es miembro de un grupo de hebras (*thread group*), que es una instancia de la clase `ThreadGroup`, la cual puede también ser dada explícitamente si no se desea el grupo por omisión.

Las hebras pueden ser creadas usando una instancia de una clase que implemente la interfaz `Runnable`, como se describe posteriormente. Esta interfaz declara un método abstracto:

```
public interface Runnable {
    public abstract void run();
}
```

La clase `Thread` puede ser usada en dos formas:

- Declarando una subclase de `Thread` que redefina el método `run`. Cuando una instancia de la subclase se crea e invoca al método `start`, una nueva hebra comienza su ejecución llamando al método `run`.

```
class MyThread extends Thread {
    ...
    public void run () {
       ... // Codigo que implementa lo que haga la hebra
    }
    ...
}

...

MyThread t = new MyThread();  // Crea una nueva hebra con
                              //  prioridad por omision
t.start();  // Inicia la nueva hebra invocando al metodo run
```

- Declarando una clase que implemente la interfaz `Runnable`, de tal modo que redefina al método `run` declarado en la interfaz. La clase no tiene que ser una subclase de `Thread`. Una nueva hebra se inicia mediante primero crear un objeto `Thread`, pasando como parámetro al constructor de `Thread` una instancia de la clase que implementa a `Runnable`, y se activa mediante invocar el método `start` del objeto tipo `Thread`. Esto resulta en la creación de la hebra, seguido de la invocación al método `run` de la clase que implementa `Runnable`. La hebra termina cuando `run` termina.

```
class MyThreadedClass implements Runnable{
    ...
    public void run () {
       ... // Codigo que implementa lo que haga la hebra
    }
    ...
}

...

Thread t = new Thread(new MyThreadedClass());
t.start();
```

En ambos casos, una nueva hebra realmente comienza a ejecutarse cuando se le incluye en la planificación por el planificador de hebras. Esto significa que una nueva hebra no necesariamente comienza a ejecutarse inmediatamente después de su creación, y puede no ejecutar nada si un diseño pobre genera hebras de control de alta prioridad que siempre se apropian del tiempo de procesador.

No hay diferencia en el comportamiento durante ejecución (*runtime behavior*) de las hebras, ya sea que fueran creadas por subclases de `Thread` o por instancias de `Thread` que mantienen un objeto implementando la interfaz `Runnable`. Esta distinción es, sin embargo, importante durante el diseño. Una subclase de `Thread` no puede ser subclase de ninguna otra clase, ya que Java solo permite la herencia de una sola superclase. Esto significa que las subclases de `Thread` sólo pueden ser usadas donde no se requiere la herencia de otra clase. De otra forma, una clase tiene que implementar `Runnable`, lo que le permite heredar lo que sea necesario.

El siguiente ejemplo ilustra una simple subclase de `Thread`, mostrando un método `run` redefinido. Una sola hebra es creada e iniciada, que despliega una secuencia de 25 mensajes. El flujo de control principal (que se ejecuta en la hebra por omisión del programa) también despliega una secuencia de 25 mensajes. Ambas hebras despliegan sus prioridades, que deben ser iguales y dispuestas a 5. Como ambas hebras se ejecutan con la misma prioridad, la salida del programa varía dependiendo de cuál Máquina Virtual de Java se utilice para ejecutarlo. Una máquina que soporte la multitarea prevenida (*pre-emptive multitasking*) de hebras con misma prioridad

despliega ambos conjuntos de mensajes entremezclados. De otra manera, un conjunto de mensajes aparecerá antes que el otro.

```
class ThreadTest1 extends Thread {
    // Metodo para desplegar un mensaje 25 veces
    public void message(String s) {
        for (int i = 0; i < 25; i++) {
            System.out.println(s+": "+(i+1));
        }
    }

    // El metodo run debe ser redefinido y se invoca cuando la
    // hebra se inicia
    public void run() {
        System.out.println("Prioridad de la hebra: " + getPriority());
        message("Salida de la hebra");
    }

    public static void main(String [] args) {
        // Crea un nuevo objeto para ejecutar una hebra separada
        // y un objeto para ejecutar el programa principal
        Thread t = new ThreadTest1();
        ThreadTest1 test = new ThreadTest1();

        // Notese como una referencia a la hebra actual se obtiene
        // usando un metodo estatico de la clase Thread
        System.out.println("Prioridad del programa principal es: " +
            Thread.currentThread().getPriority());

        // Inicia la nueva hebra
        t.start();

        test.message("Salida del programa principal");
    }
}
```

Una salida entremezclada de este programa puede verse como se muestra enseguida. Esto varía de sistema a sistema dependiendo de las propiedades y velocidad de la Máquina Virtual de Java utilizada:

```
Prioridad del programa principal es: 5
Salida del programa principal: 1
Salida del programa principal: 2
Salida del programa principal: 3
Prioridad de la hebra: 5
Salida de la hebra: 1
Salida del programa principal: 4
Salida del programa principal: 5
Salida de la hebra: 2
Salida de la hebra: 3
Salida del programa principal: 6
Salida del programa principal: 7
Salida de la hebra: 4
Salida de la hebra: 5
Salida de la hebra: 6
Salida de la hebra: 7
...
```

El siguiente ejemplo ilustra el uso de la interfaz `Runnable`. El programa funciona como un reloj muy sencillo, imprimiendo la hora una vez por segundo. También se muestra el uso del método `sleep` de la clase `Thread`. Nótese que el programa no termina con el final del método `main`, sino que permanece en ejecución dado que hay una hebra aún en ejecución.

```
import java.util.Date;

class TestThread2 implements Runnable {
    // Despliega la hora actual cada segundo, repitiendose para siempre
    public void run() {
        while(true) {
            System.out.println(new Date());
            // Espera 1 segundo llamando a sleep, que tiene
            // que estar en un bloque try. Cualquier excepcion
            // que se arroje es ignorada
            try {
                Thread.currentThread().sleep(1000);
            }
            catch (InterruptedException e) {
            }
        }
    }
```

```
    public static void main(String[] args){
        Thread t = new Thread(new TestThread2());

        t.start();
        System.out.println("Fin del programa principal");
    }
}
```

4.4.2. Métodos Sincronizados

En un programa con hebras de control, los métodos que cambian el estado de un objeto deben ser protegidos, ya que dos o más hebras de control separadas pueden intentar la ejecución de un mismo método al mismo tiempo. Un método sincronizado (*synchronized method*) puede sólo ser ejecutado por una hebra si tal hebra obtiene primero un "candado" (*lock*) sobre el objeto del que se invoca el método. Para esto, los métodos de instancia y clase pueden ser declarados con el modificador `synchronized`.

Una vez creadas, todas las hebras se ejecutan dentro del mismo programa y tienen acceso a cualquier objeto del que se pueda obtener una referencia, lo que lleva a la posibilidad de conflictos entre hebras como se describe antes. La solución a este problema se basa en candados para los objetos. Cada objeto tiene un candado (un *token*) que puede retenerse por una sola hebra en cualquier momento dado.

Cuando una hebra ejecuta una invocación a un método que ha sido declarado como `synchronized`, primero intenta obtener el candado del objeto al que pertenece el método invocado. Si se obtiene el candado, la hebra continúa y ejecuta el método, reteniendo el candado hasta retornar del método. Si el candado no puede ser obtenido, la hebra se bloquea, y se suspende su ejecución. Cuando el candado se libera, una de las hebras bloqueadas tiene la oportunidad de obtener el candado, pero sólo cuando el planificador de hebras se lo permite. Una hebra que retiene el candado de un objeto puede invocar otros métodos sincronizados en el mismo objeto, sin necesidad de liberar o re-obtener el candado.

Una vez que el candado de un objeto ha sido solicitado, ninguna otra hebra de control puede ejecutar métodos sincronizados para tal objeto. Más aún, una vez que una hebra retiene el candado de un objeto, sólo lo libera cuando la ejecución de los métodos sincronizados termina y el método que invoca no es sincronizado para el mismo objeto, o cuando se invoca el método `wait`.

Es también posible tener métodos sincronizados estáticos. Como estos métodos no se invocan para un objeto en particular, no es posible obtener un candado de objeto. En lugar de eso, cada clase tiene un candado de clase (*class lock*), el cual reemplaza al candado de objeto. Antes de que un método sincronizado estático sea

ejecutado, la hebra de control que lo invoca debe obtener primero el candado de clase. Aparte de esto, los métodos sincronizados estáticos se comportan de forma muy semejante a los métodos sincronizados de instancia.

Debe tenerse cuidado que el uso de métodos sincronizados no lleve a situaciones como "abrazos mortales" (*deadlocks*), en donde todas las hebras se bloquean tratando de obtener los candados retenidos por otras hebras. Por ejemplo, considérese el siguiente ejemplo:

```
// Declarado en una clase A
public synchronized void f(B b) {
    b.x();
}

// Declarado en una clase B
public synchronized void g(A a) {
    a.y();
}

...

A a = new A(); // Hebra 1
B b = new B(); // Hebra 2

...

// En la hebra 1
a.f(b);

...

// En la hebra 2
b.g(a);
```

Si `a` ejecuta la llamda al método `f` para un objeto de la clase `A`, primero debe obtener el candado del objeto. Una vez que el método está en ejecución, la misma hebra debe obtener el candado en un objeto de la clase `B`, a fin de realizar la llamda `b.x()`, y sin liberar el candado que ya tiene. Una situación similar se aplica para la llamada de `g` a un objeto de la clase `B`. Ambos métodos `f` y `g` requieren de obtener dos candados para terminar con éxito.

El problema surge si `f` y `g` se ejecutan en dos hebras que trabajan con los mismos objetos, como se muestra al final del ejemplo anterior. Si las llamadas de `f` y `g` se entremezclan, entonces la hebra 1 podría comenzar a ejecutar el método `f` y obtener el candado de `a`, mientras que la hebra 2 puede comenzar a ejecutar

el método `g`, obteniendo el candado de `b`, pero teniendo que suspenderse antes de alcanzar la llamada al método `a.y()`. En ese punto, ninguna de las hebras puede proceder, ya que cada una espera la liberación de un candado que es mantenido por la otra, lo que implica que un abrazo mortal ha sucedido.

El abrazo mortal puede ser difícil de detectar, y puede involucrar tres o más hebras. Además, puede ocurrir sólo cuando el planificador de hebras produce un orden específico de acciones entre hebras en un tiempo particular y relativo de las llamadas a los métodos. En general, la única forma de evitar el abrazo mortal es mediante diseño cuidadoso, y evitando llamadas a métodos que pudieran causar conflictos. Depurar programas donde ocurren abrazos mortales es notoriamente difícil, así que es mejor no permitir que el problema ocurra en primera instancia.

Una clase diseñada para que sus objetos trabajen apropiadamente en presencia de múltiples hebras se conoce como "clase segura para hebras" (*thread-safe class*). Típicamente, la mayoría de (si no todos) sus métodos públicos son sincronizados, a menos que un método sea tan simple que no se vea afectado por el estado cambiante del objeto debido a otra hebra. Si hacer un método completo sincronizado resulta poco manejable, entonces puede utilizarse enunciados sincronizados en el método, como se describe en la siguiente sección.

El siguiente ejemplo ilustra la diferencia entre utilizar métodos sincronizados y métodos ordinarios. Dos métodos, `f` y `g`, incrementan dos variables de instancia inicializadas en cero antes de que cada prueba que realiza el programa se inicie. Como `g` tiene el modificador `synchronized`, garantiza que las variables de instancia siempre mantienen el mismo valor, ya que ambas variables comienzan con el mismo valor y se incrementan juntas en el mismo método. El método `f`, sin embargo, no es sincronizado, y la hebra que lo ejecute puede ser interrumpida, permitiendo que los valores de las dos variables de instancia sean divergentes. Nótese que este ejemplo utiliza clases miembro y hebras de prioridad 8 para garantizar que la hebra de baja prioridad no sea ejecutada.

```
class ThreadTest6 {
    // Declara un conjunto de clases miembro que crean hebras
    // que invocan a metodos test diferentes, con diferentes
    // pausas
    class MyThread1 extends Thread {
        public void run() {
            for (int i = 0; i < 10; i++) {
                g();
            }
        }
    }
```

```
class MyThread2 extends Thread {
    public void run() {
        setPriority(8);
        for (int i = 0; i < 10; i++) {
            g();
            pause(700);
        }
    }
}

class MyThread3 extends Thread {
    public void run() {
        for (int i = 0; i < 100; i++) {
            f(500);
        }
    }
}

class MyThread4 extends Thread {
    public void run() {
        setPriority(8);
        for (int i = 0; i < 150; i++) {
            f(100);
            pause(200);
        }
    }
}

// Llamada al paquete de sleep, lo que hace al metodo
// mas facil
public void pause(int n) {
    // Pausa por un tiempo semi-aleatorio
    int r = (int)(Math.random() * n);
    try {
        Thread.currentThread().sleep(r);
    }
    catch (InterruptedException e) {
    }
}
```

```
// Metodo no sincronizado. Las hebras pueden entremezclarse
// al ejecutar esto
public void f(int p) {
    i++;
    pause(p);
    j++;
    System.out.println("Metodo f: " + ((i==j)? "Igual" : "Diferente"));
}

// Las hebras no pueden entremezclarse al ejecutar el siguiente
// metodo
public synchronized void g() {
    i++;
    j++;
    System.out.println("Metodo g: " + ((i==j)? "Igual" : "Diferente"));
}

// Ejecuta las pruebas
public void go() {
    // Llama al metodo g. Notese que las variables i y j
    // no divergen
    System.out.println("Llamando al metodo g");
    Thread t1 = new MyThread1();
    Thread t2 = new MyThread2();

    i = 0;
    j = 0;

    t1.start();
    t2.start();

    try {
        t1.join();
        t2.join();
    }
    catch(InterruptedException e) {
    }

    // Llama al metodo no sincronizado y notese que las
    // variables i y j divergen
    System.out.println("Llamando al metodo f");
    Thread t1 = new MyThread3();
    Thread t2 = new MyThread4();

    i = 0;
```

```
        j = 0;

        t1.start();
        t2.start();
    }

    public static void main(String[] args) {
        ThreadTest6 shared = new ThreadTest6();
        shared.go();
    }

    private int i = 0;
    private int j = 0;
}
```

4.4.3. El enunciado `synchronized`

El enunciado `synchronized` permite que un enunciado individual (incluyendo un enunciado compuesto) pueda ser protegido mediante el candado de un objeto, el cual debe ser obtenido antes de que el enunciado se ejecute.

El enunciado `synchronized` tiene la forma:

syncronized (*referencia a objeto*) *enunciado*

El enunciado puede ser un enunciado compuesto.

El enunciado `synchornized` se ejecuta para la primera hebra que obtenga el candado del objeto referenciado por el valor en la expresión entre paréntesis. Si se obtiene el candado, el enunciado se ejecuta, y se libera el candado. De otra manera, la hebra se bloquea y se forza a esperar hasta que el candado se haga de nuevo disponible.

El siguiente programa usa el enunciado `synchronized`.

```
class ThreadTest7 {
    class MyThread extends Thread {
        public void run() {
            for (int i = 0; i < 99; i++) {
                synchronized(vals) {
                    vals[i] += vals[i+1];
                }
            }
        }
    }
```

```
    public void go() {
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        }
        catch(InterruptedException e) {
        }

        for (int i = 0; i < 100; i++) {
            System.out.println(vals[i] + " ");
        }
    }

    public static void main(String[] args) {
        ThreadTest7 t7 = ThreadTest7();
        t7.go();
    }

    private int[] vals = new int[100];
    {
        for (int i = 0; i < 100; i++) {
            vals[i] = i;
        }
    }
}
```

4.5. Ventanas (*Windows*)

En la actualidad, muchos programas proveen de un formato gráfico de interacción con el usuario: las Interfaces Gráficas de Usuario (*Graphical User Interface* o GUI). Al usuario se le presentan una o más "ventanas" (*windows*) que contienen texto, dibujos, diagramas de varios tipos o imágenes. El usuario se comunica con el programa mediante escribir en áres designadas, o moviendo el "ratón" (*mouse*) a una posición en particular, y presionando un botón del mismo. Las acciones pueden ser presionar un botón en pantalla, recorrer una ventana a través de un diagrama más grande o área de texto, indicar un lugar en particular dentro de un dibujo donde una acción o re-dibujado se requiere, etc.

En el pasado, uno de los problemas en crear un GUI había sido la plataforma, es decir, la computadora y ambiente de ventanas del sistema operativo. Aún al escribir un programa en un lenguaje portable estándar como C, los diferentes sistemas operativos tenían (y en ocasiones, siguen teniendo) un conjunto de elementos diferentes para los GUI, con diferentes comportamientos.

En esta sección se describe cómo construir un GUI para un programa escrito en Java. Java provee de un conjunto de elementos contenidos en las bibliotecas del *Abstract Windowing Toolkit* (AWT), que permiten la construcción de GUIs con la misma apariencia y comportamiento en varias plataformas. Esto se logra mendiante un mapeo de los elementos de AWT a los elementos que provee el sistema operativo en particular donde el programa se ejecuta.

Existen más de cincuenta clases en las versiones actuales de AWT. No es el objetivo de esta sección describirlas en detalle, sino más bien introducir un número de programas cortos que usan AWT para proveer de GUIs simples. Los ejemplos aquí se basan en derivar nuevas clases de las clases provistas por AWT, por lo que es importante considerar las características de la herencia en Java. Para mayor información sobre AWT en Java, consúltese la documentación en línea del lenguaje.

Hay diferencias significativas entre un programa con interfaz de texto y uno con interfaz gráfica. Principalmente, se refieren al rango de eventos que el programa puede manejar. Cuando un programa con interfaz de texto espera una entrada por parte del usuario, la única cosa que puede suceder es que el usuario eventualmente escriba una secuencia de caracteres. El programa interpreta la secuencia de caracteres, decide si son o no válidos, y toma la acción apropiada. Sin embargo, en una interfaz gráfica, es posible que haya varias áreas de texto, o sea necesario presionar con el ratón en alguna parte de la ventana. Más aun, se pueden presentar al usuario más de una ventana, todas conectadas al mismo programa. El usuario puede decidir mover una de estas ventanas, obscureciendo u ocultando parcial o totalmente las otras. El usuario puede intentar recobrar ventanas obscurecidas u ocultas, y el sistema debe reconocer y redibujar las partes que se hacen visibles de nuevo. El usuario puede redimensionar la ventana, haciéndola más grande o

pequeña, y el sistema debe hacerse cargo de redibujar la interfaz gráfica a su nuevo tamaño.

Una característica central de un programa gráfico es el ciclo de evento (*event loop*). El programa espera que un evento ocurra, y cuando ocurre causa que una acción apropiada suceda. Por ejemplo, si la ventana se expone tras haber sido oculta, o si se cambia su tamaño, es necesario invocar un método para que se redibuje la ventana. Si un botón ha sido presionado, debe llamarse al método adecuado para tratar este evento (presumiblemente, debe hacerse algún cálculo, que requiere que algo o todo de la ventana se redibuje). Después de que la acción apropiada al evento se realiza, el programa vuelve a esperar en el ciclo de evento.

El ciclo de evento está dentro del código que implementa AWT, y no aparece explícitamente como parte del cógido del programador. El programador debe considerar normalmente el hacer ciertas inicializaciones que preparan a los componentes gráficos para desplegarse, y en seguida se coloca dentro del ciclo de evento de AWT. El resto del programa es entonces una serie de fragmentos (típicamente, métodos), cada uno de los cuales es invocado por el ciclo de evento de AWT para tratar cierto tipo de evento.

Hay aún otra característica importante de la programación gráfica que se ilustra en los siguientes ejemplos de programas. Normalmente, es necesario un método que redibuje una parte de la ventana, basado en el estado de varias variables del programa. En Java, este método es generalmente llamado `paint`. Cuando un evento ocurre, la acción correspondiente debe normalmente registrar los detalles del evento, y después pedir a AWT que lleve a cabo el redibujado de la parte apropiada de la pantalla en un tiempo determinado. Esta petición se realiza mediante el método `repaint`. Por ejemplo, un método invocado como resultado de presionar con el botón del ratón en cierta posición simplemente registra la posición, e invoca a `repaint`. Esto permite que la acción se realice más ágilmente, lo que permite al programa continuar recibiendo eventos como ocurran. Redibujar las parte de la ventana con el método `paint` puede hacerse, pero en general es más lento.

4.5.1. Un Programa Simple con Interfaz Gráfica

Considérese el siguiente programa simple:

```
import java.awt.*;
import java.awt.event.*;

public class Simple extends Frame
    implements WindowListener, ActionListener {

    public Simple() {
        // Prepara la ventana basica
        setTitle("Simple");
```

```
        setBackground(Color.green);
        setSize(500, 400);
        addWindowListener(this);

        // Prepara el area para un dibujo
        Canvas c = new Canvas0();
        add("Center", c);

        // Prepara una area con botones
        Panel p = new Panel();
        p.setLayout(new FlowLayout());
        Button quit = new Button("Quit");
        p.add(quit);
        quit.addActionListener(this);
        add("South", p);
    }

    public static void main (String[] args) {
        Simple s = new Simple();
        s.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        // Trata la entrada del boton quit
        dispose();
        System.exit(0);
    }

    public void windowClosing(WindowEvent event) {
        dispose();
        System.exit(0);
    }

    public void windowOpened(WindowEvent event) {}
    public void windowIconified(WindowEvent event) {}
    public void windowDeiconified(WindowEvent event) {}
    public void windowClosed(WindowEvent event) {}
    public void windowActivated(WindowEvent event) {}
    public void windowDeactivated(WindowEvent event) {}
}

class Canvas0 extends Canvas {

    public void paint(Graphics g) {
        // Prepara algunas dimensiones para el area de dibujo
```

```
Dimension d = getSize();
int cx = d.width/2,
    cy = d.height/2,
    faceRadius = 50,
    noseLength = 20,
    mouthRadius = 30,
    mouthAngle = 50,
    eyeRadius = 5;

// Dibujar el marco
g.setColor(Color.black);
g.drawRoundRect(2, 2, d.width - 5, d.height - 5, 20, 20);

// Dibujar una cara
g.setColor(Color.red);
g.drawOval(cx - faceRadius, cy - faceRadius,
           faceRadius * 2, faceRadius * 2);
g.setColor(Color.blue);
g.fillOval(cx - 30 - eyeRadius, cy - 20,
           eyeRadius * 2, eyeRadius * 2);
g.fillOval(cx + 30 - eyeRadius, cy - 20,
           eyeRadius * 2, eyeRadius * 2);
g.setColor(Color.red);
g.drawLine(cx, cy - (noseLength/2), cx, cy + (noseLength/2));
g.drawArc(cx - mouthRadius, cy - mouthRadius,
          mouthRadius * 2, mouthRadius * 2,
          270 - mouthAngle, mouthAngle * 2);

// Escribe texto
Font f1 = new Font("TimeRoman", Font.PLAIN, 14);
Font f2 = new Font("TimeRoman", Font.ITALIC, 14);
FontMetrics fm1 = g.getFontMetrics(f1);
FontMetrics fm2 = g.getFontMetrics(f2);
String s1 = "Hello, ";
String s2 = "World";
int w1 = fm1.stringWidth(s1);
int w2 = fm2.stringWidth(s2);
g.setColor(Color.black);
g.setFont(f1);
int ctx = cx - ((w1 + w2)/2);
int cty = cy + faceRadius + 30;
g.drawString(s1, ctx, cty);
ctx += w1;
g.setFont(f2);
g.drawString(s2, ctx, cty);
```

```
    }
}
```

La clase de Java que implementa una ventana independiente en la pantalla es llamada `Frame`. La versión de `Frame` del programa contiene un área donde se dibuja una "carita feliz", que es otra clase derivada de la clase `Canvas`. Se le añade un área en la parte baja, `Panel`, que contiene un botón de salida "Quit". Entonces, la clase `Simple` se deriva de `Frame`, y provee de la descripción básica de la ventana, ejecuta el programa de inicio, y especifica la acción de presionar el botón "Quit". La clase `Canvas0` se deriva de `Canvas`, y contiene la información para realizar el dibujo de la cara y el texto que la acompaña. Las clases y métodos de AWT se encuentran en dos paquetes: `java.awt` y `java.awt.event`, por lo que se importan la principio del programa.

4.5.2. Descripción de la Ventana

El método constructor `Simple()` especifica las características principales de la ventana:

- Especifica el título de la ventana, usando `setTitle("Simple")`. Si se omite este paso, un título por omisión podría parecer, como `Untitled` o `*****`.
- El método `setBackground(Color.green)` pinta de color verde el fondo de la ventana. El argumento de este método es una instancia de la clase `Color`. Esta clase provee de un número de colores como constantes. `Color.green` especifica el verde, pero las constantes pueden ser: `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white`, o `yellow`. De hecho, es posible construir combinaciones de colores con la clase `Color`.
- Especifica el tamaño de la ventana con `setSize(500, 400)`. Este método toma dos parámetros `int`, el primero especificando el ancho y el segundo la altura, ambos en unidades de "pixeles". Un pixel (o *picture element*) es la unidad básica de resolución de una pantalla; es el área de pantalla más pequeña que puede ser direccionada, y es la unidad en que las coordenadas dentro de una ventana se especifican, a partir de un origen en la esquina superior izquierda. No debe omitirse este paso, ya que AWT necesita esta información inicial para considerar la nueva ventana que se está creando.
- El método `addWindowListener` se discute más adelante.
- El área de dibujo se establece como una instancia de la clase `Canvas`. Esto se declara como `Canvas c = new Canvas0()`. Se desea que el área de dibujo aparezca con un botón en su parte inferior. Es posible indicar esto a AWT mediante una posición exacta, pero en general es posible arreglar los elementos utilizando posiciones relativas. Para esto, AWT provee de un número

de manejadores de presentación (*layout managers*), los cuales hacen la labor de posicionar los componentes de una gráfica en un "contenedor". La presentación por omisión de las clases derivadas de `Frame` se conocen como "de borde" (*border*). Se añade entonces un componente de dibujo mediante el método `add`. Utilizando el manejador de presentación, las posiciones relativas se especifican como se muestra en la figura 4.4.

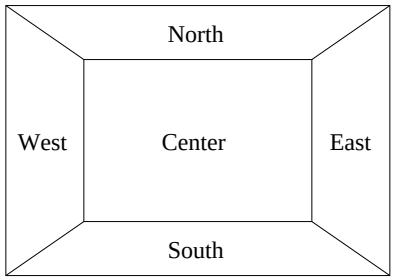

Figura 4.4: Posiciones relativas del manejador de presentaciones "de borde"

De esta forma, se hace que el área de dibujo se coloque en la posición `Center`, y el grupo de botones va en la parte `South`. No es necesario llenar todas las posiciones de la presentación "de borde".

- Se especifica el boton "Quit". Este se coloca en un panel, que es un componente general de ventanas que contiene otros componentes. Se declara como instancia de la clase `Panel`, mediante la línea `Panel p = new Panel()`. Dentro del método `setLayout`, se especifica además que el panel debe tener otro manejador de presentación, llamado de "flujo" (*flow*), el cual arregla los componentes en forma de renglón de izquierda a derecha, y en seguida, de arriba hacia abajo. Esto se hace mediante la línea `p.setLayout(new FlowLayout())`. Se crea entonces un botón etiquetado con la palabra "Quit", mediante crear una instancia de la clase `Button` con la etiqueta como argumento de su constructor, y se añade al panel `p` utilizando el método `add`:

  ```
  Button quit = new Button("Quit");
  p.add(quit);
  ```

  Finalmente, se añade el panel `p` a la clase derivada de `Frame`, posicionandolo en la parte inferior de la ventana mediante `add("South", p)`.

En este ejemplo, se utilizan los manejadores de presentaciones `FlowLayout` y `BorderLayout`. Existen otros manejadores, pero tal vez el más útil pudiera ser el manejador `GridLayout`, que arregla los componentes en forma matricial de renglones y columnas. Por ejemplo:

```
p.setLayout(new GridLayout(2, 3));   // 2 renglones, 3 columnas
```

Los componentes se añaden utilizando el método `add` sin argumentos adicionales. Es por esto que los componentes deben añadirse en un orden específico para llenar renglones y columnas. En el ejemplo anterior, los componentes se agregan utilizando las invocaciones al método `add` en el siguiente orden:

```
p.add(renglon0columna0);
p.add(renglon0columna1);
p.add(renglon0columna2);
p.add(renglon1columna0);
p.add(renglon1columna1);
p.add(renglon1columna2);
```

Esto genera una presentación de la forma que se muestra en la figura 4.5.

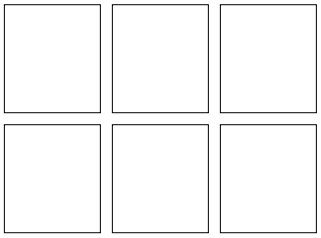

Figura 4.5: Renglones y columnas en una `GridLayout`

Se puede ser todavía más específico sobre cómo los componentes se acomodan, por ejemplo, utilizando la presentación `GridBagLayout`, o simplemente ir recursivamente añadiendo un `GridLayout` anidado en otro `GridLayout`, y así sucesivamente.

Ahora bien, el método `main` de la clase `Simple` se encarga de:

- Inicializar una variable de tipo local `Simple` con el constructor que se ha descrito, mediante la línea `Simple s = new Simple()`.
- En seguida, invoca al método `setVisible`, que despliega el objeto en pantalla, mediante la línea `s.setVisible(true)`. Finalmente, el objeto `s` entra al ciclo de evento de AWT, esperando por eventos que ocurran.

El método `actionPerformed` de la clase `Simple` debe especificarse de tal modo que defina qué sucede cuando el botón "Quit" se presiona. Cuando se presiona un botón, el systema AWT invoca al método `actionPerformed`, el cual pertenece a la interfaz `ActionListener`. Por lo tanto, primero es necesario declarar la clase `Simple` como implementación de la interface:

```
public class Simple extends Frame implements ActionListener
```

Por cada botón que se añada, es necesario definir a AWT qué clase contiene el método `actionPerformed`. En general, la clase es aquélla que contiene al constructor que establece al botón en un principio. De este modo, es posible utilizar la palabra clave `this` como argumento del método `addActionListener`, aplicado al botón que se desee utilizar:

```
quit.addActionListener(this);
```

Es necesario ahora escribir el método `actionPerformed`. Este tiene un sólo argumento, un objeto de tipo `ActionEvent`, el cual da los detalles de qué sucede:

```
public void actionPerformed(ActionEvent) {
    ...
}
```

Por el momento, este ejemplo sólo cuenta con un botón, de tal modo que si este metodo es invocado, se asume que el botón "Quit" ha sido presionado. Si esto sucede, es necesario salir del programa mediante invocar al método `System.exit`. Ya que es una salida normal del programa, se da un argumento de valor cero. Sin embargo, antes de hacer esto, es necesario asegurarse que todos los recursos que el sistema usa en este objeto sean retornados, utilizando el método `dispose`:

```
    dispose();
    System.exit(0);
```

Finalmente, es necesario considerar un método `windowClosing` para asegurar que siempre se pueda salir del programa si algo sale mal. Para esto, es necesario implementar la interfaz `WindowListener`:

```
public class Simple extends Frame
            implements WindowListener, ActionListener
```

Dado que se especifica que el método `windowClosing` debe encontrarse en esta clase, se inserta en el enunciado del constructor:

```
addWindowListener(this);
```

De este modo, el método `windowClosing` sólo sale del programa:

```
public void windowClosing(WindowEvent event) {
    dispose();
    System.exit(0);
}
```

Desafortunadamente, el implementar la interfaz `WindowListener` involucra considerar otros seis métodos para controlar varias otras cosas que pueden pasar a una ventana. En el ejempo, se proveen de versiones vacías para estos métodos:

```
public void windowOpened(WindowEvent event) { }
public void windowIconified(WindowEvent event) { }
public void windowDeiconified(WindowEvent event) { }
public void windowClosed(WindowEvent event) { }
public void windowActivated(WindowEvent event) { }
public void windowDeactivated(WindowEvent event) { }
```

En general, para varios otros programas, es posible copiar simplemente las clases derivadas de `Frame` (es decir, la clase principal) en cada programa en que se necesite utilizar ventanas. Esto termina la discusión del código para la clase `Simple`. Ahora, se comenta el código de la clase `Canvas0`.

4.5.3. El Área de Dibujo

La clase `Canvas0` no contiene campos de instancia, y sólo comprende al método `paint`. Para esta clase, se utiliza el constructor por omisión.

El método `paint` se invoca cuando la ventaja se despliega por primera vez, y en cualquier momento en que AWT requiera redibujar (por ejemplo, si alguna parte de o toda la ventana ha estado oculta y se revela otra vez, o cuando se cambie el tamaño de la ventana, etc.) se puede utilizar el método `repaint`.

Para poder dibujar el área correctamente es necesario conocer el tamaño actual de tal área. Esto se logra mediante el método `getSize`, el cual retorna una instancia de la clase `Dimension`. De esta instancia, se extraen los valores `width` y `depth`, que corresponden al ancho y altura del área de dibujo en pixeles. A partir de ellos, es posible calcular el punto central del área `(cx, cy)`, y dibujar una línea alrededor del borde del área.

El método `paint` tiene sólo un argumento, del tipo `Graphics`:

```
public void paint (Graphics g)
```

Cuando se dibujan líneas, se llenan áreas de color, se escriben caracteres, etc., se invocan los métodos asociados con el objeto `g`. De este modo, se puede dibujar un rectángulo con esquinas redondeadas mediante:

```
g.drawRoundRect(...argumentos...);
```

Hay una gran colección de métodos para dibujar de diferentes formas sobre el área de dibujo. A continuación, se listan los más útiles, y se muestra un código para la clase `Canvas0`. Cada uno se dibuja utilizando el color actual, que se modifica cuando se requiera utilizando el método `setColor`.

- El método `drawLine` dibuja una línea del color actual (por ejemplo, la nariz en el programa anterior). Los cuatro argumentos necesarios son, en orden, las coordenadas x y y de los extremos de la línea. Tales argumentos son de tipo `int`, utilizando las unidades en pixeles, considerando el origen de las coordenadas en la esquina superior izquierda del área. De este modo, la siguiente invocación dibuja una línea como se muestra en la figura 4.6.

  ```
  g.drawLine(10, 20, 30, 10);
  ```

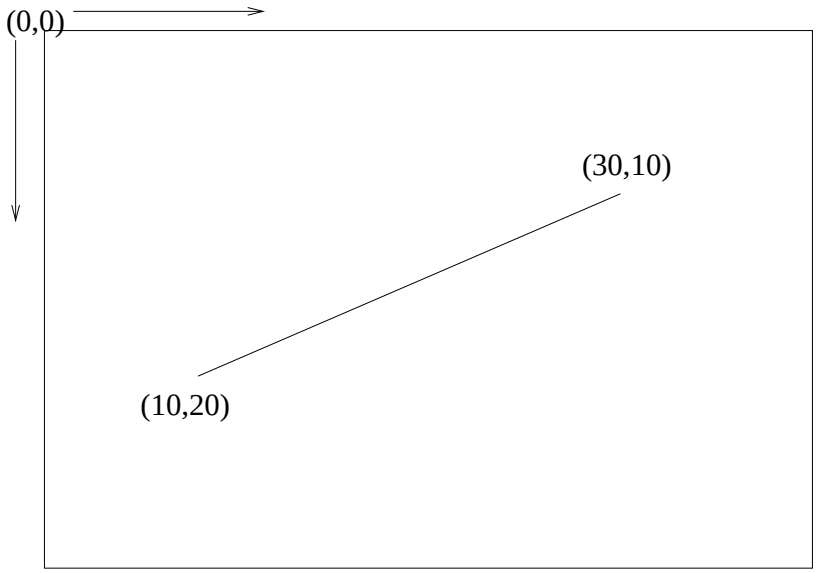

Figura 4.6: Ejemplo de `drawLine`

- El método `drawRect` dibuja un rectágulo en el color actual. Los primeros dos argumentos de este método son las coordenadas x y y de la esquina superior izquierda del rectángulo, mientras que el tercer y cuarto argumentos se refieren respectivamente al ancho y altura del rectángulo en pixeles. Así si el tercer y cuarto argumentos son iguales, el rectángulo será un cuadrado. El siguiente código dibuja un rectángulo como se muestra en la figura 4.7.

```
g.drawRect(10, 5, 20, 15);
```

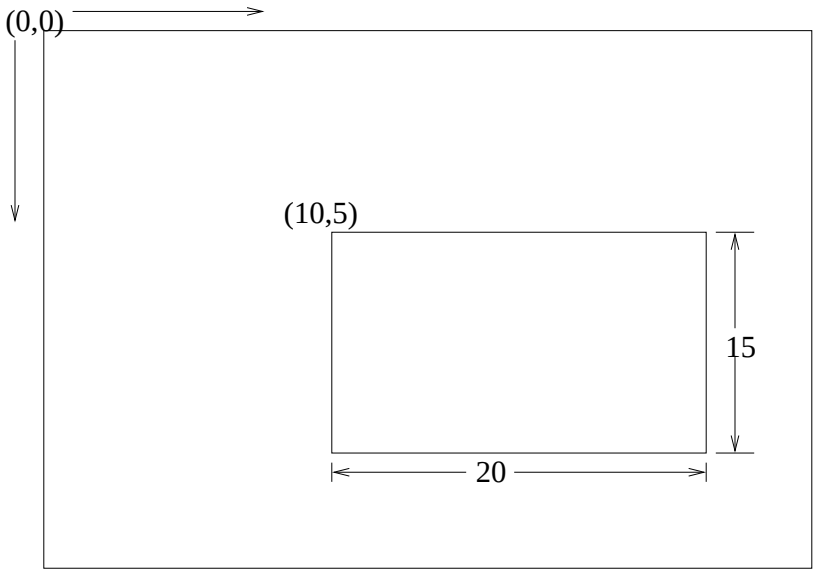

Figura 4.7: Ejemplo de `drawRect`

- El método `drawRoundRect` dibuja un rectángulo en el color actual con esquinas redondeadas. Este método se usa en el ejemplo para dibujar una línea junto al borde del área de dibujo. Los primeros dos argumentos son las coordenadas de la esquina superior izquierda, y el tercer y cuarto argumentos son el ancho y altura del rectángulo, como en el método `drawRect`. Ahora bien, este método requiere un quinto y sexto argumentos, que representan los diámetros horizontal y vertical respectivamente de los arcos de las esquinas en pixeles.

- El método `drawOval` dibuja un óvalo en el color actual. Es necesario imaginar que el óvalo se encuentra inscrito dentro de un rectángulo. De este modo, el método tiene por argumentos las coordenas x y y de la esquina superior

izquierda del rectángulo, y el tercer y cuarto argumentos corresponden al ancho y altura respectivamente del rectángulo, de manera semejante al método `drawRect`. Si el tercer y cuarto argumentos son iguales, el óvalo será un círculo. La siguiente llamada genera un óvalo como se muestra en la figura 4.8.

```
g.drawOval(10, 5, 20, 15);
```

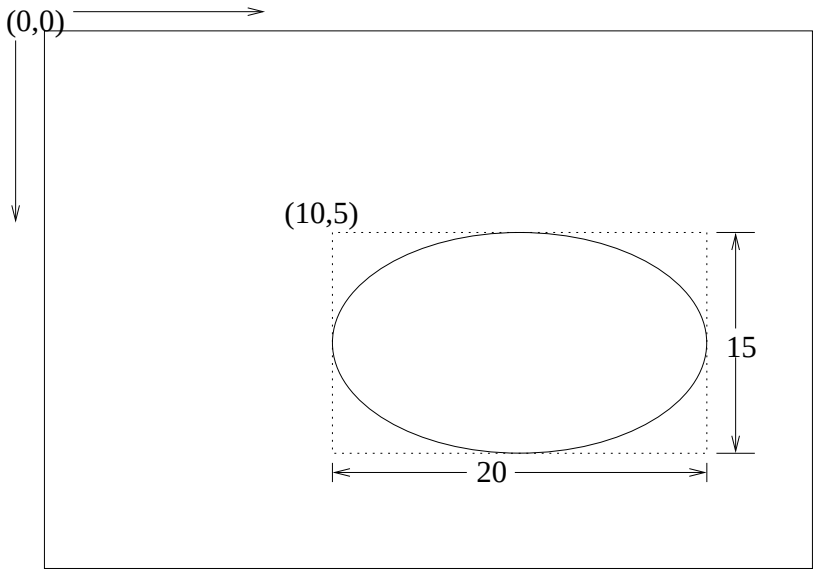

Figura 4.8: Ejemplo de `drawOval`

En ocasiones, definir la posición utilizando la esquina superior izquierda no es el procedimiento más conveniente, pero la mayoría de los métodos en AWT usan esta opción. En el ejemplo, el cálculo de las posiciones de los dibujos siempre buscan localizar la esquina superior izquierda de las formas a partir del punto central de la figura, lo que puede ser una forma más lógica de especificar la posición. Nótese que en general el código del ejemplo se ha escrito para aclarar cómo las posiciones, tamaños, ángulos, etc. se especifican en términos de las varias dimensiones básicas que definen la cara.

- El método `drawArc` dibuja parte de un óvalo o círculo. Es necesario, entonces, especificar primero la esquina superior izquierda del óvalo, y el ancho y alto del rectángulo que lo inscribe, como en el método `drawOval`. Los dos últimos argumentos (el quinto y sexto) especifican qué parte del arco se dibuja. El quinto argumento especifica un ángulo que comienza de la posición de cero

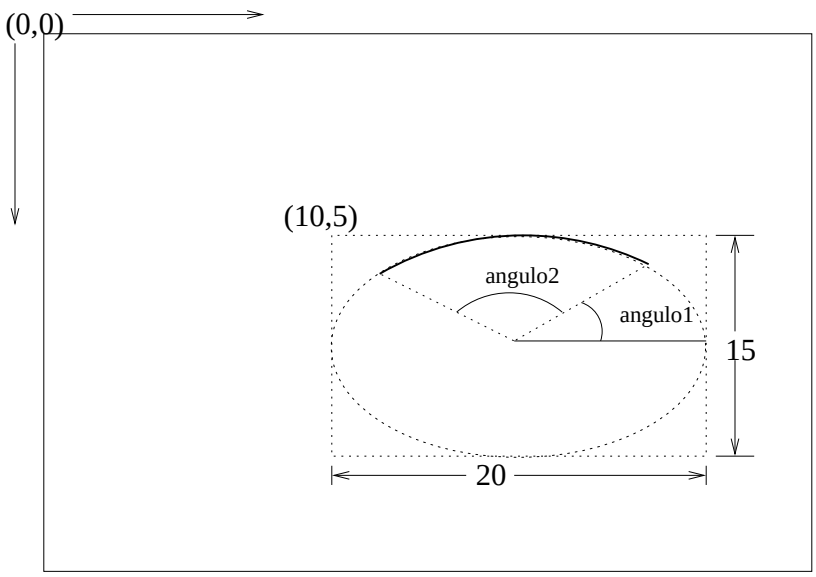

Figura 4.9: Ejemplo de `drawArc`

grados en la horizontal hacia el Este (como en la convención matemática normal) y termina donde se inicia propiamente el dibujo del arco. El arco se dibuja en sentido antihorario, por cuanto grados se especifiquen en el sexto argumento. Nótese que el sexto argumento no es el ángulo final: el ángulo final será la suma del quinto y sexto argumentos, como se muestra en la figura 4.9

Utilizando este método se dibuja la sonrisa de la carita feliz considerando el centro del área de dibujo, el radio del círculo que corresponde a la boca (`mouthRadius`), del cual la sonrisa es un arco, y el ángulo de la boca (`mouthArc`), de 50 grados, mediante el código:

```
g.drawArc(cx - mouthRadius,
      cy - mouthRadius,
      mouthRadius*2,
      mouthRadius*2,
      270 - mouthAngle,
      mouthAngle*2);
```

- También es posible añadir algún texto a un dibujo en el área. Para mostrar las capacidades de AWT en la escritura de caracteres, se escriben las palabras `Hello` en tipo normal y `World` en cursivas. El método básico es `drawString`,

el cual escribe una cadena especificada de texto (como primer argumento) en la posición que se especifica (con el segundo y tercer argumentos, como coordenadas). La siguiente llamada escribe un mensaje como se muestra en la figura 4.10.

```
g.drawString("Happy Birthday", 10, 15);
```

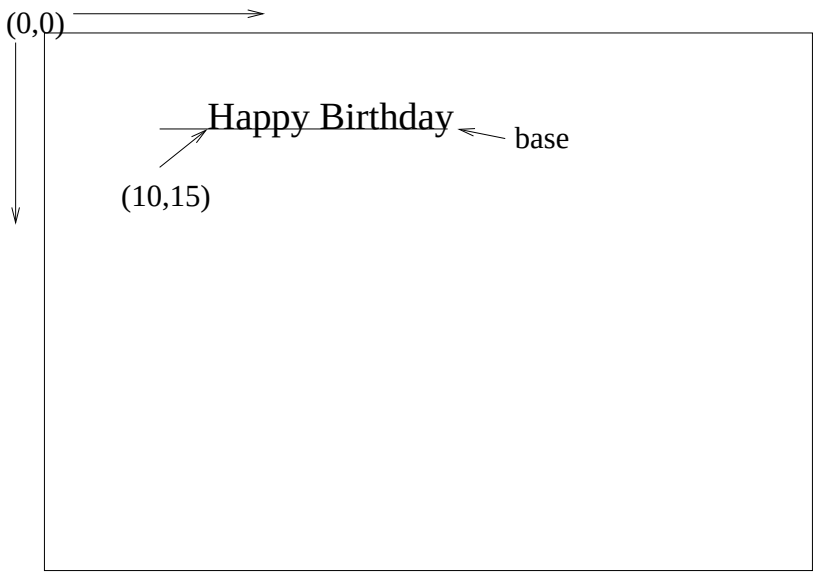

Figura 4.10: Ejemplo de `drawString`

La primera coordenada se refiere a la posición del extremo izquierdo del primer caracter. La segunda coordenada es la posición de la "base" sobre la cual los caracteres se escriben; algunos caracteres (como por ejemplo, 'g', 'y', 'j') pueden extenderse por debajo de la base.

La cadena de caracteres se dibuja en el color y tipo de letra actuales. Hasta ahora, se ha considerado el concepto de color actual, pero en cuanto al tipo de letra (o *font*) se requiere utilizar el método `setFont` para actualizarlo. Este método requiere un argumento de tipo `Font`.

Las instancias de la clase `Font` se crean mediante invocar al constructor adecuado. Por ejemplo:

```
Font f1 = new Font("TimesRoman", Font.PLAIN, 14);
```

El primer argumento de esta invocación es el nombre del tipo de letra. Otros tipos de letra comunes son `Helvetica`, `Courier`, `Dialogue` y `Symbol`. Existen otra cantidad de tipos de letra dependiendo de la plataforma utilizada.

El segundo argumento es el estilo del tipo de letra. Este debe ser uno de las siguientes constantes: `Font.PLAIN`, `Font.BOLD` o `Font.ITALICS`. Las constantes corresponden a los estilos plano, en negrita y cursiva, respectivamente. Es posible también usar expresiones mixtas, como por ejemplo `Font.BOLD + Font.ITALIC`, para el segundo argumento, lo que genera un estilo en negrita y cursiva, siempre y cuando se encuentre esta opción disponible para el tipo de letra especificado.

El tercer argumento se refiere al tamaño en puntos del tipo de letra. Tamaños típicos son de 8 (para un tamaño pequeño), 12 (de un tamaño normal) o hasta 36 o más (para tamaños grandes).

Si se desea posicionar una secuencia de cadenas de caracteres, tal vez con tipos de letra variables, AWT espera que el programador especifique la posición de cada uno de ellos, y para esto es necesario qué tanto espacio requiere una secuencia particular de caracteres de un tipo de letra. La clase `FontMetrics` provee tal información. Se establece una instancia de esta clase para un objeto `Graphics` y para un objeto `Font`:

```
FontMetrics fm1 = g.getFontMetrics(f1);
```

A partir de esta instancia, es posible obtener el ancho de una cadena de caracteres en el tipo de letra especificado, mediante:

```
int w1 = fm1.stringWidth("Se desea saber el espacio de esta cadena");
```

En el ejemplo del programa, se utilizan las longitudes de las cadenas para calcular la coordenada `ctx` para el inicio de la cadena, para que el mensaje esté posicionado centralmente en la ventana, y se actualiza este valor para obtener la posición de la siguiente porción de la cadena.

4.5.4. Entrada de Caracteres

En el programa previo, la única forma en que el usuario puede comunicarse con el programa es presionar un botón. Otro tipo deseable de comunicación con el programa es permitir al usuario escribir una secuencia de caracteres en un campo de texto, para que el programa lo interprete.

Supóngase el ejemplo de un programa que permite escribir una temperatura en grados Celsius en un campo de texto etiquetado como "Celsius", y que utiliza un botón que al ser presionado calcula la temperatura equivalente en grados Fahrenheit en un campo de texto etiquetado como "Fahrenheit". Similarmente, es posible escribir en este campo de texto, y utilizar otro botón para convertirlo en grados Celsius en el otro campo de texto. Es necesario considerar la validez de la cadena de caracteres que se escribe.

Para este ejemplo, se utilizan de nuevo una clase `Simple2` derivada de `Frame`, con seis variables de instancia:

- Una variable `temp1`, de una clase predefinida `Temperature`, que sirva para la conversión.
- Dos variables `celsiusField` y `fahrField` de tipo `TextField`. Esta clase implementa una pequeña área de texto en la que el usuario puede escribir una secuencia de caracteres que el programa puede extraer y procesar de alguna manera. Es necesario entonces declarar estas dos variables de forma que pueda invocarse a los métodos asociados con ellas en el método `actionPerformed`.
- Tres variables de la clase `Button`, llamadas `toF`, `toC` y `quit`, que permiten al método `actionPerformed` checar qué botón ha sido presionado.

La declaración de `Simple2` queda hasta ahora como sigue:

```
public class Simple2 extends Frame
                     implements WindowListener, ActionListener {

       Temperature temp1;
       TextField celsiusField, fahrField;
       Button toF, toC, quit;
       ...
}
```

El constructor de la clase inicializa las variables:

```
public Simple2 () {
    temp1 = new Temperature();
    setTitle = ("Temperatura");
    setBackground(Color.green);
    setSize(400, 600);
    addWindowListener(this);
    ...
}
```

En la ventana que se define, es necesario crear los componentes gráficos puestos en forma de tres renglones. El primero y segundo renglones son muy similares, requiriendo:

- Un campo de texto en el medio, en el cual se pueda escribir la temperatura. Este es la instancia de la clase `TextField`. El constructor de esta clase toma dos argumentos: la cadena de caracteres a aparecer inicialmente en el campo de texto (que normalmente puede ponerse en blanco), y un `int` que especifica qué tan ancho es el campo de texto en "columnas". Lo mejor es considerar este valor una o dos veces más largo que el máximo número de caracteres que se espera. Para este ejemplo, se consideran 8 columnas.
- Una cadena de caracteres a la izquierda de cada campo de texto, la cual especifica qué tipo de temperatura se trata. Se utiliza una instancia de la clase `Label`, cuya función es desplegar una cadena de caracteres. Cuando se construye una instancia de `Label`, es necesario dar como argumento una cadena de caracteres (por ejemplo, "Celsius" en el primer renglón).
- Un botón para invocar la conversión de la temperartura, como instancia de la clase `Button`. Se le coloca una etiqueta para determinar su función (en el primer renglón, se considera "Convierte a F").

Procurando mantener los componentes de cada renglón juntos, se les coloca en un panel `p1` con un manejador de flujo:

```
Panel p1 = new Panel();
p1.setLayout(new FlowLayout());
p1.add(new Label("Celsius"));

celsiusField = new TextField(" ", 8);
p1.add(celsiusField);

toF = new Button("Convierte a F");
p1.add(toF);
toF.addActionListener(this);
```

Haciendo lo mismo con el segundo renglón, en un panel `p2`:

```
Panel p2 = new Panel();
p2.setLayout(new FlowLayout());
p2.add(new Label("Fahrenheit"));

fahrField = new TextField(" ", 8);
p2.add(fahrField);
```

```
toC = new Button("Convierte a C");
p2.add(toC);
toC.addActionListener(this);
```

Finalmente, se añade un tercer panel `p3` para el botón "Salir":

```
Panel p3 = new Panel();
p3.setLayout(new FlowLayout());
quit = new Button("Salir");
p3.add(quit);
quit.addActionListener(this);
```

Para poder utilizar estos tres paneles, es necesario declarar un nuevo panel `p`, especificando un manejador de borde, y entonces añadir los tres paneles `p1`, `p2` y `p3`:

```
Panel p = new Panel();
p.setLayout(new BorderLayout());
p.add("North", p1);
p.add("Center", p2);
p.add("South", p3);
add("Center", p);
```

La ventana se ve similar a la figura 4.11.

El método `main` de la clase `Simple2` se encarga de crear una instancia de la clase, y llama al método `setVisible` para iniciar el proceso:

```
public static void main(String[] args) {
    Simple2 f = new Simple2();
    f.setVisible(true);
}
```

El método para `windowClosing` resulta ser igual que en el ejemplo anterior.

Ahora bien, para extraer la cadena de caracteres escrita en un campo de texto, debe modificarse la estructura general del método `actionPerformed`. Sin embargo, resulta un código muy similar al visto anteriormente. Dentro de este método se prueba en secuencia cada etiqueta y cada botón que podría ser presionado, tomando la acción correspondiente. A continuación, se presenta la parte para convertir de grados Celsius a grados Fahrenheit cuando el botón `toF` es presionado. La conversion inversa es muy similar. El código es como sigue:

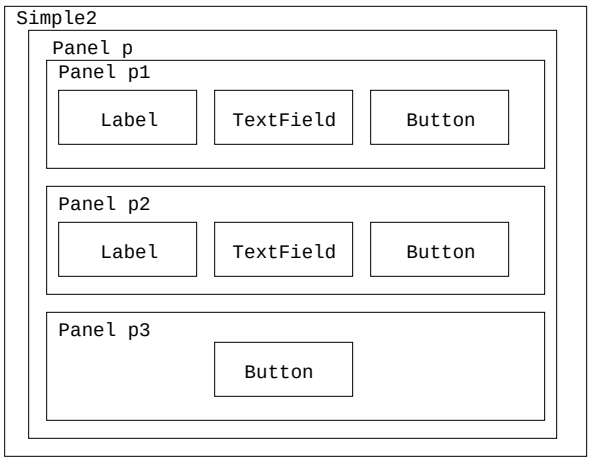

Figura 4.11: Ventana para `Simple2`

```
...
else if (event.getSource() == toF) {
    String c1 = celsiusField.getText();
    double c2 = 0.0;
    boolean isValid;

    try {
        c2 = Double.valueOf(c1).doubleValue();
        isValid = true;
    }
    catch (NumberFormatException e) {
        isValid = false;
    }

    if(isValid) {
        temp1.setCelsius(c2);
        double f2 = temp1.getFahr();
        f2 = ((double) Math.round(f2 * 100)) / 100;
        String f1 = Double.toString(f2);
        fahrField.setText(f1);
    }
    else {
        celsiusField.setText(" ");
    }
}
else ...
```

En el código anterior, se realizan las siguientes acciones:

- Se extrae la cadena del campo de texto "Celsius" usando el método `getText`, y almacenándolo en la variable `c1` de tipo `String`.
- Se convierte la cadena en un valor `double` en un bloque `try`. Si los caracteres escritos en el campo no son válidos, la conversión falla, atrapándose en el bloque `catch` como una excepción. La variable booleana `isValid` registra si la conversión es exitosa.
- Si la entrada es válida, se convierte el valor Celsius a su equivalente Fahrenheit redondeado a dos cifras decimales, y se le convierte de nuevo a una cadena de caracteres para poder escribirse en el campo de texto utilizando el método `setText`.
- Si la entrada es inválida, se borra el contenido del campo de texto utilizando una cadena en blanco.

4.5.5. Menúes, Archivos e Imágenes

Los programas presentados hasta ahora en esta sección para generar dibujos o diagramas han utilizado los métodos `draw`Algo contenidos en AWT. Sin embargo, hay otra fuente de información visual para un programa, en forma de fotografías que pueden obtenerse mediante digitalización, y que se almacenan en formatos estándar como GIF y JPEG. Como parte de AWT, Java provee un conjunto de métodos para el despliegue de tales fotografías, comúnmente conocidas como "imágenes". La mayoría de las maneras para manipular imágenes, como por ejemplo escalarlas o cortarlas, van más allá del objetivo de estas notas. Sin embargo, los dos últimos programas de este capítulo tienen la intención de mostrar cómo desplegar imágenes con los métodos de AWT. Además, se introduce a algunas otras partes de AWT que son útiles al escribir programas con ventanas. Estos son menúes, diálogos para archivos, y el manejo de los eventos del "ratón".

El primer programa, `Simple3.java`, permite al usuario seleccionar un archivo que contiene una imagen. El programa entonces intenta desplegarlo en el centro de una ventana. En lugar de tener todos los botones en despliegue continuo, se les considera dentro de un menú localizado en una barra en la parte superior de la ventana.

Colocando Menúes

En una ventana como la desarrollada en secciones anteriores se añade un sistema de menú. Este menú contiene dos botones: "Cargar", que selecciona un archivo de imagen a desplegar, y "Salir". De hecho, se ha hecho común en la mayoría de los diseños de interfaz a usuario considerar a este menú de funciones "miscelaneas" con el nombre de "Archivo" (o *"File"*), que es lo que se pretende aquí. Para esto, es necesario declarar variables de instancia de tipo `MenuItem` para los dos botones, de tal modo que se sepa cuál ha sido seleccionado:

```
public class Simple3 extends Frame
                     implements WindowListener, ActionListener{
    Canvas3 canvas;
    MenuItem load, quit;
    ...
}
```

El código que se añade al constructor de la clase `Simple3` para incluir el menú es como sigue:

```
Menu menu = new Menu("Archivo");
load = new MenuItem("Cargar");
menu.add(load);
load.addActionListener(this);
quit = new MenuItem("Salir");
menu.add(quit);
quit.addActionListener(this);
MenuBar menubar = new MenuBar();
menubar.add(menu);
setMenuBar(menubar);
```

En el código anterior, se crea un objeto de tipo `Menu` etiquetado con el nombre de "Archivo" en una barra de menú. Se añaden dos botones en el menú con las etiquetas adecuadas, y se registran para saber cuándo cualquiera de ellos se selecciona. Se crea la barra de menú, que es un objeto de tipo `MenuBar`, se le añade el menú, y se utiliza el método `setMenuBar` para añadirlo a la instancia de `Simple3` que se encuentra en construcción. Nótese que sólo es posible añadir barras de menú a las clases derivadas de `Frame`, por lo que no se permite tener menúes conectados a paneles arbitrarios.

Ahora es necesario especificar la acción a tomar cuando alguno de los botones del menú se selecciona. Esto se maneja muy parecido a la forma en que se manejan los botones en programas anteriores. Hay un método `actionPerformed` en la clase `Simple3`, el cual verifica cuál componente se ha seleccionado en la forma usual:

```
public void  actionPerformed(ActionEvent event){
    if(event.getSource()==quit) {
        dispose();
        System.exit(0);
    }
    else if(event.getSource()==load) {
        loadFile();
    }
}
```

Hay una serie de otras características útiles de los menúes. Supóngase que se declara las siguientes variables de instancia:

```
public class Simple3a extends Frame
         implements WindowListener, ActionListener, ItemListener {
    MenuItem buttonA1, buttonA2, buttonA3,
         buttonA4, buttonA5, buttonA6, buttonA7, quit,
         buttonB1p1, buttonB1p2, buttonB1p3,
         buttonB2p1, buttonB2p2, buttonB2p3, buttonB2p4;
    int noOfOptions = 7;
    CheckboxMenuItem[] options = new CheckboxMenuItem[noOfOptions];
    ...
}
```

Entonces, es necesario definir la relación entre todos estos componentes en el constructor:

```
Menu menuA = new Menu("MenuA");
buttonA1 = new MenuItem("Boton A1");
menuA.add(buttonA1);
buttonA1.addActionListener(this);
buttonA2 = new MenuItem("Boton A2");
menuA.add(buttonA2);
buttonA2.addActionListener(this);
buttonA3 = new MenuItem("Boton A3");
menuA.add(buttonA3);
buttonA3.addActionListener(this);
menuA.addSeparator();
buttonA4 = new MenuItem("Boton A4");
menuA.add(buttonA4);
buttonA4.addActionListener(this);
buttonA5 = new MenuItem("Boton A5");
menuA.add(buttonA5);
buttonA5.addActionListener(this);
buttonA6 = new MenuItem("Boton A6");
menuA.add(buttonA6);
buttonA6.addActionListener(this);
buttonA7 = new MenuItem("Boton A7");
menuA.add(buttonA7);
buttonA7.addActionListener(this);
menuA.addSeparator();
menuA.add(new MenuItem("Salir"));
```

```
Menu menuB = new Menu("MenuB");
Menu menuB1 = new Menu("MenuB1");
buttonB1p1 = new MenuItem("Boton B1.1");
menuB1.add(buttonB1p1);
buttonB1p1.addActionListener(this);
buttonB1p2 = new MenuItem("Boton B1.2");
menuB1.add(buttonB1p2);
buttonB1p2.addActionListener(this);
buttonB1p3 = new MenuItem("Boton B1.3");
menuB1.add(buttonB1p3);
buttonB1p3.addActionListener(this);
menuB.add(menuB1);
Menu menuB2 = new Menu("MenuB2");
buttonB2p1 = new MenuItem("Boton B2.1");
menuB2.add(buttonB2p1);
buttonB2p1.addActionListener(this);
buttonB2p2 = new MenuItem("Boton B2.2");
menuB2.add(buttonB2p2);
buttonB2p2.addActionListener(this);
buttonB2p3 = new MenuItem("Boton B2.3");
menuB2.add(buttonB2p3);
buttonB2p3.addActionListener(this);
buttonB2p4 = new MenuItem("Boton B2.4");
menuB2.add(buttonB2p4);
buttonB2p4.addActionListener(this);
menuB.add(menuB2);

Menu menuC = new Menu("MenuC");
options[0] = new CheckboxMenuItem("Lunes");
options[1] = new CheckboxMenuItem("Martes");
options[2] = new CheckboxMenuItem("Miercoles");
options[3] = new CheckboxMenuItem("Jueves");
options[4] = new CheckboxMenuItem("Viernes");
options[5] = new CheckboxMenuItem("Sabado");
options[6] = new CheckboxMenuItem("Domingo");

for (int i = 0; i < noOfOptions; i++) {
    options[i].addItemListener(this);
    menuC.add(options[i]);
}

MenuBar menubar = new MenuBar();
menubar.add(menuA);
menubar.add(menuB);
menubar.add(menuC);
```

- En "Menu A" se tienen varios botones, por lo que usa el método `addSeparator` para formar tres grupos de botones.
- En "Menu B" se tienen dos sub-menúes, "Menu B1" y "Menu B2", cada uno con su propio conjunto de botones. Si se presiona el botón para "Menu B1", esto hace que aparezca el sub-menú con sus tres botones.
- En "Menu C" se tienen un número de *"checkboxes"*, cada uno de los cuales independientemente registra si una opción ha sido checada (*"on"*) o no (*"off"*). Dado que se tienen varios de éstos, se declara un arreglo que los contenga, el cual llama al constructor de cada uno, y los registra con un auditor, añadiéndolos finalmente al menú. Un método, el `itemStateChanged` de la interfaz `ItemListener`, se invoca cada vez que el usuario presiona uno de los *checkboxes*, cambiando su estado:

```
public void itemStateChanged(ItemEvent event) {
    if(event.getSource()==options[0]) {
        System.out.println("La opcion Lunes ha sido presionada");
    else if (...)
    ...
    }
}
```

El estado de cada *checkbox* se considera normalmente como `FALSE`. Se puede cambiar a una estado particular con el método `setState`, y se puede averiguar su estado con el método `getState`.

Seleccionando un Archivo

Regresando al programa original, nótese que el código de la opción "Cargar" del menú invoca a un método `loadFile`, lo que permite al usuario seleccionar un archivo que contiene una imagen para ser desplegada. En seguida, se presenta el código de este método:

```
private void loadFile() {
    FileDialog d = new FileDialog(this, "Cargar Archivo", FileDialog.LOAD);
    d.setDirectory("...directorio donde comenzar la seleccion ...");
    d.setVisible(true);
    name = d.getFile();
    directory = d.getDirectory();
    canvas.repaint();
}
```

En este método se crea una instancia de la clase `FileDialog`, lo que requiere tres parámetros:

- El dueño del diálogo. Esta es una instancia de la clase `Simple3`, con la cual este diálogo debe estar asociado, por lo que se utiliza la palabra clave `this`.
- Una cadena de caracteres que aparece como título de la ventana que surge para el diálogo. Se ha escogido "Cargar Archivo".
- Una de dos constantes: `FileDialog.LOAD`, si se planea leer de un archivo (que es lo que se intenta aquí) o `FileDialog.SAVE`, si se planea escribir un archivo.

Además, se consideran un directorio de archivos inicial para que el usuario seleccione un archivo de ahí, y se invoca al método `setVisible`, lo que despliega la ventana del diálogo. El usuario selecciona un archivo y presiona un botón para indicar que el diálogo se ha completado.

En este punto, se extrae el nombre del archivo seleccionado con el método `getFile`. Si ningún archivo ha sido seleccionado, se retorna un valor de tipo `String` de longitud cero. Ya que el usuario puede cambiar de directorio en el diálogo, es necesario extraer también esta información con el método `getDirectory`.

Desplegando una Imagen

Habiendo obtenido el nombre de un archivo en un directorio, se solicita al área de dibujo redibujarse, intentando desplegar una imagen obtenida del archivo. En seguida, se presenta el método `paint` de la clase `Canvas3`:

```
public void paint(Graphics g) {
    Dimension d = getSize();
    int cx = d.width/2,
    cy = d.height/2;

    //Se dibuja una linea alrededor del area de dibujo
    g.setColor(Color.black);
    g.drawRoundRect(2, 2, d.width - 5, d.height - 5, 20, 20);

    if (parent.name != null) {
        String filename = parent.directory + parent.name;
        Image image = Toolkit.getDefaultToolkit().getImage(filename);
        g.drawImage(image,
            cx - (image.getWidth(this)/2),
            cy = (image.getHeight(this)/2),
            this);
```

```
        // Escribe el nombre del archivo bajo la imagen
        Font f1 = new Font("TimesRoman", Font.PLAIN, 14);
        FontMetrics fm1 = g.getFontMetrics(f1);
        int w1 = fm1.stringWidth(parent.name);
        g.setColor(Color.black);
        g.setFont(f1);
        int ctx = cx - (w1 / 2);
        int cty = cy + (image.getHeight(this) / 2) + 30;
        g.drawString(parent.name, ctx, cty);
    }
    else {
        // Escribe el mensaje "No es archivo" en el centro del
        // area de dibujo
        Font f1 = new Font("TimesRoman", Font.PLAIN, 14);
        ...
    }
}
```

En el código anterior se establece el nombre completo del archivo `filename` a partir de la información recabada por el método `loadFile`. Entonces se lee el archivo, y se construye una instancia de la clase `Image` con el enunciado:

```
Image image = Toolkit.getDefaultToolkit().getImage(filename);
```

Finalmente, la imagen se "dibuja":

```
g.drawImage(image,
        cx - (image.getWidth(this)/2),
        cy = (image.getHeight(this)/2),
        this);
```

El primer argumento de tipo `Image` se refiere a la imagen que se desea desplegar. El segundo y tercer argumentos son la posición en la cual debe desplegarse, a partir de la esquina superior izquierda. Las operaciones aritméticas hacen que la imagen se centre en el área de dibujo. El cuarto argumento de tipo `ImageObserver` permite llevar cuenta del progreso de crear y dibujar la imagen, verificando si se hace exitosamente o no.

Siguiendo al "Ratón"

Finalmente, se modifica el programa de tal modo que la imagen se posicione en el punto donde se presione un botón del ratón. Existen varios métodos que permiten

tomar acciones dependiendo de varias cosas que puede el usuario hace con el ratón. Aquí, se selecciona el método `mouseClicked` de la interfaz `MouseListener`, ya que interesa sólo si el ratón ha sido presionado en el área de dibujo, por lo que se especifica una clase `Canvas5` que implementa tal interface:

```
class Canvas5 extends Canvas implements MouseListener {
    ...
}
```

Es necesario mencionar que esta clase es responsable de manejar el hecho de presionar los botones del ratón, por lo que se añade un enunciado al constructor de la clase:

```
public Canvas5(Simple3  f) {
    parent = f;
    addMouseListener(this);
}
```

En seguida, se presenta el método `mouseClicked` que pertenece a la clase `Canvas5`:

```
public void mouseClicked(MouseEvent event) {
    mx = event.getX();
    my = event.getY();
    repaint();
}
```

El argumento `event` contiene las coordenadas en pantalla del punto donde se presiona un botón del ratón. Esta información se extrae con los métodos `getX()` y `getY()`. Las coordenas se almacenan en dos variables de instancia `mx` y `my`, pertenecientes a la clase `Canvas5`. En seguida se llama a redibujar el área de dibujo con `repaint`. Para colocar la imagen en el punto dado, se posiciona la imagen en el método `paint` en las coordenas `(mx, my)`, en lugar de `(cx, cy)`.

La interfaz `MouseListener` requiere de la implementación de otros métodos, los cuales para este ejemplo se proveen mediante códigos vacíos:

```
public void mousePressed(MouseEvent event){ }
public void mouseReleased(MouseEvent event){ }
public void mouseEntered(MouseEvent event){ }
public void mouseExited(MouseEvent event){ }
```

Apéndice A

Entrada por Teclado

Aun cuando se utilicen ventanas para la comunicación de los programas con el usuario, siempre es conveniente contar con un medio de comunicación simple a través de salida por pantalla en texto, y entrada por el teclado. Java considera un tipo de comunicación básica con el usuario mediante cadenas de caracteres, que se presentan en la pantalla utilizando el método `System.out.println`, que ha sido utilizado extensivamente a través de los programas anteriores. Por otro lado, en el caso de la lectura de datos a partir del teclado, Java provee de la capacidad de solicitar al usuario datos mediante un método apropiado de la clase `BasicIo`. Sin embargo, el uso directo de los métodos de la clase `BasicIo` no resulta sencillo, pues implica la manipulación directa de variables del tipo `InputStreamReader`. Debido a esto, se presenta a continuación la clase `KeyboardInput`, la cual tiene por objetivo apoyar la lectura de tipos primitivos provenientes del teclado en forma segura.

A.1. La clase `KeyboardInput`

`KeyboardInput` es una clase sencilla para leer valores de la línea de comando. Si un error ocurre durante la lectura, una excepción es arrojada y atrapada, lo que hace que un valor por omisión sea retornado.

La declaración de la clase `KeyboardInput` es como sigue:

```
import java.io.*;

public class KeyboardInput {
    private final BufferedReader in =
    new BufferedReader(new InputStreamReader (System.in)) ;
```

```
// Lectura de un valor int del teclado. El valor por omision es 0
public final synchronized int readInteger() {
    String input = "" ;
    int value = 0 ;
    try {
        input = in.readLine() ;
    }
    catch (IOException e) { }
    if (input != null) {
        try {
            value = Integer.parseInt(input) ;
        }
        catch (NumberFormatException e) { }
    }
    return value ;
}

// Lectura de un valor long del teclado. El valor por omision es 0L
public final synchronized long readLong() {
    String input = "" ;
    long value = 0L ;
    try {
        input = in.readLine() ;
    }
    catch (IOException e) { }
    if (input != null) {
        try {
            value = Long.parseLong(input) ;
        }
        catch (NumberFormatException e) { }
    }
    return value ;
}

// Lectura de un valor double del teclado. El valor por omision es 0.0
public final synchronized double readDouble() {
    String input = "" ;
    double value = 0.0D ;
    try {
        input = in.readLine() ;
    }
    catch (IOException e) { }
    if (input != null) {
        try {
            value = Double.parseDouble(input) ;
```

```
        }
        catch (NumberFormatException e) { }
    }
    return value ;
}

// Lectura de un valor float del teclado. El valor por omision es
// 0.0F
public final synchronized float readFloat() {
    String input = "" ;
    float value = 0.0F ;
    try {
        input = in.readLine() ;
    }
    catch (IOException e) { }
    if (input != null) {
        try {
            value = Float.parseFloat(input) ;
        }
        catch (NumberFormatException e) { }
    }
    return value ;
}

// Lectura de un valor char del teclado. El valor por omision es
//  ' ' (espacio)
public final synchronized char readCharacter() {
    char c = ' ' ;
    try {
        c = (char)in.read() ;
    }
    catch (IOException e) {}
    return c ;
}

// Lectura de un valor String del teclado. El valor por omision
// es "" (la cadena vacia)
public final synchronized String readString() {
    String s = "";
    try {
        s = in.readLine() ;
    }
    catch (IOException e) {}
    if (s == null) {
       s = "" ;
```

```
        }
        return s ;
    }
}
```