

## 5 Programación con Java

En este capítulo haremos una breve introducción al lenguaje de Java. Este capítulo no es sustituto de ninguna manera de un libro dedicado exclusivamente a la programación en Java. Más bien buscamos dar una introducción que permita relacionarse ante todo con el lenguaje y luego con la programación que se mostrará más adelante durante el capítulo 10 correspondiente al *Modelo de Implementación*, donde se mostrará parte del código final del *Sistema de Reservaciones de Vuelo*.

### 5.1 Sistema

Para apreciar el gran movimiento que detrás de Java hay que comprender que Java es mucho más que un lenguaje, es más bien un sistema con un alcance muy amplio. Es en cierta manera un fenómeno como el que desato Smalltalk hace 20 años gracias al sistema que tenía alrededor de su lenguaje. Por lo tanto comenzaremos esta sección analizando las características principales del sistema de Java, siguiendo con otros aspectos significativos.

#### 5.1.1 Características

El lenguaje de Java tiene ciertas características que lo han hecho un lenguaje trascendental en la actualidad para la programación de sistemas de cómputo. Estos se pueden reducir a los siguientes puntos:

- ?? **Orientado a Objetos** – Ante todo Java es un lenguaje orientado a objetos, lo cual lo pone en la misma categoría que lenguajes como C++ y Smalltalk. Como parte esta característica, se cuenta con un ligado dinámico de clases en tiempo de ejecución, herencia y polimorfismo, además de aspectos de metanivel similares a los de Smalltalk.
- ?? **Portátil** – Uno de los aspectos que han hecho de Java un lenguaje muy utilizado es su portabilidad. A diferencia de lenguajes como C y C++ que varían en su detalle dependiendo de la máquina en que sean ejecutados, Java es exactamente igual bajo cualquier plataforma. Por ejemplo, a diferencia de C y C++, el tamaño de los tipos de datos en Java es fijo, independiente de la máquina. La gran importancia de este aspecto es que si se compila el programa bajo una plataforma particular, el sistema correrá en cualquier máquina, reduciendo mucho el costo de desarrollo (tiempo y dinero). Para ello existen el concepto de la máquina virtual de Java (JVM – Java Virtual Machine) que debe existir en cada plataforma donde se ejecute un programa de Java.
- ?? **Abierto** – El aspecto de portabilidad se da gracias a su diseño abierto que permite a cualquier compañía, e incluso desarrollador, tomar el código fuente, para adaptarlo a una nueva plataforma donde aún no se ha probado. Ninguno de los demás lenguajes ofrecen tal diseño abierto. Otra razón para la gran aceptación de Java.
- ?? **Gratis** – Muy de la mano con el aspecto “abierto” de Java es que el lenguaje se ofrece gratis aunque bajo licencia a cualquier usuario. Esto reduce obviamente el costo de la aplicación y fortalece la decisión para su utilización bajo distintas plataformas, donde no se incurre en el gran número de licencias pagadas, típicamente por máquina, que la mayoría de los demás productos obligan.
- ?? **Integrado al Web** – Entre todos los aspectos mencionados hasta ahora, quizá el de su integración al Web, ha sido la razón para su gran difusión en una época donde el Internet ha sido de tanta importancia. Java es el único lenguaje, con excepción de algunos lenguajes de *scripts*, que viene integrado con los *browsers* más utilizados en el Web.
- ?? **Simple** – Otro aspecto para lograr la gran aceptación de Java es su similitud con C y C++ en relación a las expresiones básicas del lenguaje. Esto ha permitido a los programadores aprender Java más rápidamente, a diferencia de lenguajes como Smalltalk que requieren un cambio en la manera de pensar para programadores ya acostumbrados a C y C++. Sin embargo, Java se considera más *puro* que C++, ya que un programa en Java no contiene más que clases, simplificando el programa y al propio compilador. Java elimina mucha de la complejidad de C++, como es la aritmética de apuntadores lo cual agrega mucha complejidad en la administración de memoria. Se elimina la complejidad adicional de tipos como estructuras y el uso de asociaciones de tipo a través de **typedefs**, junto con el preprocesador de C++ con palabras reservadas como **#define**, **?include** y **?ifdef**. Otro aspecto que es eliminado es la sobrescritura de operadores. También se eliminan aspectos de manejo complicado como es la herencia múltiple.
- ?? **Robusto** – En contraste a C++ y en especial a C, Java es *fuertemente tipificado*, lo que ayuda a encontrar más fácilmente los errores de programación durante la etapa de compilación. Java también incluye manejo de excepciones y recolección de basura para lograr programas más robustos.

- ?? **Seguro** – Gracias a la eliminación de los apuntadores de C y C++, Java logra un modelo de manejo de memoria mucho más seguro. Esta seguridad es además apoyado por el modelo de verificación de código en tiempo de ejecución, como veremos más adelante en la descripción del modelo completo de Java.
- ?? **Eficiencia** – Java en la actualidad se le considera un lenguaje eficiente. Y aunque nunca llegue a la eficiencia de C si se le compara en la actualidad con C++ en relación a esto. Esta eficiencia se basa, en que se cuenta con un compilador para la generación de código en contraste con aquellos lenguajes completamente interpretados donde el rendimiento es menor. En Java se cuenta en la actualidad con un compilador incremental (JIT – Just-in-Time Compiler), que ayuda a lograr estos objetivos.
- ?? **Bibliotecas** – Otro aspecto que ha hecho de Java un lenguaje de mucha aceptación es la gran riqueza de sus bibliotecas, llamadas *paquetes* (“*package*”). Esto es en radical contraste con C y C++ donde las bibliotecas realmente no existen. Al contrario, Java contiene un sin fin de bibliotecas que facilitan de gran manera la creación de programas, además de asegurar una estandarización entre aplicaciones. Existen bibliotecas para el manejo de estructuras de datos avanzadas, manejo de multimedia, manejo de redes como TCP/IP, procedimientos remotos y concurrencia mediante múltiples hilos. En la actualidad, aprender el lenguaje de Java como tal es sólo un 10% del esfuerzo, el 90% restante debe dedicarse a aprender a utilizar sus bibliotecas. Obviamente se estudian sólo aquellas que se deseen utilizar. Por ejemplo, una biblioteca importante es la del sistema de ventanas que puede correr bajo cualquier plataforma. Existe el AWT (Abstract Window Toolkit) desde la primera versión de Java, y se cuenta en la actualidad con las bibliotecas *JFC* (Java Foundation Classes), también conocidas como *SWING*. Además de éstas existen bibliotecas para manejo de gráficas en 2 y 3 dimensiones. Incluso existen versiones para correr en plataformas móviles, por ejemplo, como asistentes personales.
- ?? **Tecnología** – Existe una gran número de productos y tecnología en general desarrollada alrededor de Java. Aparte de Java como lenguaje se cuenta con productos tales como *EJB* (Enterprise JavaBeans), *JSP* (Java Server Pages), Java Servlets y *JDBC* (Java Data Base Connectors). Además de estas y otras, existen productos relacionados con estándares tales como *CORBA* (Common Object Request Browser Architecture) y *XML* (eXtended Markup Language). En la actualidad se cuenta con tres ediciones principales Java: *J2EE* (Java2 Enterprise Edition), *J2SE* (Java2 Standard Edition) y *J2ME* (Java2 Micro Edition).

### 5.1.2 Ambiente de Compilación y Ejecución

El modelo de compilación y ejecución de Java se muestra en la Figura 5.1. Del lado izquierdo se muestra los pasos para la compilación de un programa en Java, mientras que del lado derecho se muestran los pasos para su ejecución.

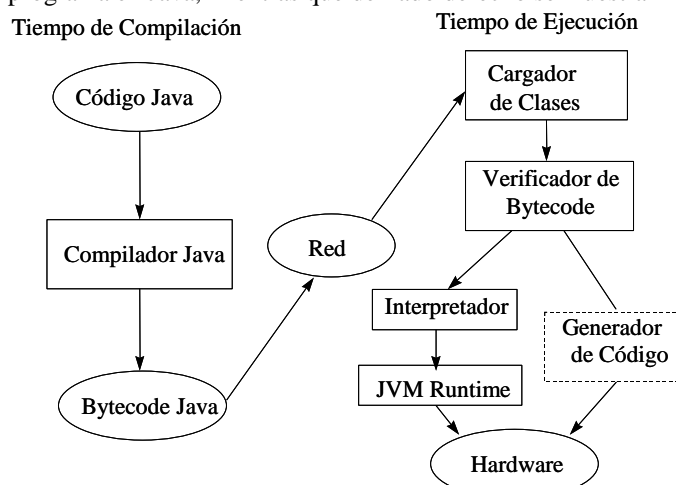


Figura 5.1 Modelo de compilación y ejecución de Java.

#### Compilación

Se escribe un programa en código Java. Este programa, que tiene como extensión el sufijo “.java”, es compilado por cualquiera de los compiladores de Java en alguna de las distintas plataformas. En general debe existir un archivo “.java” por cada clase que exista en el programa, donde el archivo debe tener el mismo nombre que la clase contenida. El compilador genera el código final, conocido como *bytecode*, a ser interpretado por la máquina virtual

de Java. El programa generado tiene como extensión el sufijo “.class”. Se genera un archivo “.class” por cada clase que se tenga en la aplicación.

Por ejemplo, si se tiene una clase llamada “ej”, el nombre del archivo debe ser “ej.java”. El archivo se compilaría mediante algún ambiente de desarrollo o utilizando el comando `javac` que viene incluido en los “kit” de desarrollo de Java como *JDK* (Java Development Kit) o *SDK* (Standard Development Kit). Por ejemplo, para compilar el archivo anterior, para compilar el archivo anterior se ejecutaría

```
javac ej.java
```

Esta compilación resultaría en el archivo “ej.class”.

### Ejecución

Durante la ejecución se obtiene el *bytecode*, guardado en los archivos “.class”, que puede ya estar en la plataforma actual o haber sido enviado por la red, como en el caso de un *browser*. El *bytecode* es cargado en la máquina virtual por el cargador de clases. A continuación este código es verificado por el verificador de *bytecode*, y dependiendo del hardware con que se cuenta, puede ser interpretado y ejecutado por el procesador virtual de la máquina virtual o traducido a código de un procesador de Java mediante el generador de código.

?? Existen dos maneras de ejecutar (y estructurar) un programa dependiendo de su ambiente de ejecución. En el caso de una aplicación “normal” (“standalone”, esta se ejecuta mediante el siguiente interpretador de Java, llamado simplemente `java`:

```
java ej2
```

?? En el caso de una aplicación que se ejecuta desde un *browser*, llamado un *applet*, el contenido de los archivos “.class” que están almacenados en el servidor, son transmitidos a través de la red y ejecutados en la máquina cliente (que puede ser la misma máquina que el servidor). Dado que un browser sólo comprende archivo “.html”, el applet debe ser relacionado con un archivo llamado, por ejemplo “ej.html”. Este archivo debe contener la siguiente línea:

```
<applet code=ej.class width=200 height=200></applet>
```

Dado que pueden haber múltiples archivos “.class”, sólo el principal es el que se incluye en la línea anterior. Otra forma adicional de ejecutar el applet es mediante el comando `appletviewer`, de la siguiente forma:

```
appletviewer ej.html
```

A lo largo del capítulo iremos describiendo con mayor detalle el desarrollo de programas en Java junto con ejemplos.

### 5.1.3 Paquetes

Java lleva a un nuevo nivel el concepto de bibliotecas o *paquetes*. Estos paquetes proveen una amplia funcionalidad para crear nuevas aplicaciones para Java. Además de servir como bibliotecas, definen un *API* (Application Program Interface) que permite al desarrollador extender las clases de estos paquetes para adaptarlos a las necesidades básicas de un programa. Java organiza estos paquetes en componentes jerárquicos a partir de dos directorios raíz principales. El primero es “java” siendo parte esencial de lo que actualmente se conoce como el API 1.1 de Java. Los paquetes de este API se muestran en la Tabla 5.1.

Paquete	Contenido
java.applet	Clases para implementar <i>applets</i> correspondientes a aplicaciones que corren dentro de los browsers.
java.awt	Clases para gráficas, componentes (GUI – Graphic User Interface) y administradores de control de ventanas, además de clases más especializadas como para procesamiento de imágenes (AWT – Advanced Window Toolkit).
java.beans	Clases e interfaces para construir <i>JavaBeans</i> correspondientes a GUIs independientes de plataformas.
java.io	Clases para control de entradas y salidas, tales como archivos y <i>streams</i> .
java.lang	Clases que componen el núcleo del lenguaje.
java.math	Clases para aritmética avanzada, incluyendo manejo de precisión numérica arbitraria.
java.net	Clases relacionadas con el manejo de redes, tales como datagramas y <i>sockets</i> .
java.rmi	Clases para el manejo de métodos remotos.
java.security	Clases para aspectos de seguridad, tales como criptografía.
java.sql	Clases para acceso a base de datos con el lenguaje <i>SQL</i> .
java.text	Clases para internacionalización del idioma, independiente del lenguaje particular.
java.util	Clases adicionales, tales como estructuras de datos avanzadas y compresión de datos.

Tabla 5.1 Paquetes básicos de Java.

En la actualidad se cuenta con el API 1.2 de Java, mejor conocido como Java2, el cual incluye además del paquete “java”, el paquete “javax” donde se encuentran componentes más avanzados, como se muestra en la Tabla 5.2.

Paquete	Contenido
javax.accessibility	Clases que definen contratos entre componentes de interfaces de usuario y una tecnología asistente que provee acceso a esos componentes.
javax.activation	Clases que definen activación de los componentes de <i>JavaBeans</i> .
javax.ejb	Clases para el manejo de <i>EJB (Enterprise JavaBeans)</i> .
javax.jms	Clases para el manejo de <i>JMS (Java Message Server)</i> .
javax.mail	Clases para el manejo de correo.
javax.naming	Clases para el acceso de los servicios de nombres.
javax.rmi	Clases para la invocación de métodos remotos incluyendo <i>CORBA</i> .
javax.servlet	Clases para el manejo de <i>servlets</i> y <i>JSP (Java Server Pages)</i> .
javax.sql	Clases para el acceso a base de datos con <i>SQL</i> .
javax.swing	Clases que proveen un conjunto de componentes para GUIs que trabajan en cualquier plataforma.
javax.transaction	Clases para el manejo de transacciones entre componentes.

Tabla 5.2 Paquetes extendidos de Java.

En Java, cada clase debe ser parte de un *paquete (package)*, y esta clase puede ser referida por su nombre completo “calificado”, el cual consiste de la jerarquía del paquete y el nombre de la clase, todos separados por puntos. Los propios nombres de paquetes generalmente están compuestos de múltiples componentes separados por puntos. Por ejemplo, la clase `PixelGrabber` que se encuentra en el paquete `java.awt.image` sería accesado mediante:

```
java.awt.image.PixelGrabber
```

Vale la pena notar que los paquetes se guardan en distintos directorios, donde el “.” realmente corresponde a “/” (“\” en la PC), donde se traduce, por ejemplo `java.awt.image` a `java/awt/image`. Por lo tanto la clase `PixelGrabber` estaría guardada dentro del directorio anterior.

Además de los paquetes mencionados en las Tablas 5.1 y 5.2 existe un número muy extenso de productos y productos adicionales desarrollados por Sun y por otras compañías como los paquetes para gráficas en 2 y 3 dimensiones que son también parte de Java y los paquetes para acceso a bases de datos de Oracle y Sybase.

## 5.2 Lenguaje

El lenguaje de Java contiene muchos aspectos que deben dominarse por un programador. Comenzaremos por los aspectos más básicos hasta llegar a lograr programas completos.

### 5.2.1 Aspectos Generales

Existen ciertos aspectos básicos del lenguaje que se requieren describir antes de poder proseguir con los tipos de datos y expresiones que se pueden aplicar para generar un programa completo.

#### Comentarios

El primer aspecto que debe conocerse en cualquier lenguaje es como distinguir entre código y comentarios. En Java existen tres tipos distintos para la especificación de comentarios, como se muestra en la Tabla 5.3.

Notación	Descripción
// línea comentada	Las dos diagonales indican el comienzo de un comentario que tiene efecto hasta el final de la línea.
/* párrafo comentado */	La diagonal seguida por el asterisco indica el inicio de un párrafo comentado. Para terminarse el comentario debe añadirse un asterisco seguido de otra diagonal. Estos comentarios no pueden anidarse uno dentro del otro.
/** párrafo comentado */	La diagonal seguida por dos asteriscos indica el inicio de un párrafo comentado. Para terminarse el comentario debe añadirse un asterisco seguido de otra diagonal. A diferencia del comentario anterior, este es un comentario documentable que puede ser extraído mediante el comando <i>javadoc</i> para producir un documento de documentación sencillo a partir del código fuente de Java. Estos comentarios no pueden anidarse uno dentro del otro.

Tabla 5.3 Notación para comentarios.

#### Caracteres

La especificación de caracteres en Java es distinta a la mayoría de los demás lenguajes. Java utiliza 16 bits en lugar de los más comunes 8 bits correspondientes al código ASCII para especificar caracteres. Este código de 16 bits es conocido en Java como *Unicode*, el cual mantiene compatibilidad con ASCII.

Existen actualmente 34,000 caracteres definidos en Unicode pero no todos pueden ser desplegados en todas las plataformas, por lo cual se utilizan secuencias especiales de *escape* con el siguiente formato: “\uxxxx“, donde xxxx representa una secuencia de uno a cuatro dígitos hexadecimales. Por ejemplo, el caracter nulo es “\u0000“. Además de este formato de especificación, también se apoya las secuencia especiales de escape, como en C, que son “\n“ y “\t“, y de manera más general “\xxx“, donde xxx representa un dígito octal.

#### Palabras reservadas

Todo lenguaje de programación cuenta con un número de palabras reservadas que tienen un significado especial y predefinido para Java. Tales palabras incluyen **if**, **else**, etc. En Java son 48 las palabras reservadas, las cuales el usuario no puede utilizar para sus propios usos.

#### Identificadores

Dentro de las palabras que el usuario puede definir se encuentran los *identificadores*. Los identificadores sirven para relacionarse con estructuras del programa. Por ejemplo, para poder definir una clase o acceder un objeto, se requiere definir un identificador. Identificadores que guardan o se refieren a valores de algún tipo son también conocidos como *variables*. Los nombres de los identificadores son cualquier palabra, con excepción de las reservadas por Java, que inician con cualquier letra del alfabeto o que comienzan con los siguientes símbolos: “\$” o “\_“. Estos identificadores pueden ser de cualquier tamaño. Por ejemplo, identificadores aceptables son: “ID”, “nombre”, “\_temp”, “\$dolar”, etc.

#### Oraciones

Toda *oración* en Java correspondiente a una línea de ejecución tiene como caracter final el “;“. Una notación relacionada el *bloque* que utiliza de llaves “{“ y “}” para especificar el inicio y fin de un grupo de oraciones.

#### Paquetes

Como mencionamos anteriormente, Java organiza sus bibliotecas alrededor del concepto de *paquetes* (“packages”). Dado el amplio número de paquetes que existen en el sistema de Java, además de aquellos que son generados por los propios desarrolladores, es necesario tener un manejo modular de manera que clases dentro de un paquete no tengan

conflictos con clases en otros paquetes, inclusive con clases que pudieran tener el mismo nombre. Por tal motivo, Java utiliza la expresión `package` que debe aparecer como primera instrucción en un archivo de código fuente en Java. Por ejemplo, un paquete con el nombre “ej” se incluiría como primera instrucción de todos los archivos que contengan clases de este paquete:

```
package ej;
```

De tal manera se especifica de qué paquete es componente la clase (y el archivo correspondiente). Las clases que son parte de un paquete particular tienen acceso a todas las demás clases del mismo paquete, algo que discutiremos con mayor detalle más adelante. Cuando un archivo es parte del un paquete, la clase compilada debe ubicarse de manera apropiada dentro de la jerarquía de directorios para poder ser accesada de manera correcta, como se mencionó anteriormente.

### Importar

Muy relacionada al concepto de paquetes es la instrucción `import`, la cual permite utilizar clases definidas en otros paquetes bajo nombres abreviados. Aunque siempre es posible utilizar otras clases por medio de sus nombres calificados completos, `import`, que no lee la clase ni la incluye, permite ahorrar escritura haciendo al código más legible. En otras palabras, sin la expresión de `import`, se puede utilizar la clase `PixelGrabber` siempre y cuando al utilizarla se le llame por su nombre calificado completo `java.awt.image.PixelGrabber`.

Existen tres formatos para `import`:

?? “`import package;`” que permite que el paquete especificado sea conocido por el nombre de su último componente. Por ejemplo, la siguiente expresión

```
import java.awt.image;
```

permite que la clase `java.awt.image.PixelGrabber` se le llame de la siguiente manera

```
image.PixelGrabber
```

?? “`import package.class;`” que permite que la clase especificada en el paquete sea conocida por su nombre de clase directamente. Por lo tanto, la expresión

```
import java.awt.image.PixelGrabber;
```

permite que la clase `java.awt.image.PixelGrabber` se le llame de la siguiente manera

```
PixelGrabber
```

?? “`import package.*;`” que permite que todas las clases en un paquete sean accesibles por medio del nombre de su clase directamente. Por ejemplo, la expresión

```
import java.awt.image.*;
```

permite que la clase `java.awt.image.PixelGrabber` y cualquier otras dentro de ese mismo paquete se le llame mediante su nombre directo, como

```
PixelGrabber
```

Se pueden incluir cualquier número de expresiones `import`, aunque todas deben aparecer después de la expresión inicial de `package` y antes de cualquier definición de clases o código en general. Si dos paquetes importados mediante esta forma contiene clases con el mismo nombre, es un error usar sus nombres ambiguos sin usar el nombre calificado completo.

### 5.2.2 Estructuras Básicas

Una vez mencionados los aspectos básicos del lenguaje de Java, proseguimos con la definición de las estructuras o tipos de datos que se pueden definir dentro de Java. Comenzamos con los tipos de datos primitivos.

#### Tipos Primitivos

En Java existen un número de tipos primitivos o predefinidos como parte del lenguaje. Estos tipos se muestran en la Tabla 5.4.

Tipo	Descripción	Valor Omisión
byte	Es un estructura de 8 bits.	0
char	Es una estructura en base a Unicode de 16 bits (sin signo).	\u0000
short	Es una estructura numérica de tipo entero de 16 bits.	0
int	Es una estructura numérica de tipo entero de 32 bits.	0
long	Es una estructura numérica de tipo entero de 64 bits.	0
float	Es una estructura numérica de tipo real de 32 bits.	0.0
double	Es una estructura numérica de tipo real de 64 bits.	0.0
boolean	Es una estructura de 1 bits, con valores true o false.	false

Tabla 5.4 Tipos primitivos predefinidos en Java.

Los tipos primitivos son estructuras que guardan un valor primitivo y que son manipulados a través de variables, declaradas de manera correspondiente, como veremos más adelante. Los valores asignados por omisión a variables de estos tipos se muestra en la última columna. Los nombres correspondientes a tipos primitivos comienzan con una letra minúscula. Existe un “sin-tipo”, llamado “void”.

Los primeros cinco tipos en la tabla `byte`, `char`, `short`, `int` y `long`, son conocidos como tipos integrales.

### Tipos No-Primitivos

Los tipos “no-primitivos” corresponden a clases los cuales son instanciadas en objetos. Los objetos son referenciados a través de variables que guardan referencias de estos. A diferencia de las variables correspondientes a tipos primitivos que guardan un “valor”, las variables correspondientes a tipos “no-primitivos” guardan una “referencia” al objeto instanciado. En Java no existe un número predeterminado de tipos “no-primitivos”, aunque si existen algunos predefinidos a través de las bibliotecas del lenguaje. Existe una clase muy particular que vale la pena mencionar. Esta clase es `Object`, que tiene como particularidad servir como la *superclase* a todas las demás clases del sistema, clases definidas en una biblioteca de Java o clases definidas directamente por el programador. Por ejemplo, una de estas clases ya definidas en Java es `String`. Esta clase es esencial en el manejo de cadenas.

### Variables

Para poder utilizar uno de estos tipos de datos primitivos es necesario primero definir alguna variable que guarde el valor del tipo correspondiente. Las variables se representan por medio de un nombre seleccionado dentro de los posibles identificadores. Por ejemplo, dos nombres de variables válidos serían, “x” y “y”. Habiendo definido el nombre de la variable se prosigue a *declararla* de acuerdo a algún tipo de dato. Como nota importante, todas las variables existen dentro de las clases y no pueden existir “sueltas” en un archivo como ocurre en C++.

### Declaraciones

Una declaración consiste en relacionar una variable con el tipo de dato que va a guardar. Por ejemplo si consideramos las dos variables, “x” y “y”, donde se desea que cada una de ellas guarde un valor entero, tendríamos que hacer la siguiente declaración:

```
int x,y;
```

Estas variables guardan inicialmente el valor de “0”, hasta que se les haga una *asignación* con un nuevo valor, algo que veremos más adelante.

Por ejemplo, una variable *valor* de tipo `boolean` se declararía de la siguiente forma:

```
boolean valor;
```

Las variables que hacen referencias a objetos de alguna clase se declaran de la misma forma. Si *obj* representa una variable de tipo `ClassX`, su declaración será la siguiente:

```
ClassX obj;
```

A diferencia de las variables de tipos primitivos que guardan valores, las variables de clases guardan únicamente referencias a los objetos instanciados de estas clases. Esto se hace por la sencilla razón de que los objetos son estructuras más complejas que los tipos primitivos por lo cual estos se guardan en memoria y las variables se refieren a esta ubicación. De tal manera, una variable de clase guarda por omisión una referencia vacía o nula que corresponde a `null` en Java. Vale la pena resaltar que esta referencia nula no equivale al “0” como ocurre en C. En particular, `null` es una palabra reservada que significa ausencia de referencia.

### Constantes

Como parte de las declaraciones de variables, Java permite agregar distintos tipos de modificadores, que afectan diversos aspectos de las variables. Existen por ejemplo modificadores para la visibilidad, correspondiente al manejo

del encapsulamiento, que serán mostrados más adelante cuando se describan objetos y clase. Sin embargo, vale la pena mencionar un modificador particular que hace que una variable se vuelva una constante. Esto se hace a través del modificador `final` como se muestra a continuación. Por ejemplo, la siguiente declaración haría que la variable `c` no pueda cambiar de valor.

```
final int c = 5;
```

Dado que la variable no puede cambiar de valor, es necesaria inicializarla con algún valor en el momento de su declaración. Sería un error tratar de cambiar luego este valor.

### Arreglos

El arreglo es una estructura presente en la gran mayoría de los lenguajes por lo cual tampoco falta en Java. Aunque se utiliza una notación similar a los demás lenguajes, su manejo es un poco diferente. Esto último se debe a que los arreglos se manipulan por referencia, al igual que los objetos. La declaración básica de un arreglo es de la siguiente forma:

```
int numeros[];
```

Esta declaración específica que la variable `numeros` hará referencia a un arreglo de números enteros. Estos números serán luego accesados mediante la siguiente notación, donde `i` representa el elemento del arreglo:

```
numeros[i];
```

Para que esto funcione correctamente, es necesario inicializar la variable `numeros` para que se refiera a algún arreglo, ya que hasta ahora sólo hemos hecho una declaración. Existen dos formatos diferentes para hacer esta inicialización. El primer formato es al similar al de C:

```
int numeros[] = {1, 2, 4, 8, 16, 32, 64, 128}
```

Esta declaración crea un arreglo de 8 elementos inicializando sus elementos a los valores especificados. La variable `numeros` guarda la referencia a dicho arreglo.

La segunda manera de inicializar un arreglo es mediante la palabra `new`, que como veremos más adelante se utiliza para instanciar cualquier tipo de objetos. La inicialización de un arreglo mediante la palabra `new` es más común y especifica el tamaño del arreglo, como se muestra a continuación.

```
int numeros[] = new int[50];
```

Por ejemplo, esta declaración incluye la inicialización de `numeros` como referencia a un arreglo de 50 enteros donde cada uno es inicializado con el valor de "0".

Un aspecto importante con los arreglos es que se puede conocer su largo utilizando el modificador `length`, como se muestra a continuación:

```
numeros.length
```

Esto permite obtener el largo definido para el arreglo.

Los arreglos no tienen que ser de una sola dimensión, ya que Java apoya múltiples dimensiones. Estos arreglos son implementados como "arreglos de arreglos". Por ejemplo, un arreglo de dos dimensiones de enteros se declara e inicializa de la siguiente manera:

```
int numeros2[][] = new int[10][20];
```

Esta declaración genera un arreglo con 200 elementos inicializados todos a "0". Como mencionamos antes, el manejo de Java implica que existen 10 arreglos cada uno de ellos refiriéndose a un arreglo de una dimensión de 20 elementos.

Cuando se asigna un arreglo multidimensional, no se tiene que especificar el número de elementos que se contiene en cada dimensión. Por ejemplo, la siguiente es una declaración válida:

```
int numero3[][][] = new int[10][][];
```

Esta declaración asigna un arreglo que contiene 10 arreglos de una dimensión, donde cada uno se refiere a un arreglo de tipo `int[][]`. La regla básica en Java es que las primeras  $n - 1$  dimensiones, con  $n \geq 1$ , deben especificar el número de elementos.

Nótese que en el caso de funciones (que veremos más adelante en la sección de clases), la declaración de un arreglo como argumento de la función es similar al manejo descrito aquí. Sin embargo, existen dos formatos aceptados:

?? Se puede utilizar la notación similar a C:

```
void func(char buf[]) {
    char s[] = new char [50]; ... }
```

?? Pero también se puede utilizar la siguiente notación:

```
void func(char[] buf) {
    char[] s = new char [50]; ... }
```

En el caso de arreglos de objetos, el manejo de arreglos es bastante similar al de los tipos primitivos. La consideración principal es que crear un arreglo de objetos no crea todos los objetos que se guardan en el arreglo.



Esto se hace principalmente para ofrecer mayor flexibilidad en la creación del arreglo, como permitir al programador hacer las llamadas deseadas a los constructores. Por lo tanto, si se quiere crear, por ejemplo un arreglo de 20 objetos de tipo `Persona`:

```
Persona p[] = new Persona[20];
for (int i = 0; i < i.length; i++)
    p = new Persona();
```

Nótese que la primera declaración e instanciación del arreglo genera el arreglo de 20 elementos de tipo `persona`, aunque cada elemento está vacío. Dentro del ciclo “for” se instancia cada elemento, en este caso con los valores de omisión de la clase. Algunos de los detalles utilizados en este ejemplo quedarán más claros más adelante en el capítulo.

### Cadenas

Como en la mayoría de los lenguajes, se pueden definir cadenas como arreglos de caracteres. En otras palabras se puede definir una cadena de la siguiente manera:

```
char s[] = "Prueba";
```

La problemática con esta manera de definir cadenas es que su manipulación se vuelve complicada, por ejemplo comparar o concatenar cadenas. Para lograr un mejor manejo, Java define la cadena como una clase con todos los beneficios que esto significa. Aunque el tema de clases y objetos será tratado más adelante, vale la pena mencionar algunos aspectos del manejo de las cadenas en Java.

Los objetos tipo cadenas son instancias de la clase `java.lang.String`. Por ejemplo, una cadena se puede instanciar cadenas de la siguiente forma:

```
String s = "Prueba";
```

Un aspecto que resalta en Java en relación al manejo de cadenas es el operador “+” que corresponde a una concatenación. Por ejemplo, la siguiente expresión es válida y resultaría en las dos cadenas concatenadas:

```
String s1 = "Hola";
String s2 = "Mundo";
String s3 = s1 + s2;
```

Esta operación tiene efectos profundos para el compilador. Por ejemplo, la siguiente función de impresión en C es muy problemática para el compilador ya que involucra un número de argumentos variables (al igual que muchas otras funciones similares):

```
printf( "%s%s", s1, s2 );
```

Esto se resuelve en Java mediante el operador de concatenación:

```
print(s1 + s2);
```

(En realidad la función `print` requiere del prefijo `System.out` para accederse correctamente. Este prefijo será explicado más adelante.) Este es sólo un ejemplo de las facilidades para el manejo de cadenas en Java (al igual que de muchos otros aspectos). La única restricción importante en los objetos de tipo `String` es que estos son inmutables, no pueden cambiarse una vez asignados. Por ejemplo, para poder manejar cadenas modificables se debe instanciar objetos de la clase `StringBuffer` a partir de un objeto `String`, algo que va más allá del alcance introductorio de este capítulo.

Existen ciertas funciones que se pueden aplicar a las cadenas para manipularlas, como son: `length()`, `charAt()`, `equals()`, `compareTo()`, `indexOf()`, `lastIndexOf()`, `substring()`. Estas tipo de funciones hacen de Java un lenguaje muy poderoso.

### Expresiones Básicas

Una vez que se haya declarado variables y posiblemente asignado un valor inicial, estas variables pueden ser manipuladas mediante diversos tipos de expresiones. En esta sección describiremos las expresiones más importantes que pueden ser aplicadas a variables de tipos primitivos. Más adelante describiremos las expresiones que se pueden aplicar a objetos.

#### Asignación

Las variables de tipos primitivos son utilizadas para guardar valores de sus respectivos tipos. Por lo tanto, la expresión más importante que hay es la de asignación de valores. Por ejemplo, para asignarle un valor a una variable `x` de tipo entero se haría lo siguiente:

```
x = 10;
```

Obviamente tuvo que haberse hecho antes la declaración correspondiente. Incluso puede hacerse la asignación al mismo momento de la declaración.

```
int x = 10;
```

La asignación es similar para todos los demás tipos primitivos.

### Operadores

Java apoya todos los operadores estándares de C, con la misma precedencia. La Tabla 5.5 muestra todos los operadores que se pueden aplicar a tipos primitivos (incluyendo aritméticos, integrales y booleanos) en orden de precedencia.

Operador	Tipo de Operando(s)	Descripción de la Operación
++, --	aritmético	incremento, decremento (unario) (pre o post)
+, -	aritmético	más, menos (unario)
?	integral	complemento de bit (unario)
!	booleano	complemento lógico (unario)
(tipo)	cualquiera	“cast”
*, /, %	aritmético	multiplicación, división, resto
+, -	aritmético	suma, resta
+	cadena	concatenación de cadenas
<<	integral	desplazamiento hacia la izquierda
>>	integral	desplazamiento hacia la derecha con signo
>>>	integral	desplazamiento hacia la derecha con extensión de cero
<, <=	aritmético	menor que, menor o igual que
>, >=	aritmético	mayor que, mayor o igual que
==	primitivo	igual (tienen valores idénticos)
!=	primitivo	no igual (tienen valores diferentes)
&	integral	AND para bits
&	booleano	AND booleano
^	integral	XOR para bits
^	booleano	XOR booleano
	integral	OR para bits
	booleano	OR booleano
&&	booleano	AND condicional
	booleano	OR condicional
?:	boolean, cualquiera, cualquiera	operador condicional (ternario)
=	variable, cualquiera	asignación
*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=,  =	variable, cualquiera	asignación con operación

Tabla 5.5 Operadores para tipos primitivos predefinidos en Java.

Por ejemplo, la siguiente es una expresión de multiplicación,

```
int x;
x = 23*54;
```

Aunque no es el objetivo mostrar ejemplos de todos los posibles operadores, vale la pena resaltar algunas diferencias de Java con C y C++:

?? Java no apoya los operadores de apuntadores “\*”, “&” o `sizeof`.

?? Java no considera a “.” (acceso a campo) y “[ ]” (acceso a arreglo) como operadores.

?? Java no apoya la sobrecarga de operadores.

Como comentario adicional sobre los operadores en Java, dado que todos los tipos en Java son valores con signo, el operador >> se define como un corrimiento a la derecha con extensión de signo, mientras que el operador >>> trata el valor de corrimiento lógico como un valor sin signo y lo corre a la derecha con extensión de cero.

### Control

Aparte de los operadores, existen un número de expresiones de control que son utilizadas por los lenguajes para controlar el flujo de la lógica del programa. Estas expresiones son bastante estandarizadas en los lenguajes modernos aunque varían en ciertos detalles. La tabla 5.6 muestra las expresiones de control.

Expresión	Descripción de la Expresión
if (condición-1) bloque-1 else if (condición-i) bloque-i else bloque-n	Si condición-1 es verdadera se ejecuta el bloque-1. De lo contrario se prosigue con condición-i de manera similar para ejecutar el bloque-i. Puede haber un número infinito de estas condiciones. Si ninguna condición es verdadera se ejecuta el bloque-n.
while (condición) bloque do bloque while (condición)	Mientras condición sea verdadera se ejecuta el bloque. A diferencia de la expresión anterior, primero se ejecuta el bloque y luego se revisa la condición para la siguiente ejecución.
switch (variable) case valor-i: bloque-i default: bloque-n	Se verifica el valor de la variable (tipo integral). Se compara a los valores especificados para los diversos casos, <i>valor-i</i> , hasta encontrar uno igual. En ese momento se ejecuta <i>bloque-i</i> . Si no se encuentra ninguno igual, se ejecuta <i>bloque-n</i> .
for (expr-1; condición-2; expr-3) bloque	Se ejecuta <i>expr-1</i> al inicio de esta expresión. Si <i>condición-2</i> es verdadera se ejecuta el <i>bloque</i> . A continuación se ejecuta <i>expr-3</i> y se prueba <i>condición-2</i> . Si esta es verdadera nuevamente se ejecuta el <i>bloque</i> . Se sigue ejecutando el <i>bloque</i> , precedido de la ejecución de <i>expr-3</i> , mientras <i>condición-2</i> sea verdadera.
break label;	Esta expresión permite interrumpir el flujo de una estructura de control con la opción de saltar fuera de la sección etiquetada por “ <i>label:</i> ”, o en su ausencia, saltar fuera de la estructura de control actual.
continue label;	Esta expresión permite interrumpir el ciclo actual de una estructura de control con la opción de saltar a la última línea de la sección etiquetada por “ <i>label:</i> ”, o en su ausencia, saltar a la última línea de la estructura de control actual.
label: expr	Etiqueta asignada a una expresión, utilizada en conjunción con <i>break</i> y <i>continue</i> .
return expr;	Esta expresión devuelve el valor generado por <i>expr</i> .

Tabla 5.6 Expresiones de control en Java.

Si los bloques en la tabla involucran más de una oración, estos deben incluir llaves para especificar el inicio y fin del bloque. Todas las condiciones deben ser expresiones que devuelvan un valor booleano. Nuevamente, hacemos énfasis en que un valor numérico no puede utilizarse como uno booleano en Java, ni siquiera haciendo un “cast”. Los valores *false* y “0” no son equivalentes.

En el caso de la expresión de *for*, en Java no se permite el uso de la coma para separar múltiples expresiones dentro de la condición, aunque si es permitido dentro de las dos secciones, *expr-1* y *expr-3*. El resto de las expresiones, con excepción de *label*, son similares en su uso a las de C.

### 5.2.3 Objetos y Clases

Todo programa de Java debe incluir clases. Consideremos los diversos aspectos de las clases como se describió inicialmente en el Capítulo 4. Utilizando la notación UML, una clase se representa como se muestra en la Figura 5.2.

NombreClase

Figura 5.2 Notación de UML para una clase.

En Java, el código correspondiente a una clase se muestra continuación:

```
class NombreClase {
}
```

Nótese que el nombre de la clase siempre comienza con mayúscula. Si el nombre es compuesto, como en este caso, para facilitar su lectura, la siguiente palabra debe iniciarse también con mayúsculas. No deben haber espacios dentro del nombre de la clase.

La anterior es probablemente la definición más sencilla que puede asignarse a una clase. La primera palabra `class`, sirve de prefijo para indicar el inicio de una clase.

Por ejemplo, consideremos la clase `Persona` como se muestra en la Figura 5.3.

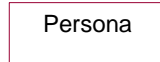


Figura 5.3 Notación de UML para una clase llamada `Persona`.

La clase `Persona` se define de la siguiente manera.

```
class Persona {
}
```

En las siguientes secciones mostramos de manera incremental el resto de las definiciones relacionadas con la clase.

### Atributos

El siguiente paso en la definición de una clase es indicar sus atributos, estos se muestran nuevamente en la Figura 5.4.

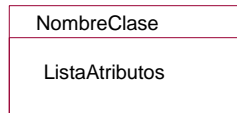


Figura 5.4 Notación de UML para una clase con atributos.

En Java, el código correspondiente a una clase con atributos se muestra continuación:

```
class NombreClase {
// atributos
    tipoAtributo1 nombreAtributo1;
    ...
    tipoAtributoi nombreAtributoi;
    ...
    tipoAtributoN nombreAtributoN;
}
```

La lista de atributos corresponde a declaraciones de tipos primitivos, compuestos de un tipo, *tipoAtributoi*, seguido de un nombre, *nombreAtributoi*, (los “...” son únicamente para resaltar que es una lista de atributos, y la línea “// atributos” representa un comentario únicamente). Nótese que los atributos comienzan siempre con una letra minúscula, aunque las siguientes palabras en el caso de nombres compuestos, pueden comenzar con mayúsculas. Como con los nombres de clases, no deben haber espacios dentro del nombre y en especial no deben haber nombres repetidos.

Por ejemplo, consideremos la clase `Persona` con varios atributos como se muestra en la Figura 5.5.

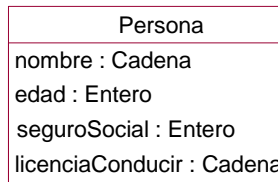


Figura 5.5 Notación de UML para una clase llamada `Persona`, que contiene atributos.

La clase `Persona` y sus atributos se definen de la siguiente manera.

```
class Persona {
// atributos
    String nombre;
    int edad;
    int seguroSocial;
    String licenciaConducir;
}
```

El orden de los atributos no tiene ninguna importancia dentro de la clase. Nótese que los tipos de los atributos no necesariamente tienen que ser tipos primitivos, como es el caso de `String`.

### Operaciones

El siguiente paso en la definición de una clase es indicar sus operaciones, estos, junto con los atributos se muestran en la Figura 5.6.

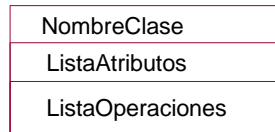


Figura 5.6 Notación de UML para una clase con atributos y operaciones.

En Java, el código correspondiente a una clase con atributos se muestra continuación:

```
class NombreClase {
  // atributos
  tipoAtributo1 nombreAtributo1;
  ...
  tipoAtributoi nombreAtributoi;
  ...
  tipoAtributoN nombreAtributoN;
  // operaciones
  tipoRetorno1 nombreMétodo1 ( listaParámetrosMétodo1 )
    { cuerpoMétodo1 }
  ...
  tipoRetornoj nombreMétodoj ( listaParámetrosMétodoj )
    { cuerpoMétodoj }
  ...
  tipoRetornoM nombreMétodoM ( listaParámetrosMétodoM )
    { cuerpoMétodoM }
}
```

Aunque conceptualmente se habla de operaciones, en los lenguajes de programación es más preciso hablar de métodos. La relación entre estos dos términos es que múltiples métodos pueden corresponder a una misma operación. La lista de métodos anterior está compuesta por el tipo de valor de retorno, *tipoRetornoj*, el nombre del método, *nombreMétodoj*, los parámetros que recibe el método, *listaParámetrosj*, y finalmente el cuerpo del método, *nombreCuerpoj*. (Nuevamente, los “...” son únicamente para resaltar que es una lista de métodos.) Nótese que los nombres de los métodos comienzan siempre con una letra minúscula, aunque las siguientes palabras en el caso de nombres compuestos, pueden comenzar con mayúsculas. Como con los nombres de clases y atributos, no deben haber espacios dentro del nombre. En particular, *listaParámetros*, tiene el siguiente formato:

```
tipoRetorno nombreMétodo ( tipo1 par1, tipo2 par2, ..., tipoN parN )
  { cuerpoMétodo }
```

Por otro lado, *cuerpoMétodo*, es una lista de expresiones similares a las descritos en la sección correspondiente además de llamadas a otros métodos.

A diferencia de los atributos, pueden haber nombres repetidos para los métodos. A esto se le conoce como *sobrecarga* de métodos.

Por ejemplo, consideremos la clase `Persona` con varios métodos, además de los atributos anteriores, como se muestra en la Figura 5.7.

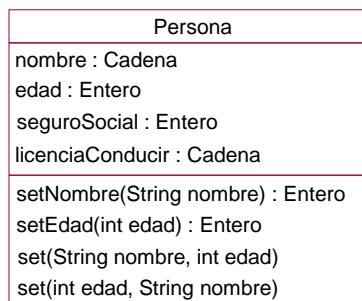


Figura 5.7 Notación de UML para una clase `Persona` que contiene atributos y métodos.

La clase `Persona`, con sus atributos y métodos, se define de la siguiente manera.

```
class Persona {
  String nombre;
  int edad;
  int seguroSocial;
  String licenciaConducir;
```

```

    int setNombre(String nom) {
        nombre = nom; return 1; }
    int setEdad(int ed) {
        edad = ed; return 1; }
    void set(String nom, int ed) {
        setNombre(nom); setEdad(ed); }
    void set(int ed, String nom) {
        setNombre(nom); setEdad(ed); }
}

```

El orden de los métodos no tiene ninguna importancia dentro de la clase. Nótese que para evitar posibles conflictos, el nombre de un parámetro debe ser diferente del de un atributo. Si los dos nombres fueran iguales, la variable a ser utilizada se resuelve según su alcance (“scope”). (Se utiliza la variable cuya definición sea la más cercana a su lugar de utilización, en este caso el parámetro del método tendría precedencia.) Otro aspecto a notar son el uso del “return” en el caso de métodos que devuelven algún tipo que no sea “void”. Adicionalmente, los últimos dos métodos tienen nombre similar, por lo cual realmente corresponden a una misma operación que es “asignar el valor al nombre y edad” sin importar el orden de los parámetros. Este uso de la sobrescritura de métodos es muy común. Por último vale la pena resaltar el manejo de parámetros. Todos los parámetros relacionados a tipos primitivos son pasados “por valor”. Esto significa que si el valor del parámetro es cambiado dentro del método, esto no afectaría de ninguna manera su valor original. Por ejemplo, consideremos la siguiente versión de los métodos anteriores:

```

    int setEdad(int ed) {
        edad = ed; ed = 0; return 1; }
    void set(String nom, int ed) {
        setEdad(ed); setEdad(ed); }

```

En el primer método, se asigna el valor de “ed” a “edad” y luego se asigna “0” a “ed”. En el segundo método, se llama dos veces al método “setEdad”. Dado que “ed” fue pasado “por valor”, el “0” nunca fue devuelto al llamado original y el segundo “setEdad” vuelve a asignar la “edad” correcta.

Sin embargo, este no es el caso con los objetos, ya que estos son pasados “por referencia”. En otras palabras, aunque las variables no sean globales, los objetos a los que las variables se refieren sí lo son, como es el caso de un objeto de tipo `String`. Consideremos ahora la siguiente modificación a los métodos originales:

```

    int setNombre(String nom) {
        nombre = nom; nom = null; return 1; }
    void set(String nom, int ed) {
        setNombre(nom); setNombre(nom); }

```

En el primer método, “setNombre”, se asigna la referencia que guarda la variable “nom” a “nombre”. A continuación se asigna el valor “null” a “nom”, o sea una referencia nula. En el segundo método, “set”, existen dos llamadas al primer método, “setNombre”. La primera llamada asigna el valor original del parámetro “nom”. En la segunda llamada a la función “setNombre”, se vuelve a enviar la referencia guardada por “nom”, aunque se debe considerar si su valor ya ha cambiado. Dado que “nom” fue pasado “por referencia”, el “null” fue reasignado a la variable de “nom” dentro del método “set”. Por lo tanto la segunda llamada a la función “setNombre” asigna un “nom” nulo, proveniente de esta reasignación, a la variable “nombre” dentro del método “setNombre”.

## Encapsulamiento

En Java, como en la mayoría de los lenguajes orientados a objetos, es muy importante considerar el encapsulamiento de los atributos y métodos definidos en la clase. Aunque todos los campos de una clase son accesibles dentro de esa clase.

Para ello, Java define tres modificadores básicos para el manejo del encapsulamiento y que pueden ser aplicados a los campos o miembros (atributos y métodos) de una clase y a la propia clase completa: `public`, `private` y `protected`, como se muestra a continuación:

?? `public` - se agrega a los campos de la clase que pueden ser accedidos fuera de la clase. En general, deben ser públicos los métodos de la clase, aunque no necesariamente todos sus métodos.

?? `private` - se agrega a los campos de la clase que son accedidos únicamente desde dentro de la clase, o sea, dentro de sus propios métodos. En general, deben ser privados los atributos de la clase, y posiblemente algunos métodos de uso interno.

?? `protected` - se agrega a los campos de la clase que son accedidos únicamente desde dentro de la clase o dentro de una subclase que hereda de la actual, o sea, dentro de sus propios métodos o métodos de alguna de sus

subclase. En general, deben ser protegidos los atributos de la clase, y posiblemente algunos métodos de uso interno.

La distinción entre estos modificadores de encapsulamiento puede volverse un poco confusa dado que además de afectar el encapsulamiento de los campos entre clases, también afecta la el acceso dependiendo si las clase son, o no, campos del mismo paquete. Por lo tanto, Java define dos maneras generales de aplicar estos modificadores, como se muestra a continuación:

?? *Modificador de encapsulamiento para campo de una clase* – se aplica únicamente a un atributo o método de una clase y puede consistir de cualquiera de los tres modificadores: `public`, `private` y `protected`. Este modificador se añade al inicio de una declaración, sea atributo o método, como se muestra a continuación:

```
class Persona {
    private String nombre;
    protected int edad;
    public int seguroSocial;
    public String licenciaConducir;

    private int setNombre(String nom) {
        nombre = nom; return 1; }
    protected int setEdad(int ed) {
        edad = ed; return 1; }
    public void set(String nom, int ed) {
        setNombre(nom); setEdad(ed); }
    public void set(int ed, String nom) {
        setNombre(nom); setEdad(ed); }
}
```

?? *Modificador de encapsulamiento para una clase* – se aplica a toda la clase como tal y puede consistir únicamente del modificador: `public` y afecta la visibilidad de la clase entre paquetes. Este modificador se añade al inicio de la especificación de la clase, como se muestra a continuación:

```
public class Persona {
    private String nombre;
    protected int edad;
    public int seguroSocial;
    public String licenciaConducir;

    private int setNombre(String nom) {
        nombre = nom; return 1; }
    protected int setEdad(int ed) {
        edad = ed; return 1; }
    public void set(String nom, int ed) {
        setNombre(nom); setEdad(ed); }
    public void set(int ed, String nom) {
        setNombre(nom); setEdad(ed); }
}
```

En general, una vez la clase es pública en otra paquete, entra en rigor la visibilidad de sus campos. La tabla 5.7 muestra los diferentes efectos de estos modificadores dependiendo de cuatro formas de acceder campos de una clase: dentro de la misma clase, dentro de una clase en el mismo paquete, dentro de una subclase en otro paquete, o dentro de una clase en otro paquete pero que no es una subclase de la actual. Se consideran cuatro niveles de encapsulamiento: `public`, `protected` y `private` para campos de clase y paquete, correspondiente a la visibilidad existente si se omite el modificador de sus campos. Esto último resalta que no es obligatorio utilizar los modificadores de encapsulamiento. Sin embargo, su efecto no corresponde a ninguno de los tres casos como a menudo ocurre con otros lenguajes. Esto es debido principalmente a la existencia de paquetes.

Es accesible por:	Encapsulamiento			
	public	protected	paquete	private
Misma clase	sí	sí	sí	sí
Clase en el mismo paquete	sí	sí	sí	no
Subclase en un paquete diferente	sí	sí	no	no
No-subclase en un paquete diferente	sí	no	no	no

Tabla 5.7 Modificadores de encapsulamiento y su efecto sobre las diversas estructuras.

La explicación de la Tabla 5.7 es la siguiente:

- ?? Todos los campos o miembros de una clase son siempre accesibles dentro de una misma clase, sin importar el modificador de sus campos.
- ?? Todos los campos de una clase son siempre accesibles por cualquier otra clase, incluyendo subclases, dentro del mismo paquete, siempre y cuando el campo no sea `private`.
- ?? Una subclase en un paquete distinto, sólo puede acceder campos `public` o `protected`. Nótese que la clase debe ser `public` para poder ser vista en otro paquete.
- ?? Una clase, que no sea subclase, en un paquete distinto, sólo puede acceder campos `public`. Nuevamente la clase debe ser `public` para poder ser vista en otro paquete.

### Constructores

Para completar los aspectos fundamentales de una clase se debe considerar sus *constructores*. Estos son métodos especiales que pueden ser llamados únicamente durante la instanciación de un nuevo objeto (esto último lo veremos en la siguiente sección.) El constructor lleva el mismo nombre de la clase y puede incluir parámetros. A diferencia del resto de los métodos, un constructor no especifica ningún tipo de retorno, ya que de por sí, el objeto recién creado es lo que se devuelve. Su formato es como se muestra a continuación:

```
class NombreClase {
    // atributos
    ...listaAtributos...
    // constructor
    NombreClase ( listaParámetrosConstructor1 )
        { cuerpoConstructor1 }
    ...
    NombreClase ( listaParámetrosConstructori )
        { cuerpoConstructori }
    ...
    NombreClase ( listaParámetrosConstructorN )
        { cuerpoConstructorN }
    // operaciones
    ...listaMétodos...
```

Pueden existir múltiples constructores, donde todos deben tener el mismo nombre que es idéntico al nombre de la clase. Este es otro ejemplo de sobrecarga, en este caso del constructor de la clase. Como con los métodos, no puede existir dos constructores con una lista de parámetros exactamente iguales. Como los métodos, la lista de parámetros puede estar vacía. El constructor no es obligatorio en Java, ya que por omisión se generaría uno con lista de parámetros vacía y un cuerpo vacío. A continuación se muestra un ejemplo del uso de los constructores:

```
class Persona {
    private String nombre;
    private int edad;
    private int seguroSocial;
    private String licenciaConducir;

    public Persona(String nom, int ed, int seg, String lic) {
        set(nom, ed); seguroSocial = seg; licenciaConducir = lic; }

    public int setNombre(String nom) {
        nombre = nom; return 1; }
    public int setEdad(int ed) {
        edad = ed; return 1; }
    public void set(String nom, int ed) {
        setNombre(nom); setEdad(ed); }
    public void set(int ed, String nom) {
        setNombre(nom); setEdad(ed); }
}
```

Nótese que cambiamos los modificadores de encapsulamiento de los atributos y métodos para volverlos privados y públicos, respectivamente. Nótese además que los constructores también aceptan los modificadores de encapsulamiento de manera similar a los métodos. Un constructor `private` nunca puede ser llamado (un ejemplo



de una clase que nunca podrá ser instanciada) y un constructor `protected` sólo puede ser instanciado por una subclase. En el ejemplo anterior, el constructor acepta valores de inicialización para todos los atributos de la clase.

A diferencia de los lenguajes como C++, Java no requiere un destructor, aunque si existen la función especial `finalize` que permite manejo avanzado de recolección de basura.

### Instanciación

Una vez definidos los aspectos esenciales de la clase, el siguiente paso es poder instanciar objetos. Java, para crear un nuevo objeto, se debe utilizar el operador `new` seguido por la clase a la que pertenece el objeto. Además de la clase, puede haber una lista de argumentos opcionales entre paréntesis, que son asignados y deben corresponder a la lista de parámetros de algún constructor de la clase. Si la clase no tiene un constructor, la lista deberá estar vacía. Debe quedar muy claro que este operador permite instanciar un nuevo objeto, pero si no existe una variable que guarde la referencia al nuevo objeto, el objeto prácticamente será perdido por ser imposible su acceso. Por lo tanto, antes de proceder con la instanciación debe declararse una variable que guarde la referencia al nuevo objeto, como se muestra a continuación:

```
Persona p1 = new Persona("Juan", 35, 1234567, "x254f");
```

Esta instanciación asigna los valores especificados a cada una de los atributos.

En el caso de no haber especificado ningún constructor, Java permite instanciar un nuevo objeto utilizando la llamada a un constructor vacío generado implícitamente, como se muestra a continuación:

```
Persona p2 = new Persona();
```

Esta instanciación crea un objeto tipo `Persona` donde los atributos toman como valor aquellos asignados por Java por omisión. Esta generación implícita por parte de Java no ocurre si ya se ha definido al menos otro constructor para esa clase. En este último caso, de no haber un constructor que no tome argumentos, la llamada anterior hubiese ocasionado un error.

Como nota adicional, si se utilizara una sola variable para guardar la referencia a ambos objetos, la referencia del primera se perdería ya que la variable siempre guarda el último valor asignado. Más aún, no es esencial que una variable explícitamente guarde la referencia a un objeto. Siempre cuando esa referencia esté guardada y sea accesible, por ejemplo dentro de alguna lista o como parámetro de un método, el objeto no será eliminado por el *recolector de basura*. En cambio, si no existe ninguna forma de acceder al objeto, este será automáticamente borrado.

### Acceso a Campos

Una vez instanciado un objeto, lo más importante es poder acceder los campos de la clase los cuales ya han sido definidos previamente.

#### Atributos

Los atributos (tipos primitivos) son manipulados como se explicó en la sección correspondiente. Estas manipulaciones se hacen mediante expresiones tales como asignación o flujos de control. Lo particular a considerar en los objetos es el acceso mediante la variable que guarda la referencia al objeto seguido por un “.” y el nombre del atributo. Por ejemplo, para asignar el número del `seguroSocial` del objeto `p1` al valor `8888888`, donde el objeto es de tipo `Persona` se hace lo siguiente:

```
p1.seguroSocial = 8888888;
```

Nótese que esta asignación puede ser hecha si el atributo no es privado, ya que de lo contrario el encapsulamiento no lo permitiría. Si esto ocurriese, que debiera ser la norma, el acceso se haría a través de un acceso a un método no privado del objeto, como veremos a continuación.

#### Métodos

Los métodos son llamados a partir de la variable que guarda la referencia al objeto seguido por un “.” y el nombre del método. Por ejemplo, para copiar el número del `edad` del objeto `p1` al valor `25`, donde el objeto es de tipo `Persona` se hace lo siguiente:

```
p1.setEdad(25);
```

Nótese que esta llamada al método para la asignación puede ser hecha si el método no es privado, ya que de lo contrario el encapsulamiento no lo permitiría.

### Referencia Propia

Existe una palabra reservada, `this`, que es muy importante para referirse al objeto actual. Se utiliza de dos maneras distintas: como referencia a algún campo del objeto o cómo llamada a un constructor. Veamos los dos casos:

?? Como ejemplo de referencia a un campo de un objeto, el atributo `edad` puede ser accesado dentro del método `setEdad` mediante el siguiente código:

```
protected int setEdad(int edad) {
    this.edad = edad; return 1; }
```

Nótese como el `this` no cambia en absoluto la lógica original del método, aunque evita un posible conflicto cuando un argumento del método tiene el mismo nombre que el atributo de la clase. Obviamente, este posible conflicto se resuelve utilizando nombres diferentes. (Para Java esto no es un conflicto, más bien pudiera ser base de confusión para el programador.) Sin embargo, el uso más importante para esta referencia es como referencia a ser devuelta por un método. Por ejemplo, modifiquemos el método anterior, en particular el tipo devuelto, de la siguiente manera:

```
protected Persona setEdad(int ed) {
    edad = ed; return this; }
```

En este ejemplo, el `this` juega un papel primordial ya que permite que el método devuelva la referencia del propio objeto para ser usado por otras variables. Obviamente, la referencia debe ser de tipo `Persona` para que el código sea correcto.

?? Como ejemplo de llamada a un constructor de un objeto, consideremos el siguiente constructor adicional para la clase `Persona`:

```
public Persona() {
    this(null, 0, 0, null); }
```

Este segundo constructor no tiene argumentos y se aprovecha del primer constructor para redirigir la llamada utilizando "`this()`", en este caso con parámetros de omisión. En este ejemplo particular no es necesario hacer esta última llamada por asignar valores similares a los que asigna Java por omisión a las variables, sin embargo, esta es una buena práctica. Por otro lado, sería necesario incluir esta llamada si quisiéramos asignar valores de omisión diferentes a los que asigna Java. La llamada a "`this()`" debe utilizarse dentro de un constructor y debe aparecer como su primera línea.

### Expresiones para Objetos

Las expresiones que se aplican a los objetos son más bien limitadas en el sentido de que las manipulaciones principales en el control de un programa son sobre los tipos primitivos. Los objetos como tales se usan más bien como encapsuladores de estos tipos primitivos y no tanto como la base de expresiones. Sin embargo existe algunos operadores que vale la pena mencionar aquí y que se muestra en la Tabla 5.8.

Operador	Descripción de la Operación
(tipo)	"cast"
instanceof	comparación de tipo
==	igual (se refieren al mismo objeto)
!=	no igual (se refieren a distintos objetos)

Tabla 5.8 Operadores que pueden ser aplicados a objetos.

Nótese la lista reducida de operadores en relación a lista extensa de operadores aplicables a tipos primitivos. De esta lista vale la pena resaltar los siguientes aspectos:

?? Los operadores "`==`" y "`!=`" comparan referencias. Para comparar los propios valores a donde apuntan las referencias, se debe utilizar el método "`equals()`".

?? El operador "`instanceof`" devuelve `true` si el objeto a la izquierda es una instancia de la clase especificada a la derecha, de lo contrario devuelve `false`. Tiene la misma precedencia que los operadores de comparación.

Además de esto, los objetos como tales son utilizados muy comúnmente en expresiones que involucran funciones, donde las referencias a los objetos son sus argumentos.

### 5.2.4 Ligas, Asociaciones y Composición

Hasta ahora hemos mostrado como se definen las clases y como se crean los objetos. Para poder generar una aplicación completa es necesario poder relacionar clases, o sea objetos, entre si. Esto corresponde a los conceptos de ligas y asociaciones entre objetos y clases, respectivamente. En la gran mayoría de los lenguajes orientados a objetos no existe ninguno de estos dos conceptos. Por lo tanto estos deben ser implementados por algún mecanismo existente en el lenguaje. Típicamente se describen asociaciones mediante la especificación de referencias a otras clases, donde las referencias son guardadas como atributos de la clase. En general asociaciones de grados mayores a

dos se implementan mediante asociaciones binarias, por lo cual analizaremos éstas últimas. Consideremos la relación entre las dos clases mostradas en el diagrama de la Figura 5.8.

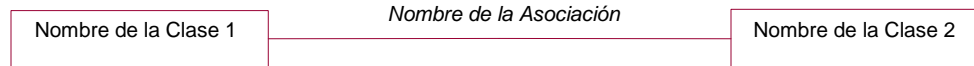


Figura 5.8 Asociación entre clases.

Una asociación binaria es implementada mediante un atributo correspondiente a cada clase de la asociación, como se muestra a continuación:

```
class Clase1 {
    Clase2 ref;
}
class Clase2 {
    Clase1 ref;
}
```

El mayor problema que existe con este tipo de implementación para las asociaciones es mantener la consistencia entre las referencias. En otras palabras, si la asociación, o liga, deja de existir, es importante que las referencias sean actualizadas de la manera correspondiente. Es importante que estos atributos de referencia no sean accesibles externamente para evitar actualizaciones de forma independiente. Por otro lado, los métodos accesibles externamente para actualizar los atributos no deberían ser añadidos a una de las clases de la asociación sin acceder la implementación del otro objeto, ya que los atributos están mutuamente restringidos. Cuando una nueva liga se añade a la asociación, ambos apuntadores deben ser actualizados, y cuando la liga es removida, ambos apuntadores también deben ser también removidos. Este es un ejemplo de la complejidad que le agrega a un programa por la falta de un mecanismo que implemente asociaciones y ligas de manera natural.

### Rol

En el código de Java anterior no hay ningún indicio al concepto de la asociación más que las dos referencias mencionadas. Por lo tanto, aunque la asociación tuviera un nombre, este nombre sería asignado a ninguno de las dos clases ya que el concepto como tal de la asociación se ha perdido. Sin embargo, el nombre de rol, es más fácil de asignar. Consideremos el diagrama de la Figura 5.9.



Figura 5.9 Asociación entre clases con nombres de rol.

Los nombres de rol pueden ser aprovechados para nombrar a los de las dos referencias, como se muestra a continuación:

```
class Clase1 {
    Clase2 rol2;
}
class Clase2 {
    Clase1 rol1;
}
```

Nótese que se utilizan los nombres de rol opuestos a la ubicación del atributo de referencia.

### Acceso

Analicemos ahora que ocurre si integramos el concepto del acceso o navegación para las asociaciones apoyado por UML. El diagrama de la Figura 5.10 muestra una asociación con navegación bidireccional, que es equivalente al código anterior. Nótese que se agregó el nombre de la asociación, aunque esto no afecta al código.

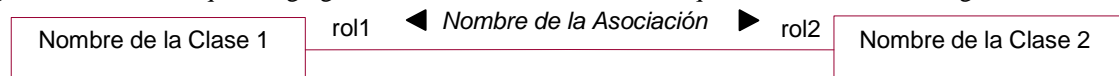


Figura 5.10 Asociación entre clases con nombres de rol y navegación bidireccional.

Simplifiquemos un poco la asociación mediante la navegación de una sola dirección, como se muestra en la Figura 5.11.

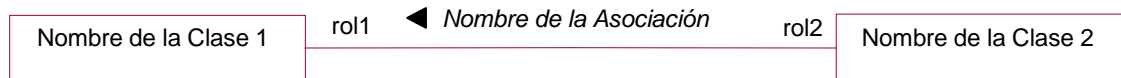


Figura 5.11 Asociación entre clases con nombres de rol y navegación en una sola dirección.

Con este último diagrama la navegación es de la clase 2 a la clase 1 pero no viceversa. Por lo tanto la relación puede implementarse mediante el siguiente código simplificado:

```
class Clase1 {
}
class Clase2 {
    Clase1 rol1;
}
```

### Multiplicidad

En todos los ejemplos anteriores la multiplicidad de la relación fue de “uno-uno”. La multiplicidad de mucho agrega cierta complicación ya que requiere de estructuras adicionales en lugar de la referencia directa. Consideremos el diagrama de la Figura 5.12.



Figura 5.12 Asociación entre clases con nombres de rol y multiplicidad “uno-muchos”.

El código para el lado "muchos" requieren un conjunto de objetos, o un arreglo de referencias, como se muestra a continuación:

```
class Clase1 {
    Clase2 rol2[];
}
class Clase2 {
    Clase1 rol1;
}
```

Obviamente, eventualmente se requiere instanciar los propios objetos, por lo cual, en el caso de un arreglo, el número máximo de posibles ligas debe ser conocido con anterioridad. Una opción más eficaz es utilizar estructuras de datos más avanzadas, como un objeto contenedor `Vector` o algún otro ofrecido por Java o escrito por el programador que evite predefinir el número máximo de relaciones que pueden haber para una clase particular.

El caso de multiplicidad “muchos-muchos” es simplemente una extensión del caso anterior. Consideremos el diagrama de la Figura 5.13.



Figura 5.13 Asociación entre clases con nombres de rol y multiplicidad “muchos-muchos”.

El código para ambos lados de la asociación requieren un conjunto de objetos, o un arreglo de referencias, como se muestra a continuación:

```
class Clase1 {
    Clase2 rol2[];
}
class Clase2 {
    Clase1 rol1[];
}
```

### Asociación Reflexiva

Una asociación reflexiva se implementa como caso especial de una asociación entre clases distintas. Consideremos el diagrama de la Figura 5.14.

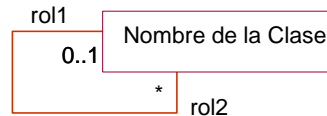


Figura 5.14 Asociación reflexiva con nombres de rol y multiplicidad.

El código para ambos lados de la asociación se muestra a continuación:

```
class Clase {
    Clase rol1;
    Clase rol2[];
}
```

Nótese que las referencias son ahora a la misma clase. Por otro lado, la multiplicidad “0..1” se implementa como la de “uno” (realmente la multiplicidad es cierta forma de restricción que debe ser asegurada mediante lógica adicional).

### Asociación como Clase

Como se describió en el Capítulo 4, se pueden modelar las asociaciones como clases. Esto en cierta manera puede simplificar la implementación de las asociaciones en especial el manejo de la consistencia entre las referencias de las clases. Aunque estos objetos no son difíciles de implementar, siempre ayuda que la biblioteca de clases las contenga o al menos que contenga clases que ayuden a la implementación final. El enfoque más sencillo es implementar un objeto de asociación como un objeto *diccionario* que hace un mapa entre la dirección hacia delante y la dirección hacia atrás de la asociación. El diccionario debe ser actualizado cuando la asociación es actualizada. Consideremos el diagrama de la Figura 5.15.

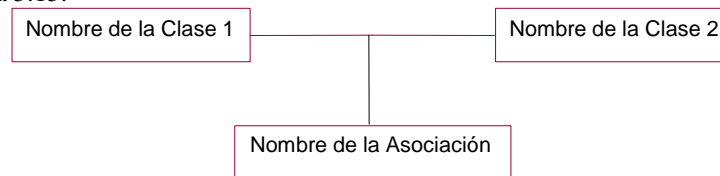


Figura 5.15 Asociación como clase.

El código para ambos lados de la asociación se muestra a continuación:

```
class Clase1 {
    Asociacion ref;
}
class Clase2 {
    Asociacion ref;
}
class Asociacion {
    Clase1 ref1[];
    Clase2 ref2[];
}
```

Nótese que las clases se refieren a la asociación (diccionario), mientras que la asociación es responsable de la consistencia en la información. Cualquier modificación en la multiplicidad de la asociación sólo afecta a la clase asociación. Además la clase asociación puede guardar atributos y operaciones particulares de la relación que a su vez es especializada de una clase diccionario genérico. De tal manera, no hay necesidad de añadir atributos a las clases originales en la asociación que realmente dependen de la asociación. Si una pequeña fracción de los objetos participan en la asociación, entonces objetos de asociación separados aprovechan mejor el espacio que clases que incluyen todos los atributos relacionados con la asociación y que de manera poco frecuente son utilizados.

### Composición

La composición es básicamente una extensión del concepto de asociación. Dado que la asociación no tiene ningún mecanismo que la soporte en Java, la composición tampoco. Consideremos el diagrama de la Figura 5.16.

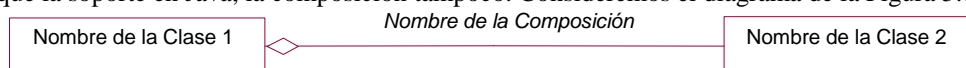


Figura 5.16 Composición entre clases.

El código para la composición se muestra a continuación:

```
class Clase1 {
    Clase2 ref;
}
class Clase2 {
    Clase1 ref;
}
```

Como se puede ver, no hay diferencia de implementación con la asociación, y todas las consideraciones descritas en la sección de ligas y asociaciones se aplica.

### Clases Contenedoras

Las *clases contenedoras* son un buen ejemplo de clases que contienen a otras clases y que utilizan asociaciones y composición como base. Dos de los ejemplos más importantes de estas clases son la lista o *lista ligada* (“LinkedList”) y la *pila* (“stack”).

#### Lista

El siguiente diagrama muestra un diseño genérico para una *lista* que puede contener cualquier tipo de objeto, como se muestra en la Figura 5.17.

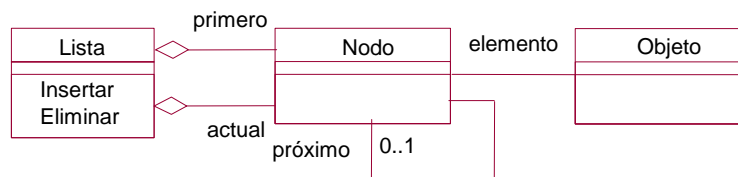


Figura 5.17 Diagrama para una Lista Ligada.

Se utilizan tres clases:

- ?? *Lista* que agrupa las operaciones de *insertar* y *eliminar* los objetos contenidos a través de un número indefinido de nodos. Se guarda la referencia al *primero* y al *actual* (corresponde al *último* elemento). Se utiliza una relación de composición para resaltar que la lista contiene nodos.
- ?? *Nodo* que son los contenedores para cada uno de los objetos. Cada nodo tiene una referencia al *próximo* o *siguiente* nodo (que puede ser nula), además de una referencia al propio objeto. Es muy importante esta clase ya que ofrece la funcionalidad de liga entre nodos.
- ?? *Objeto* que corresponde a la propia información que la lista guarda. Es importante separarla de los nodos para no requerir ninguna funcionalidad que no sea exclusiva al objeto, a diferencia del nodo que guarda funcionalidad propia de la lista.

Aunque pudiesen utilizarse menos clases (también pudieran ser más), este diseño es muy compacto evitando cualquier mezcla de la información con la lista, un requisito importante de una clase *contenedora*.

Comenzamos la descripción del código a partir de la clase *Nodo* ya que la clase *Objeto* la representamos por la clase *Object*, la superclase de todas las clases en Java, como describiremos en mayor detalle en la sección de herencia más adelante. Esto último facilita mucho el diseño y el manejo de las clases contenedoras. Nótese la correspondencia entre atributos y métodos con el diagrama. Obviamente el código tiene el detalle completo.

```
class Nodo
{
    private Nodo proximo;
    private Object elemento;

    public Nodo(Object elem) {
        setElemento(elem); }

    public void setElemento(Object elem) {
        elemento = elem; }
    public Object getElemento() {
        return elemento; }
    public void setProximo(Nodo prox) {
        proximo = prox; }
    public Nodo getProximo() {
        return proximo; }
}
```

En el código anterior, todos los atributos son privados, por lo cual se agregan métodos `get` y `set` para consultar y modificar sus valores. Se tiene un sólo constructor para inicializar el nodo con el elemento respectivo. Como debe ocurrir con cualquier clase contenedora, la lógica del modelo debe guardarse en el agregado, o sea la clase `Lista`:

```
public class Lista {
    private Nodo primero;
    private Nodo actual;

    private Nodo getPrimero() {
        return primero; }
    private Nodo getActual() {
        return actual; }
    private Object getElemento() {
        if (actual != null)
            return actual.getElemento();
        else
            return null;
    }
    public void insertar(Object elem) {
        Nodo tmp = new Nodo(elem);
        if (actual != null) {
            tmp.setProximo(actual.getProximo());
            actual.setProximo(tmp); }
        if (primero == null)
            primero = tmp;
        actual = tmp;
    }
    public Object eliminar() {
        Nodo tmp = null;
        Object elem = null;
        if (primero != null)
        {
            tmp = primero.getProximo();
            elem = primero.getElemento();
            primero = tmp;
        }
        return elem;
    }
}
```

Nuevamente, nótese la correspondencia con el diagrama. Vale la pena resaltar ciertos aspectos del código. No hubo necesidad de agregar un constructor ya que los atributos son inicializados por omisión a un valor nulo. Los tres métodos `get` son privados, mientras que los únicos métodos públicos para la manipulación de la lista son `insertar` y `eliminar`. Esto es importante para un buen manejo del encapsulamiento de la lista. Más aún, sólo la clase `Lista` es pública, mientras que `Nodo` es privada si se accesa desde otro paquete. En el ejemplo, se inserta elementos al final de la lista y se elimina del inicio de la lista. Un comentario adicional, es que esta lista sólo inserta y elimina elementos. Su funcionalidad puede ser fácilmente extendida mediante operaciones, por ejemplo, para revisar o imprimir los elementos de la lista.

### Pila

El siguiente diagrama muestra un diseño genérico para una *pila* que puede contener cualquier tipo de objeto, como se muestra en la Figura 5.18.

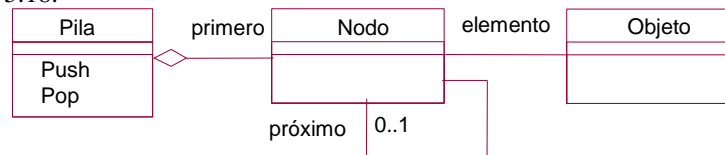


Figura 5.18 Diagrama para una Lista Ligada.

Nótese la similitud con el diagrama de la Figura 5.17 para la *lista*. Se utilizan nuevamente tres clase:

- ?? Pila que agrupa las operaciones de *push* (meter) y *pop* (sacar) los objetos contenidos a través de un número indefinido de nodos. Se guarda la referencia al *primero* únicamente. Se utiliza una relación de composición para resaltar que la pila contiene nodos.
- ?? Nodo que son los contenedores para cada uno de los objetos. Cada nodo tiene una referencia al *próximo* o *siguiente* nodo (que puede ser nula), además de una referencia al propio objeto. Es similar al nodo de la lista.
- ?? Objeto que corresponde a la propia información que la pila guarda.

En nuestro ejemplo de la pila reutilizaremos el diseño de la clase `Nodo` como se verá a continuación. La clase `Objeto` es nuevamente implementada por la clase `Object`, la superclase a todas las clases en Java.

```
class Nodo
{
    private Nodo proximo;
    private Object elemento;

    public Nodo(Object elem) {
        setElemento(elem); }

    public void setElemento(Object elem) {
        elemento = elem; }
    public Object getElemento() {
        return elemento; }
    public void setProximo(Nodo prox) {
        proximo = prox; }
    public Nodo getProximo() {
        return proximo; }
}
```

En código anterior es exactamente igual al `Nodo` para el ejemplo de la lista. Todos los atributos son privados, por lo cual se agregan métodos `get` y `set` para consultar y modificar sus valores. Se tiene un sólo constructor para inicializar el nodo con el elemento respectivo. Como debe ocurrir con cualquier clase contenedora, la lógica del modelo debe guardarse en el agregado, o sea la clase `Pila`:

```
public class Pila
{
    private Nodo primero;

    public void push(Object elem) {
        Nodo tmp = new Nodo(elem);
        if (primero != null)
            tmp.setProximo(primero);
        primero = tmp;
    }
    public Object pop() {
        Nodo tmp;
        Object elem;
        if (primero != null)
        {
            elem = primero.getElemento();
            tmp = primero;
            primero = tmp.getProximo();
            return elem;
        }
        return null;
    }
}
```

Nuevamente, nótese la correspondencia con el diagrama. Vale la pena resaltar ciertos aspectos del código. No hubo necesidad de agregar un constructor ya que los atributos son inicializados por omisión a un valor nulo. El código de la clase `Pila` es más sencillo que el de la clase `Lista`. Se omitieron los métodos `get`, mientras que los únicos métodos públicos para la manipulación de la lista son `push` y `pop`. Esto es importante para un buen manejo del encapsulamiento de la pila. Y de manera similar al ejemplo de la lista, sólo la clase `Pila` es pública, mientras que `Nodo` es privada si se accesa desde otro paquete.



### 5.2.6 Generalización y Herencia

La herencia es un aspecto fundamental de Java y de los lenguajes orientados a objetos. Tomemos el diagrama de herencia (sencilla) que se muestra en la Figura 5.19.

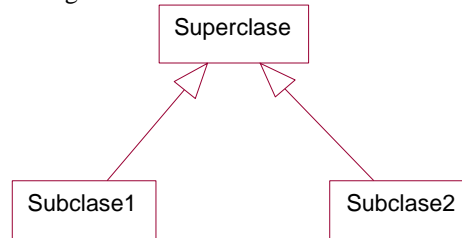


Figura 5.19 Herencia de clases.

En la figura se muestra una superclase de la cual heredan dos subclase. La herencia es codificada utilizando la palabra `extends` como se muestra a continuación:

```

class Superclase {
}
class Subclase1 extends Superclase {
}
class Subclase2 extends Superclase {
}
  
```

Un comentario general sobre el esquema de herencia en Java es que de no ser especificada una superclase, Java genera implícitamente una herencia a la clase `Object`. De tal manera `Object` es la superclase, directa o indirectamente, de todo el resto de las clase en una aplicación. De tal forma, la clase `Object` es la única que no tiene una superclase.

Consideremos el siguiente ejemplo particular de uso de herencia como se muestra en el diagrama de la Figura 5.20.

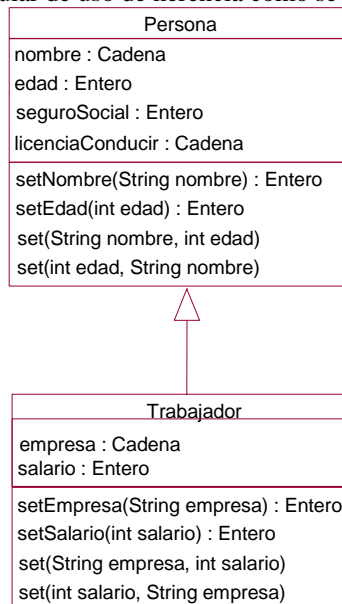


Figura 5.20 Herencia de Persona a Trabajador.

El código para la herencia entre `Persona` y `Trabajador` se muestra a continuación. El código para la clase `Persona` es ligeramente modificado para que sus atributos sean `protected` en lugar de `private`, de tal manera que la clase `Trabajador` pueda luego utilizarlos:

```

class Persona {
    protected String nombre;
    protected int edad;
    protected int seguroSocial;
    protected String licenciaConducir;

    public Persona(String nom, int ed, int seg, String lic) {
        set(nom, ed); seguroSocial = seg; licenciaConducir = lic; }
  
```

```

public Persona() {
    Persona(null, 0, 0, null); }

public int setNombre(String nom) {
    nombre = nom; return 1; }
public int setEdad(int ed) {
    edad = ed; return 1; }
public void set(String nom, int ed) {
    setNombre(nom); setEdad(ed); }
public void set(int ed, String nom) {
    setNombre(nom); setEdad(ed); }
}

```

El código para la clase Trabajador se muestra a continuación:

```

class Trabajador extends Persona {
    private String empresa;
    private int salario;

    public Trabajador(String emp, int sal) {
        empresa = emp; salario = sal; }
    public Trabajador() {
        this(null,0); }

    public int setEmpresa(String emp) {
        empresa = emp; return 1; }
    public int setSalario(int sal) {
        salario = sal; return 1; }
    public void set(String emp, int sal) {
        setEmpresa(emp); setSalario(sal); }
    public void set(int sal, String emp) {
        setEmpresa(emp); setSalario(sal); }
}

```

Nótese la similitud entre ambas clases, aunque Trabajador en este caso hereda de Persona. La instanciación de un objeto de tipo Trabajador es similar a la que se hizo anteriormente para Persona, aunque obviamente cambiando el nombre de la clase. A continuación hacemos dos instancias como ejemplo:

```

Trabajador t1 = new Trabajador ();
Trabajador t2 = new Trabajador ("IBM", 35000);

```

Hay un pequeño detalle que vamos a remediar en la siguiente sección: no se está asignando ningún valor a los atributos de Persona cuando se instancia un nuevo objeto. En otras palabras, se le asigna valores a los atributos de Trabajador pero no a los de su superclase Persona.

Existe un modificador especial, `final`, que si se agrega de prefijo en la primera línea de la definición de una clase, hace que la clase no pueda ser heredada por otras.

### Referencia a la Superclase

Existe en Java una palabra reservada llamada `super` que es algo similar en su uso al `this` descrito anteriormente. La palabra `super` se utiliza de dos maneras distintas: como referencia a algún campo de la superclase del objeto o cómo llamada a un constructor de la superclase. Veamos los dos casos:

?? Como ejemplo de referencia a un campo de la superclase, consideremos que se define un segundo atributo `edad` dentro de la clase Trabajador:

```

class Trabajador extends Persona {
    ...
    private int edad;
    ...
}

```

Para poder acceder el atributo de la superclase agregamos un nuevo método `setSuperEdad` dentro de la clase Trabajador de la siguiente forma:

```

private int setSuperEdad(int edad) {
    super.edad = edad; return 1; }

```

Nótese que el método `setEdad` de la clase `Persona` modificaría el atributo `edad` de la clase `Trabajador`. Este es un ejemplo del gran cuidado que se debe tener cuando se usa herencia. Nótese que es ilegal especificar `super.super.edad`.

?? Como ejemplo de llamada a un constructor de la superclase, consideremos el siguiente constructor adicional para la clase `Trabajador` que incluye parámetros para inicializar valores en los atributos heredados de `Persona`:

```
public Trabajador (String emp, int sal,
    String nom, int ed, int seg, String lic) {
    super(nom, ed, seg, lic);
    set(emp, sal);
}
```

Un nuevo objeto de tipo `Trabajador` utilizando el constructor anterior sería el siguiente:

```
Trabajador t3 = new Trabajador ("IBM", 35000, "Juan", 35, 1234567, "x254f");
```

Este segundo constructor agrega argumentos para los atributos de ambas clase y se aprovecha del constructor de la superclase para redirigir la llamada utilizando `super()`. De manera análoga a `this()`, existe la restricción de que `super()` sólo puede utilizarse dentro de un constructor y debe aparecer como su primera línea. Dada la restricción de ambas llamadas, no es posible combinarlas dentro de un mismo constructor. Por omisión, Java siempre llama al constructor vacío de la superclase, por lo cual este debe existir de manera explícita si existen otros constructores en la superclase, o en el caso de no haber constructores, Java genera uno de manera implícita para la superclase. La única excepción que hace Java de no llamar a `super()` de manera implícita es cuando ya existe una llamada a `this()` en el constructor.

### Sobrescritura y Polimorfismo

En los ejemplos de las secciones anteriores ya se han mostrado algunos casos de sobrescritura de atributos y métodos. A continuación describimos estos casos con mayor detalle.

#### Atributos

La sobrescritura de atributos (*“shadowed”*) corresponde a dos atributos, uno definido en la superclase y otro en la subclase, ambos con el mismo nombre. Esto es útil si desea utilizar en la subclase la misma variable definida con un tipo diferente y también es útil cuando éstas son inicializadas con valores distintos. En nuestros ejemplos anteriores se dio el caso de la sobrescritura del atributo `edad` en la clase `Trabajador` con respecto a la ya definida en la clase `Persona`. En este caso no hay distinción ni de tipo ni de valor de inicialización entre ambas. Para distinguir entre ambos atributos es necesario utilizar `this.edad` o `super.edad` para referirse a la edad definida en `Trabajador` o en `Persona`, respectivamente. Esto se aplica únicamente si el objeto donde se encuentra las llamadas fue instanciado como `Trabajador` (realmente no importa si las llamadas están dentro de un método que pertenece a la superclase o a la subclase). Por ejemplo, el siguiente código para un objeto de tipo `Trabajador` corresponde al caso `this.edad` donde se accesa la edad del `Trabajador` a pesar de que el método `seEdad` está definido dentro de la clase `Persona`:

```
private int setEdad(int ed) {
    edad = ed; return 1; }
```

Si el objeto fue instanciado de la clase `Persona`, la situación de sobrescritura ya no existe.

Otra forma de distinguir entre los dos atributos es mediante un *cast* utilizando `this`: `((Trabajador)this).edad` o `((Persona)this).edad`, donde el atributo se refiere a la clase correspondiente al *cast*.

#### Métodos

La sobrescritura de es la base del *polimorfismo* es los lenguajes orientados a objetos. La sobrescritura se base en definir métodos con la misma firma exacta en la superclase al igual que la subclase (análoga al uso de *virtual* en C++). En los ejemplos de la clase `Trabajador` y la clase `Persona` se sobrescribió el método `set` como se puede ver a continuación. La clase `Persona` incluía los dos siguiente métodos `set`:

```
class Persona {
    ...
    public void set(String nom, int ed) {
        setNombre(nom); setEdad(ed); }
    public void set(int ed, String nom) {
        setNombre(nom); setEdad(ed); }
}
```

La clase `Trabajador` incluía los dos siguiente métodos `set`:

```

class Trabajador extends Persona {
    ...
    public void set(String emp, int sal) {
        setEmpresa(emp); setSalario(sal); }
    public void set(int sal, String emp) {
        setEmpresa(emp); setSalario(sal); }
}

```

La sobrescritura de los métodos anteriores se puede apreciar mejor con el siguiente ejemplo:

```

Trabajador t4 = new Trabajador ();
t4.set("Perez", 50);

```

¿A cuál método `set` se llama, al de `Persona` o al de `Trabajador`? La respuesta es que se llama al método `set` que sobrescribe al de su superclase, por lo tanto es el de `Trabajador`. El que los argumentos tengan nombres diferentes no afecta, únicamente afectan sus tipos. (Cómo comentario adicional, es un error tener dos métodos con firmas similares, sea en la misma o en la superclase, pero con tipos de retorno diferente.)

El ejemplo anterior no es demasiado descriptivo para poder apreciar el poder de la sobrescritura y del polimorfismo. Consideremos el siguiente ejemplo que consiste de las clases que se muestran en el diagrama de la Figura 5.21.

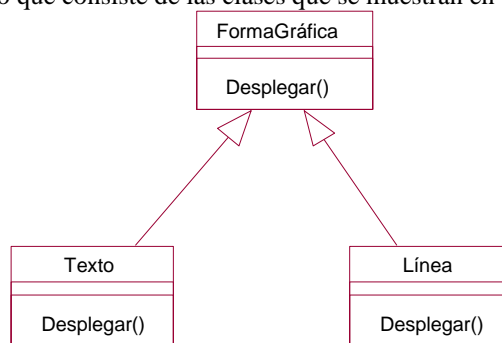


Figura 5.21 Ejemplo de polimorfismo.

A continuación definiremos los aspectos esenciales de estas clases.

```

public class FormaGrafica {
    ...
    public void desplegar(int x, int y) {
    }
    ...
}
public class Texto extends FormaGrafica {
    ...
    public void desplegar(int x, int y) {
        desplegarTexto(); }
    ...
}
public class Línea extends FormaGrafica {
    ...
    public void desplegar(int x, int y) {
        desplegarLínea(); }
    ...
}

```

Sin entrar en detalles y omitiendo otros, las tres clases definen el método `desplegar` con exactamente la misma firma, aunque la superclase la tiene vacía mientras que las dos subclases, `Texto` y `Línea`, solicitan `desplegarTexto` y `desplegarLínea`, respectivamente. Ahora, aprovechemos la lista que definimos anteriormente y escribamos el siguiente código un método `desplegarVentana` de alguna otra clase:

```

public void desplegarVentana (Lista l) {
    ...
    FormaGrafica fg;
    while ((fg = (FormaGrafica)l.eliminar()) != null) {
        fg.desplegar();
    }
}

```

El método `desplegarVentana` tiene como argumento una `Lista l` que para nuestro ejemplo suponemos que esta llena, donde anteriormente se la insertado un número de objetos de tipo `Texto` o `Línea`. Como simple ejercicio, iremos eliminando cada uno de estos objetos de la lista (dentro del “while”) y si el objeto resultante no es nulo, lo desplegaremos. Lo interesante del ejercicio es que la variable `fg` fue declarada del tipo de la superclase que tiene el método `desplegar` a ser sobrescrito, sin en ningún momento mencionar en este método el tipo de las dos subclases, `Texto` y `Línea`. Sin embargo, el despliegue es correcto ya que Java reconoce dinámicamente el tipo verdadero (no el declarado por `fg` que simplemente guarda la referencia al objeto) del objeto en la lista y hace el llamado de acuerdo a la sobrescritura correspondiente. Esto significa que si en un futuro definimos nuevas subclases de `FormaGrafica` y sobrescribimos de manera adecuada el método `desplegar`, el método `desplegarVentana` no tendrá que ser modificado para el manejo adecuado de la nueva clase. Esto es “extensibilidad al máximo”, el código nuevo no afecta en absoluto al código viejo. Cuando se logra manejar y aprovechar adecuadamente el polimorfismo, se puede uno considerar que domina la programación orientada a objetos.

Como ejercicio mental, consideremos que ocurriría si el lenguaje no apoyara el polimorfismo, o sea, la sobrescritura de métodos. Dentro del método `desplegarVentana` tendríamos que revisar el tipo verdadero de cada objeto al que se refiere `fg`, posiblemente mediante múltiples expresiones “if else” que permitan conocer su tipo y hacer la llamada adecuada de manera explícita. Esto sería muy tedioso y requeriría de modificaciones constantes para adecuarse a nuevas subclases.

Vale la pena destacar, como se vio en los ejercicios de las secciones anteriores, que se puede invocar un método sobrescrito por medio de la referencia `super` seguido por el nombre del método.

Como comentario final a esta sección, métodos que tengan el modificador `final` en su definición en la superclase, no pueden ser sobrescritos. Más aún, todos los métodos de una clase con el prefijo `final` también se consideran `final`. Además de no poder ser sobrescritos, los métodos `final` son más eficientes ya que no participan en la sobrescritura que es un proceso dinámico de búsqueda en Java como en la mayoría de los demás lenguajes orientados a objetos.

### Clases Abstractas

Las clases abstractas son un aspecto básico de la generalización dado que definen clases que requieren subclases para poder utilizarse. De manera básica, se puede definir una clase como abstracta mediante el modificador `abstract`. Una clase definida de esta manera no puede ser instanciada, requiriendo una subclase para poder ser utilizada. Una clase abstracta se define de la siguiente manera:

```
abstract class NombreClase
```

Por ejemplo, podríamos modificar la definición de la clase `FormaGrafica` para volverla una clase abstracta que no pudiera instanciarse.

```
public abstract class FormaGrafica {
    ...
    public void desplegar(int x, int y) {
    }
    ...
}
```

Fuera de esta restricción de no poder ser instanciada directamente, una clase abstracta puede contener atributos y métodos como cualquier otra clase normal (concreta).

### Métodos Abstractos

La utilización del modificador `abstract`, como se mostró en la sección anterior, define una clase como abstracta. Además de esto, se pueden definir métodos abstractos, utilizando el modificador `abstract` en los propios. El uso sería por ejemplo el siguiente:

```
public abstract class FormaGrafica {
    ...
    public abstract void desplegar(int x, int y);
    ...
}
```

Si el método `desplegar` de la clase `FormaGrafica` de los ejemplos anteriores fuera definido de esta forma, cualquier clase que herede de `FormaGrafica` debería forzosamente sobrescribir el método `desplegar`. Efectivamente, cualquier clase con un método abstracto, automáticamente se vuelve una clase abstracta, la cual no puede ser instanciada. Nótese que es obligatorio que la clase se defina como abstracta si esta incluye algún método

abstracto. El opuesto no es obligatorio. También nótese que al volverse el método abstracta, se elimina su implementación (que anteriormente estaba vacía).

Como se puede apreciar del ejemplo anterior, un método abstracto no tiene cuerpo, solo una firma. Todas las subclases que hereden de esta clase tienen que sobrescribir los métodos abstractos definidos en la superclase, si no la subclase se consideraría también abstracta. (Esto es similar a una función en C++ igualada a "0" en su definición, por ejemplo, "void func()=0".)

### Interfaces

Como alternativa a la definición de clases y métodos abstractos, Java ofrece otra estructura que la interface. Las interfaces son similares a clases abstractas, excepto que se utiliza la palabra `interface` en lugar de `abstract` y `class`. Una interface se define de la siguiente manera:

```
public interface NombreInterface {
    ...listaMétodos...
}
```

Una interface sólo permite definir métodos pero no atributos. Estos métodos son implícitamente abstractos y no pueden contener una implementación dentro de la interface. (Al igual que una clase, una interface puede incluso estar completamente vacía.) La única otra estructura que puede definirse dentro de la interface es una *constante estática* (`static final`) un tema que trataremos en la siguiente sección. Consideremos la modificación de la clase `FormaGrafica` para volverse una interface:

```
public interface FormaGrafica {
    ...
    public void desplegar(int x, int y);
    ...
}
```

Nótese que ya no se utiliza el modificador `abstract` dentro de la declaración del método `desplegar`. ¿Como habría que modificar a las clases `Texto` y `Linea` para poder utilizar `FormaGrafica` si esta se vuelve una interface? La respuesta es que en lugar de utilizar la palabra `extends` ahora se debe usar la palabra `implements`. Por lo tanto, la nueva definición de `Texto` y `Linea` sería la siguiente:

```
public class Texto implements FormaGrafica {
    ...
    public void desplegar(int x, int y) {
        desplegarTexto();
    }
    ...
}
public class Linea implements FormaGrafica {
    ...
    public void desplegar(int x, int y) {
        desplegarLinea();
    }
    ...
}
```

A diferencia de que se permite un sólo `extends` para la herencia de clases en Java (o sea herencia sencilla), Java permite utilizar múltiples `implements` dentro de una clase. Por ejemplo, consideremos la siguiente interface:

```
public interface FormaEscalable {
    ...
    public void escalar(double s);
    ...
}
```

Esta interface define el método `escalar` que permite a un objeto gráfico cambiar su tamaño. Las clases `Texto` y `Linea` se podrían modificar de la siguiente forma:

```
public class Texto implements FormaGrafica, FormaEscalable {
    ...
    public void desplegar(int x, int y) {
        desplegarTexto();
    }
    public void escalar(double s) { ... }
    ...
}
public class Linea implements FormaGrafica, FormaEscalable {
    ...
    public void desplegar(int x, int y) {
```

```

        desplegarLinea(); }
    public void escalar(double s) { ... }
    ...
}

```

De tal forma, una clase puede implementar cualquier número de interfaces.

También es posible que una clase herede de su superclase mediante el `extends` y a la vez implemente a su interface mediante el `implements`, donde el número de interfaces implementadas no tiene límite. Por ejemplo, volvamos a la definición original de la clase `FormaGrafica`:

```

public class FormaGrafica {
    ...
    public void desplegar(int x, int y);
    ...
}

```

Las clases `Texto` y `Linea` se podrían modificar de la siguiente forma:

```

public class Texto extends FormaGrafica implements FormaEscalable {
    ...
    public void desplegar(int x, int y) {
        desplegarTexto(); }
    public void escalar(double s) { ... }
    ...
}
public class Linea extends FormaGrafica implements FormaEscalable {
    ...
    public void desplegar(int x, int y) {
        desplegarLinea(); }
    public void escalar(double s) { ... }
    ...
}

```

Las clases `Texto` y `Linea` pueden efectivamente considerarse una instancia de ambos tipos `FormaGrafica` y `FormaEscalable`.

De manera análoga a que las clases pueden extenderse de manera jerárquica a través de subclases, las interfaces pueden extenderse en subinterfaces. Una subinterface hereda todos los métodos abstractos y constantes estáticas de la superinterface, y puede definir nuevos métodos abstractos y constantes estáticas. Una interface puede extender más de una interface a la vez. Por ejemplo consideremos la siguiente interface que permite rotar objetos gráficos:

```

public interface FormaRotable {
    ...
    public void rotar(double r);
    ...
}

```

Ahora definamos una nueva interface `FormaTransformable` que extiende a `FormaEscalable` y `FormaRotable`:

```

public interface FormaTransformable extends FormaEscalable, FormaRotable {}

```

Las clases `Texto` y `Linea` se podrían modificar de la siguiente forma:

```

public class Texto extends FormaGrafica implements FormaTransformable {
    ...
    public void desplegar(int x, int y) {
        desplegarTexto(); }
    public void escalar(double s) { ... }
    public void rotar(double r) { ... }
    ...
}
public class Linea extends FormaGrafica implements FormaTransformable {
    ...
    public void desplegar(int x, int y) {
        desplegarLinea(); }
    public void escalar(double s) { ... }
    public void rotar(double r) { ... }
    ...
}

```

Este manejo de jerarquías de interfaces permite consolidar múltiples interfaces en una para ser luego implementadas a través de una sola interface.

### Herencia Múltiple

El tema de la herencia múltiple es uno de los aspectos más complejos en los lenguajes de programación orientados a objetos. Esta complejidad radica en las dificultades de implementación por parte de los compiladores de estos lenguajes. Los distintos lenguajes toman diferentes enfoques con respecto a la herencia múltiple. Como se discutió inicialmente en el capítulo 4, existe una problemática de heredar atributos y métodos similares de distintas superclases, ocasionando el problema de resolver cuales de estos se va a utilizar. Por ejemplo, consideremos el diagrama de la Figura 5.22 donde las Transformable, Escalable y Rotable se vuelven clases en lugar de interfaces por lo cual aceptan atributos e implementación de métodos.

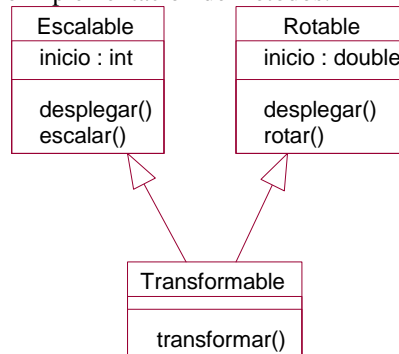


Figura 5.22 Ejemplo de herencia múltiple.

Ahora definamos el método transformar para la clase Transformable:

```

public void transformar() {
    inicio = 4.5;
    desplegar();
}
  
```

A cual atributo se refiere inicio, al de la clase Escalable que es un int, o al de la clase Rotable que es un double. Además, a cual desplegar se llama, al definido en la clase Escalable o al definido en la clase Rotable. Esta es la base de la complejidad que ocasiona la herencia múltiple y que requiere de mecanismos adicionales, que pueden ser bastante complejos, para ser resueltos por un lenguaje de programación. Por lo tanto, los distintos lenguajes toman diversos enfoques. Por ejemplo, C++ apoya la herencia múltiple aunque con ciertas dificultades para el usuario (tales como conflictos con el manejo de apuntadores y referencias especiales para resolver la herencia de atributos y métodos), mientras que Smalltalk directamente no apoya la herencia múltiple. Por otro lado, Java toma un enfoque muy original de herencia múltiple “restringida”. Java, como hemos visto, permite herencia sencilla de clases pero implementación de múltiple interfaces. Si nos olvidamos de la nomenclatura especial por un momento, o sea, interface e implements, estas estructuras son simplemente clases sin atributos ni implementación de métodos. Lo que estas estructuras ofrecen es una solución a la herencia múltiple pero sin los conflictos de herencia de múltiples atributos e implementación de métodos de múltiples superclases. En otras palabras, Java elimina la complejidad de herencia múltiple pero aún ofreciendo un mecanismo similar.

En general, como muchos lenguajes de programación orientados a objetos, tales como Java, no apoyan la herencia múltiple, es necesario en tales casos implementar herencia múltiple a través de herencia sencilla y posiblemente agregación (*delegación*). Los siguientes tres casos describen el enfoque general:

- ?? Implementación de herencia múltiple usando agregación. Una superclase con múltiples generalizaciones individuales se puede redefinir como un agregado en el cual cada componente del agregado reemplaza una de las ramas de la generalización. Se reemplaza las posibles instancias de la herencia múltiple por un grupo de instancias que componen el agregado. La herencia de las operaciones a través del agregado no es automática, debiendo ser delegadas a los componentes apropiados. Si una subclase tiene varias superclases, todas de igual importancia, es mejor usar delegación y preservar la simetría.
- ?? Implementación de herencia múltiple heredando de la clase más importante y delegando el resto. Se toma una como subclase de la superclase más importante, combinándose con un agregado correspondiendo a las generalizaciones restantes. Si se tiene una superclase principal, se implementa la herencia múltiple a través de herencia sencilla y agregación. Si el número de combinaciones es pequeño, se puede usar generalización



anidada (siguiente caso). Si el número de combinaciones, o el tamaño del código, es grande se debe evitar este tipo de implementación.

- ?? Implementación de herencia múltiple usando generalización anidada. Se crean varios niveles de generalización, terminando la jerarquía con subclases para todas las posibles combinaciones de clases unidas. En este caso no se utiliza agregación. Se preserva la herencia pero se duplica las declaraciones, rompiendo con el espíritu de la orientación a objetos. Se debe factorizar primero según el criterio de herencia más importante, y luego según el resto. Si una superclase tiene bastantes más características que las otras superclases, o si una superclase es el cuello de botella en el rendimiento, se debe preservar la herencia en relación a esa clase.

### 5.2.7 Entidades Estáticas

Existe en Java el concepto de estructuras estáticas de clases. A diferencia de los *atributos (atributos de instancia o atributo de objeto)* y *métodos (métodos de instancia o método de objeto)* descritos anteriormente, los cuales requieren de un objeto instanciado de la clase que los define para poder ser utilizados, los atributos estáticos (*atributos de clase*) y métodos estáticos (*métodos de clases*) no requieren de la existencia de los objetos y pueden ser utilizados directamente a partir de las clases que los definen. Nótese que un objeto siempre puede acceder a sus campos de clase (estáticos), mientras que los campos estáticos no pueden acceder los campos del objeto. Los campos estáticos pueden recibir todos los modificadores aplicables a los no estáticos, incluso se aplican las mismas operaciones. Para ello se utiliza la palabra `static` que convierte un atributo o un método en estático, como veremos a continuación. Nótese, que ni los atributos ni los métodos estáticos pueden ser sobrescritos.

#### Atributos

Los atributos estáticos o atributos de clase se distinguen de los atributos de objeto en que se tiene una sola copia para todos los objetos de una clase. Por ejemplo, consideremos el siguiente atributo estático:

```
class Persona {
    public static String nacionalidad;
    ...
}
```

Definamos el siguiente método (fuera de la clase `Persona`) que muestra el manejo de los atributos estáticos:

```
public void print () {
    Persona.nacionalidad = "mexicano";
    ...
}
```

Como se puede ver, el acceso de la variable `nacionalidad` es por medio de la clase y no por medio de un objeto. Es importante resaltar que todos los objetos instanciados de la clase `Persona` tienen acceso a una sola copia de `nacionalidad`, por lo cual cualquier cambio a su valor afectaría a todos estos objetos. Los atributos de clase se inicializan cuando la clase se carga por primera vez, a diferencia de las variables de instancia que se inicializan solo cuando se instancian nuevos objetos.

Como se mencionó antes, los atributos estáticos aceptan todos los modificadores que los atributos normales. Por lo tanto, se pueden definir atributos estáticos constantes utilizando el `static final`. Este tipo de constantes, que como todos los atributos, es declarado dentro de la definición de clase, es equivalente al `“?define”` en C y C++. El compilador de Java utiliza el valor asignado a la constante para calcular inicialmente otras constantes de “tiempo de compilación”, algo que no se puede hacer con constantes no estáticas. Además, el `“static final”`, también puede ser utilizado en el caso de compilación condicional. Por ejemplo:

```
public static final boolean DEBUG = false;
```

puede ser utilizado en secciones `“if”` para que se compilen o no.

#### Métodos

Los métodos de clase se declaran también con `static` y se invocan con el nombre de la clase de manera similar a los atributos de clase. (Estos métodos no pueden pasar el `this` como referencia ya que pueden existir sin que se hayan instanciado objetos. Por ejemplo, la siguiente es una declaración de un método estático:

```
class Persona {
    public static String nacionalidad;
    public static String getNacionalidad() {
        return nacionalidad;
    }
    ...
}
```

Nótese que los métodos estáticos tienen acceso a los atributos estáticos dentro de una misma clase. Modifiquemos el método `print` descrito en la sección anterior que muestra el manejo de los métodos estáticos:

```
public void print () {
    Persona.getNacionalidad();
    ...
}
```

Nuevamente, el acceso es mediante el nombre de la clase. Muchas de las bibliotecas de Java aprovechan los métodos estáticos para definir funciones que no requieren instanciación de objetos para utilizarse. Por ejemplo, todos los métodos de la clase `System` son métodos de clase, tales como “`System.out.print()`”, al igual que los métodos de la clase `Math`, que funciona como una biblioteca de funciones más que como instancias de objetos.

Existe un método estático extremadamente importante, “`main`”, que indica el inicio de la aplicación, como explicaremos en la sección de aplicaciones y applets.

### Inicializador

Para mantener el máximo posible de similitud con el manejo de clases “normales”, existe un inicializador, análogo al constructor, que permite inicializar los aspectos estáticos de la clase (no de las instancias). No tiene argumentos ya que automáticamente se carga cuando la clase se carga. Un inicializador estático de clase tiene el siguiente formato:

```
static {
    ...
}
```

A diferencia de los constructores, no tiene nombre ni se pasan argumentos. Java permite múltiples bloques estáticos como el anterior, los cuales se llaman todos al cargar la clase. Una de las aplicaciones es cargar *métodos nativos* de la máquina virtual, típicamente en C.

### 5.2.8 Meta Clases

Existe en Java el concepto de meta clases, o sea clases de clases. Si un *objeto* es la instancia de una *clase*, entonces la propia *clase* es la instancia de una *meta clase*. Este concepto se muestra en el diagrama de la Figura 5.23.

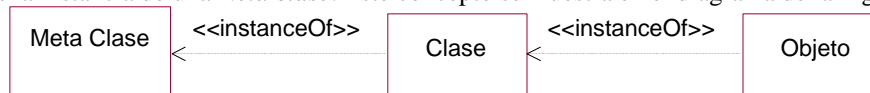


Figura 5.23 Concepto de *meta clase*.

El concepto de *meta clase* es extremadamente útil, en particular el hecho de poder tratar a una clase como si fuera un objeto. Por ejemplo, la Figura 5.24 muestra las relaciones anteriores adaptadas por Java, donde `Class` corresponde a la meta clase y la relación está dada con cualquier *Clase* y *Objeto* definido por el programador.

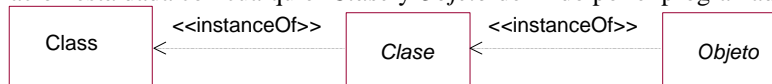


Figura 5.24 Concepto de *meta clase* en Java.

Por ejemplo, veamos el siguiente código:

```
Class miClase = Class.forName(nombre_clase);
Object miObjeto = miClase.newInstance();
```

En la primera línea se traduce el `nombre_clase` (definido como `String`) a una clase `miClase`. Nótese que esta variable se refiere a una clase manipulada como objeto! En la segunda línea se instancia `miObjeto` a partir de `miClase`. Nótese que este proceso se puede aplicar a cualquier clase en Java dada que la instanciación es efectuada de manera totalmente anónima, todo gracias a la manipulación de la clase como si fuese un objeto. De hecho este es un ejemplo también de polimorfismo (a través del método `newInstance`).

### 5.2.9 Aspectos Adicionales

En esta sección describimos algunos aspectos adicionales que se pueden considerar básicos en el lenguaje de Java.

#### Archivos

En Java es relativamente sencillo acceder archivos. Veamos el siguiente código en Java:

```
File file = new File(dir, archivo);
BufferedReader is = new BufferedReader(new FileReader(file));
String s = is.readLine();
...
is.close();
```

Se instancia un archivo `file` de tipo `File` especificando su ubicación en el sistema, `dir`, y su nombre, `archivo`. A continuación se instancia un objeto de tipo `FileReader` el cual se conecta al archivo `file` y luego, en la misma línea, se instancia el objeto `is` de tipo `BufferedReader` que permite conectarse a través de un búfer de lectura. La llamada `is.readLine()` hace una lectura de una línea completa y la guarda en una variable `s` de tipo `String`. Luego de terminar de leer la información deseada del archivo, éste se cierra mediante la llamada `is.close()`.

La escritura es análoga a la lectura, como se muestra a continuación:

```
BufferedWriter os = new BufferedWriter(new FileWriter(file));
os.write(s);
...
os.close();
```

Se instancia un archivo `file`, como se mostró anteriormente. A continuación se instancia un objeto de tipo `FileWriter` el cual se conecta al archivo `file` y luego, en la misma línea, se instancia el objeto `os` de tipo `BufferedWriter` que permite conectarse a través de un búfer de escritura. La llamada `os.write(s)` hace una escritura de una cadena referida por la variable `s` de tipo `String`. Luego de terminar de escribir la información deseada en el archivo, éste se cierra mediante la llamada `os.close()`.

### Bases de Datos

Mostramos a continuación el manejo básico para acceder una base de datos en Java. Lo primero que generalmente se hace es checar que exista el paquete de Java que permite administrar la conexión a las bases de datos, como se muestra a continuación:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
```

La propia conexión a la base de datos se hace a través de una llamada similar a la siguiente:

```
Connection con = DriverManager.getConnection("jdbc:odbc:nombre", log, pass);
```

Lo anterior genera una conexión a una base de datos llamada *nombre* que puede accesarse opcionalmente a través de un nombre de usuario *log* y una contraseña *pass*. Esta conexión queda abierta hasta que se cierre mediante la siguiente llamada:

```
con.close();
```

Luego de esto se debe instanciar una variable de tipo `Statement` la cual es utilizada como contenedor de la llamada en *SQL*:

```
Statement stmt = con.createStatement();
```

Por ejemplo, si se quisiera hacer una consulta a una *tabla* con un nombre de usuario *log*, se generaría inicialmente una cadena *query* para guardar la llamada de *SQL* para luego ejecutarse mediante la llamada `stmt.executeQuery(query)` como se muestra a continuación:

```
String query = "SELECT * FROM tabla WHERE (login = 'log')";
ResultSet rs = stmt.executeQuery(query);
```

Esta llamada regresa un resultado `rs` de tipo `ResultSet`. De este resultado se obtiene la estructura de la tabla, o sea su *meta dato*, incluyendo el número de columnas, para luego poder leer de manera correcta sus propios datos, como se muestra a continuación:

```
ResultSetMetaData rsmd = rs.getMetaData();
int numCols = rsmd.getColumnCount();
while (rs.next()) {
    for (int i = 1; i <= numCols; i++)
        String str = rs.getString(i);
    ...
}
```

La propia lectura, en el caso de una cadena, se hace mediante la llamada `rs.getString(i)`, donde *i* identifica la columna de la tabla a partir del valor 1. Mientras `rs.next()` no sea nulo esto significa que existen resultados adicionales para seguir leyéndose.

A diferencia de la consulta, las inserciones o actualizaciones de las tablas se hacen utilizando la llamada `stmt.executeUpdate(update)` como se muestra a continuación:

```
String update = "INSERT INTO tabla ...";
int n = stmt.executeUpdate(update);
```

Nótese que la llamada `stmt.executeUpdate(update)` regresa ahora un entero correspondiente al número de *records* que fueron insertados o modificados exitosamente. El formato de las inserciones y actualizaciones corresponden a las especificaciones de *SQL* y no serán tratadas aquí en más detalle.

## Manejo de excepciones

Un aspecto de suma importancia y que es integral en los lenguajes como en Java, es el manejo de excepciones. Cuando un error ocurre en un programa, el sistema lanza una excepción que puede ser atrapada por el programa. Considerando que los errores se generan por diversas razones, incluso por razones externas a una aplicación, como al acceder el sistema operativo, es esencial que el programador tenga un manejo efectivo. Esto se hace en Java a través de las tres palabras reservadas:

- ?? `try` – define un bloque de código donde pudieran ocurrir excepciones.
- ?? `catch` – define una sección para el manejo de las excepciones (a través de la clase `Throwable` o alguna de sus subclases). Este bloque es opcional.
- ?? `finally` - código a ejecutarse como finalización del bloque, ocurran o no excepciones. Este bloque es opcional.

Por ejemplo, veamos el siguiente caso para la lectura de un archivo:

```
try {
    BufferedReader is = new BufferedReader(new FileReader(file));
    leerArchivo(is);
    is.close();
}
catch(IOException e) {
    System.out.print("Error Lectura Registro: " + e);
    System.exit(1);
}
finally {
    System.out.print("Lectura Archivo: " + file);
}
```

El bloque `try` abre la conexión al archivo de lectura `file`, como se describió anteriormente. En este bloque agregamos una llamada al método que hace la lectura, `leerArchivo` (este método se muestra más adelante). El bloque `catch` hace el manejo de la posible excepción, `IOException`, la cual debe ser pasada como argumento único dentro del bloque. En este ejemplo, se imprime el tipo de excepción y se sale del programa. El bloque `finally` siempre se ejecuta al finalizar los anteriores (a menos que se salga de la aplicación). En este ejemplo se imprime la información general del archivo del cual se está leyendo.

Java requiere que las excepciones que no son “normales” (típicamente las que provienen de la interacción con el sistema operativo) deben ser manejadas mediante la palabra `throws` en la definición del método afectado. Por ejemplo, el acceso a archivos puede ocasionar una excepción de entrada/salida que no es considerada “normal” por Java. Tal error se da cuando, por ejemplo, el archivo no existe o se trata de leer de un archivo vacío. Nótese como se debe definir un método como `leerArchivos`:

```
public void leerRegistros(BufferedReader is) throws IOException {
    String s = is.readLine();
    ...
}
```

Por suerte Java sabe cuales excepciones no son “normales” avisando al programador que debe incluirlas en la definición del método afectado. La excepción `IOException` que se incluye con el `throws` debe también aparecer en el `catch` del `try` correspondiente, sino un error de compilación ocurriría.

El manejo de excepciones es requerido para cualquier código que trate de acceder “situaciones peligrosas”, en particular es obligatorio cuando se trate de acceder elementos externos al programa, como son los archivos y las bases de datos.

## Modificadores Adicionales

Existen algunos modificadores adicionales que vale la pena mencionar:

- ?? `native` es un modificador que se puede aplicar a las declaraciones de los métodos y que significa que el método está implementado de manera nativa en C o alguna otra plataforma pero no en Java. Como un método abstracto, se agrega un punto y coma al final de su declaración.
- ?? `synchronized` es un modificador que especifica una *sección crítica* que no puede ser interrumpida en aplicaciones que usan *múltiples hilos (threads)* de control. Puede ser utilizado para métodos de instancia (objetos) o métodos de clase.

- ?? `transient` es un modificador que puede ser aplicado a campos de instancia de una clase que indica que un campo no es parte del estado persistente del objeto y por lo tanto no tiene que ser *serializado* con el objeto en tales situaciones.
- ?? `volatile` es un modificador que se puede aplicar a todos los campos y especifica que el campo es usado por hilos sincronizados y que por lo tanto el compilador no lo optimizará en su manejo, al contrario del resto de los campos.

### Finalizador

Finalmente describimos un método de finalización de clase llamado `finalize`. Análogo al constructor que inicializa el objeto, el finalizador lo finaliza. Esto no está relacionado con la recolección de basura, sino con otros aspectos, como cerrar archivos o *sockets* abiertos. Un finalizador se vería de la siguiente forma:

```
protected void finalize() throws IOException {
    if (fd != null) close();
}
```

Java nunca llama a un finalizador más de una vez para un mismo objeto. El método finalizador se llama por lo general antes de la recolección de basura. Sin embargo, Java no garantiza el orden en que ocurran estos, por lo cual puede que no se llame la recolección de basura o el finalizador. Cualquier recurso adicional aún no recolectado sería liberado al terminar el programa. Inclusive, los objetos pueden “resucitarse” guardando una referencia al propio objeto (`this`) desde el finalizador.

### 5.2.10 Aplicaciones y Applets

Existen dos maneras de estructurar un programa en Java: aplicaciones o *applets*. Ambos siguen el mismo proceso de desarrollo con Java incluyendo la gran mayoría de las facilidades que Java ofrece. La diferencia es que las aplicaciones se ejecutan como cualquier programa “normal” mientras que los *applets* están específicamente diseñados para correr en el Web a través de un *browser*.

#### Aplicaciones

Las aplicaciones utilizan un método especial para iniciar el programa: el método `main`. La aplicación más sencilla es la famosa “Hola Mundo” la cual se programaría de la siguiente manera como una aplicación en Java:

```
class ej {
    public static void main(String args[]) {
        System.out.println("Hola Mundo!");
    }
}
```

Al ejecutar el programa escribiría “Hola Mundo”.

El método `main` debe aparecer en toda aplicación y realmente no afecta a que clase se le asigne el método, ya que este no tiene acceso a las estructuras internas de la clase, además de ser accesado sólo internamente por Java. Por ejemplo,

```
public class Persona {
    public static void main(String args[]) {
        for (int i = 0; i < args.length; i++)
            System.out.print(args[i] + " ");
        System.out.print("\n");
        System.exit(0);
    }
}
```

Nótese el argumento “args” de “main” que recuerdan cierta similitud a “argc” y “argv”, sólo que integrándolos en un sólo.

#### Applets

Tomamos ahora el programa anterior de “Hola Mundo” el cual se programaría de la siguiente manera como un applet en Java:

```
public class ej extends Applet {
    public void paint(Graphics g){
        g.drawString("Hola Mundo!", 25, 25);
    }
}
```

En lugar del método `main`, un applet requiere una clase que herede de `Applet` y que sobrescriba el método `paint` para poder desplegar textos o gráficas en la pantalla. (En la siguiente sección extenderemos más sobre el tema de interfaces gráficas.) Todo applet requiere de una página “html” para su ejecución. La página “html”, por ejemplo “ej.html”, que va a desplegar el applet requeriría de la siguiente línea para poder ejecutarse correctamente:

```
<applet code=ej.class width=200 height=200></applet>
```

El archivo “html” puede ejecutarse en un browser o mediante la aplicación `appletviewer` de la siguiente forma:

```
appletviewer ej.html
```

A diferencia del método `main`, el paso de argumentos iniciales es a través de parámetros del archivo “html”.

### 5.3 Interfaces Gráficas de Usuario

Programar en Java sin utilizar *Interfaces Gráficas de Usuario* (GUI – por sus siglas en inglés) es no aprovechar uno de los aspectos más importantes que ofrece la programación y en particular Java. Comenzamos describiendo despliegues gráficos sencillos para luego explicar el manejo de ventanas, textos, botones y paneles en Java a través de la biblioteca AWT.

#### 5.3.1 Despliegues Gráficos

Antes (GUI – por sus siglas en inglés) es no aprovechar uno de los aspectos más importantes que ofrece la programación y en particular Java.

#### Ocho Reinas

El siguiente ejercicio se conoce como el juego de las “ocho reinas” y es muy interesante porque es un ejemplo de lo compacto que puede ser un programa, en especial aquellos que utilizan mucho la recursividad, como es el caso con este ejercicio. El problema tiene origen en el ajedrez, donde una reina puede atacar a cualquier otra pieza que quede en la misma fila, en la misma columna o en una diagonal. El problema de las ocho reinas consiste simplemente en colocar ocho reinas en un tablero de ajedrez, en forma tal que ninguna reina pueda atacar a ninguna otra reina. En la Figura 5.25 se muestra un ejemplo de como se vería un tablero de tal manera. La solución no es la única y depende de las condiciones de inicio.

q									
				q					
								q	
						q			
		q							
	q						q		
			q						

Figure 5.25 Problema de las Ocho Reinas.

La esencia de la solución orientada a objetos, propuesta aquí, consiste en crear las reinas y otorgarles ciertos poderes para que ellas mismas puedan descubrir la solución. Para lograr esta solución, la primera observación que hay que hacer es que en ninguna de las soluciones pueden quedar dos reinas en la misma columna y, en consecuencia, ninguna columna puede estar vacía. Por lo tanto, se puede asignar inicialmente una columna para cada reina, y reducir así el problema para dar a cada reina la tarea de encontrar una fila apropiada. Se tiene una solución al acertijo de las ocho reinas cuando todas las reinas guardan cierta relación entre sí. De esta manera, está claro que las reinas necesitan comunicarse. Dado esto, se puede hacer una segunda observación. Cada reina necesita enviar mensajes sólo a una de sus vecinas. En forma más precisa, cada reina necesita saber sólo acerca de la reina que está inmediatamente a su izquierda. Así, los datos para cada reina constan de tres valores: un valor de columna, que es inmutable (se establece una vez y nunca se altera); un valor de fila, que se altera en la búsqueda de la solución, y la reina vecina a su izquierda inmediata.

Se define una “solución aceptable para una columna  $i$ ” como la configuración de las columnas 1 hasta  $i-1$  en la cual ninguna reina puede atacar a otra reina en tales columnas. A cada reina se le encargará que encuentre soluciones aceptables entre ella y su vecina a la izquierda. Se encuentra una solución al acertijo en su conjunto pidiendo a la reina del extremo derecho que descubra una solución aceptable. El diagrama de clases para la clase `Reina` se muestra en la Figura 5.26.

Reina
fila columna vecina
primero siguiente puedeAtacar pruebaOAvanza

Figure 5.26 Clase Reina para el problema de las Ocho Reinas.

Los atributos y métodos son los siguientes:

- ?? fila - número de fila actual (cambia)
- ?? columna - número de columna (fija)
- ?? vecina - vecina de la izquierda (fija)
- ?? primera - inicia fila, luego encuentra la primera solución aceptable para sí misma y vecinas
- ?? siguiente - avanza fila y encuentra la siguiente solución aceptable
- ?? puedeAtacar - ve si una posición puede ser atacada por sí misma o por vecinas
- ?? pruebaOAvanza - comparte código común para la verificación de posición.

La estructura del código en Java es la siguiente (sin especificar ni los argumentos ni la implementación de los métodos):

```
class Reina {
    private int columna, fila;
    private Reina vecina;
    public Reina () {
        public boolean primera() { ... }
        private boolean puedeAtacar() { ... }
        private boolean pruebaOAvanza() { ... }
        private boolean siguiente() { ... }
    }
}
```

Empecemos por lo más general hasta terminar con lo más detallado tratando de seguir la lógica lo más ordenado posible, algo que se complica dada la recursividad. Para ello definimos un método main:

```
public static void main (String args[]){
    Reina ultimaReina = null;
    for (int i=1; i<= 8; i++)
        ultimaReina = new Reina(i, ultimaReina);
    if ((ultimaReina != null) && ultimaReina.primera())
        ultimaReina.imprime();
}
```

Se tiene un ciclo “for” donde se instancia una nueva reina y se la asigna a una columna particular junto con la referencia de su vecina a la izquierda correspondiente a la última reina instanciada. Una vez se hayan ubicado las ocho reinas en sus columnas correspondientes, con sus referencias entre si, se inicializa el algoritmo mediante “ultimaReina.primera()”. La última línea del main imprime el resultado final.

En cuestión de métodos, el primero que se requiere es el constructor que ubica a las reinas en su columna particular, aunque fuera del tablero (fila 0), junto a sus referencias, como se puede ver a continuación:

```
public Reina (int c, Reina r ) {
    columna=c;
    fila=0;
    vecina=r;
}
```

Luego se ejecuta primera el cual inicializa a todas las reinas a la fila 1 de manera recursiva comenzando desde la última instanciada. Gracias a la recursión, llegamos de “ida” hacia la izquierda hasta la primera reina, la de la columna 1, donde paramos ya que esta no tiene vecina a la izquierda. Una vez que se llega al extremos izquierdo comenzamos a avanzar de “regreso” hacia la derecha, una reina a la vez, haciendo la prueba, pruebaOAvanza, para cada reina para ver si esta tiene conflictos con el resto de sus vecinas a la izquierda. Este método se muestra a continuación:

```
public boolean primera() {
    fila = 1;
    if (vecina != null) {
```

```

        if (vecina.primera())
            return pruebaOAvanza();
    }
    return true;
}

```

El siguiente método en la lógica es `pruebaOAvanza` el cual verifica si una reina puede atacar a su vecina a la izquierda, mediante el método `puedeAtacar`. En caso de que si la pueda atacar, la reina debe avanzar a la siguiente fila donde la prueba se volverá a realizar. Este método se describe a continuación:

```

private boolean pruebaOAvanza() {
    if (vecina != null) {
        if (vecina.puedeAtacar(fila, columna))
            return siguiente();
    }
    return true;
}

```

El procedimiento `puedeAtacar` compara la posición de la fila de la reina actual con su vecina a la izquierda, empleando el hecho de que, para un movimiento en diagonal, las diferencias en las filas deben ser iguales a las diferencias en las columnas. Como el algoritmo es recursivo, cada reina debe compararse también con todo el resto de las reinas hasta el extremo izquierdo, algo que se hace mediante el `puedeAtacar` interno. El método se describe a continuación:

```

private boolean puedeAtacar(int f, int c) {
    int dc;
    dc=columna-c;
    if (f==fila)
        return true;
    if ( (fila+dc==f) || (fila-dc==f) )
        return true;
    if (vecina !=null)
        return vecina.puedeAtacar(f,c);
    return false;
}

```

Tras encontrar una solución para las primeras columnas, la reina trata cada fila en orden, empezando con la fila 1, por medio del procedimiento `puedeAtacar`. Resulta una de dos cosas: se encuentra una solución aceptable o la reina intenta todas las posiciones sin encontrar una solución. En el último caso, la reina pide a su vecina que encuentre otra solución aceptable, y la reina empieza de nuevo a probar valores de fila potenciales. Cuando se pide a una reina que encuentre otra solución, esta simplemente incrementa su número de fila y verifica con sus vecinas, suponiendo que no se encuentre ya en la última fila. Si ya está en la última fila, la reina no tiene más remedio que pedir a su vecina una nueva solución y empezar la búsqueda una vez más desde la primera fila. Esto se implementa mediante el método siguiente:

```

private boolean siguiente() {
    if (fila==8) {
        if (vecina != null) {
            if (!vecina.siguiente())
                return false;
            fila=0;
        }
    }
    fila++;
    return pruebaOAvanza();
}

```

Aunque no lo crea el algoritmo ha sido resuelto. Sólo nos falta imprimir el resultado, lo cual se hace con el siguiente método:

```

public void imprime() {
    if (vecina != null)
        vecina.imprime();
    System.out.println("Reina: "+ columna + " "+ fila);
}

```

La impresión del resultado final sería:



```

Reina: 1 1
Reina: 2 5
Reina: 3 8
Reina: 4 6
Reina: 5 3
Reina: 6 7
Reina: 7 2
Reina: 8 4

```

Esta salida no es demasiado amigable por lo cual agreguemos un pequeño despliegue gráfico y corramos el programa como una Applet. Para tal objetivo, definimos un método `despliega`, similar en lógica a `imprime`:

```

public void despliega (Graphics g){
    if (vecina != null)
        vecina.despliega (g);
    g.setColor(Color.gray);
    g.fillOval(((fila - 1) * 50)+10, ((columna - 1) * 50)+10, 25, 25);
}

```

Definimos una nueva clase `ReinaApplet` que herede del `Applet` (no queremos múltiples applets). A esta clase le asignamos un método `init` análogo al `main`, y otro método `paint` que despliegue el resultado final, el tablero más las reinas. Esto se hace de la siguiente manera:

```

public class ReinaApplet extends Applet {
    private Reina ultimaReina;
    public void init (){
        ultimaReina = null;
        for (int i = 1; i <= 8; i++)
            ultimaReina = new Reina (i, ultimaReina);
    }
    public void paint (Graphics g) {
        for (int i = 0; i < 8; i+=2)
            for (int j = 0; j < 8; j+=2) {
                g.fillRect(50 * (j+1), 50 * i, 50, 50);
                g.fillRect(50 * j, 50 * (i+1), 50, 50);
            }
        if ((ultimaReina != null) && ultimaReina.primera ())
            ultimaReina.despliega (g);
    }
}

```

El resultado final gráfico es el que se muestra en la Figura 5.27.

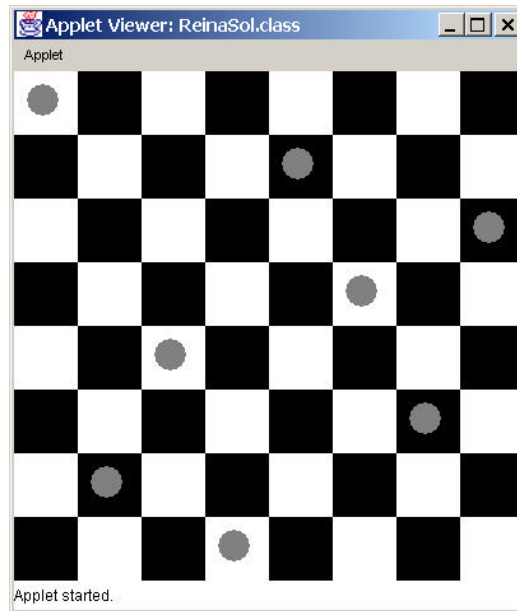


Figura 5.27 Applet con despliegue final para el problema de las ocho reinas.

### Dominó

Presentamos en esta sección un ejemplo de juego del dominó. Este juego consiste de 28 fichas de las cuales 7 son entregadas inicialmente a cada jugador. Cada jugador tratará de ubicar una de sus fichas en el tablero de manera que uno de sus valores coincida con alguno de los dos valores en los extremos del tablero. El turno pasa de jugador en jugador hasta que alguno de los dos jugadores agote sus fichas o que ninguno de los dos pueda continuar. El ganador será aquel que termine sus fichas o que tenga un menor puntaje al finaliza el juego. En el caso de no poder ubicar ninguna de sus fichas, el jugador en turno deberá obtener una nueva ficha del montón de fichas sobrantes. Si aún no es posible ubicar esta nueva ficha se deberá obtener otra adicional hasta poder ubicarla en el tablero, o hasta agotar el montón (la sopa) en cuyo caso pasará el turno al siguiente jugador. El objetivo de este ejercicio es implementar en Java un programa que simule el juego de Dominó entre 2 jugadores. El programa deberá contener de manera general las clases, atributos y métodos, descritos en la Figura 5.28.

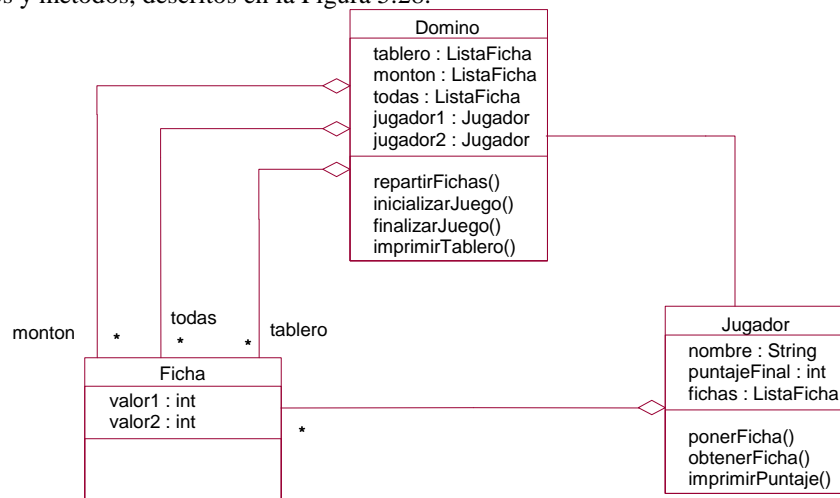


Figura 5.28 Diagrama de clase para el juego del Dominó.

Los detalles de este ejercicio lo dejamos al lector que los desarrolle. Sin embargo mostramos gráficamente la salida del programa durante su ejecución. La obtención e inicialización de fichas deberá ser aleatoria. El inicio, con las piezas repartidas, se muestra en la Figura 5.29.

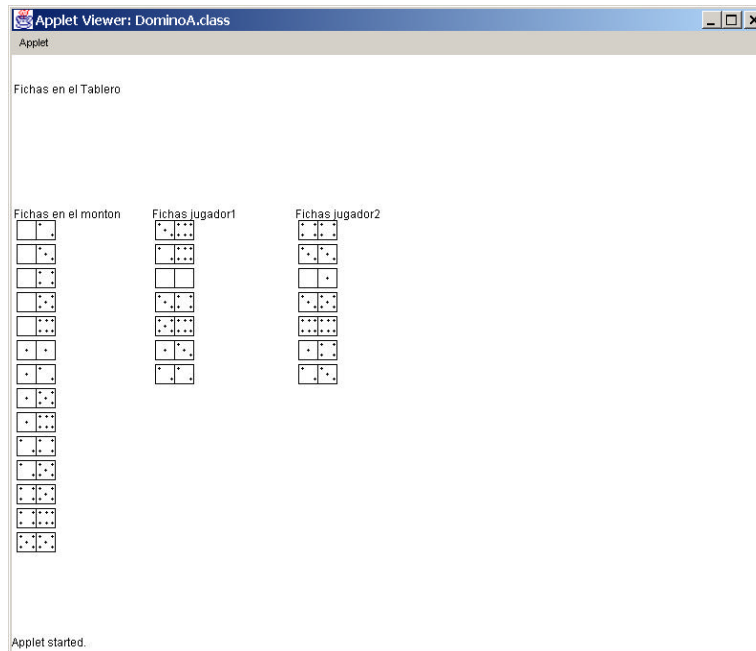


Figura 5.29 Pantalla del juego del Dominó durante el inicio.

Cada jugador puede ubicar de manera arbitraria sus fichas en el caso de tener múltiples opciones para hacerlo, como se muestra en la Figura 5.30.

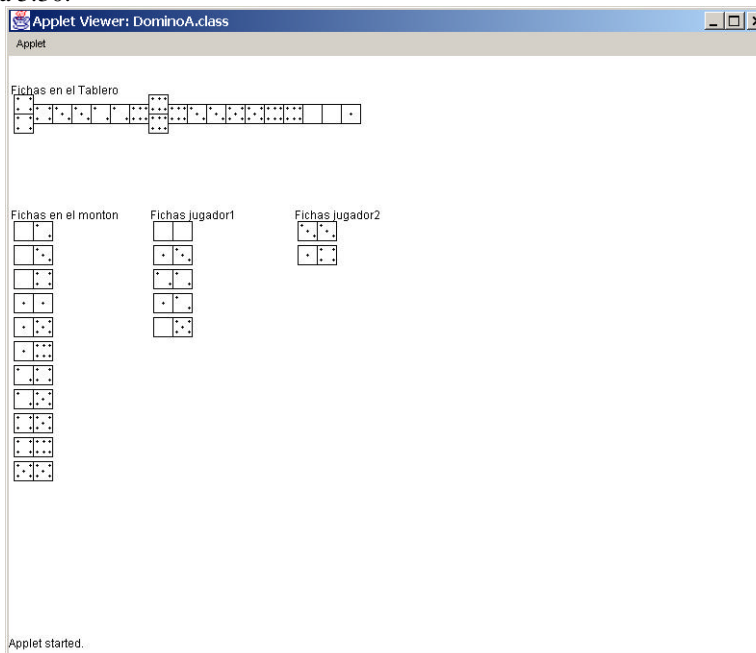


Figura 5.30 Pantalla del juego del Dominó durante su desarrollo.

El juego se acaba cuando ya no hay más fichas para ubicar, como se muestra en la Figura 5.31.

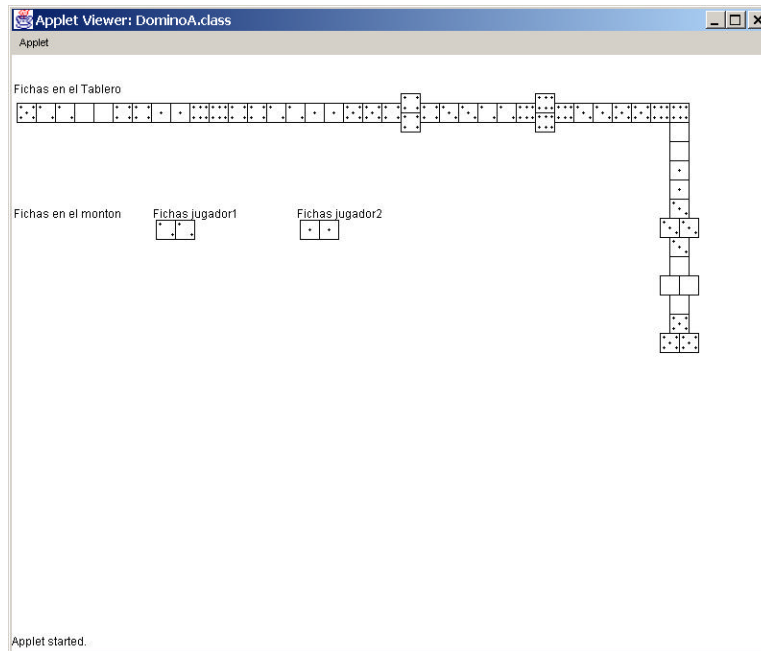


Figura 5.31 Pantalla del juego del Dominó al final.

### 5.3.2 Ventanas, Textos, Botones y Paneles

En esta sección introducimos rápidamente el diseño de GUIs en Java mediante la biblioteca *AWT*, la primera biblioteca gráfica de Java como parte de Java 1, y que tiene la gran ventaja de que, a diferencia de la biblioteca *Swing* y en general Java 2, esta biblioteca corre en la actualidad en todos los browsers, incluso los de Microsoft, sin necesidad de un *plug-in* especial. El ejemplo a desarrollarse en esta sección es muy importante ya que servirá de base para prototipos posteriores en el libro.

Todo sistema de ventanas requiere alguna ventana donde desplegar la información. Existen dos filosofía separadas aunque relacionadas en Java que afectarán el diseño como ya hemos mencionado anteriormente: aplicaciones y applets. Las aplicaciones requieren de un *Frame* (*marco*) donde desplegar y permitir la interacción con el usuario, mientras que un applet puede directamente hacer esto en la pantalla del browser o mediante nuevos marcos de manera similar a las aplicaciones. Empecemos por lo más básico y mostremos el uso de los marcos a través de la siguiente clase *InterfaceUsuario*:

```
class InterfaceUsuario extends Frame
```

Esta clase heredará todas las características de un marco para manejarse como una ventana. Antes de proseguir, es necesario comprender que toda aplicación con ventanas requiere un manejo de eventos de entradas y salidas. Esto significa que la aplicación antes de hacer nada debe de alguna manera especificar como controlará eventos generados por el teclado y en especial el ratón. Java define dos interfaces muy importantes en *AWT* que son *WindowListener* y *ActionListener*, donde *WindowListener* define los métodos relacionados con el manejo de eventos para una ventana, como abrir y cerrar, mientras que *ActionListener* define los métodos para manejar eventos dentro de la ventana, como apretar un botón. De tal manera, es necesario que la clase *InterfaceUsuario* implemente estas dos interfaces si se tiene planeado manejar estos tipos de eventos. Por lo tanto, extendemos la definición de la clase *InterfaceUsuario* de la siguiente manera:

```
class InterfaceUsuario extends Frame implements WindowListener,
ActionListener
```

Dado que las interfaces deben tener su métodos sobrescritos, hagamos esto con los siguiente métodos de *WindowListener* inicialmente:

```
public void windowClosed(WindowEvent event) {}
public void windowDeiconified(WindowEvent event) {}
public void windowIconified(WindowEvent event) {}
public void windowActivated(WindowEvent event) {}
public void windowDeactivated(WindowEvent event) {}
public void windowOpened(WindowEvent event) {}
public void windowClosing(WindowEvent event) {
```

```
System.exit(0); }
```

Estos son todos los métodos definidos por la interface `WindowListener`:

- ?? `windowClosed` para el manejo de eventos a partir de una ventana cerrada.
- ?? `windowDeiconified` para el manejo de eventos a partir de una ventana no iconificada.
- ?? `windowIconified` para el manejo de eventos a partir de una ventana iconificada.
- ?? `windowActivated` para el manejo de eventos a partir de una ventana activada.
- ?? `windowDeactivated` para el manejo de eventos a partir de una ventana desactivada.
- ?? `windowOpened` para el manejo de eventos a partir de una ventana abierta.
- ?? `windowClosing` para el manejo de eventos en el momento que se cierra una ventana.

Todos estos métodos deben sobrescribirse aunque queden vacíos, como es el caso con la mayoría de los anteriores para nuestro ejemplo. El único que realmente hemos sobrescrito es `windowClosing` para permitir salir de la aplicación cuando este evento ocurra.. (Existen alternativas para sobrescribir únicamente los métodos deseados utilizando *adaptadores* en lugar de *interfaces*, algo que no mostraremos en este libro.)

La sobrescritura de la interface `ActionListener` es mucho más importante para las ventanas, ya que a través del método `actionPerformed` se debe especificar que hacer cuando, por ejemplo, se presiona algún botón dentro de la ventana. El siguiente método muestra la sobrescritura de `actionPerformed`, imprimiendo el evento ocurrido

```
public void actionPerformed(ActionEvent event) {
    System.out.println("Action: "+event.getActionCommand());
}
```

Esta es la lógica básica del manejo de eventos para un marco. Para completar la funcionalidad necesitamos inicializar la clase `InterfaceUsuario` y definir alguna pantalla para desplegar. Para ello definimos el siguiente constructor:

```
public InterfaceUsuario() {
    setSize(800,600);
    setBackground(Color.lightGray);
    addWindowListener(this);
    pantalla = new PantallaPrincipal(this);
    desplegarPantalla(pantalla);
}
```

Describamos las llamadas dentro de este constructor, las cuales son la mayoría opcionales:

- ?? `setSize` define el tamaño del marco.
- ?? `setBackground` asigna un color de fondo de manera opcional.
- ?? `addWindowListener` registra el marco (`this`) con el administrador de ventanas de Java para que podamos manejar los eventos. Este método es necesario.
- ?? `PantallaPrincipal` instancia la pantalla a ser desplegada, algo que veremos a continuación. Nótese que la variable `pantalla` la definimos como un atributo de la clase `InterfaceUsuario`, “`private Pantalla pantalla;`”, algo que explicaremos también más adelante junto con la clase `Pantalla`.
- ?? `desplegarPantalla` despliega la pantalla recién instanciada.

Antes de mostrar los detalles de `PantallaPrincipal`, veamos como desplegaríamos una pantalla mediante el método `show` definido por la clase `Frame`:

```
protected void desplegarPantalla(Pantalla p) {
    show();
}
```

Nótese que no estamos utilizando el argumento de tipo `Pantalla` aún. Esto lo remediamos en un momento para cuando dejemos más clara la lógica que utilizaremos mediante la explicación de la clase `PantallaPrincipal`.

Antes de hacer eso mostremos el método `main` para instanciar la clase `InterfaceUsuario`:

```
public static void main(String[] args) {
    System.out.println("Starting System...");
    InterfaceUsuario iu = new InterfaceUsuario();
}
```

Si quisiéramos instanciar el marco bajo control de un applet, algo que también es aceptable, definiríamos la siguiente clase `InterfaceUsuarioApplet` que hereda de la clase `Applet` y sobrescribiendo el método `init`, en lugar del método `main`:

```
public class InterfaceUsuarioApplet extends Applet
{
```

```

public void init() {
    showStatus("Starting System...");
    InterfaceUsuario iu = new InterfaceUsuario();
}
}

```

Otro comentario es que definiremos las gran mayoría de los métodos como `protected` dado que son accedados dentro de un mismo paquete. Sólo los métodos sobrescritos de las interfaces deben definirse como públicos. También definiremos los constructores como públicos aunque algunos pudieran también definirse como `protected` si son los objetos instanciados dentro del mismo paquete. y los constructores deben Como ejemplo vamos a crear un `PantallaPrincipal` que genere el despliegue que se muestra en la Figura 5.32.

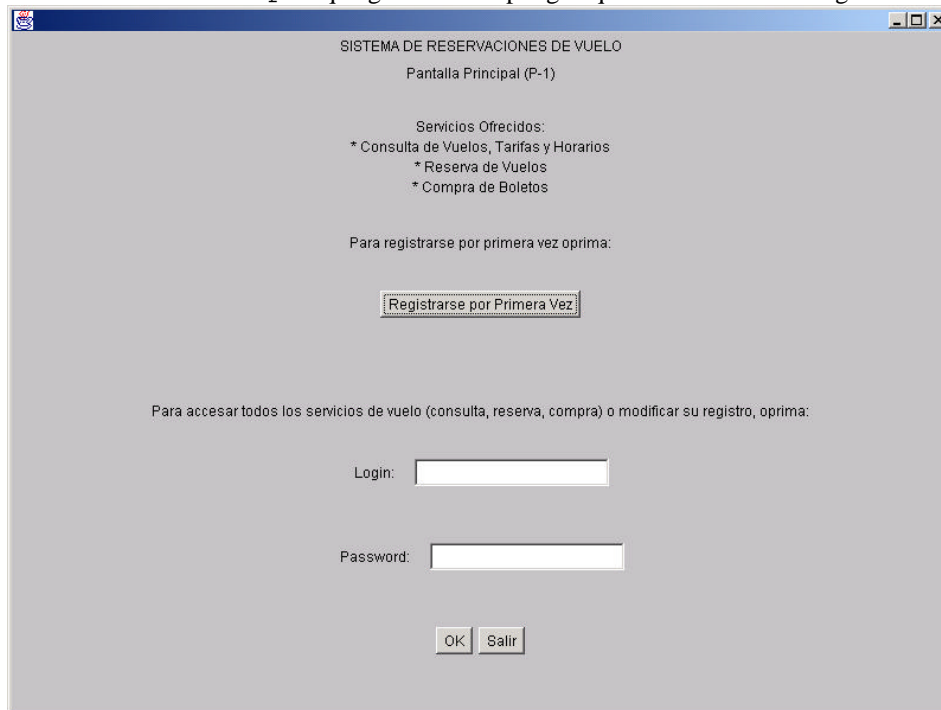


Figura 5.32 Ejemplo de marco desplegando `PantallaPrincipal`.

Para simplificar el diseño de una de estas pantallas, es posible dividir las en secciones lógicas llamadas *paneles* que visualmente no afectan la presentación de la pantalla, pero son una buena guía para el diseñador. En nuestro caso dividiremos la pantalla en paneles horizontales que contengan los diversos elementos de la pantalla, o sea, textos y botones.

Dado que por lo general se desean desplegar múltiples pantallas, definamos una superclase `Pantalla`, como algunos de los lectores ya se habrán imaginado, de la siguiente manera:

```

class Pantalla {
    protected InterfaceUsuario interfaceUsuario;
    protected Panel panel;
    protected Button boton;
    protected Vector paneles, botones;
}

```

Lo primero que haremos dentro de esta clase es definir un atributo de tipo `InterfaceUsuario`, el cual guardará la información sobre el la clase encargada de administrar el despliegue. Los tributos de tipo `Panel` y `Button` para que sirvan para referencias en cualquiera de las subclases de `Pantalla`. El aspecto más importante para esta clase es que definiremos dos arreglos de tamaño dinámico, mediante la clase `Vector`, para guardar la lista de todos los paneles y botones que sean instanciados en las diversas pantallas, ya que estos requieren un manejo especial, y los arreglos facilitarán su manipulación como veremos más adelante.

Definimos el constructor para la clase `Pantalla` de manera que reciba una referencia a guardarse de la clase `InterfaceUsuario`.

```

public Pantalla (InterfaceUsuario ui) {
    interfaceUsuario ui;
}

```

```

        inicializarPantalla();
        crearPantalla();
    }

```

Adicionalmente, este constructor inicializará la pantalla mediante la llamada `crearPantalla`, la cual está sobrescrita por las pantallas particulares. En la superclase, el método se puede definir como abstracto y protegido.

```
protected abstract void crearPantalla ();
```

Este método es el corazón de la lógica de despliegue. Antes de ello se reinician los vectores para los paneles y botones que están definidos por el método `inicializarPantalla`.

```
protected void inicializarPantalla() {
    paneles = new Vector();
    botones = new Vector();
}

```

Definamos ahora la clase `PantallaPrincipal` como subclase de `Pantalla`:

```
class PantallaPrincipal extends Pantalla
```

El constructor de `PantallaPrincipal` llamará al constructor de la superclase mediante la llamada `super`, a la cual pasará el parámetro `ui` de tipo `InterfaceUsuario`.

```
public PantallaPrincipal (InterfaceUsuario ui) {
    super(ui);
}

```

El diseño de las pantallas será utilizando paneles, o sea secciones de la pantalla. Esto se guardará dentro del método `crearPantalla`, en este caso de la clase `PantallaPrincipal`, correspondiente a la Figura 5.30. Esta clase no tiene que ser pública ya que se llama dentro del constructor de la superclase. Dado que existe una sobrescritura del método, este debe definirse como protegido.

```
protected void crearPantalla ()
```

El primer panel contiene el título de la pantalla, como se muestra a continuación:

```
panel = new Panel();
panel.setLayout(new GridLayout(2,1));
panel.add(new Label("SISTEMA DE RESERVACIONES DE VUELO",
    Label.CENTER));
panel.add(new Label("Pantalla Principal (P-1)", Label.CENTER));
paneles.addElement(panel);

```

Luego de instanciar el objeto de tipo `Panel`, se le asigna a este un administrador para la organización interna del panel, por ejemplo `GridLayout` que define en este caso una cuadrícula de 2x1. Luego añadimos los elementos del panel, en este caso un título (`Label`) que corresponde a un texto que no es modificable el cual es centrado dentro de la cuadrícula. Una vez completado el panel, lo agregamos a la lista de paneles.

El siguiente panel contiene 4 filas y una sola columna, donde vamos a insertar cuatro líneas de texto como se ve a continuación:

```
panel = new Panel();
panel.setLayout(new GridLayout(4,1));
panel.add(new Label("Servicios Ofrecidos:", Label.CENTER));
panel.add(new Label("* Consulta de Vuelos, Tarifas y Horarios",
    Label.CENTER));
panel.add(new Label("* Reserva de Vuelos", Label.CENTER));
panel.add(new Label("* Compra de Boletos", Label.CENTER));
paneles.addElement(panel);

```

Nuevamente agregamos el panel a la lista de paneles. El siguiente panel es similar al primero y agrega una etiqueta, como se ve a continuación:

```
panel = new Panel();
panel.setLayout(new GridLayout(1,1));
panel.add(new Label("Para registrarse por primera vez oprima:",
    Label.CENTER));
paneles.addElement(panel);

```

En el siguiente panel hacemos algo diferente. Instanciamos un botón (`Button`), al cual le ponemos como etiqueta "Registrarse por Primera Vez", como se ve a continuación:

```
panel = new Panel();
panel.setLayout(new GridLayout(1,1));
boton = new Button ("Registrarse por Primera Vez");
botones.addElement(boton);

```

```

panel.add(boton);
paneles.addElement(panel);

```

Además de agregar el panel a la lista de paneles, agregamos el botón a la lista de botones. El siguiente panel es similar al primero y agrega una etiqueta, como se ve a continuación:

```

panel = new Panel();
panel.setLayout(new GridLayout(1,1));
panel.add(new Label("Para acceder todos los servicios de vuelo
    (consulta, reserva, compra) o modificar su registro, oprima:",
    Label.CENTER));
paneles.addElement(panel);

```

En el siguiente panel también hace algo diferente. Instanciamos un campo de texto (TextField), además de agregarle una etiqueta "Login:", como se ve a continuación:

```

panel = new Panel();
panel.setLayout(new GridLayout(1,1));
panel.add(new Label("Login:", Label.LEFT));
panel.add(new TextField(20));
paneles.addElement(panel);

```

En el siguiente panel instanciamos un campo de texto adicional (TextField), que además de agregarle una etiqueta "Password:", como se ve a continuación:

```

panel = new Panel();
panel.setLayout(new GridLayout(1,1));
panel.add(new Label("Password:"));
panel.add(new TextField(20));
paneles.addElement(panel);

```

En el último panel instanciamos dos botones adicionales, "OK" y "Salir", como se ve a continuación:

```

panel = new Panel();
panel.setLayout(new GridLayout(1,1));
boton = new Button("OK");
botones.addElement(boton);
panel.add(boton);
boton = new Button("Salir");
botones.addElement(boton);
panel.add(boton);
paneles.addElement(panel);

```

Ahora viene la gran pregunta: ¿cómo hacemos para desplegar todo esto y manejar de manera adecuada los eventos relacionando? Para ello regresamos al método desplegarPantalla inicialmente definido para la clase InterfaceUsuario y le agregamos algunas líneas adicionales antes del show, como se ve a continuación:

```

protected void desplegarPantalla(Pantalla p) {
    if (pantalla != null)
        pantalla.borrarPantalla();
    if (p != null)
        pantalla = p;
    if (pantalla != null)
        pantalla.desplegarPantalla();
    show();
}

```

Dado que estamos en proceso de desplegar una nueva pantalla, lo primero que tenemos que hacer es borrar la anterior, tanto paneles como registro de botones. Nótese que se guarda en p la nueva ventana mientras que debemos borrar la pantalla anterior. Eso lo hacemos a través del método borrarPantalla dentro de la clase Pantalla, algo que describiremos en un momento. Utilizamos siempre un "if" para asegurar que no existan valores nulos. Luego de borrar la pantalla actual, cambiamos el valor del atributo pantalla al de la nueva pantalla la cual será desplegada mediante el método desplegarPantalla. El método borrarPantalla se muestra a continuación, el cual puede definirse como protegido por ser llamado dentro del mismo paquete:

```

protected void borrarPantalla() {
    interfaceUsuario.removeAll();
    int bs = botones.size();
    for (int i = 0; i < bs; i++)
        if ((boton = (Button)botones.elementAt(i)) != null)
            boton.removeActionListener(interfaceUsuario);
}

```



```
}

```

El método `removeAll` borra todo lo que hay en la pantalla, mientras que `removeActionListener` borra el registro para manejo de eventos de los botones de la pantalla anterior.

El despliegue de la pantalla se hace mediante el método `desplegarPantalla` perteneciente a la clase `Pantalla`, como se ve a continuación:

```
protected void desplegarPantalla() {
    System.out.println("Desplegando: "+ this);
    int ps = paneles.size();
    interfaceUsuario.setLayout(new GridLayout(ps,1));
    for (int i = 0; i < ps; i++)
        interfaceUsuario.add((Panel)paneles.elementAt(i));
    int bs = botones.size();
    for (int i = 0; i < bs; i++)
        if ((boton = (Button)botones.elementAt(i)) != null)
            boton.addActionListener(interfaceUsuario);
}
```

Se obtiene el número de paneles, `ps`, de la lista de `paneles`, para el cual se le solicita a la `interfaceUsuario` que organice la pantalla en una cuadrícula mediante `GridLayout`, que conste en un número de filas `ps` y una sola columna. De tal manera se agrega a la `interfaceUsuario`, cada uno de los paneles mediante la llamada `interfaceUsuario.add`. Con esto es suficiente para desplegar los paneles junto con todos los campos definidos para cada uno de ellos anteriormente, en el momento se ejecuta la llamada `show`. Sin embargo, falta algo importante, registrar los botones con el manejador de eventos. Para hacer esto obtenemos el número de botones `bs`. Luego, obtenemos cada uno de ellos de la lista y lo registramos con el sistema mediante la llamada `boton.addActionListener`. De tal manera ya podemos desplegar la `PantallaPrincipal` con todos sus elementos, incluyendo botones registrados para luego saber cual de ellos fue oprimido.

Ahora viene el siguiente paso luego de desplegar la pantalla anterior. El usuario puede llenar los campos de texto, que no generan ningún evento, y desplegar alguno de los tres botones. ¿Qué hace el sistema en el momento que se oprima alguno de estos botones? Recordemos el método `actionPerformed` de la clase `InterfaceUsuario` definida precisamente para ello, para la cual agregamos una nueva línea que será la encargada de manejar el evento llamando a `manejarEventos` a partir de cada pantalla desplegada:

```
public void actionPerformed(ActionEvent event) {
    System.out.println("Action: "+event.getActionCommand());
    pantalla.manejarEvento(event.getActionCommand());
}
```

Para que esto funcione primero debemos definir el método `manejarEvento` dentro de la clase `Pantalla` (recuerden la explicación de *polimorfismo*) y lo hacemos de manera abstracta:

```
protected abstract Pantalla manejarEvento(String str);
```

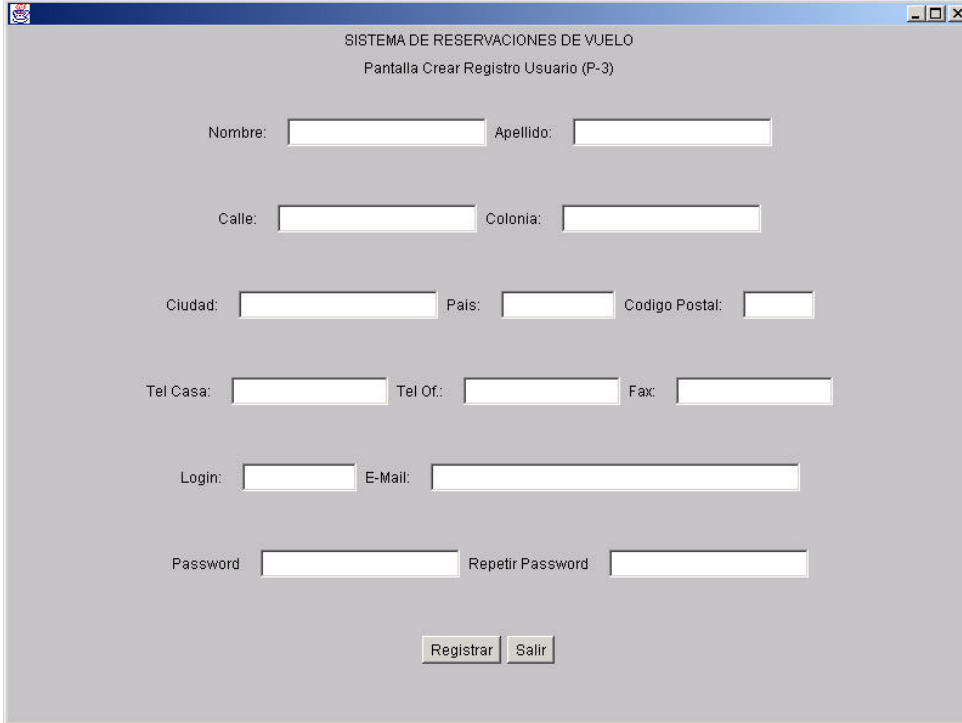
Luego sobrescribimos el método `manejarEvento` dentro de la clase `PantallaPrincipal` de manera que según se oprima un botón la pantalla instanciará la siguiente a desplegarse. Esto se hace comparando el nombre del botón presionado contra las distintas posibilidades que se destacan en los siguientes `if`:

```
protected Pantalla manejarEvento(String str) {
    if (str.equals("Registrarse por Primera Vez")) {
        if (pantallaCrearRegUsuario == null)
            pantallaCrearRegUsuario = new PantallaCrearRegUsuario(this);
        return pantallaCrearRegUsuario;
    }
    else if (str.equals("OK")) {
        if (pantallaServicio == null)
            pantallaServicio = new PantallaServicio(this);
        return pantallaServicio;
    }
    else if (str.equals("Salir")) {
        System.exit(0);
    }
    else
        System.out.println("Error en PantallaPrincipal: "+str);
    return this;
}
```

Por ejemplo, si el usuario presiona el botón "Registrarse por Primera Vez" (nótese que la comparación de cadenas se hace mediante el método "equals") entonces se instancia un pantalla de tipo `PantallaCrearRegUsuario`, que explicaremos en un momento. Si es un "OK" se solicita la pantalla `PantallaServicio` y si es "Salir" simplemente se sale del programa. Antes de definir nuestra siguiente pantalla, veamos que ocurre con la lógica principal. La llamada `manejarEvento` fue hecha desde `actionPerformed` en la clase `InterfaceUsuario`. Extendamos este método con una nueva versión de la siguiente manera:

```
public void actionPerformed(ActionEvent event) {  
    System.out.println("Action: "+event.getActionCommand());  
    Pantalla p = pantalla.manejarEvento(event.getActionCommand());  
    desplegarPantalla(p);  
}
```

La siguiente llamada es `desplegarPantalla` y el ciclo se completa. En otras palabras volvimos al inicio de nuestra lógica. La pantalla `PantallaCrearRegUsuario` se muestra en la Figura 5.33.



The screenshot shows a Java Swing window with a title bar that reads "SISTEMA DE RESERVACIONES DE VUELO" and "Pantalla Crear Registro Usuario (P-3)". The window has a light gray background and contains several text input fields arranged in a form. The fields are labeled as follows: "Nombre:" and "Apellido:" (two fields side-by-side), "Calle:" and "Colonia:" (two fields side-by-side), "Ciudad:", "Pais:", and "Codigo Postal:" (three fields side-by-side), "Tel Casa:", "Tel Of:", and "Fax:" (three fields side-by-side), "Login:" and "E-Mail:" (two fields side-by-side), and "Password" and "Repetir Password" (two fields side-by-side). At the bottom center of the window, there are two buttons: "Registrar" and "Salir".

Figura 5.33. Ejemplo de marco desplegando `PantallaCrearRegUsuario`.

En la Figura 5.34 se muestran las clases principales, junto con sus atributos y métodos principales, para el ejemplo de las pantallas mostrado aquí. Nótese que en el diagrama se muestra la notación de implementar para las interfaces como una herencia "punteada" a partir de clases abstractas.

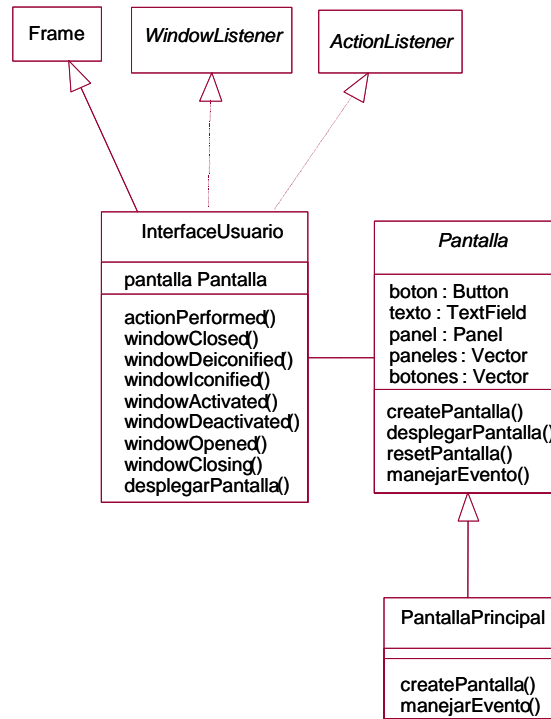


Figura 5.34 Diagrama de clases para el ejemplo de las pantallas.

Como veremos en el siguiente capítulo, estas pantallas y esta lógica nos servirá para crear el prototipo inicial del sistema de reservaciones de vuelos que utilizaremos a lo largo del libro.

