ESTRUCTURA DE DATOS CON ORIENTACIÓN A OBJETOS

Introducción al diseño y Programación Orientada a Objetos.

1.1 Diseño Orientado a Objetos.

Metodología enfocada a la solución de problemas complejos.

Complejidad del software.

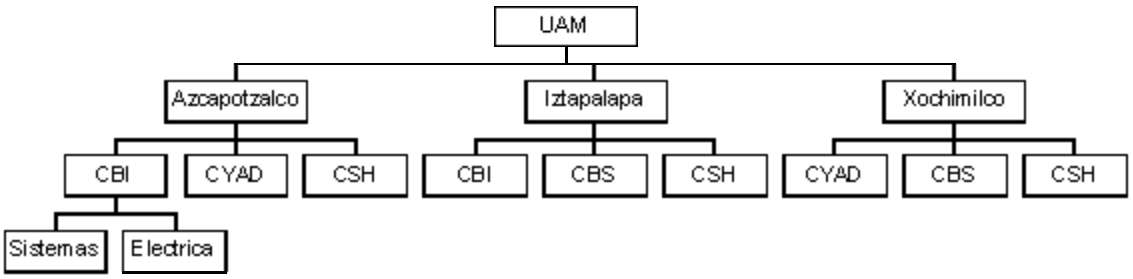
- Problemas difíciles de precisar.
- Definición de requerimientos vago y cambio a la par que se desarrolla el sistema.
- Proceso de desarrollo difícil.
- Verificación del software poco viable.

Para manejar la complejidad se utiliza alguna forma de organización jerárquica.

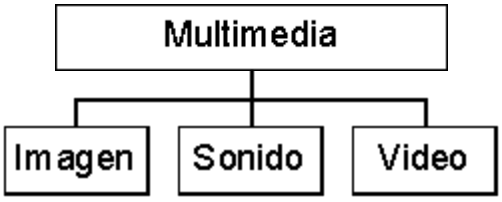
Dependencias.

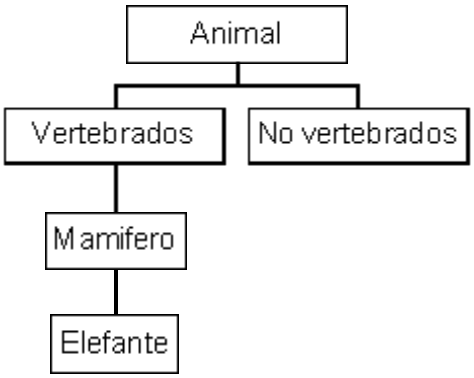
- Jerárquica estructural(parte de)
- Jerárquica especialización (tipo de)

Ejemplos.



Dependencia estructural: Existen dependencias muy fuertes.



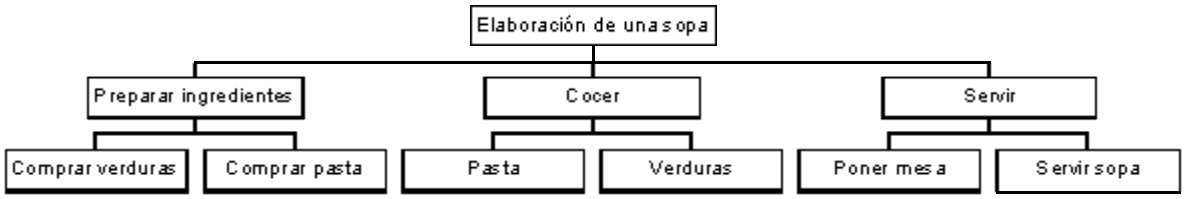


Jerarquía de especialización: Heredan las características.

1.2 Descomposición algorítmica.

Algorítmico: Método utilizado en el diseño estructurado ?Top-down?. Se basa en la identificación de las operaciones en la descripción del problema y su expresión en términos de operaciones más sencillas.

Ejemplo:



1.3 Orientado a objetos.

Método que permite resolver un problema alrededor de abstracciones representadas por objetos. Los objetos se comportan como agentes, contienen un estado y un comportamiento.

1.4 Modelo orientado a objetos.

Abstracción: Caracterización de un objeto de acuerdo a las propiedades que nos interesen en un instante de tiempo.

Encapsulación: Manera de ocultar los detalles de la representación interna de un objeto presentando solo la interfase para el usuario.

Modularidad: Un módulo es una agrupación de abstracciones lógicamente relacionadas.

1.5 Introducción a la Programación Orientada a Objetos (OOP).

El objetivo de la POO consiste en organizar los programas de modo que reflejen la forma de organización de los objetos en el mundo real.

Un **objeto** es un elemento independiente de un programa de computadora, que representa un grupo asociado de características y está diseñado para realizar tareas específicas. A los objetos también se les conoce como **instancias**.

Cada objeto tiene un papel específico en un programa, y todos los objetos pueden funcionar con otros objetos en maneras definidas.

Una **clase** es una plantilla que se utiliza para crear múltiples objetos con características similares. Las clases engloban todas las características de un conjunto particular de objetos. Cuando se escribe un programa en un lenguaje orientado a objetos, no se definen objetos individuales, sino que se definen clases de objetos.

Los conceptos de *herencia, interfaces y paquetes,* conforman los mecanismos para organizar clases y su comportamiento. La biblioteca de clases de Java se apoya en estos conceptos.

La herencia: Es un mecanismo que hace posible que una clase herede todo el comportamiento y los atributos de otra clase.

A través de la herencia, una clase tiene inmediatamente toda la funcionalidad de una clase existente. Debido a esto, las nuevas clases se pueden crear indicando únicamente en que se diferencian de la clase existente.

Con la herencia, todas la clases se acomodan en una jerarquía estricta, las que uno mismo creó y aquellas que provienen dela biblioteca de clases de Java y otras bibliotecas.

A una clase que hereda de otra clase se le llama **subclase**, y a la clase que proporciona la herencia se le llama **superclase**.

Una clase puede tener únicamente una superclase, pero cada clase tiene una cantidad ilimitada de subclases. Las subclases reciben por herencia todos los atributos y comportamiento de sus superclases.

Modularidad: Un modulo es una agrupación de abstracciones lógicamente relacionadas. La interfaz de unmódulo es la descripción de los servicio que ofrece a otros módulos.

Jerarquía: Permite organizar y ordenar las abstracciones. Hay dos tipos: por estructura y por especialización.

Tipo: Caracterización precisa de propiedades sobre la estructura y el comportamiento de una colección de unidades. Esta asociado íntimamente con la clases. Va a indicarnos cual es el ambiente que va a englobar al modulo.

Persistencia: Propiedad de un objeto de poder quedar vigente en el tiempo y el espacio (independiente de 00). Indicara que tiempo el objeto permanecerá vigente.

Concurrencia: Que simultáneamente se pueden llevar múltiples procesos.

1.6 Clases y objetos.

Objeto = Instancia.
Un objeto tiene un estado, un comportamiento y una identidad.

Estado: El estado de un objeto se compone de las propiedades estáticas de un objeto con valores dinámicos asociados a ellas en un momento dado.

Estado ? variables de instancia.

Comportamiento: Forma en que actúa y reacciona el objeto, expresado en términos de cambios de estado y envío de mensajes a otros objetos.

Comportamiento ? Método (es lo que nos indica qué puede hacer la clase)

Identidad: Propiedad que distingue a un objeto de los demás.

1.7 Relaciones entre objetos.

Relación de uso. Expresan la manera en que unos objetos usan a otros.

Formas de uso:

- **Actor**.-Objeto que puede operar sobre otros. (nunca es operado por los demás).
- **Servidor**.-Objeto que puede ser operado por otros. (nunca opera sobre los demás).
- **Agente.-** Objeto que puede operar sobre los demás y también puede ser operado por otros.

Relación de Contención. Se crea la relación si un objeto esta compuesto por otros objetos.

1.8 Clases.

Conjunto de objetos que tienen una estructura común y un comportamiento común. Por tanto es una abstracción del concepto objeto.

*Un objeto es una instancia de una clase.

Interfaz de una clase. Proporciona la vista externa o interfaz con el usuario.

Se divide en:

*Pública:*visible para todas las clases.

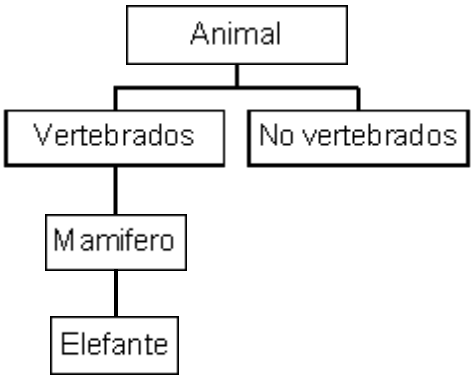
Protegida: visible sólo para subclases.

*Privada:*NO visible para ninguna otra clase.

Para garantizar el encapsulamiento y la privacidad se usaran variables privadas.

Relación de Herencia. Una clase comparte la estructura y/o comportamiento definidos en una o más clases.

Herencia Sencilla.



NOTA. El ?elefante? es una subclase, y ?animal? es una superclase.

Herencia Multiple.

Superclase: Clase de la cual otra clase hereda.

Subclase: Clase que hereda de una o más clases.

subclase ?especialización de la superclase.

NOTA. NO todos los lenguajes de programación orientada a objetos usan la herencia múltiple.

Clase abstracta. Su comportamiento no esta totalmente definido. Sirven para captar las propiedades generales que posteriormente son particularizadas en las subclases.

Clase Base. Clase que no tiene superclases.

Relación de instanciación. Relación entre clases donde el comportamiento de una clase puede ser parametrizada por otra clase.

Ejemplo.

> Colecciones de objetos homogéneas (listas, colas, pilas, árboles, etc. [operaciones genéricas para todos]).

La parametrización de una clase con otra es un mecanismo muy flexible que permita crear clases reutilizables.

Relación de metaclases. A una clase se le puede ver como a un objeto por lo tanto a la clase de los objetos de clases se les llama ***Metaclase***.

Metaclase. Es una clase cuyos objetos son clases.

La idea central de la instanciación y la parametrización: poder llegar a tener clases reutilizables.

1.9 Relación entre clases y objetos.

Las clases son estáticas, forman parte del texto del programa y no cambian durante su ejecución.

Los objetos por el contrario son totalmente dinámicos, se crean, cambian su estado y se destruyen.

Clasificación.

Forma de ordenar el conocimiento. No existen reglas simples que permitan la mejor selección de clases y objetos para generar una clasificación. Clasificar es agrupar cosas que tienen estructura común o comportamiento similar. La clasificación es un proceso difícil, por lo que conviene hacerlo de manera incremental iterativa.

Al iniciar el diseño, se define una estructura inicial para las clases. En base a la experiencia adquirida se crean nuevas **subclases** a partir de las existentes o bien es muy común que se requiera dividir una clase en otras más pequeñas (pero más generales), inclusive es posible requerir de la unión de varias clases.

1.10 Autoevaluación.

1. Da 3 ejemplos de jerarquía de especialización y 3 de dependencias estructuras.
2. Buscar toda la metodología de diagramación para la programación orientada a objetos.

3.Realizar una pequeña investigación comparativa de los diferentes lenguajes de programación (orientada a objetos) que existen, y su evolución:

*Historia

*Evolución

*Ventajas y desventajas.

*Conclusiones.

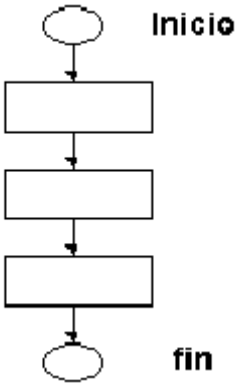
4.¿Cómo clasificaríamos el entorno de tu casa para que funcione correctamente? Pista: Clase base: la casa.Luego la recamara, la sala, el baño, la cocina, etc? Luego, tomando la recamara: ¿qué hay en ella?.- cama, cómoda/closet, mesa, silla, etc.

2. Desarrollo Orientado a Objetos.

2.1 PROGRAMACION PROCEDURAL.

La programación procedural fue la primer manera en como se programaron las aplicaciones como Windows.Las características de este lenguaje de programación son:

- Programa paso a paso que guía a la aplicación por una serie de instrucciones
- Ejecuta cada sentencia en el orden en que fue establecida
- Los datos (variables) y las funciones (métodos) son dos partes independientes

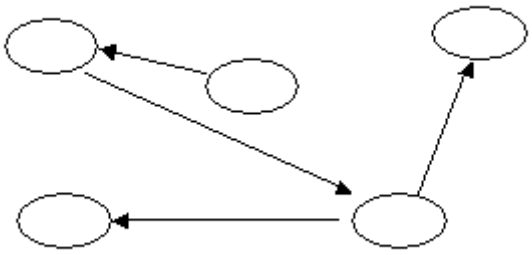


2.2 PROGRAMACION ORIENTADA A OBJETOS.

La programación orientada a objetos es el segundo paso en la evolución del diseño de aplicaciones. Su diseño se asemeja mas a la realidad y por tanto es mas sencillo de programar y reutilizar.Su auge a permitido la creación de una gran variedad de lenguajes de programación como Java, C++, etc.

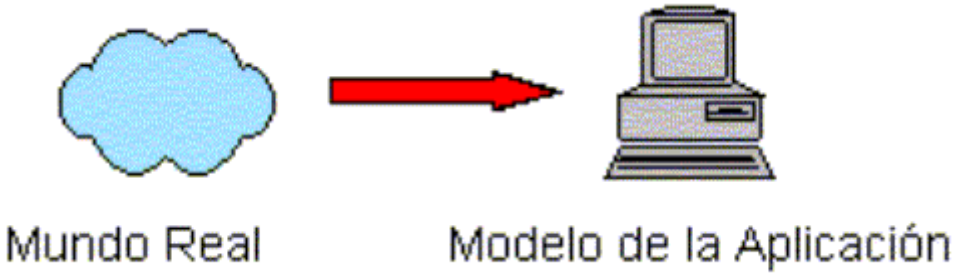
Las características de la programación orientada a objetos son:

- Programa en base a objetos
- Combina los datos (variables) y las funciones (métodos) en una misma unidad
- Su ejecución depende de los eventos y estados de la aplicación



EL enfoque del diseño orientado a objetos es representar el mundo real a través de entidades con datos y funciones, de manera que se comporten de manera inteligente de acuerdo a los eventos o estados que se presentan en una aplicación.

Ejemplo: Una tienda de libros esta compuesta de: Libros, clientes, ordenes, etc. Cada uno de ellos representa una entidad dentro del modelo, analizando a un libro, podemos decir que es una entidad inteligente, ya que podemos pedirle que se imprima, que se elimine, que se cambie el titulo, etc. de acuerdo a eventos o estados que se presenten dentro de la ejecución de un proceso en la aplicación.



La tecnología de Objetos consiste, como su nombre lo dice, en desarrollar software basada en objetos.La tecnología de Objetos nos permite:

- Un nuevo lenguaje de programación, evolucionando los lenguajes procedurales existentes como C, pascal, fortran, cobol, etc.

- Proporciona un completo ciclo de vida en los cambios de software, que consiste del diseño, prueba y ejecución. Permitiendo la separación de roles de manera independiente pero trabajando en conjunto, lo que permite que sea muy fácil de mantener.

La forma de programación de esta tecnología es muy parecida al mundo real, por lo que es muy fácil de leer y entender un programa elaborado por otro programador. Cuando se inicia con esta tecnología, los beneficios obtenidos no son automáticos, pero reduce considerablemente los costos, la construcción y mantenimiento del sistema.

Los principales beneficios que nos proporciona el diseño orientado a objetos son:

- **Adaptabilidad**: Es la habilidad de incorporar un nuevo cambio ya sea en los requerimientos, en la tecnología, en el desarrollo, en el tipo de cliente, etc.

- **Mantenimiento**: Su mantenimiento es fácil, ya que se puede tener una clara separación de roles (modelo-vista-control) y un cambio en alguna parte de la aplicación, no provoca cambios en las demás

- **Escalabilidad**: Dado que los objetos manejan el encapsulamiento, estos pueden ser simples o complejos como sea necesario, sin que el usuario del objeto conozca estos detalles.

- **Fácil de Utilizar**: Los objetos presentan una interface para el acceso a la lógica de su programación de manera sencilla, por lo que el cliente del objeto solo requiere consultar y a través de la interface solicitar los servicios que proporciona el objeto sin conocer lo que existe detrás en el objeto.

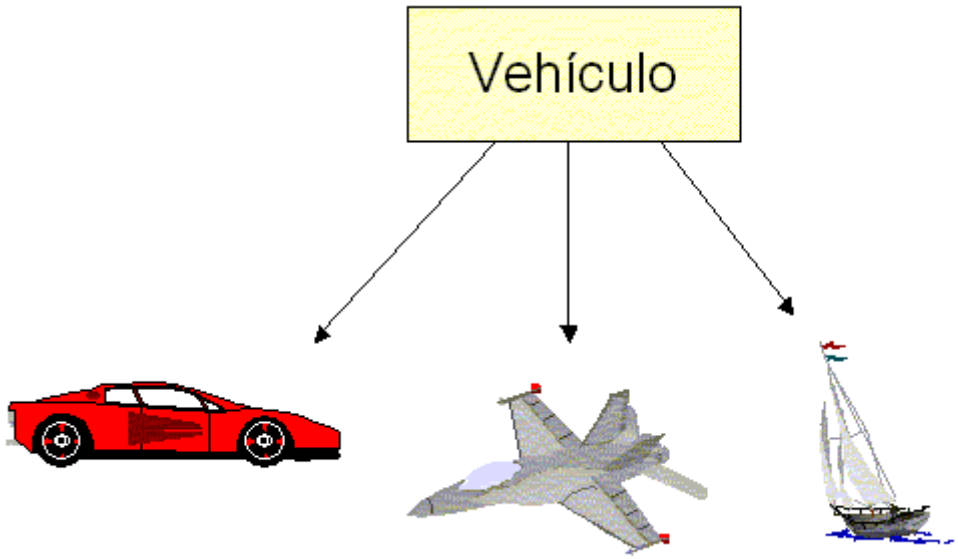
- **Reutilización**: La reutilización del código es uno de los principales beneficios, ya sea utilizada a través de la herencia o con objetos helpers que proporcionen funciones programadas en aplicaciones anteriores.

- Debido a que el diseño se asemeja al mundo real, facilita la comunicación entre el analista, diseñador, programador y usuario.

2.3 OBJETOS.

Un objeto en una aplicación orientada a objetos representa un objeto del mundo real involucrada en la solución.

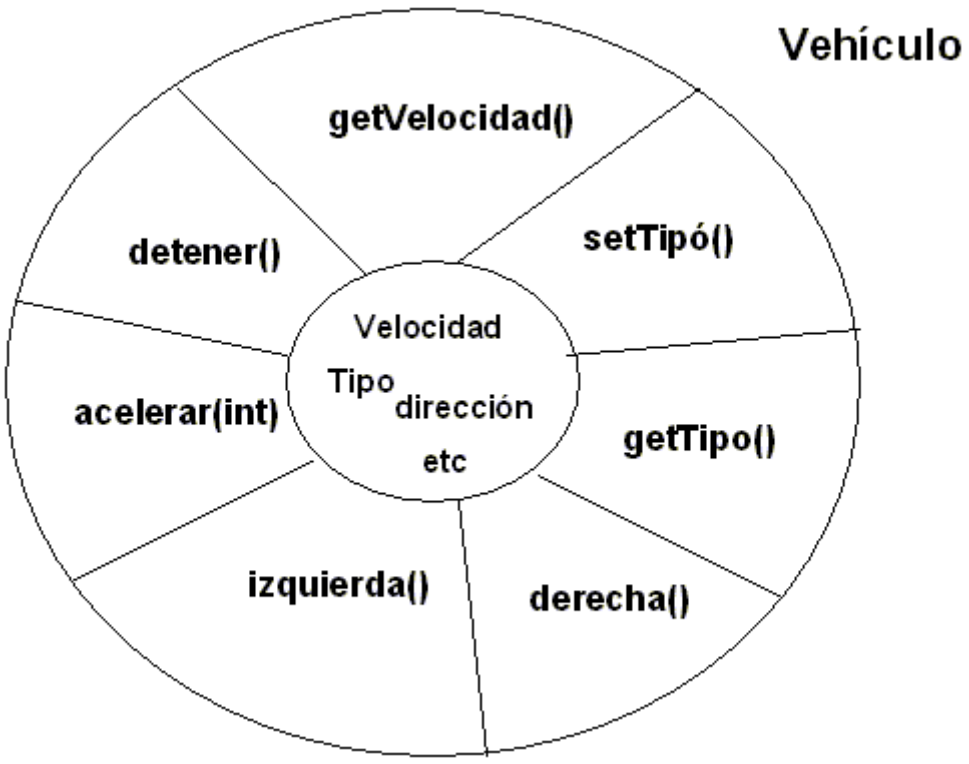
Un Objeto es una representación de una entidad del mundo real. En el ejemplo se muestra la entidad de vehículo, esta representa una entidad en el mundo real, como puede ser un auto, un avión, un barco, etc. Cada uno de ellos tendrá datos y funciones que dan funcionalidad al objeto vehículo, de esta forma, el cliente de la aplicación puede pedirle al vehículo que avance a la izquierda, a la derecha, hacia atrás, hacia delante, se detenga, etc.



Un objeto es comúnmente llamado instancia de clase o instancia.

Un objeto es una entidad que contiene datos y funciones que operan sobre los datos.

En la figura se muestra al objeto vehículo, el cual contiene los atributos de velocidad, tipo, dirección. Que son los que le dan las características al objeto.Cuenta además con funciones o métodos que proporcionan la funcionalidad al objeto, y contiene métodos para leer o modificar los atributos del objeto. Los métodos se muestran en la figura y protegen los atributos del objeto.

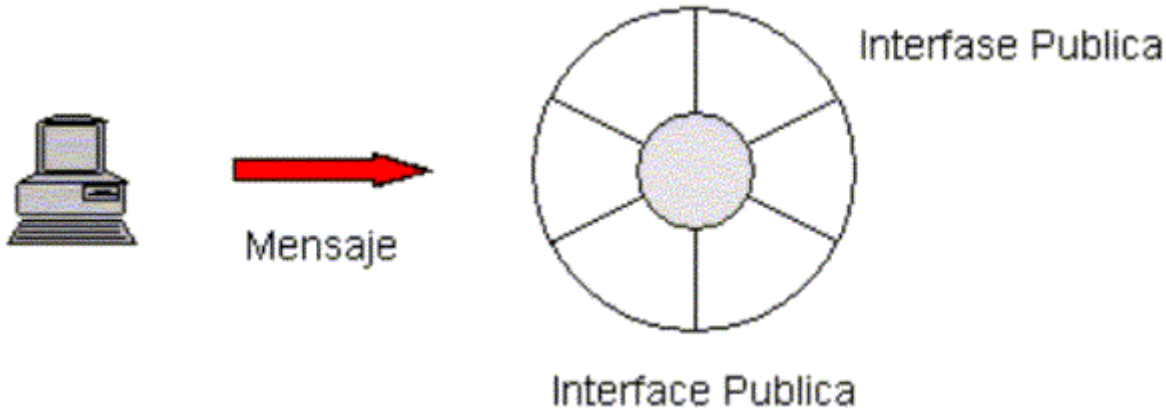


2.4 ENCAPSULAMIENTO.

Una de las tantas definiciones de lo que es el encapsulamiento se muestra en la figura, el objetivo del encapsulamiento es ocultar a los demás objetos la información internad de su funcionamiento, no permitirles acceder sus detalles internos, solo le permite conocer una interface publica que le permita a los demás objetos manejarlos. Por tanto permita conocer el QUE, pero no el COMO.

El encapsulamiento oculta la implementación de los mensajes o métodos internos de lógica de la clase, así como de los atributos de mismo, los cuales se recomienda, deben ser siempre privados y de uso exclusivo de la clase.

De esta manera separa al cliente de conocer la complejidad del objeto, proporcionándole solo una interface publica para el uso de manera transparente



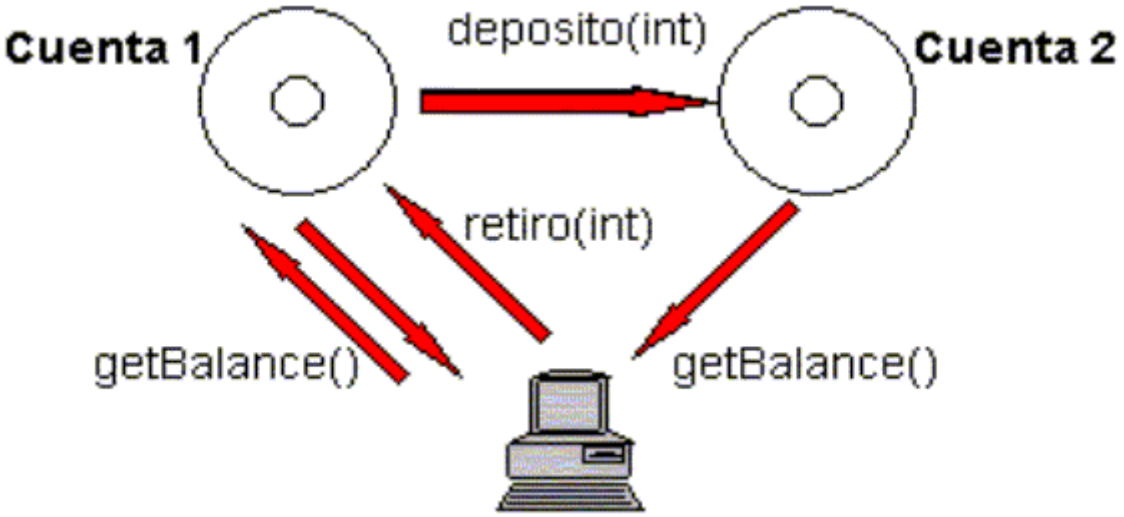
2.5 MENSAJES

La función del significado de un mensaje en la programación orientada a objetos se muestra en la figura. Es a través de los mensajes, que pueden cambiar el estado de un objeto a través de la invocación de métodos.

Los objetos en el mundo real, no son independientes unos de otros, sino que son influenciados. Un ejemplo serio una transacción bancaria, desde donde se desea hacer un retiro de una cuenta y transferirlo a otra, esta requiere de dos entidades ?cuenta? de las cuales se hará el traspaso de dinero. Estas instancias de cuenta deben comunicarse a través de mensajes para llevar a cabo su objetivo.

El modelo del mundo real, los objetos se envían solicitudes unos a otros, en las aplicaciones orientadas a objetos, la única manera en como se comunican los objetos es a través de mensajes.

Los mensajes invocan métodos, y existen métodos que al termino de su ejecución, no regresan mensajes, por lo que el envió de un mensaje no asegura que se lleve a cabo la acción correctamente.



2.6 ATRIBUTOS

Los objetos están compuestos de atributos y métodos.

Los atributos dentro de un objeto representan el estado actual del objeto, y dichos estados son los que dan el flujo a la ejecución de una aplicación OO. Por lo tanto es importante que siempre sean protegidos del resto de los objetos que interactúan con él.

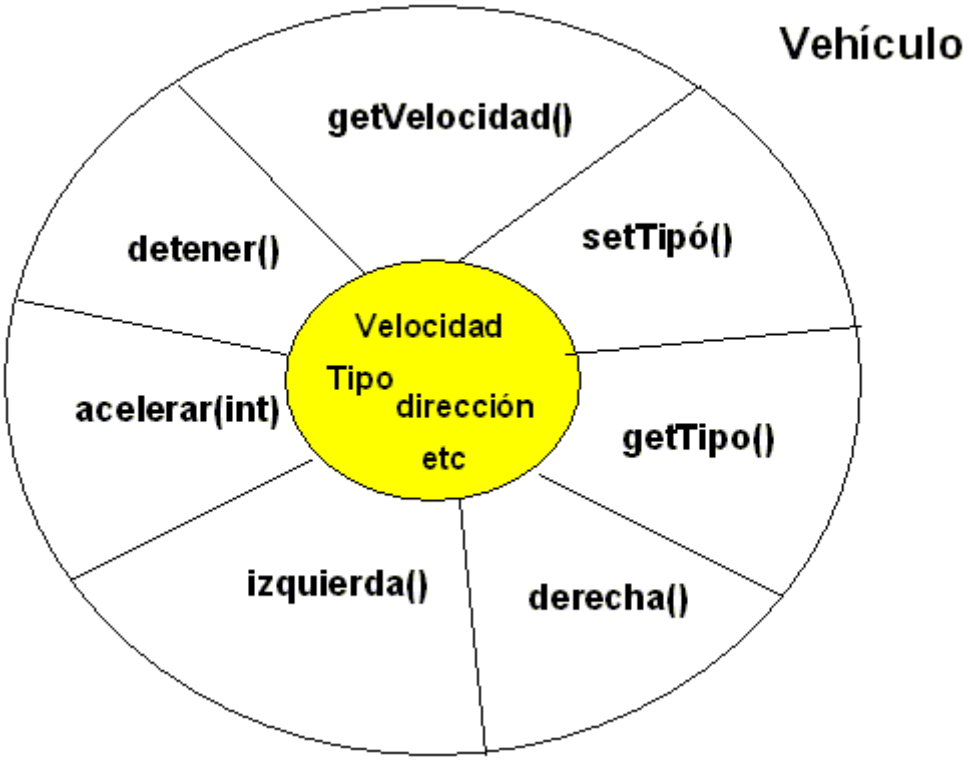
Es a través del encapsulamiento, que dichos atributos son ocultos a los demás.

En la figura los atributos que conforman el objeto son:

Velocidad: Que representa la velocidad con el que se mueve el objeto vehículo

Tipo: Representa el tipo de vehículo del objeto, como auto, avión, barco, etc.

Dirección: Representa el ángulo sobre el cual se mueve el vehículo



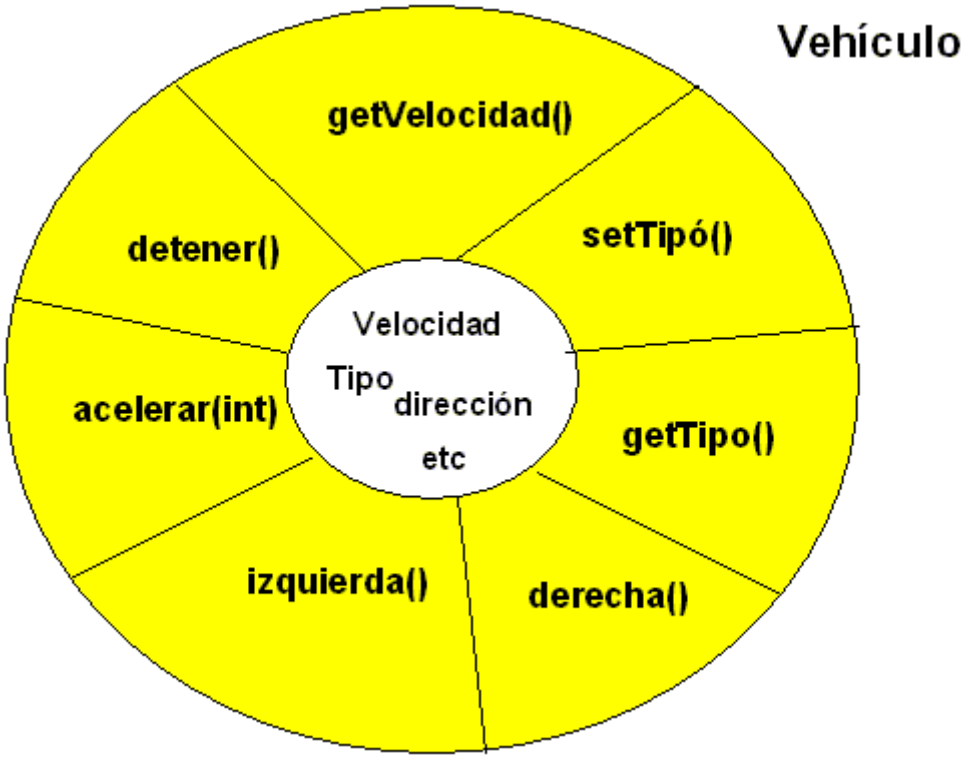
2.7 METODOS

Los objetos están compuestos de atributos y métodos.

Los métodos representan la implementación de cada uno de los mensajes que puede recibir el objeto.

Los métodos que pueden ser invocados por otros objetos a través de mensajes, dichos métodos forman parte de la interface publica para el manejo del objeto. Pero también pueden existir métodos que ayudan a la ejecución de los métodos públicos para que cumplan con sus objetivos, estos deben de estar protegidos por el encapsulamiento.

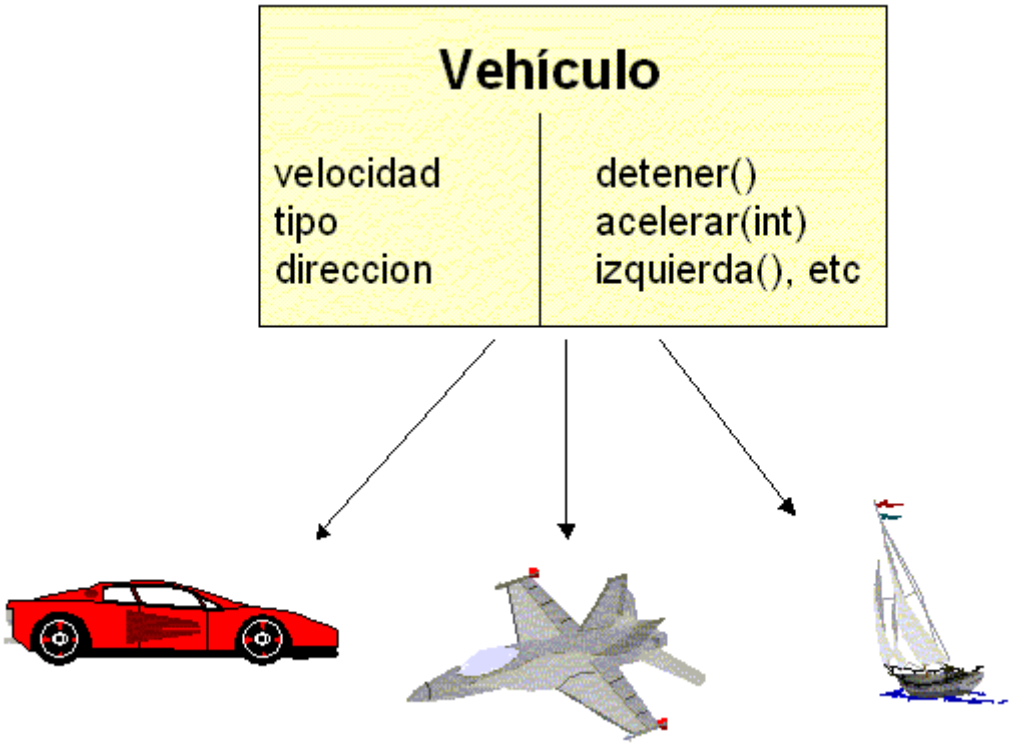
No todos lo métodos, regresan algún mensaje al objeto que lo invoco, eso dependerá de su implementación.



2.8 CLASES

Una clase es un template o abstracción de un objeto, la cual define:

- Los datos que tendrá cada objeto o instancia
- Los mensajes que cada instancia puede recibir
- La implementación de los mensajes que cada instancia puede recibir
- Cada instancia creada con la clase lucirá como se especifico en la clase



En la figura se muestra un ejemplo de una clase denominada vehículo, esta presenta atributos (datos) y métodos (funciones) que conforman cada uno de los objetos creados a partir de la clase.

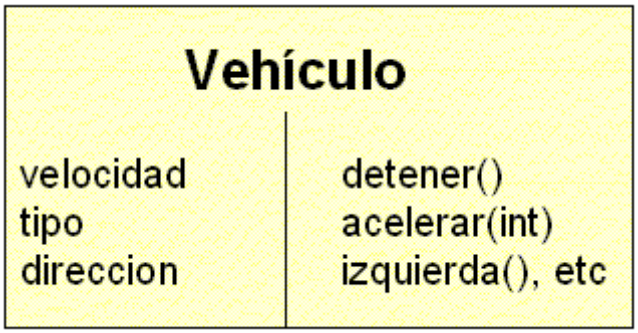
Por tanto, cada instancia creada con la clase (auto, avión, barco, etc.) lucirá como se especifico en la clase.

Lo que se requiere por tanto para definir una clase es:

Nombre: El nombre de la clase debe ser singular, que claramente defina la entidad o concepto que representa, siendo pequeño y conciso

Lista de elementos de datos: Piezas de datos que necesitan ser capturados y que definen las características que distinguen a las instancias creadas a partir de la clase

Lista de mensajes y sus implementaciones (métodos) que se pueden recibir, conformando la funcionalidad y lógica que proporcionaran los objetos creados a partir de la clase.



2.9 HERENCIA

La gente aprende mas rápido cuando los conceptos nuevos son comparados con los ya conocidos. Cuando se diseñan aplicaciones, los programadores, con frecuencia usan código de programas ya existentes.

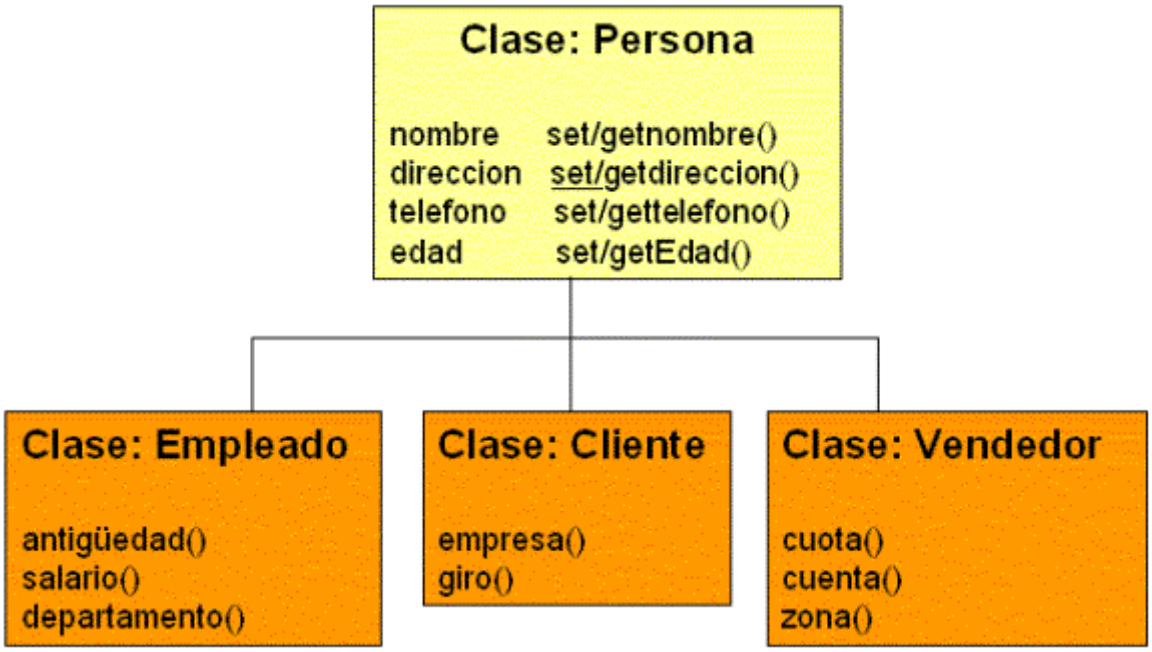
La definición de Herencia, podríamos decir que es el mecanismo para utilizar una clase con datos y métodos a través de otras clases de manera automática.

La herencia utiliza el concepto ?ES UN...? en las relaciones entre clases, por tanto, si tenemos una clase base ?Persona? y una clase ?Empleado? que hereda de la clase base ?Persona?, podremos decir que un Empleado ES UNA Persona.

En la clase padre conocida en la programación orientada a objetos como SuperClase, debe contener código base o genérico de las clases hijas o SubClases. Las SubClases deben de dar el detalle de cada una de ellas heredando las características de la Superclase, reutilizando código.

Recordar que entre menos acopladas estén las clases, el sistema es mas flexible.

Por tanto, la herencia es el mecanismo para el reusó de código.

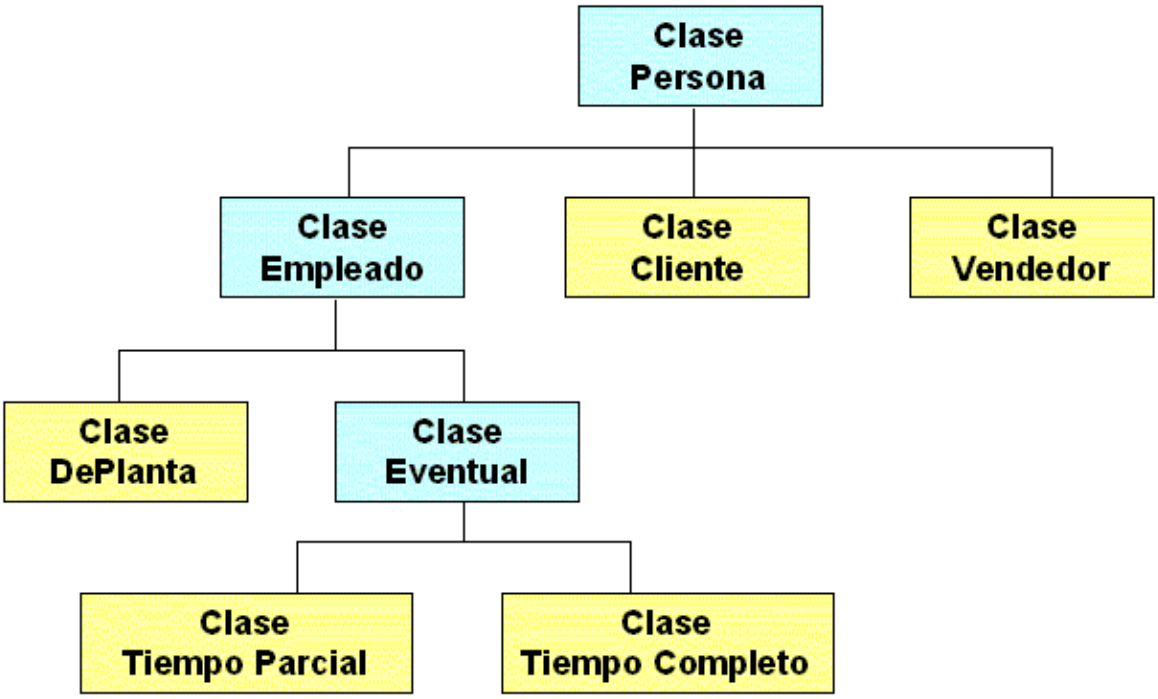


2.10 JERARQUIA DE HERENCIA

El conjunto completo de clases derivadas, desde la clase base hasta todas sus clases derivadas es llamado Jerarquía de Clases.

La herencia de la OO no es simplemente un atajo para la programación, ya que la herencia permite agrupar datos similares en unidades relacionadas.

Así, los objetos que requieran de herencia, se iniciara con una clase básica y derivara nuevas clases de esa clase común. Con los datos y funciones comunes de la clase base podrá luego concentrarse en programar solamente los cambios que aparecen en todas las clases derivadas que haya a continuación.



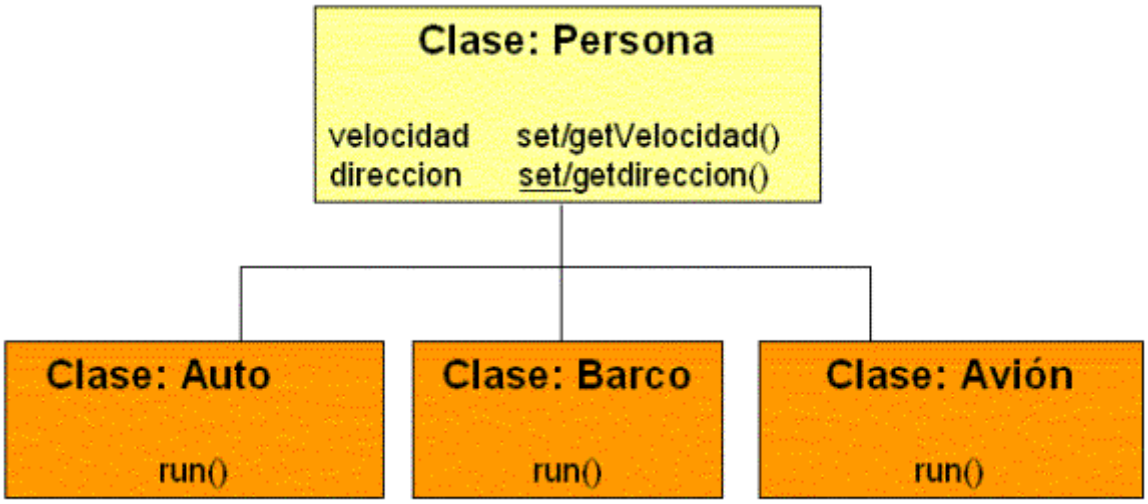
Mediante la herencia, la programación orientada a objetos ayuda a que se produzca código mas rápido, permitiéndole la reutilización de código en diferentes aplicaciones, por lo que se puede heredar cualquier cosa que pueda ser representada como objeto.

Otra razón para la herencia es la facilidad de cambio que proporciona la herencia, si se cambia una clase base, también cambian automáticamente todas las clases derivadas. Si se usara el enfoque de copiar y pegar, el cambio significaría una búsqueda y cambio manual en cualquier lugar en donde se hubiera usado el código.

No se debe de utilizar la herencia de manera indiscriminada, ya que eso provoca que las clases estén muy acopladas y esto provoca que el sistema no sea flexible.

La herencia habilita el polimorfismo a través de las interfaces.

Algunos lenguajes de programación como Java no soportan la herencia múltiple.



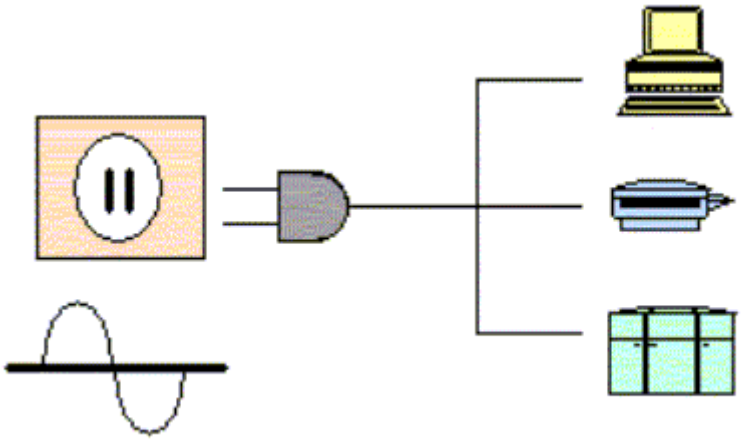
Algunas veces el reutilizar código, moviendo métodos dentro de una superclase no es la mejor opción para solucionar un problema, sobre todo cuando la implementación del método no es exactamente la misma para las subclases.

Cuando estos ocurre el método es colocado en las subclases para que cada una de ellas defina la implementación, esto permite a las instancias de diferentes clases tener diferentes implementaciones para el mismo mensaje o invocación del método.

Por lo tanto, el polimorfismo depende de una buena interface.

La herencia habilita el polimorfismo, el cual significa ?Muchas formas?, y a través de ella, se puede trabajar con un objeto sin conocerlo, yo puedo invocar el movimiento del vehículo invocando el método run() sin importar si se trata de un barco o un avión, y cada uno de ellos tendrá una implementación diferente de acuerdo a sus funciones de movimiento.

Si el sistema crece y se requiere de un nuevo vehículo llamado ?cohete?, solo es necesario implementar la interface (método run()) para que pueda ser utilizado por el sistema, sin que se requiera modificar el código existente.



2.11 UML

UML Unified Modeling Language ó Lenguaje unificado de modelado, es un lenguaje de modelado que proporciona la notación gráfica de que se valen los métodos para expresar los diseños, decir es un lenguaje grafico para visualizar, especificar construir y documentar sistemas de software.

Es la parte más importante para comunicación de un diseño, en donde las parte involucradas necesita conocer el lenguaje de modelado y no el proceso que se utilizó para lograr el diseño. Por tanto UML ayuda a la comunicación entre analistas, diseñadores y programadores.

UML define una notación y un metamodelo.

Notación: Es el material gráfico que se ven en los modelos, es la sintaxis del lenguaje de modelado.

Metamodelo: Diagrama que define la notación anteriormente descrita.

UML ha sido establecido por el OMG (Object Management Group) como la notación estándar para sistemas de objetos distribuidos.

Las técnicas en el UML fueron diseñadas en cierta medida para ayudar a los usuarios a hacer un buen desarrollo de OO. Algunas de sus técnicas son:

Diagramas de Interacción

Diagramas de clases

Patrones

etc.

2.12 CASOS DE USO

El caso de uso es un documento narrativo que describe la secuencia de eventos de un actor (agente externo) que utiliza un sistema para completar un proceso.

Los casos de uso se representan con una elipse que contiene el nombre del caso de uso

Los casos de uso son historias o casos de utilización de un sistema, no son exactamente los requerimientos no las especificaciones funcionales.

Un caso de uso es un elemento primario de la planificación y el desarrollo de proyectos, es una interacción típica entre un usuario y un sistema.

Un caso de uso es una toma instantánea de algún aspecto del sistema. La suma de todos los casos de uso constituyen la vista externa del sistema.

Un buen conjunto de casos de uso sirven para la comunicación de los elementos superficiales hasta las cuestiones mas profundas.

2.13 ACTOR

Un actor es la entidad externa del sistema que de alguna manera participa en la historia del caso de uso. Por lo regular estimula el sistema con eventos de entrada o recibe algo de él. Los actores están representados por el papel que desempeñan en el caso de uso. Conviene escribir su nombre con mayúsculas en la narrativa del caso de uso para facilitar la identificación.

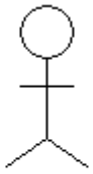
Un Actor es un usuario que desempeña un papel con respecto al sistema. Cuando se trata con actores, se debe pensar en los papeles y no en las personas ni en los títulos de sus puestos.

Los actores llevan a cado casos de uso. Un mismo actor puede realizar varios casos de uso y a la inversa, un caso de uso puede ser realizado por varios actores.

Los actores son útiles cuando se trata de definir los casos de uso. No es necesario que los actores sean seres humanos, a pesar de que los actores estén representados por figuras humanas. El actor puede ser también un sistema externo que necesite cierta información del sistema actual.

Se deben cuestionar los casos de uso con los actores para descubrir los objetivos reales del usuario y considerar formas alternas para lograrlos.

Un actor es el que obtiene un valor del caso de uso y es el interesado en la construcción de un caso de uso. Lo importante es comprender los casos de uso y los objetivos del usuario que satisfacen.



Instructor

2.14 RELACIONES

Las relaciones pueden ser de:

Uso (<<usa>>)

Herencia (<<extiende>>)

Las relaciones especifican quienes interactúan en el sistema.

Además de los vínculos entre los actores y los casos de uso, hay otros dos tipos de vínculos, estos representan las relaciones de usos (usa y extends (entiende) entre los casos de uso.

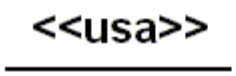
Se usa la relación extiende cuando se tiene un caso de uso que es similar a otro, pero que hace un poco mas, es decir, no se efectúa el comportamiento habitual genérico asociado con dicho caso de uso, en ese momento se efectúa una variación. El modo de abordar la variación es poner la conducta normal en un caso y la conducta inusualen cualquier otro lado. Por tanto, la esencia de una relación extends consiste primero en obtener el caso de uso simple y normal y en cada paso del caso de uso preguntarse si algo puede fallar o comportarse de manera diferente.

Las relaciones de uso ocurren cuando se tiene una porción de comportamiento que es similar en más de un caso de uso y que no se quiere copias la descripción de tal conducta

Tratándose de la relación extends, los actores tienen que ver con los casos de uso que están extendiendo. Un actor se encargará tanto del caso de uso base como de todas las extensiones. En las relaciones de uso, es frecuente que no haya un actor asociado con el caso de uso común, si lo hay no se considera que esté llevando a cabo los demás casos de uso.

En resumen, podemos decir que se utilizaran relaciones extends cuandodescriba una variación de conducta normal y relaciones de usos para repetir cuando se trate de uno o varios casos de uso y desee evitar repeticiones.

Relación Normal



Relación de Uso

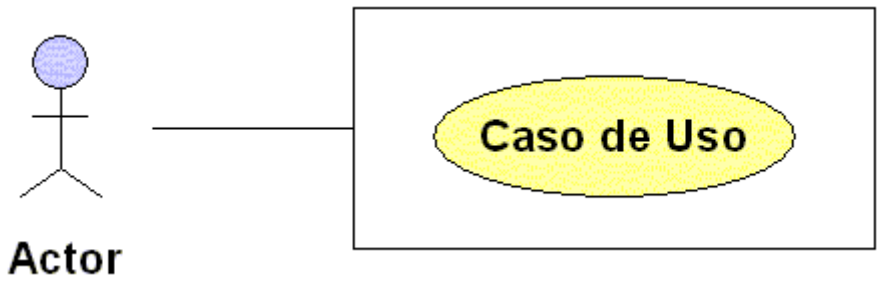
Relación de Herencia

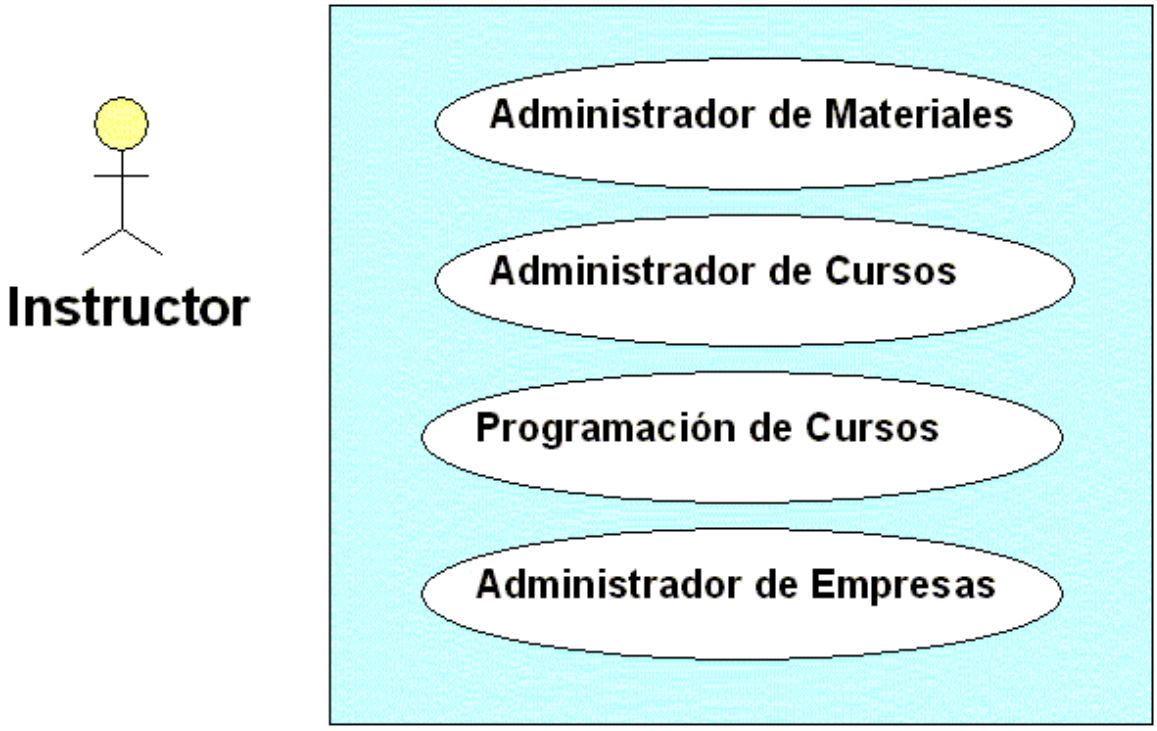
2.15 DIAGRAMA DE CASOS DE USO

El diagrama de casos de uso es un diagrama para la representación grafica de los casos de uso y es también parte de UML.

Un diagrama de casos de uso muestra por tanto, actores y casos de uso juntos con sus relaciones.

El término escenario en UML, se refiere a una sola ruta a través de un caso de uso, una ruta que muestra una particular combinación de condiciones dentro de dicho caso de uso, por ejemplo el caso de uso en que todo salga bien, en que se presente alguna falla, o que sea rechazado.





2.16 DIAGRAMA DE CLASE

El diagrama de clase describe los tipos de objetos que hay en el sistema y las relaciones estáticas que existen entre ellos. Hay dos tipos de relaciones estáticas:

asociaciones (cliente tome distintos cursos)

subtipos (un cliente es una persona)

Los diagramas de clase también muestran los atributos y operaciones de una clase y las restricciones a que se ven sujetos, según la forma en que se concentren los objetos.

2.17 Autoevaluación

1. Investigar los diagramas UML de:
    a. Casos de uso
    b. Clases
    c. Secuencia
    d. Colaboración

e.Estado

f.Actividades

3. Introducción al lenguaje de programación Java.

3.1 JAVA.

Es un lenguaje orientado a objetos que fue desarrollado por Sun Microsystem. Se diseño para ser pequeño, sencillo y portable a través de plataformas y sistemas operativos.
Los programas en Java pueden ser aplicaciones y applets. Las aplicaciones son como cualquier programa desarrollado en un lenguaje orientado a objetos. Los applets son programas que se bajan de World Wide Web por medio de un navegador Web y se ejecutan dentro de páginas Web HTML. El navegador debe de soportar Java, como el Netscape Navigator. Aparecen como imágenes, son dinámicos e interactivos y se puede utilizar para crear animaciones , figuras, formas que pueden responder de inmediato a entradas del usuario, juegos u otros efectos interactivos en la misma página Web entre texto y gráficos.

Los programas desarrollados en ***Java*** presentan diversas ventajas frente a los desarrollados en otros lenguajes como C/C++. La ejecución de programas en ***Java*** tiene muchas posibilidades: ejecución como aplicación independiente ***(Stand-alone Applicatio***n), ejecución como ***applet***, ejecución como ***servle***t, etc. Un ***applet*** es una aplicación especial que se ejecuta dentro de un navegador o navegador (por ejemplo *Netscape Navigator* o *Internet Explore*r) al cargar una página HTML desde un servidor ***We***b. El ***applet*** se descarga desde el servidor y no requiere instalación en la computadora donde se encuentra el navegador. Un ***servlet*** es una aplicación sin interfase gráfica que se ejecuta en un servidor de Internet. La ejecución como aplicación independiente es análoga a los programas desarrollados con otros lenguajes.

Además de incorporar la ejecución como ***Apple***t, ***Java*** permite fácilmente el desarrollo tanto de arquitectura cliente-servidor como de aplicaciones distribuidas, consistentes en crear aplicaciones capaces de conectarse a otros ordenadores y ejecutar tareas en varios ordenadores simultáneamente, repartiendo por lo tanto el trabajo. Aunque también otros lenguajes de programación permiten crear aplicaciones de este tipo, ***Java*** incorpora en su propio ***API*** estas funcionalidades.

3.2 Historia de Java

Java surgió en 1991 cuando un grupo de ingenieros de *Sun Microsystems* trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Esto hizo de Java un lenguaje ideal para distribuir programas ejecutables vía la WWW, además de un lenguaje de programación de propósito general para desarrollar programas que sean fáciles de usar y portables en una gran variedad de plataformas. Desarrollaron un código "neutro" que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una *"máquina hipotética o virtua*l" denominada ***Java Virtual Machine (JV***M). Era la ***JVM*** quien interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: *"Write Once, Run Everywhere*". A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadoras, ***Java*** se introdujo a finales de 1995. Se uso en varios proyectos de Sun (en aquel entonces se llamaba Oak) sin mucho éxito comercial. Se difundió más cuando se unió con HotJava, un navegador Web experimental, para bajar y ejecutar subprogramas (los futuros applets). Cuando se incorporo a Netscape. La clave del éxito fue la incorporación de un intérprete ***Java*** en la versión 2.0 del programa Netscape Navigator en 1994, produciendo una verdadera revolución en Internet. Obtuvo tanta atención que en Sun la división de Java se separo en la subsidiaria JavaSot.

Java 1.1 apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje. ***Java 1.***2, más tarde rebautizado como ***Java*** 2, nació a finales de 1998.

A nivel de código fuente, los tipos de datos primitivos de Java tienen tamaños consistentes en todas las plataformas de desarrollo. Las bibliotecas de clases fundamentales en Java facilitan la escritura de código, el cual puede ser transportado de una plataforma a otra sin necesidad de rescribirlo para que trabaje en la nueva plataforma. Lo anterior también se aplica al código binario. Se puede ejecutar sin necesidad de recompilarlo.

Con los lenguajes convencionales al compilar se genera código binario para una plataforma en particular. Si se cambia la plataforma se tendrá que recompilar el programa. Por ejemplo un programa en C para una PC, no servirá en un servidor UNIX y viceversa.

Al programar en ***Java*** no se parte de cero. Cualquier aplicación que se desarrolle "cuelga" (o se apoya, según como se quiera ver) en un gran número de ***clases*** preexistentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el ***API*** o ***Application Programming interfase*** de ***Jav***a). ***Java*** incorpora en el propio lenguaje muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de software o fabricantes de ordenadores (threads, ejecución remota, componentes, seguridad, acceso a bases de datos, etc.). Por eso muchos expertos opinan que ***Java*** es el lenguaje ideal para aprender la informática moderna, porque incorpora todos estos conceptos de un modo estándar, mucho más sencillo y claro que con las citadas extensiones de otros lenguajes. Esto es consecuencia de haber sido diseñado más recientemente y por un único equipo.

El principal objetivo del lenguaje ***Java*** es llegar a ser el "nexo universal" que conecte a los usuarios con la información, esté ésta situada en la computadora local, en un servidor de ***We***b, en una base de datos o en cualquier otro lugar.

Java es un lenguaje muy completo (de hecho se está convirtiendo en un macro lenguaje: ***Java 1.0*** tenía 12 paquetes; ***Java 1.1*** tenía 23 y ***Java 1.2*** tiene 59). En cierta forma casi todo depende de casi todo. Por ello, conviene aprenderlo de modo *iterativ*o: primero una visión muy general, que se va refinando en sucesivas iteraciones. Una forma de hacerlo es empezar con un ejemplo completo en el que ya aparecen algunas de las características más importantes.

Java 2 ***(antes llamado Java 1.2 o JDK 1.2) es la tercera versión importante del lenguaje de programación Java.***No hay cambios conceptuales importantes respecto a ***Java 1.1*** (en ***Java 1.1*** sí los hubo respecto a ***Java 1.***0), sino extensiones y ampliaciones, lo cual hace que a muchos efectos, sea casi lo mismo trabajar con ***Java 1.1*** o con ***Java 1.***2.

La compañía ***Sun*** describe el lenguaje ***Java*** como *"simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámic*o". Además de una serie de halagos por parte de ***Sun*** hacia su propia criatura, el hecho es que todo ello describe bastante bien el lenguaje ***Jav***a, aunque en algunas de esas características el lenguaje sea todavía bastante mejorable.

3.3 Características de Java

Muchos de los conceptos de programación orientada a objetos de Java se heredan de C++, aunque también se toman conceptos de otros lenguajes orientados a objetos. Java incluye un conjunto de bibliotecas de clase que ofrecen tipos de datos básicos, capacidades de entrada y salida y del sistema y otras funciones de utilidad. Estas bibliotecas son parte del ambiente estándar de Java, el cual también incluye herramientas sencillas de conectividad de red, protocolos comunes de Internet y funciones del kit de herramientas para la interfase de usuario.
Además de ser portable y estar orientado a objetos, uno de los objetivos iniciales de Java era ser pequeño y simple, y por lo tanto fácil de escribir, compilar, depurar y, sobre todo, fácil de aprender. Java es poderoso y flexible.

Java esta fundamentado en C y C++ y mucha de su sintaxis y estructura orientada a objetos se tomó de este ultimo. Aunque excluye los conceptos más complejos de éstos, haciendo un lenguaje más simple sin sacrificar mucho de su poder. No existen apuntadores, ni su aritmética. Las cadenas y los arreglos son objetos reales en Java. La administración de la memoria es automática.

3.4 EL ENTORNO DE DESARROLLO DE JAVA

Existen distintos programas comerciales que permiten desarrollar código ***Jav***a. La compañía ***Su***n, creadora de ***Jav***a, distribuye gratuitamente el *Java(Tm) Development Kit (JD*K). Se trata de un conjunto de programas y librerías que permiten desarrollar,

compilar y ejecutar programas en ***Jav***a. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (con el denominado ***Debugge***r). Cualquier programador con un mínimo de experiencia sabe que una parte muy importante (muchas veces la mayor parte) del tiempo destinado a la elaboración de un programa se destina a la ***detección y corrección de errore***s. Existe también una versión reducida del ***JD***K, denominada ***JRE*** *(Java Runtime Environmen*t) destinada únicamente a ejecutar código ***Java*** (no permite compilar).
Los ***ID***Es *(Integrated Development Environmen*t), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código ***Jav***a, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar ***Debug*** gráficamente, frente a la versión que incorpora el ***JDK*** basada en la utilización de una consola (denominada habitualmente ventana de comandos de MS-DOS, en ***Windows NT/95/9***8) bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con ***componentes*** ya desarrollados, los cuales se incorporan al proyecto o programa. Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas, y archivos resultantes de mayor tamaño que los basados en clases estándar.

3.5 El compilador de Java

Se trata de una de las herramientas de desarrollo incluidas en el ***JD***K. Realiza un análisis de sintaxis del código escrito en los archivos fuente de ***Java*** (con extensión ****.jav***a). Si no encuentra errores en el código genera los archivos compilados (con extensión ****.clas***s). En otro caso muestra la línea o líneas erróneas. En el ***JDK*** de ***Sun*** dicho compilador se llama ***javac.ex***e (para ambientes windows en UNÍX se elimina la extensión .exe). Tiene numerosas opciones, algunas de las cuales varían de una versión a otra. Se aconseja consultar la documentación de la versión del ***JDK*** utilizada para obtener una información detallada de las distintas posibilidades.

3.6 La Máquina Virtual de Java

La existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de ***Sun*** a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se planteó la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código "neutro" el cual estuviera preparado para ser ejecutado sobre una *"máquina hipotética o virtua*l", denominada ***Java Virtual Machine (JV***M). Es esta ***JVM*** quien ***interpreta*** este código neutro convirtiéndolo a código particular de la CPU utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.
La ***JVM*** es el intérprete de ***Jav***a. Ejecuta los ***"bytecode***s" (archivos compilados con extensión ****.clas***s) creados por el compilador de ***Java (javac.ex***e o javac). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado ***JIT*** *(Just-In-Time Compile*r), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

3.7 Las variables PATH y CLASSPATH

El desarrollo y ejecución de aplicaciones en ***Java*** exige que las herramientas para compilar ***(javac.ex***e o javac) y ejecutar ***(java.ex***e o java) se encuentren accesibles. La computadora, desde una ventana de comandos de MS-DOS, sólo es capaz de ejecutar los programas que se encuentran en los directorios indicados en la variable ***PATH*** de la computadora (o en el directorio activo). Si se desea compilar o ejecutar código en ***Jav***a, el directorio donde se encuentran estos programas ***(java.exe*** y ***javac.ex***e o java y javac) deberá encontrarse en el ***PAT***H. Tecleando ***PATH*** en una ventana de comandos de MS-DOS se muestran los nombres de directorios incluidos en dicha variable de entorno. Para UNIX en una terminalk usar env|grep PATH.
Java utiliza además una nueva variable de entorno denominada ***CLASSPAT***H, la cual determina dónde buscar tanto las clases o librerías de ***Java*** (el ***API*** de ***Jav***a) como otras clases de usuario. A partir de la versión 1.1.4 del JDK no es necesario indicar esta variable, salvo que se desee añadir conjuntos de clases de usuario que no vengan con dicho ***JD***K. La variable ***CLASSPATH*** puede incluir la ruta de directorios o archivos ****.zip*** o ****.jar*** en los que se encuentren los archivos ****.clas***s. En el caso de los archivos ****.zip*** hay que observar que los archivos en él incluidos no deben estar comprimidos. En el caso de archivos ****.jar*** existe una herramienta ***(jar.ex***e o jar), incorporada en el ***JD***K, que permite generar estos archivos a partir de los archivos compilados ****.clas***s. Los archivos ****.jar*** son archivos comprimidos y por lo tanto ocupan menos espacio que los archivos ****.class*** por separado o que el archivo ****.zip*** equivalente.

Una forma general de indicar estas dos variables es crear un archivo ***batch*** de MS-DOS ***(*.ba***t) donde se indiquen los valores de dichas variables. Cada vez que se abra una ventana de MS-DOS será necesario ejecutar este archivo ****.bat*** para asignar adecuadamente estos valores. Un posible archivo llamado ***jdk117.ba***t, podría ser como sigue:

```
set JAVAPATH=C:\jdk1.1.7

set PATH=.;%JAVAPATH%\bin;%PATH%

set CLASSPATH=.\;%JAVAPATH%\lib\classes.zip;%CLASSPATH%
```

lo cual sería válido en el caso de que el ***JDK*** estuviera situado en el directorio ***C:\jdk1.1.***7.

Para el caso de UNIX se puede usar un shell script llamado por ejemplo jdk.1.7.sh:

```
#!/bin/sh

JAVAPATH=/usr/local/bin/jdk.1.7

PATH=$PATH:$JAVAPATH/bin:

CLASSPATH=$CLASSPATH:$JAVAPATH/lib/classes.zip:

export JAVAPATH PATH CLASSPATH
```

lo cual seria valido si el JDK estuviera situado en el directorio /usr/local/bin/jdk.1.7. Para correr este shell script se haria:

$chmod 700 jdk.1.7.sh

$. ./jdk.1.7.sh

Si no se desea tener que ejecutar este archivo cada vez que se abre una consola de MS-DOS es necesario indicar estos cambios de forma "permanente". La forma de hacerlo difiere ***entre Windows 95/98*** y ***Windows N***T. En ***Windows 95/98*** es necesario modificar el archivo ***autoexec.bat*** situado en C:\, añadiendo las líneas antes mencionadas. Una vez reiniciada la computadora estarán presentes en cualquier consola de MS-DOS que se cree. La modificación al archivo ***autoexec.bat*** en ***Windows 95/98*** será la siguiente:

```
set JAVAPATH=C:\jdk1.1.7

set PATH=.;%JAVAPATH%\bin;%PATH%

set CLASSPATH=
```

donde en la tercera línea debe incluir la ruta de los archivos donde están las clases de ***Jav***a. En el caso de utilizar **Windows NT** se añadirá la variable **PATH** en el cuadro de diálogo que se abre con*Start* -> Settings -> Control Panel -> System -> Environment -> User Variables for NombreUsuario. En Español *Inicio*-> Configuración -> Panel de Control-> Sistema -> Entorno -> Variables de usuario parar NombreUsuario

Para el caso de UNIX se puede colocar el código del shell script en el archivo .profile, si se tiene Bourne o Korn shell. Para el caso de C shell se debe de modificar y poner en .login:

setenv JAVAPATH/usr/local/bin/jdk.1.7

```
set path = ( $JAVAPATH /bin $path )

setenv CLASSPATH
```

También es posible utilizar la opción ***classpath*** en el momento de llamar al compilador ***javac*** o al intérprete ***java***. En este caso los archivos ****.jar*** deben ponerse con el nombre completo en el ***CLASSPAT***H: no basta poner el ***PATH*** o directorio en el que se encuentra. Por ejemplo, si se desea compilar y ejecutar el archivo ***ContieneMain.jav***a, y éste necesitara la librería de clases ***G:\MyProject\OtherClasses.ja***r, además de las incluidas en el ***CLASSPAT***H, la forma de compilar y ejecutar sería:

```
javac -classpath .\;G:\MyProject\OtherClasses.jar ContieneMain.java

java -classpath .\;G:\MyProject\OtherClasses.jar ContieneMain
```

Se aconseja consultar la ayuda correspondiente a la versión que se esté utilizando, debido a que existen pequeñas variaciones entre las distintas versiones del JDK.

Cuando un archivo ***filename.java*** se compila y en ese directorio existe ya un archivo ***filename.class***, se comparan las fechas de los dos archivos. Si el archivo ***filename.java*** es más antiguo que el ***filename.class*** no se produce un nuevo archivo ***filename.class***. Esto sólo es válido para archivos ****.class*** que se corresponden con una clase ***public***.

3.8 Obtención del ambiente de desarrollo de Java

La maquina virtual de Java viene incluida con los navegadores. Para una aplicación que no se corra en navegador se puede obtener fácilmente del sitio Web de JavaSoft http://www.javasoft.com. De ahí se puede bajar el JDK. Se debe procurar bajar la versión mas reciente, para la plataforma en la que se trabajara.
Existen también ambientes integrados de desarrollo integrados o IDEs, disponibles para desarrollar en Java. Se incluyen el Java Workshop de Sun para Solaris, Windows NT y Windows 95, 98. Se pueden bajar de http://www.sun.com/developer-products/java; Café de Symantec para Windows 95,98 Windows NT y Macintosh http://cafe.symantec.com; Roaster de Natural Intelligence http://www.natural.com/pages/products/roaster/index.html

Para instalar:

1. Se debe desempacar y correr el script de instalación en UNIX. Para Windows se desempacay se ejecuta el instalador.

2. Se debe de incluir en el PATH el directorio bin bajo el directorio donde se instalo el JDK.

3. Se define la variable de ambiente CLASSPATH con la ruta absoluta al archivo classes.zip

4. Incluir las definiciones anteriores en el archivo de configuración de la cuenta o maquina en donde se instalo: .profile, .login o autoexec.bat según sea el caso.

Para crear, compilar y ejecutar un programa:

1. Crear el archivo fuente con un editor de texto (vi, bloc de notas, etc.)

2. Compilar con el compilador javac (javac Prog.java)

3. Si no hay problemas se generara el archivo Prog.class

4. Para ejecutar una aplicación usar el interprete java (java Prog)

5. Si es un applet se hace referencia a el desde una pagina HTML con la etiqueta APPLET

```
<HTML>

        <HEAD>
```

```
        <TITLE> HOLA </TITLE>
    </HEAD>
    <BODY>
        <P> Salida del applet:
        <APPLET CODE="Prog.class" WIDTH=200 HEIGTH=50
        </APPLET>
    </BODY>
</HTML>
```

Si es un applet se ejecuta por medio de un navegador o del appletviewer, al cargar la pagina que hace referencia al applet.

3.9 NOMENCLATURA HABITUAL EN LA PROGRAMACIÓN EN JAVA

Los nombres de ***Java*** son sensibles a las letras mayúsculas y minúsculas. Así, las variables ***mas***a, ***Masa*** y ***MASA*** son consideradas variables completamente diferentes. Las reglas del lenguaje respecto a los nombres de variables son muy amplias y permiten mucha libertad al programador, pero es habitual seguir ciertas normas que facilitan la lectura y el mantenimiento de los programas de ordenador. Se recomienda seguir las siguientes instrucciones:

1. En ***Java*** es habitual utilizar nombres con minúsculas, con las excepciones que se indican en los puntos siguientes.

2. Cuando un nombre consta de varias palabras es habitual poner una a continuación de otra, poniendo con mayúscula la primera letra de la palabra que sigue a otra (Ejemplos: *elMayor()*, *VentanaCerrable*, *RectanguloGrafico*, *addWindowListener()*,setHora()).

3. Los nombres de ***clases*** e ***interfaces*** comienzan siempre por mayúscula (Ejemplos: *Geometria*, *Rectangulo*, *Dibujable*, *Graphics*, *ArrayList*, *Iterator*,Tiempo).

4. Los nombres de ***objetos***, los nombres de ***métodos*** y ***variables miembro***, y los nombres de las ***variables locales*** de los métodos, comienzan siempre por minúscula (Ejemplos: *main()*, *dibujar()*, *numRectangulos*, x, y, r, hora, minuto, segundo).

5. Los nombres de las ***variables finales***, es decir de las constantes, se definen siempre con mayúsculas (Ejemplo: *PI*)

3.10 VARIABLES

Una variable es un nombre que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

1. Variables de ***tipos primitivos***. Están definidas mediante un valor único que puede ser entero, de punto flotante, carácter o booleano. ***Java*** permite distinta precisión y distintos rangos de valores para estos tipos de variables (***char***, ***byte***, ***short***, ***int***, ***long***, ***float***, ***double***, ***boolean***). Ejemplos de variables de tipos primitivos podrían ser: 123, 3456754, 3.1415, 12e-09, 'A', True, etc.

2. Variables ***referencia***. Las variables referencia son referencias o nombres de una información más compleja: ***arreglos*** u ***objetos*** de una determinada clase.

Desde el punto de vista del papel o misión en el programa, las variables pueden ser:

1. Variables ***miembro*** de una clase: Se definen en una clase, fuera de cualquier método; pueden ser *tipos primitivos* o *referencias*.

2. Variables ***locales***: Se definen dentro de un método o más en general *dentro de cualquier bloque* entre ***llaves*** {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también *tipos primitivos* o *referencias*.

3.11 Nombres de Variables

Los nombres de variables en ***Java*** se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por ***Java*** como operadores o separadores (,.+-*/ etc.).
Existe una serie de ***palabras reservadas*** las cuales tienen un significado especial para ***Java*** y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

Tabla 3.1 Palabras reservadas de Java

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

3.12 Tipos Primitivos de Variables

Se llaman ***tipos primitivos*** de variables de ***Java*** a aquellas variables sencillas que contienen los tipos de información más habituales: valores *boolea*n, *caracteres* y *valores numéricos* enteros o de punto flotante. ***Java*** dispone de ocho tipos primitivos de variables: un tipo para almacenar valores ***true*** y ***false (boolea***n); un tipo para almacenar caracteres ***(cha***r), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros ***(byt***e, ***shor***t, ***int*** y ***lon***g) y dos para valores reales de punto flotante ***(float*** y ***doubl***e). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la Tabla 3.2.

Tipo de variable	Descripción
Boolean	1 byte. Valores true y false
Char	2 bytes. Unicode. Comprende el código ASCII
Byte	1 byte. Valor entero entre -128 y 127
Short	2 bytes. Valor entero entre -32768 y 32767
Int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes. Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tabla 3.2. Tipos primitivos de variables en Java.

Los tipos primitivos de ***Java*** tienen algunas características importantes que se resumen a

continuación:

1.El tipo ***boolean*** no es un valor numérico: sólo admite los valores ***true*** o ***fals***e. El tipo ***boolean*** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un ciclo o en una selección debe ser ***boolea***n.

2.El tipo ***char*** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.

3.Los tipos ***byt***e, ***shor***t, ***int*** y ***long*** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en ***Java*** no hay enteros ***unsigne***d.

4.Los tipos ***float*** y ***double*** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.

5.Se utiliza la palabra ***void*** para indicar la ausencia de un tipo de variable determinado.

6.A diferencia de C/C++, los tipos de variables en ***Java*** están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, un ***int*** ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador.

7.Existen extensiones de ***Java 1.2*** para aprovechar la arquitectura de los procesadores ***Inte***l, que permiten realizar operaciones de punto flotante con una precisión extendida de 80 bits.

3.13 ¿Cómo se definen e inicializan las variables?

Una variable se define especificando el ***tipo*** y el ***nombre*** de dicha variable. Estas variables pueden ser tanto de tipos ***primitivos*** como ***referencias*** a objetos de alguna clase perteneciente al ***API*** de ***Java*** o generada por el usuario. Si no se especifica un valor en su declaración, las variable ***primitivas*** se inicializan a cero (salvo *boolean* y *cha*r, que se inicializan a ***false*** y '\0'). Análogamente las variables de tipo ***referencia*** son inicializadas por defecto a un valor especial: ***nul***l.
Es importante distinguir entre la ***referencia*** a un objeto y el ***objeto*** mismo. Una ***referencia*** es una variable que indica dónde está guardado un objeto en la memoria del ordenador (a diferencia de C/C++, ***Java*** no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los ***apuntador***es). Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor ***nul***l. Si se desea que esta ***referencia*** apunte a un nuevo objeto es necesario crear el objeto utilizando el operador ***ne***w. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones). También es posible igualar la ***referencia*** declarada a otra referencia a un objeto existente previamente.

Un tipo particular de referencias son los ***arreglos*** o vectores, sean éstos de variables primitivas (por ejemplo, un vector de enteros) o de objetos. En la declaración de una referencia de tipo ***arreglo*** hay que incluir los ***corchetes*** []. En los siguientes ejemplos aparece cómo crear un vector de 10 números enteros y cómo crear un vector de elementos ***MyClas***s. ***Java*** garantiza que los elementos del vector son inicializados a ***null*** o a cero (según el tipo de dato) en caso de no indicar otro valor.

Ejemplos de declaración e inicialización de variables:

```
1.int x; // Declaración de la variable primitiva x. Se inicializa a 0

2.int y = 5; // Declaración de la variable primitiva y. Se inicializa a 5

3.MyClass unaRef; // Declaración de una referencia a un objeto MyClass.

4.// Se inicializa a null

5.unaRef = new MyClass(); // La referencia "apunta" al nuevo objeto creado

6.// Se ha utilizado el constructor por defecto

7.MyClass segundaRef = unaRef; // Declaración de una referencia a un
   objeto

8.//MyClass. Se inicializa al mismo valor que unaRef

9.int [] vector; // Declaración de un arreglo. Se inicializa a null

10.vector = new int[10]; // Vector de 10 enteros, inicializados a 0
```

```
11.double [] v={1.0, 2.65, 3.1};// Declaración e inicialización de un
   vector de 3

12.// elementos con los valores entre llaves

13.MyClass [] lista=new MyClass[5];// Se crea un vector de 5 referencias a

14.// objetos Las 5 referencias son inicializadas a null

15.lista[0] = unaRef; // Se asigna a lista[0] el mismo valor que unaRef

16.lista[1] = new MyClass(); // Se asigna a lista[1] la referencia al
   nuevo

17.//objeto El resto (lista[2]?lista[4] siguen con valor null
```

En el ejemplo mostrado las referencias ***unaRe***f, ***segundaRef*** y ***lista[0]*** actuarán sobre el mismo objeto. Es equivalente utilizar cualquiera de las referencias ya que el objeto al que se refieren es el mismo.

3.14 Visibilidad y vida de las variables

Se entiende por ***visibilida***d o ***ámbito o alcance*** de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en una expresión. En ***Java*** todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves **{}**, es decir dentro de un ***bloqu***e, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque ***if*** no serán válidas al finalizar las sentencias correspondientes a dicho ***if*** y las variables miembro de una ***clase*** (es decir declaradas entre las llaves {} de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.
Las variables miembro de una clase declaradas como ***public*** son accesibles a través de una ***referencia*** a un objeto de dicha clase utilizando el operador punto (.). Las variables miembro declaradas como ***private*** no son accesibles directamente desde otras clases. Las ***funciones miembro*** de una clase tienen acceso directo a ***todas*** las variables miembro de la clase sin necesidad de anteponer el nombre de un objeto de la clase. Sin embargo las funciones miembro de una clase ***B*** derivada de otra A, tienen acceso a todas las variables miembro de ***A*** declaradas como ***public*** o ***protecte***d, pero no a las declaradas como ***privat***e. Una clase derivada sólo puede acceder directamente a las variables y funciones miembro de su clase base declaradas como ***public*** o ***protecte***d. Otra característica del lenguaje es que es posible declarar una variable dentro de un bloque con el mismo nombre que una variable miembro, pero no con el nombre de otra variable local que ya existiera. La variable declarada dentro del bloque oculta a la variable miembro en ese bloque. Para acceder a la variable miembro oculta será preciso utilizar el operador ***thi***s, en la forma ***this.varnam***e.

Uno de los aspectos más importantes en la programación orientada a objetos (OOP) es la forma en la cual son creados y eliminados los objetos. En ***Java*** la forma de crear nuevos ***objetos*** es utilizando el operador ***ne***w. Cuando se utiliza el operador ***ne***w, la variable de tipo ***referencia*** guarda la posición de memoria donde está almacenado este nuevo objeto. Para cada objeto se lleva cuenta de por cuántas variables de tipo ***referencia*** es apuntado. La eliminación de los objetos la realiza el programa

denominado ***recolector de basura***, quien automáticamente libera o borra la memoria ocupada por un ***objeto*** cuando no existe ninguna ***referencia*** apuntando a ese objeto. Lo anterior significa que aunque una variable de tipo referencia deje de existir, el objeto al cual apunta no es eliminado si hay otras referencias apuntando a ese mismo objeto.

3.14.1 Casos especiales: Clases BigInteger y BigDecimal

Java 1.1 incorporó dos nuevas clases destinadas a operaciones aritméticas que requieran gran precisión: ***BigInteger*** y ***BigDecimal***. La forma de operar con objetos de estas clases difiere de las operaciones con variables primitivas. En este caso hay que realizar las operaciones utilizando métodos propios de estas clases ***(add()*** para la suma, ***subtract()*** para la resta, ***divide()*** para la división, etc.). Se puede consultar la ayuda sobre el paquete ***java.math***, donde aparecen ambas clases con todos sus métodos.
Los objetos de tipo ***BigInteger*** son capaces de almacenar cualquier número entero sin perder información durante las operaciones. Análogamente los objetos de tipo ***BigDecimal*** permiten trabajar con el número de decimales deseado.

3.15 OPERADORES DE JAVA

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++. Estos operadores se describen brevemente en los apartados siguientes.

3.15.1 Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: ***suma*** (+), ***resta*** (-), ***multiplicación*** (*), ***división*** (/) y ***resto de la división*** (%).

3.15.2 Operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el ***operador igual*** (=). La forma general de las sentencias de asignación con este operador es:

```
variable = expresión;
```

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones "acumulativas" sobre una variable. La Tabla 3.3 muestra estos operadores y su equivalencia con el uso del ***operador igual*** (=).

Operador	Utilización	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

Tabla 3.3. Otros operadores de asignación.

3.15.3Operadores unarios

Los operadores ***más*** (+) y ***menos*** (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en ***Java*** es el estándar de estos operadores.

3.15.4Operador instanceof

El operador ***instanceof*** permite saber si un objeto pertenece o no a una determinada clase. Es un operador binario cuya forma general es, `objectName instanceof ClassName` y que devuelve ***true*** o ***false*** según el objeto pertenezca o no a la clase.

3.15.5Operador condicional ?:

Este operador, tomado de C/C++, permite realizar selecciones condicionales sencillas. Su forma general es la siguiente:

```
ExpresiónBooleana ? res1 : res2
```

donde se evalúa ***ExpresiónBooleana*** y se devuelve ***res1*** si el resultado es ***true*** y ***res2*** si el resultado es ***fals***e. Es el único operador ternario (tres argumentos) de ***Jav***a. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:

```
x=1 ; y=10; z = (x<y)?x+3:y+8;
```

asignarían a ***z*** el valor 4, es decir x+3.

3.15.6Operadores incrementales

Java dispone del operador ***incremento*** (++) y ***decremento*** (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:
1.Precediendo a la variable (por ejemplo: *++*i). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.

2.Siguiendo a la variable (por ejemplo: ***i++***). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en ciclos ***for*** es una de las aplicaciones más frecuentes de estos operadores.

3.15.7Operadores relacionales

Los ***operadores relacionales*** sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor ***boolean (true*** o ***fals***e) según se cumpla o no la relación considerada. La Tabla 3.4 muestra los operadores relacionales de ***Jav***a.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Tabla 3.4. Operadores relacionales.

Estos operadores se utilizan con mucha frecuencia en las ***selecciones*** y en los ***ciclo***s, que se verán en próximos apartados de este capítulo.

3.15.8Operadores lógicos

Los operadores lógicos se utilizan para construir ***expresiones lógica***s, combinando valores lógicos ***(true*** y/o ***fals***e) o los resultados de los operadores ***relacionale***s. La Tabla 3.5 muestra los operadores lógicos de ***Jav***a. Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser ***true*** y el primero es ***fals***e, ya se sabe que la condición de que ambos sean ***true*** no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalúa op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

Tabla 3.5. Operadores lógicos.

3.15.9 Operador de concatenación de cadenas de caracteres (+)

El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método ***println()***. La variable numérica ***result*** es convertida automáticamente por ***Java*** en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

3.15.10 Operadores que actúan a nivel de bits

Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o ***bandera***s, esto es, variables de tipo entero en las que cada uno de sus bits indican si una opción está activada o no. La Tabla 3.6 muestra los operadores de ***Java*** que actúan a nivel de bits.

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2 (positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits (1 si sólo uno de los operandos es 1)
~	~op2	Operador complemento (invierte el valor de cada bit)

Tabla 3.6. Operadores a nivel de bits.

En binario, las potencias de dos se representan con un único bit activado. Por ejemplo, los números (1, 2, 4, 8, 16, 32, 64, 128) se representan respectivamente de modo binario en la forma (00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000), utilizando sólo 8 bits. La suma de estos números permite construir una variable ***banderas*** con los bits activados que se deseen. Por ejemplo, para construir una variable ***banderas*** que sea 00010010 bastaría hacer ***bandera***s=2+16. Para saber si el segundo bit por la derecha está o no activado bastaría utilizar la sentencia,

```
if (banderas & 2 == 2) {...}
```

La Tabla 3.7 muestra los operadores de asignación a nivel de bits.

Operador	Utilización	Equivalente a
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Tabla 3.7. Operadores de asignación a nivel de bits.

3.15.11 Precedencia de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de *x/y*z* depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en un sentencia, de ***mayor a menor*** precedencia:

operadores postfijos `[] . (parámetros) expresión++ expresión--`

operadores unarios `++expresión --expresión +expresión -expresión`

`~ !`

creación o cast `new (type)expresión`

productos `* / %`

aditivos `+ -`

corrimiento `<< >> >>>`

relacional `< > <= >= instanceof`

igualdad `== !=`

AND de bits `&`

OR exclusivo de bits `^`

OR inclusivo de bits `|`

AND lógico `&&`

OR lógico `||`

condicional `? :`

asignación = += -= *= /= %= &= ^= |= <<= >>= >>>=

En ***Jav***a, todos los operadores binarios, excepto los operadores de asignación, se evalúan de ***izquierda a derech***a. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

3.16 ESTRUCTURAS DE PROGRAMACIÓN

En este apartado se supone que el lector tiene algunos conocimientos de programación y por lo tanto no se explican en profundidad los conceptos que aparecen.
Las ***estructuras de programación*** o ***estructuras de control*** permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados ***selecciones*** y ***ciclo***s. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a *concepto*, aunque su *sintaxis* varía de un lenguaje a otro. La sintaxis de ***Java*** coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no suponga ninguna dificultad adicional.

3.17 Sentencias o expresiones

Una ***expresión*** es un conjunto variables unidos por ***operadore***s. Son órdenes que se le dan a la computadora para que realice una tarea determinada.
Una ***sentencia*** es una ***expresión*** que acaba en ***punto y coma*** (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:

```
i = 0; j = 5; x = i + j;// Línea compuesta de tres sentencias
```

3.18 Comentarios

Existen dos formas diferentes de introducir comentarios entre el código de ***Java*** (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.
Java interpreta que todo lo que aparece a la derecha de dos barras "//" en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos /***?***/. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
1.// Esta línea es un comentario
```

```
2.int a=1; // Comentario a la derecha de una sentencia

3.// Esta es la forma de comentar más de una línea utilizando

4.// las dos barras. Requiere incluir dos barras al comienzo de cada línea

5./* Esta segunda forma es mucho más cómoda para comentar un número elevado
   de líneas ya que sólo requiere modificar el comienzo y el final. */
```

En ***Java*** existe además una forma especial de introducir los comentarios (utilizando /**?*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las ***clases*** y ***paquetes*** desarrollados por el programador. Una vez introducidos los comentarios, el programa ***javadoc.exe*** (incluido en el ***JDK***) genera de forma automática la información de forma similar a la presentada en la propia documentación del ***JDK***. La sintaxis de estos comentarios y la forma de utilizar el programa ***javadoc.exe*** se puede encontrar en la información que viene con el ***JDK***.

3.18.1Estructuras Selectivas

Las selecciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el *flujo de ejecución* de un programa. Existen dos selecciones diferentes: ***if*** y ***switch***.

3.18.1.1 if

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor ***true***). Tiene la forma siguiente:

```
1.if (ExpresiónBooleana)

2.{

2.instrucciones;

3.}
```

Las ***llaves*** {} sirven para agrupar en un ***bloque*** las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del ***if***.

3.18.1.2 if else

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el ***else*** se ejecutan en el caso de no cumplirse la expresión de comparación ***(false)***,

```
1.if (ExpresiónBooleana)

2.{

2.instrucciones1;

3.}

3.else
```

```
4.{

5.instrucciones2;

6.}
```

3.18.1.3 if elseif else

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al ***else***.

```
1.if (ExpresiónBooleana1)

2.{

2.instrucciones1;

3.}

3.else if (ExpresiónBooleana2)

4.{

5.instrucciones2;

5.}

6.else if (ExpresiónBooleana3)

7.{

8.instrucciones3;

9.}

10.else

11.{

11.instrucciones4;

12.}
```

Véase a continuación el siguiente ejemplo:

```
1.int numero = 61; // La variable "numero" tiene dos dígitos

2.if(Math.abs(numero) < 10) // Math.abs() calcula el valor absoluto.
   (false)

3.System.out.println("Numero tiene 1 dígito ");

4.else if (Math.abs(numero) < 100) // Si numero es 61, estamos en
   este caso

5.System.out.println("Numero tiene 1 dígito ");

5.else
```

```
6.{ // Resto de los casos

7.System.out.println("Numero tiene mas de 3 dígitos ");

8.System.out.println("Se ha ejecutado la opción por defecto ");

9.}
```

3.18.1.4 switch

Se trata de una alternativa a la selección ***if elseif else*** cuando se compara la ***misma expresión*** con distintos valores. Su forma general es la siguiente:

```
1.switch (expresión)
2.{

2.case valor1:

3.instrucciones1;

4.break;

5.case valor2:

6.instrucciones2;

7.break;

8.case valor3:

9.instrucciones3;

10.break;

11.case valor4:

12.instrucciones4;

13.break;

14.case valor5:

15.instrucciones5;

16.break;

17.case valor6:

18.instrucciones6;

19.break;

20.[default:

21.instrucciones7;]

22.}
```

Las características más relevantes de ***switch*** son las siguientes:

1.Cada sentencia ***case*** se corresponde con un único valor de ***expresión***. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos. El ejemplo del Apartado 4.3.3.3 no se podría realizar utilizando ***switc***h.

2.Los valores no comprendidos en ninguna sentencia ***case*** se pueden gestionar en ***defaul***t, que es opcional.

3.En ausencia de ***brea***k, cuando se ejecuta una sentencia ***case*** se ejecutan también todas las ***case*** que van a continuación, hasta que se llega a un ***break*** o hasta que se termina el ***switc***h.

Ejemplo:

```
1.char c = (char)(Math.random()*26+'a'); // Generación aleatoria de
   letras

2.// minúsculas

3.System.out.println("La letra " + c );

3.switch (c)

4.{

5.case 'a': // Se compara con la letra a

6.case 'e': // Se compara con la letra e

7.case 'i': // Se compara con la letra i

8.case 'o': // Se compara con la letra o

9.case 'u': // Se compara con la letra u

10.System.out.println(" Es una vocal ");

11.break;

12.default:

13.System.out.println(" Es una consonante ");

14.}
```

3.18.2Ciclos

Un ***ciclo*** se utiliza para realizar un proceso repetidas veces. Se denomina también ***lazo*** o ***loo***p. El código incluido entre las ***llaves*** {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los ciclos infinitos, hecho que ocurre cuando la condición de finalizar el ciclo ***(ExpresiónBooleana***) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.
Ciclo while

Las sentencias ***instrucciones*** se ejecutan mientras ***ExpresiónBooleana*** sea ***tru***e.

```
1.while (ExpresiónBooleana)
2.{
2.instrucciones;
3.}
```

3.18.2.1 Ciclo for

La forma general del ciclo ***for*** es la siguiente:

```
for (inicialización; ExpresiónBooleana; incremento)
{
        instrucciones;
}
```

que es equivalente a utilizar ***while*** en la siguiente forma,

```
1.inicialización;
3.while (ExpresiónBooleana)
4.{
3.instrucciones;
4.incremento;
5.}
```

La sentencia o sentencias ***inicialización*** se ejecuta al comienzo del ***fo***r, e ***incremento*** después de ***instrucciones***. La ***ExpresiónBooleana*** se evalúa al comienzo de cada iteración; el ciclo termina cuando la expresión de comparación toma el valor ***fals***e. Cualquiera de las tres partes puede estar vacía. La ***inicialización*** y el ***incremento*** pueden tener varias expresiones separadas por comas.

Por ejemplo, el código situado a la izquierda produce la salida que aparece a la derecha:

Código:	***Salida:***
for(int i = 1, j = i + 10; i < 5; i++, j = 2*i)	i = 1 j = 11
{	
System.out.println(" i = " + i + " j = " + j);	i = 2 j = 4
}	i = 3 j = 6
	i = 4 j = 8

3.18.2.2 Ciclo do while

Es similar al ciclo ***while*** pero con la particularidad de que el control está al final del ciclo (lo que hace que el ciclo se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados los ***instrucciones***, se evalúa la condición: si resulta ***true*** se vuelven a ejecutar las sentencias incluidas en el ciclo, mientras que si la condición se evalúa a ***false*** finaliza el ciclo. Este tipo de ciclos se utiliza con frecuencia para controlar la satisfacción de una determinada condición de error o de convergencia.

```
1.do

2.{

2.instrucciones

3.} while (ExpresiónBooleana);
```

3.18.3Estructuras Misceláneas

3.18.3.1 break y continue

La sentencia ***break*** es válida tanto para las selecciones como para los ciclos. Hace que se salga inmediatamente del ciclo o bloque que se está ejecutando, sin realizar la ejecución del resto de las sentencias.
La sentencia ***continue*** se utiliza en los ciclos (no en selecciones). Finaliza la iteración "i" que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del ciclo). Vuelve al comienzo del ciclo y comienza la siguiente iteración (i+1).

3.18.3.2 break y continue con etiquetas

Las ***etiquetas*** permiten indicar un lugar donde continuar la ejecución de un programa después de un ***break*** o ***continu***e. El único lugar donde se pueden incluir etiquetas es ***justo delante de un bloque*** de código entre llaves {} *(i*f, *switc*h, *d*o...*whil*e, *whil*e, *f*or) y sólo se deben utilizar cuando se tiene uno o más ciclos (o bloques) dentro de otro ciclo y se desea salir *(break*) o continuar con la siguiente iteración *(continu*e) de un ciclo que no es el actual.
Por tanto, la sentencia ***break Etiqueta*** finaliza el bloque que se encuentre a continuación de ***Etiqueta***. Por ejemplo, en las sentencias,

```
1.ciclo1: // etiqueta

3.for ( int i = 0, j = 0; i < 100; i++)

4.{

 3.while ( true )

 4.{

5.if( (++j) > 5)

6.{
```

```
5.break ciclo1;

6.} // Finaliza ambos ciclos

7.else

8.{

8.break;

9.} // Finaliza el ciclo interior (while)

10.}

11.}
```

la expresión `break ciclo1;` finaliza los dos ciclos simultáneamente, mientras que la expresión `break;` sale del ciclo ***while*** interior y seguiría con el ciclo ***for*** en i. Con los valores presentados ambos ciclos finalizarán con i = 5 y j = 6 (se te invita a comprobarlo).

La sentencia ***continue*** (siempre dentro de al menos un ciclo) permite transferir el control a un ciclo con nombre o etiqueta. Por ejemplo, la sentencia,

```
continue ciclo1;
```

transfiere el control al ciclo ***for*** que comienza después de la etiqueta ***ciclo1:*** para que realice una nueva iteración, como por ejemplo:

```
1.ciclo1:

2.for (int i=0; i<n; i++)

3.{

3.ciclo2:

4.for (int j=0; j<m; j++)

5.{

5....

6.if (expresión)

7.continue ciclo1;

8.then continue ciclo2;

9....

10.}

11.}
```

3.18.3.3 return

Otra forma de salir de un ciclo (y de un método) es utilizar la sentencia ***retur***n. A diferencia de ***continue*** o ***brea***k, la sentencia ***return*** sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return `(return valor;)`.

3.18.4Bloque try {...} catch {...} finally {...}

Java incorpora en el propio lenguaje el manejo de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.
En el lenguaje ***Jav***a, una ***Excepción*** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas ***excepciones*** son ***fatales*** y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras ***excepcione***s, como por ejemplo no encontrar un archivo en el que hay que leer o escribir algo, pueden ser ***recuperable***s. En este caso el programa debe dar al usuario la oportunidad de corregir el error (definiendo por ejemplo una nueva ***trayectoria*** del archivo no encontrado).

Los errores se representan mediante clases derivadas de la clase ***Throwabl***e, pero los que tiene que chequear un programador derivan de ***Exception (java.lang.Exception*** que a su vez deriva de ***Throwabl***e). Existen algunos tipos de excepciones que ***Java*** obliga a tener en cuenta. Esto se hace mediante el uso de bloques ***tr***y, ***catch*** y ***finall***y.

El código dentro del bloque ***try*** está "vigilado". Si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque ***catc***h, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques ***catch*** como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque ***finall***y, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error.

En el caso en que el código de un método pueda generar una ***Exception*** y no se desee incluir en dicho método el manejo del error (es decir los ciclos ***try/catch*** correspondientes), es necesario que el método pase la ***Exception*** al método desde el que ha sido llamado. Esto se consigue mediante la adición de la palabra ***throws*** seguida del nombre de la ***Exception*** concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques ***try/catch*** o volver a pasar la ***Exceptio***n. De esta forma se puede ir pasando la ***Exception*** de un método a otro hasta llegar al último método del programa, el método ***main()***.

En el siguiente ejemplo se presentan dos métodos que deben "controlar" una ***IOException***

relacionada con la lectura archivos y una ***MyException*** propia. El primero de ellos *(metodo*1) realiza el manejo de las excepciones y el segundo *(metodo*2) las pasa al siguiente método.

```
1.void metodo1()
```

```
2.{
2....
3.try
4.{
4....// Código que puede lanzar las excepciones IOException y MyException
5.}
5.catch (IOException e1)
6.{// Se ocupa de IOException simplemente dando aviso
7.System.out.println(e1.getMessage());
7.}
8.catch (MyException e2)
9.{
10.// Se ocupa de MyException dando un aviso y finalizando la función
10.System.out.println(e2.getMessage()); return;
11.}
12.finally
13.{ // Sentencias que se ejecutarán en cualquier caso
14....
15.}]
16....
17.} // Fin del metodo1
18.void metodo2() throws IOException, MyException
19.{
19....
20.// Código que puede lanzar las excepciones IOException y MyException
21....
22.} // Fin del metodo2
```

3.19 Un Ejemplo completo

A continuación se muestra una clase de ejemplo.

3.19.1 Clase Reloj

A continuación se muestra el programa principal, contenido en el archivo ***Reloj.jav***a. En realidad, este programa principal lo único que hace es utilizar la clase ***Tiempo***. Es pues un programa puramente "usuario", a pesar de lo cual hay que definirlo dentro de una clase, como todos los programas en ***Jav***a.

```
1./** Reloj.java */
2.class Reloj
3.{
4.public static void main(String arg[]) throws InterruptedException
5.{
6.System.out.println("Comienza main()...");
7./** Crear 5 Objetos Tiempo */
8.Tiempo t = new Tiempo(10, 20, 30);
9.Tiempo t1 = new Tiempo();
10.Tiempo t2 = new Tiempo(20);
11.Tiempo t3 = new Tiempo(14,43);
12.Tiempo t4 = new Tiempo(14,45,22);
13.Tiempo t5 = new Tiempo(25,60,70);
14.System.out.println("Hora: " + t.getHora() +
15."; Minuto: " + t.getMinuto() +
16."; Segundo: " + t.getSegundo());
17./** Cambiar Valores */
18.t.setHora(20);
19.t.setMinuto (30);
20.t.setSegundo(22);
21.System.out.println("Hora: " + t.getHora() +
22."; Minuto: " + t.getMinuto() +
23."; Segundo: " + t.getSegundo());
24.System.out.println("La hora estándar es: " +
25.toCadenaEstandar() +
26." La hora militar es: " +
27.t.toCadenaMilitar());
28.System.out.println("Varias Horas: ");
29.System.out.println("Todos los argumentos por omisión: ");
30.System.out.println("" + t1.toCadenaMilitar());
31.System.out.println("" + t1.toCadenaEstandar());
32.System.out.println("Hora especificada,minuto y segundo por omisión: ");
33.System.out.println("" + t2.toCadenaMilitar());
34.System.out.println("" + t2.toCadenaEstandar());
35.System.out.println("Hora, minuto especificados segundo por omisión: ");
36.System.out.println("" + t3.toCadenaMilitar());
37.System.out.println("" + t3.toCadenaEstandar());
38.System.out.println("Hora, minuto,segundo especificados : ");
39.System.out.println("" + t4.toCadenaMilitar());
40.System.out.println(" " + t4.toCadenaEstandar());
41.System.out.println("Valores invalidos especificados : ");
42.System.out.println("" + t5.toCadenaMilitar());
43.System.out.println("" + t5.toCadenaEstandar());
44./** Definir valores invalidos */
45.t.setTiempo(14,27,6);
46.System.out.println("La hora militar después de definirla es: "
47.+t.toCadenaMilitar());
48.System.out.println("La hora estándar después de definirla es:"
49.+t.toCadenaEstandar());
50./** Definir valores invalidos */
51.t.setTiempo(99,99,99);
52.System.out.println("Después de dar valores invalidos es: ");
```

```
53. System.out.println("Hora militar : " +t.toCadenaMilitar());
54. System.out.println("Hora estándar : " +t.toCadenaEstandar());
55. System.out.println("Termina main()...");
56.  } // fin de main()
57.} // fin de clase Reloj
```

La sentencia 1 es simplemente un comentario que contiene el nombre del archivo. El compilador de ***Java*** ignora todo lo que va desde los caracteres /** hasta /. Este comentario es apropiado para la documentación del código. Existen utilerías que buscan estos comentarios y generan la documentación del programa. Otros comentarios son /* */ y // los cuales NO se emplean con la herramienta antes mencionada

Después de la sentencia 1 se indicaría la importación de las clases de los ***paquetes*** de ***Jav***a, esto es, hacer posible acceder a dichas clases utilizando nombres cortos. Por ejemplo, se podría acceder a la clase ***ArrayList*** simplemente con el nombre ***ArrayList*** en lugar de con el nombre completo ***java.util.ArrayLis***t, con la sentencia ***import***.

```
import java.util.ArrayList;
```

Un ***paquete*** es una agrupación de clases que tienen una finalidad relacionada. Existe una jerarquía de ***paquetes*** que se refleja en nombres compuestos, separados por un punto (.). Es habitual nombrar los ***paquetes*** con letras minúsculas (como ***java.util*** o ***java.aw***t), mientras que los nombres de las clases suelen empezar siempre por una letra mayúscula (como ***ArrayLis***t). El asterisco (*) se puede usar para indicar que se importan todas las clases del ***paquete***.

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

Hay un ***paquete***, llamado ***java.lan***g, que se importa siempre automáticamente. Las clases de ***java.lang*** se pueden utilizar directamente, sin importar el ***paquete***. De hecho en este programa se usan.

La sentencia 2 indica que se comienza a definir la clase ***Reloj***. La definición de dicha clase va entre ***llaves*** {}. Como también hay otras construcciones que van entre llaves, es habitual indentar o sangrar el código, de forma que quede claro donde empieza (línea 3) y donde termina (línea 57) la definición de la clase.En ***Java*** todo son clases: no se puede definir una variable o una función que no pertenezca a una clase. En este caso, la clase ***Reloj*** tiene como única finalidad acoger al método ***main()***, que es el programa principal del ejemplo. Las clases utilizadas por ***main()*** son mucho más importantes que la propia clase ***Reloj***. Una ***clase*** es una agrupación de ***variables miembro*** *(dato*s) y ***funciones miembro*** *(método*s) que operan sobre dichos datos y permiten comunicarse con otras clases. Las clases son verdaderos ***tipos de variables*** o datos, creados por el usuario. Un ***objeto*** (en ocasiones también llamado ***instanci***a) es una variable concreta de una clase, con su propia copia de las variables miembro.

Las líneas 4-56 contienen la definición del programa principal de la aplicación, que en ***Java*** siempre se llama ***main()***. La ejecución siempre comienza por el programa o método ***main()***. La palabra ***public*** indica que esta función puede ser utilizada por cualquier clase; la palabra ***static*** indica que es un ***método de clas***e, es decir, un método que puede ser utilizado aunque no se haya creado ningún objeto de la clase ***Reloj*** (que de hecho, no se han creado); la palabra ***void*** indica que este método no tiene valor de retorno. A continuación del nombre aparecen, entre paréntesis, los argumentos del

método. En el caso de ***main()*** el argumento es siempre un *vector* o *array* (se sabe por la presencia de los corchetes []), en este caso llamado ***arg***, de cadenas de caracteres (objetos de la clase ***String***). Estos argumentos suelen ser parámetros que se pasan al programa en el momento de comenzar la ejecución (por ejemplo, el nombre del archivo donde están los datos).

El ***cuerpo*** *(body)* del método ***main()***, definido en las líneas 5-56, va también encerrado entre ***llaves*** {...}. A un conjunto de sentencias encerrado entre llaves se le suele llamar ***bloque***. Es conveniente *indentar* para saber dónde empieza y dónde terminan los bloques del método ***main()*** y de la clase ***Reloj***. Los bloques nunca pueden estar entrecruzados; un bloque puede contener a otro, pero nunca se puede cerrar el bloque exterior antes de haber cerrado el interior.

La sentencia 6 (`System.out.println("Comienza main()...");`) ***imprime*** una *cadena de caracteres* o *String* en la salida estándar del sistema, que normalmente será una ventana de MS-DOS o una ventana especial del entorno de programación que se utilice (por ejemplo JBuilder, de Inprise Corporation). Para ello se utiliza el método ***println()***, que está asociado con una variable ***static*** llamada ***out***, perteneciente a la clase ***System*** (en el ***paquete*** por defecto, ***java.lang***). Una ***variable miembro static***, también llamada ***variable de clase***, es una variable miembro que es única para toda la clase y que existe aunque no se haya creado ningún objeto de la clase. La variable ***out*** es una variable ***static*** de la clase ***System***. La sentencia 6, al igual que las que siguen, termina con el carácter ***punto y coma*** (;).

La sentencia 8 (`Tiempo t = new Tiempo(10, 20, 30);`) es muy propia de ***Java***. En ella se crea un ***objeto*** de la clase ***Tiempo***. Esta sentencia es equivalente a las dos sentencias siguientes:

```
Tiempo t;

t = new Tiempo(10, 20, 30);
```

que quizás son más fáciles de explicar. En primer lugar se crea una ***referencia*** llamada ***t*** a un objeto de la clase ***Tiempo***. Crear una referencia es como crear un "nombre" válido para referirse a un objeto de la clase ***Tiempo***. A continuación, con el operador ***new*** se crea el objeto propiamente dicho. Puede verse que el nombre de la clase va seguido por tres argumentos entre paréntesis. Estos argumentos se le pasan al ***constructor*** de la clase como datos concretos para crear el objeto (en este caso los argumentos son la hora, el minuto y el segundo).

Interesa ahora insistir un poco más en la diferencia entre ***clase*** y ***objeto***. La *clase* ***Tiempo*** es lo genérico: es el patrón o modelo para crear tiempos concretos. El *objeto* ***t*** es un tiempo concreto, con su hora, minuto y segundo. De la clase ***Tiempo*** se pueden crear tantos objetos como se desee; la clase dice que cada objeto necesita tres datos (la hora, el minuto y el segundo) que son las ***variables miembro*** de la clase. Cada objeto tiene sus propias copias de las variables miembro, con sus propios valores, distintos de los demás objetos de la clase.

La sentencia 14 imprime por la salida estándar una cadena de texto que contiene el valor del tiempo. Esta cadena de texto se compone de tres subcadenas, unidas mediante el ***operador de concatenación*** (+). Observa cómo se accede a la hora del objeto t: el

nombre del objeto seguido del nombre del metodo getHora, unidos por el operador punto . El valor numérico de la hora se convierte automáticamente en cadena de caracteres. Otro tanto ocurre para el minuto y el segundo.

Las sentencias 9 a 13 crean nuevos objetos de la clase ***Tiempo***, llamados ***t1, t2, t3, t4*** y ***t5***.

La sentencia 18 utiliza el método ***setHora()*** de la clase ***Tiempo***. Este método asigna una hora a la variable miembro hora. Un punto importante es que todos los métodos de ***Java*** (excepto los *métodos de clase* o *stati*c) se aplican a un objeto de la clase por medio del operador punto (por ejemplo, `t.setHora()`).

La sentencia 9 crea un nuevo objeto de la clase ***Tiempo*** guardándolo en la referencia t1. En este caso no se pasan argumentos al constructor de la clase. Eso quiere decir que deberá utilizar algunos valores "por defecto" para la hora, minuto y segundo.

La sentencia 10 vuelve a utilizar un método llamado ***Tiempo()*** para crear otro tiempo ¿Se trata del mismo método de la sentencia 9, utilizado de otra forma? No. Se trata de un método diferente, aunque tenga el mismo nombre. A las funciones o métodos que son diferentes porque tienen distinto código, aunque tengan el mismo nombre, se les llama ***funciones sobrecargadas*** *(overloade*d). Las funciones sobrecargadas se diferencian por el numero y tipo de sus argumentos. El método de la sentencia 9 no tiene argumentos, mientras que el de la sentencia 10 tiene uno (en todos los casos objetos de la clase ***Tiempo***). En realidad, el método de la sentencia 10 es un ***método static*** (o ***método de clas***e), esto es, un método que no necesita ningún objeto como argumento implícito. Los métodos ***static*** suelen ir *precedidos por el nombre de la clase* y el operador punto ***(Java*** también permite que vayan precedidos por el nombre de cualquier objeto, pero es considerada una nomenclatura más confusa.). La sentencia 10 es absolutamente equivalente a la sentencia 9, pero el método ***static*** de la sentencia 10 es más "simétrico". Las sentencias 11 y 12 no requieren ya comentarios especiales.

Las sentencias 24-27 tienen que ver con la parte de formateo del ejemplo.

En las líneas 24-27 se crea 2 cadenas para dar Tiempo resultante. toCadenaEstandar proporciona la hora en el formato HH:MM:SS AM/PM. toCadenaMilitar proporciona el tiempo en el formato HH:MM:SS donde las horas estan en el rango 0-23.

3.19.2 Clase Tiempo

A continuación se muestra la definición de la clase ***Tiempo en el archivo*** Tiempo***.jav***a.

```
1. /** Definición de la clase tiempo
2. @author Hugo Pablo Leyva*/
3. public class Tiempo
4. {
5. private int hora;// 0-23
6. private int minuto;// 0-59
7. private int segundo;// 0-59
```

```
8.  /** El constructor Tiempo inicializa variables */
9.  public Tiempo (int h,int m,int s)
10. {
11. setTiempo(h,m,s);
12. }
13. /** El constructor Tiempo inicializa las variables a 0 */
14. public Tiempo ()
15. {
16. this(0,0,0);
17. }
18. /** El constructor Tiempo inicializa variables minuto y segundo a 0*/
19. public Tiempo (int h)
20. {
21. this(h,0,0);
22. }
23. /** El constructor Tiempo inicializa variablesegundo a 0 */
24. public Tiempo (int h,int m)
25. {
26. this(h,m,0);
27. }
28. /* Fijar hora en horario militar, en caso de valores
29. invalidos fijar a 0 */
30. public void setTiempo(int hora,int minuto,int segundo)
31. {
32. this.hora=((hora>=0 && hora<24)?hora:0);
33. this.minuto=((minuto>=0 && minuto<60)?minuto:0);
34. this.segundo=((segundo>=0 && segundo<60)?segundo:0);
35. }
36. /** Poner la hora */
37. public void setHora(int h)
38. {
39. hora=((h>=0 && h<24)?h:0);
40. }
41. /** Poner el minuto */
42. public void setMinuto(int m)
43. {
44. minuto=((m>=0 && m<60)?m:0);
45. }
46. /** Poner el segundo */
47. public void setSegundo(int s)
48. {
49. segundo=((s>=0 && s<60)?s:0);
50. }
51. /** Métodos get
52. Obtener la hora */
53. public int getHora()
54. {
55. return (hora);
56. }
57. /** Obtener el minuto */
```

```
58. public int getMinuto()
59. {
60. return (minuto);
61. }
62. /** Obtener el segundo */
63. public int getSegundo()
64. {
65. return (segundo);
66. }
67. /** Convertir Tiempo a una cadena en horario militar */
68. public String toCadenaMilitar()
69. {
70. return(hora<10?"0":"") + hora +
71. ":" + (minuto<10?"0":"") + minuto +
72. ":" + (segundo<10?"0":"") + segundo ;
73. }
74. /** Convertir Tiempo a una cadena en horario estándar */
75. public String toCadenaEstandar()
76. {
77. return((hora==10 || hora==0)?12:hora % 12) +
78. ":" + (minuto<10?"0":"") + minuto +
79. ":" + (segundo<10?"0":"") + segundo +
80. (hora <12 ? " AM" : " PM");
81. }
82. }
```

La sentencias 1-2 son un comentario relativo a quien hizo la clase.
Las sentencias 5-7 definen las ***variables miembro de objet***o, que son la hora, minuto y segundo.

Las sentencias 24-27 define el constructor general de la clase ***Tiempo***. Este a su vez se vale del método setTiempo para construir el objeto. Este método se halla en las sentencias 30-35. En este caso tiene una peculiaridad y es que el nombre de los argumentos ***(hora***, minuto, segundo) coincide con el nombre de las variables miembro. Esto es un problema, porque como se verá más adelante ***los argumentos de un método son variables locales*** que sólo son visibles dentro del bloque {...} del método, que se destruyen al salir del bloque y que *ocultan otras variables* de ámbito más general que tengan esos mismos nombres. En otras palabras, si en el código del constructor se utilizan las variables ***(hora***, minuto, segundo) se está haciendo referencia a los argumentos del método y no a las variables miembro. La sentencia 32 indica cómo se resuelve este problema. Para cualquier método ***no static*** de una clase, la palabra ***this*** es una referencia al objeto ;el ***argumento implícit***o, sobre el que se está aplicando el método. De esta forma, ***this.hora*** se refiere a la variable miembro, mientras que hora es el argumento del método.

Las sentencias 13-27 representan otros tres constructores de la clase (métodos sobrecargados), que se diferencian en el número y tipo de argumentos. Los tres tienen en común el realizar su papel llamando al constructor general previamente definido, al que se hace referencia con la palabra ***this*** (en este caso el significado de ***this*** no es exactamente el del argumento implícito). Al constructor de la sentencia 19 sólo no se le

pasa la hora, asignado 0 al minuto y segundo. Al constructor de la sentencia 24 se le pasa solo la hora y el minuto de la clase ***Tiempo***, al segundo se le asigna 0. El constructor de la sentencia 13 es un ***constructor por defect***o, al que no se le pasa ningún argumento, que crea un tiempo de hora, minuto, y segundo en las 0 horas con 0 minutos y 0 segundos.

Las sentencias 67 a 814 definen los métodos ***toCadenaEstandar()*** y ***toCadenaMilitar()***. Estos métodos regresan una cadena formateada en formato estándar (AM/PM) y militar (0-24 hrs.).

Las sentencias de 28-50 definen los métodos set. Estos métodos se usan para poder modificar las variables miembro. Estas variables solo deben y pueden ser modificadas por estos métodos.

Las sentencias 51-66 definen los métodos get. Estos métodos se usan para obtener los valores de las variables miembro. Estas variables solo deben y pueden obtenerse mediante estos métodos.

Tanto los métodos set como get deben de empezar con set y get respectivamente. NO se recomienda traducir set y get ya que existen herramientas que al hacer la documentación buscan los métodos con estos prefijos para indicar cuales son los métodos con los que se puede obtener y modificar el valor de las variables miembro.

3.20 Auto evaluación.

1. Menciona las principales características de Java
2. Menciona las ventajas de Java sobre C++ o lenguajes similares
3. ¿ Por qué Java puede correr en cualquier plataforma sin modificar o recompilar el código ?

4. ¿ Qué es un paquete ?

5. ¿ De qué clase se derivan implícitamente una clase en Java ?

6. ¿ Para qué se emplea this ?

7. ¿ Qué es una interface ?

8. Investiga la jerarquía de clases de Java.

9. ¿ Cuál es la estructura general de un programa en Java?

10. ¿ Cuáles son los tipos de variables con los que cuenta Java ?

11. ¿ Cuáles son los tipos primitivos de Java ?

12. ¿ En que difiere el tipo char de Java del de otros lenguajes ?

13. ¿ Cuál es el alcance de las variables en Java ?

14.¿ Cuál es la precedencia de operadores en Java?

15.¿ Cuál es la diferencia entre los operadores && y &?

16.¿ Cuáles son los tipos de comentarios que maneja Java?

17.Explica como se hace el manejo de errores en Java

18.¿ Cuál es la diferencia entre break y continue?

19.¿ Diferencia entre break y continue y sus contrapartes etiquetadas?

20.¿ Cuál es la estructura general de un programa en Java?

---

4. Estructura de Datos

4.1 Tipos de Datos Abstractos

Cuando implementamos programas que modelen el mundo real nos topamos con un problema, en los lenguajes de programación sólo contamos con unos cuantos tipos de datos y en el mundo real no es suficiente con esto. En problemas reales es frecuente que la información que se maneje no sea de tipo elemental, es decir; no solamente es de tipo entero, real, carácter, etc., sino que es mucho más compleja. Por esta y otras razones debemos de ampliar nuestra noción de tipo de dato.
Un **tipo de dato** es el conjunto de valores que puede tomar una variable. Como ejemplos tenemos a los boléanos que solo pueden tomar los valores CIERTO y FALSO.

Matemáticamente podemos extender el concepto de tipo de dato a **Tipo de Datos Abstractos.**Un TDA es un modelo matemático que define un tipo de dato con un conjunto de operaciones definidas sobre ese tipo de dato. Es posible que esas operaciones den resultados pertenecientes a otro tipo de dato, pero por lo menos una de ellas, debe de dar un resultado del tipo del TDA en cuestión. Para poder implantar un TDA mediante un lenguaje de programación se requiere de una **Estructura de Datos.**Una ED es un conjunto de variables de distintos tipos interconectadas entre sí de diferentes maneras. Las formas de representar una ED y sus interconexiones depende del lenguaje de programación con el que trabajes.

La unidad básica de una ED es llamada celda. En un lenguaje de programación las celdas se representan comúnmente mediante **registros.** Un registro es un conjunto de variables de diferente tipo referenciadas por un mismo nombre, cada variable se denomina campo. Los mecanismos mediante los cuales se interconectan las celdas son:

- **Arreglos**

- **Colecciones**

En el caso de los arreglos se maneja un arreglo de celdas.

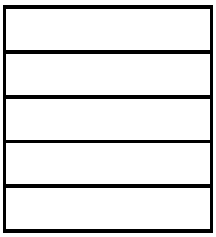


Fig. 1. Arreglo de tipo celda

Si manejamos apuntadores, las celdas tienen 2 campos: uno para el dato que se almacena y el otro es un apuntador a la siguiente celda. La última celda apunta al apuntador NULO, para indicar que no sigue otra. Esto puede visualizarse de la siguiente manera:

manzana

mandarina

naranja

piña

NULO

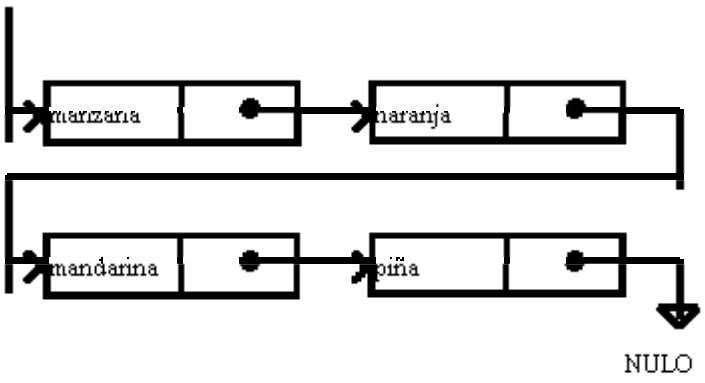


Fig. 2. Lista ligada mediante apuntadores

Existen varios TDA pero hay 3 de ellos que son los más básicos, puesto que se emplean para construir otros TDA. Estos son:

- Listas
- Pilas
- Colas

4.2 Listas

Matemáticamente, una lista es una secuencia de cero o más elementos de un tipo determinado. A menudo se representa una lista como una sucesión de elementos separados por comas:

$$a_1, a_2, a_3, a_4, \ldots a_i, \ldots a_n.$$

donde: cada a_i es del tipo elemento. Al número n de elementos se le llama longitud de la lista. Se dice que $a_1$es el primer elemento y $a_n$el último elemento. Si n=0, se tiene una lista vacía, es decir que no tiene elementos.

Una propiedad importante de una lista es que sus elementos están ordenados en forma lineal de acuerdo con sus posiciones en la misma. Se dice que $a_i$precede a a_{i+1} para i=1,2,...,n-1, y que a_i sucede a a_{i-1} para i=2,3,...,n. Se dice que el elemento a_i está en la posición i. Es conveniente postular también la existencia de una posición que sucede a la del último elemento de la lista. La función FIN(L) devolverá la posición que sigue a la posición n en una lista L de n elementos. Debemos observar que la posición FIN(L), con respecto al principio de la lista, está a una distancia que varía conforme la lista crece ose reduce, mientras que las demás posiciones guardan una posición fija con respecto al principio de la lista.

Las listas constituyen una estructura flexible en particular, porque pueden crecer y acortarse según se requiera; los elementos son accesibles y se pueden insertar y suprimir en cualquier posición de la lista. Las listas también pueden concatenarse entre sí o dividirse en sublistas. Se presentan de manera rutinaria en aplicaciones como recuperación de información, traducción de lenguajes de programación y simulación. Te presentaremos ahora un conjunto representativo de operaciones con listas. Ahí, L es una lista de objetos de tipo elemento, x es un objeto de ese tipo y p es de tipo posición. Observa que "posición" es otro tipo de datos cuya implantación cambiará con aquella que se haya elegido para las listas. Aunque de manera informal se piensa en las posiciones como enteros, en la práctica pueden tener otra representación.

1. FIN(L). Esta función regresa por definición, la posición siguiente al último elemento de la lista.

2. PRIMERO(L).Está función devuelve la primera posición de la lista L. Si L está vacía, la posición que se devuelve es FIN(L).

3.INSERTA(x,p,L). Este procedimiento inserta el elemento x en la posición p de lista L, pasando los elementos de la posición p y siguientes a la posición inmediata posterior. Esto quiere decir que si L es a_1, a_2, a_3, a_4,...,a_n. Se convierte en a_1, a_2, a_3, a_4,... a_{p-1}, x, a_{p+1},..,a_n.. Si p es FIN(L), entonces L se convierte en a_1, a_2, a_3, a_4,...,a_n ,x. Si la lista L no tiene posición p, el resultado es indefinido.

4.SUPRIME(p,L). Esta función elimina el elemento en la posición p de la lista L. Si L es a_1, a_2, a_3, a_4,... a_{p-1}, a_p, a_{p+1},..,a_n L se convierte en a_1, a_2, a_3, a_4,... a_{p-1}, a_{p+1},..,a_n . El resultado no está definido si L no tiene posición p o si p es FIN(L).

5.LOCALIZA(x,L). Esta función devuelve la posición de x en la lista L. Si x figura más de una vez en L, la posición de la primera aparición de x es la que se devuelve. Si x no figura en la lista, entonces se devuelve FIN(L).

6.RECUPERA(p,L).Esta función devuelve el elemento que está en la posición p de la lista L. El resultado no está definido si p=FIN(L) o si L no tiene posición p. Observa que si se utiliza RECUPERA, los elementos deben ser de un tipo que pueda ser devuelto por una función.

7.SIGUIENTE(p,L). Esta función devuelve la posición siguiente a p en la lista L. Si p es la última posición de L, SIGUIENTE(p,L)=FIN(L). SIGUIENTE no está definida si p es FIN(L) o si L no tiene posición p.

8.ANTERIOR(p,L). Esta función devuelve la posición anterior, a p en la lista L. ANTERIOR no está definida si p es PRIMERO(L). Está indefinida cuando L no tiene posición p.

9.ANULA_LISTA(L). Este procedimiento ocasiona que L se convierta en la lista vacía y devuelve la posición FIN(L).

10.IMPRIME_LISTA(L). Imprime los elementos de L en su orden de aparición en la lista.

11.LISTA_VACIA(L). Esta función es bolean. Regresa CIERTO si la lista esta vacía y FALSO en caso contrario.

Recordemos que todo lo anterior es matemática, las cuales definen un nuevo tipo de dato. Esto es análogo a cuando se define un vector y sus operaciones, o cuando se define una matriz junto con sus operaciones.

4.3 Pilas

Las pilas son un caso particular del TDA lista. En una lista se pueden insertar y suprimir elementos en cualquier posición. En una pila sólo se permite suprimir e insertar en un sólo extremo. Este extremo se denomina tope.

Matemáticamente, una pila es una secuencia de cero o más elementos de un tipo determinado, en la cual sólo se permite insertar y suprimir elementos en un sólo extremo denominado tope. Las pilas también se denominan ?lista mete y saca? y ?UEPS? o ?último en entrar primero en salir. Dado que sólo se permite insertar y suprimir en el tope, el concepto de posición en una pila no existe, ya el único punto de referencia de interés no esta fijo.

Las pilas se emplean en situaciones donde solo conviene trabajar con un solo extremo. Por ejemplo, en una cocina los platos de la vajilla se suelen acomodar uno sobre otro. En este caso solo conviene, cuando nos piden un plato retirarlo de la parte superior.

Las pilas se utilizan mucho en compiladores. Se emplea una pila para controlar el orden en que se llaman y terminan de ejecutarse los procedimientos y las funciones. También se emplean para evaluar expresiones aritméticas.

Las operaciones que definen completamente al TDA pila son 5. P es de tipo pila y x es de tipo elemento.

1.ANULA_PILA(P). Convierte la pila P en la pila vacía. Esta operación es exactamente la misma que para las listas generales.

2.TOPE(P). Regresa el elemento que se halla en el tope de la pila. Si la pila esta vacía el resultado es indefinido.

3.METE(x,P). Mete el elemento x en el tope de la pila P.

4.SACA(P). Saca el elemento de la pila P, que se halle en el tope de la pila. Si la pila esta vacía el resultado es indefinido.

5.PILA_VACIA(P). Esta función es bolean. Regresa CIERTO si la pila esta vacía y devuelve FALSO en caso contrario.

Dado que la pila es un caso particular de la lista, es posible expresar las operaciones de la pila en términos de las operaciones de la lista. Considerando lo anterior, tenemos las siguientes relaciones entre las operaciones de pilas y listas:

- ANULA_PILA(P)=ANULA_LISTA(P)

- TOPE(P)=REGRESA(PRIMERO(P),P)

- METE(x,P)=INSERTA(x,PRIMERO(P),P)

- SACA(P)=SUPRIME(PRIMERO(P),P)

- PILA_VACIA(P)=LISTA_VACIA(P)

Lo anterior es valido si consideramos que el tope esta al ?principio? de la pila. Si consideramos que el tope esta al ?final? de la pila tenemos:

- ANULA_PILA(P)=ANULA_LISTA(P)
- TOPE(P)=REGRESA(ANTERIOR(FIN(P)),P)
- METE(x,P)=INSERTA(x,FIN(P),P)
- SACA(P)=SUPRIME(ANTERIOR(FIN(P)),P)
- PILA_VACIA(P)=LISTA_VACIA(P)

Las pilas al igual que las lista se pueden implementar con arreglos y colecciones.

4.4 Colas

La cola es una colección ordenada de elementos, de las que se pueden borrar elementos en un extremo o insertarlos en el otro. También es un caso particular de la lista. El primer elemento insertado en una cola, es el primero en ser eliminado. Por esta razón algunas veces las colas son denominadas como listas FIFO, en oposición a la pila que es una lista LIFO. Análogamente con las pilas, no existe el concepto de posición.
Las operaciones para una cola son análogas a las de las pilas; las diferencias substanciales son que las inserciones se hacen al final de la lista y no al principio, y la terminología tradicional no son las mismas para las colas y listas. Se usaran las siguientes operaciones con colas:

1. ANULA_COLA(C). Convierte la cola C en una cola vacía.
2. FRENTE(C). Es una función que devuelve el valor del primer elemento de la cola C. Si la cola esta vacía el resultado es indefinido.
3. PONE_EN_COLA(x,C). Inserta el elemento x al final de la cola C.
4. QUITA_DE_COLA(C). Suprime el primer elemento de la cola C. Si la cola esta vacía el resultado es indefinido.
5. COLA_VACIA(C). Es bolean. Devuelve verdadero si, y solo si, C es una cola vacía.

Dado que la cola también es un caso particular de la lista, también se pueden expresar sus operaciones en términos de las operaciones de listas. Si consideramos que se inserta al final y que se suprime al principio:

- ANULA_COLA(C)=ANULA_LISTA(C)

- FRENTE(C)=REGRESA(PRIMERO(C),C)
- PONE_EN_COLA(x,C)=INSERTA(x,FIN(C),C)
- QUITA_DE_COLA(C)=SUPRIME(PRIMERO(C),C)
- COLA_VACIA(P)=LISTA_VACIA(P)

  Lo anterior es valido si consideramos que quitamos del ?final? y que insertamos al ?principio? de la cola.

- ANULA_COLA(P)=ANULA_LISTA(P)
- FRENTE(P)=REGRESA(ANTERIOR(FIN(C)),C)
- PONE_EN_COLA(x,P)=INSERTA(x,PRIMERO(C),C)
- QUITA_DE_COLA(C)=SUPRIME(ANTERIOR(FIN(C)),C)
- COLA_VACIA(P)=LISTA_VACIA(P)

  Las colas al igual que las lista se pueden implementar con arreglos y colecciones.

4.5 Árboles

Un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue de los demás llamándolo raíz. Existe una relación de parentesco entre los nodos y la raíz, que establece una estructura jerárquica. Los nodos como los elementos de una lista pueden ser de cualquier tipo que se desee. Formalmente un árbol se puede definir recursivamente como:

1. Un nodo es por sí mismo un árbol. Este nodo es también la raíz del árbol.
2. Dado un nodo n y un conjunto de árboles A_1, A_2, ?, A_m con raíces n_1, n_2, ?, n_m. Podemos construir un árbol nuevo haciendo que n sea la raíz del árbol y A_1, A_2, ?, A_m sean subárboles del nuevo árbol. Se dice que el nodo n es el padre de los subárboles y que las raíces de estos n_1, n_2, ?, n_m son los hijos.
3. Existe un árbol sin nodos denominado árbol nulo, que se representa con $\wedge$.

Como ejemplo de un árbol podemos tener la tabla de contenido de un libro. Un libro tiene un titulo y esta dividido en capítulos, as su vez los capítulos se dividen en secciones y estas en subsecciones. El titulo del libro es la raíz y los

capítulos sus hijos. A su vez las secciones serian los hijos de los capítulos y las subsecciones los hijos de las secciones.

Un Arbol de una tabla de contenido

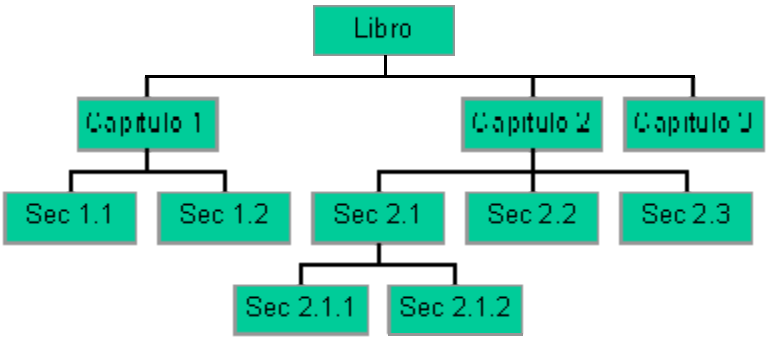


Fig. 3

4.6 Conceptos Básicos de árboles

4.6.1 Hermanos

2 o más nodos con el mismo padre son hermanos.

4.6.2 Abuelo

El padre del padre de un nodo.

4.6.3 Nieto

El hijo del hijo de un nodo.

4.6.4 Trayectoria

Si n_1, n_2, ?, n_m es una secuencia de nodos en un árbol tal que n_i es el padre de n_{i+1} con i=1,?,m entonces se dice que existe una trayectoria del nodo n_1 al nodo n_m. La longitud de la trayectoria es el numero de nodos de la misma menos 1.

4.6.5 Ancestros y Descendientes

Si existe una trayectoria de un nodo n_a a un nodo n_b se dice que n_a es un ancestro de n_b , y n_b es un descendiente de n_a .

Un ancestro o descendiente de un nodo que no sea el mismo, se dice que es un ancestro o descendiente propio respectivamente. Un nodo sin

descendientes propios es llamado una hoja. El único nodo sin ancestro propio es la raíz.

4.6.6 Peso y Profundidad

El peso de un nodo es la longitud más larga del nodo a una hoja. La profundidad de un nodo es la longitud de la única trayectoria de la raíz al nodo.

4.7 El orden de los nodos

Normalmente los hijos de un nodo se ordenan de izquierda a derecha.
Si lo deseas se puede ignorar el orden de los nodos.

4.7.1 Preorden, Enorden, Postorden

Existen 3 formas de ordenar todos los nodos de un árbol: Preorden, enorden y postorden.

Los 3 recorridos son los mismos si se tiene un solo nodo o el árbol es nulo.

El recorrido preorden es:

1. Se visita la raíz.
2. Se visita el primer hijo en preorden, luego el segundo en preorden y así sucesivamente.

El recorrido enorden es:

1. Se visita el primer hijo en enorden.
2. Se visita la raíz
3. Se visita el segundo hijo enorden, luego el tercero enorden y así sucesivamente.

El recorrido postorden es:

1. Se visita el primer hijo en postorden, luego el segundo en postorden y así sucesivamente.
2. Se visita la raíz.

4.8 Etiquetas de árboles y árboles etiquetados

Frecuentemente es útil asignar una etiqueta a cada nodo de un árbol. La etiqueta es el nombre del nodo. Se dice que un árbol es etiquetado si cada nodo tiene un nombre. Este nombre también puede interpretarse como el valor del nodo.

4.9 Determinación de Ancestros

Los recorridos preorden y postorden de un árbol tienen una propiedad muy útil para saber si un nodo es un ancestro de otro. Si postorden(n) es la posición del nodo n en un listado preorden de los nodos del árbol, y si además des(n) es el numero de descendientes propios del nodo n, entonces los números postorden asignados a los nodos en el subárbol con raíz n están numerados consecutivamente desde postorden(n)- des(n) hasta postorden(n). Para probar si un nodo x es un descendiente de un nodo y, solo se necesita comprobar sí

postorden(y) - des(y) <= postorden(x) <= postorden(y)

4.10 TDA Árbol

Matemáticamente, un árbol puede definirse como un conjunto de elementos llamados nodos, uno de los cuales es llamado la raíz. Existe una relación jerárquica de parentesco entre los nodos. Cada nodo con excepción de la raíz tiene un padre. Los nodos que tienen el mismo padre son hijos de este.

Los árboles constituyen una estructura flexible en particular, porque pueden crecer y acortarse según se requiera. Los árboles también pueden agruparse entre sí o dividirse en subárboles. Se presentan de manera rutinaria en aplicaciones como recuperación de información, traducción de lenguajes de programación y simulación. Te presentaremos ahora un conjunto representativo de operaciones con árboles. Ahí, A es un árbol, n es un nodo, x es un objeto de tipo elemento y representa la etiqueta del nodo. Existe un árbol sin nodos que se denomina árbol nulo, se representa con ^.

1. PADRE(n, A). Esta función regresa el padre de un nodo n en el árbol A. Sí el nodo es la raíz, la cual no tiene padre por definición regresa ^.

2. HIJO_MAS_A_LA_IZQUIERDA(n, A).Está función devuelve el hijo mas a la izquierda del nodo n en el árbol A. Regresa ^si el nodo es una hoja, la cual no tienen hijos.

3. HERMANO_DERECHO(n, A). Regresa el hermano derecho del nodo n en el árbol A, definido como el nodo k con el mismo padre p que el nodo n tal que k esta inmediatamente a la derecha de n en el ordenamiento de los hijos del padre p. Si n es hijo único se devuelve ^.

4. ETIQUETA(n, A). Esta función devuelve la etiqueta del nodo n en el árbol A. Es posible que existan árboles donde las etiquetas no están definidas.

5.CREAi(x,A_1,A_2,...,A_i). Es una de las funciones de una familia para cada valor de i=0, 1, ..., CREAi crea un nodo nuevo r con etiqueta x para i hijos especificados, los cuales son las raíces de los árboles A_1,A_2,...,A_i, en orden de izquierda a derecha. Se regresa la raíz r del nuevo árbol. .

6.RAIZ(A). Regresa la raíz del árbol A. Se devuelve ^ si el árbol esta vacío.

7.ANULA_ARBOL(A). Este procedimiento ocasiona que A se convierta en el árbol nulo ^.

8.ARBOL_VACIO(A). Esta función es bolean. Regresa CIERTO si el árbol esta vacío y FALSO en caso contrario.

Como ejemplo del uso de estas funciones a continuación se muestra el pseudo código del recorrido en preorden, suponiendo que las etiquetas sean cadenas. NULO es el árbol nulo ^.

```
procedimiento preorden(nodo n, arbol A)

Inicio

nodo rama

escribe(etiqueta(n))

rama=hijo_mas_a_la_izquierda(n, A)

mientras(rama<>NULO)

preorden(rama, A)

rama=hermano_derecho(rama, A)

fin_mientras

fin_procedimiento
```

Fig. 4

Como veremos mas adelante la mayoría de las operaciones de árboles se implementan en forma recursiva por naturaleza, ya que son más simples y fáciles que las no recursivas.

4.11 Árboles Binarios

Un árbol binario es aquel donde cada nodo a lo mas puede tener 2 hijos. Estos se denominan respectivamente hijo izquierdo e hijo derecho.

Para las implantaciones de los árboles solo consideraremos árboles binarios.

4.12 COLECCIONES

Java dispone también de clases e interfaces para trabajar con colecciones de objetos. En primer lugar se verán las clases Vector y Hashtable, así como la interface Enumeration. Estas clases están presentes en lenguaje desde la primera versión. Después se explicará brevemente la Java Collections Framework, introducida en la versión JDK 1.2.

Estas colecciones a su vez se pueden usar para emplear TDAS.

4.12.1 Clase Vector

La clase java.util.Vector deriva de Object, implementa Cloneable (para poder sacar copias con el método clone()) y Serializable (para poder ser convertida en cadena de caracteres).

Como su mismo nombre sugiere, Vector representa un arreglo de objetos (referencias a objetos de tipo Object) que puede crecer y reducirse, según el número de elementos. Además permite acceder a los elementos con un índice, aunque no permite utilizar los corchetes [].

El método capacity() devuelve el tamaño o número de elementos que puede tener el vector.

El método size() devuelve el número de elementos que realmente contiene, mientras que capacityIncrement es una variable que indica el salto que se dará en el tamaño cuando se necesite crecer. La Tabla 4.8 muestra los métodos más importantes de la clase Vector. Puede verse que el gran número de métodos que existen proporciona una notable flexibilidad en la utilización de esta clase.

Métodos	Función que realizan
Vector(), Vector(int), Vector(int, int)	Constructores que crean un vector vacío, un vector de la capacidad indicada y un vector de la capacidad e incremento indicados
void addElement(Object obj)	Añade un objeto al final
boolean removeElement(Object obj)	Elimina el primer objeto que encuentra como su argumento y desplaza los restantes. Si no lo encuentra devuelve false
void removeAllElements()	Elimina todos los elementos
Object clone()	Devuelve una copia del vector
void copyInto(Object anArray[])	Copia un vector en un array
void trimToSize()	Ajusta el tamaño a los elementos que tiene
void setSize(int newSize)	Establece un nuevo tamaño
int capacity()	Devuelve el tamaño (capacidad) del vector
int size()	Devuelve el número de elementos
boolean isEmpty()	Devuelve true si no tiene elementos
Enumeration elements()	Devuelve una Enumeración con los elementos
boolean contains(Object elem)	Indica si contiene o no un objeto
int indexOf(Object elem, int index)	Devuelve la posición de la primera vez que aparece un objeto a partir de una posición dada
int lastIndexOf(Object elem, int index)	Devuelve la posición de la última vez que aparece un objeto a partir de una posición, hacia atrás
Object elementAt(int index)	Devuelve el objeto en una determinada posición
Object firstElement()	Devuelve el primer elemento
Object lastElement()	Devuelve el último elemento
void setElementAt(Object obj, int index)	Cambia el elemento que está en una determinada posición
void removeElementAt(int index)	Elimina el elemento que está en una determinada posición
void insertElementAt(Object obj, int index)	Inserta un elemento por delante de una determinada posición

Tabla 1. Métodos de la clase Vector.

Además de capacityIncrement, existen otras dos variables miembro: elementCount, que representa el número de componentes válidos del vector, y elementData[] que es el arreglo de Objects donde realmente se guardan los elementos del objeto Vector (capacity es el tamaño de este arreglo). Las tres variables citadas son protected.

4.12.2 Interface Enumeration

La interface java.util.Enumeration define métodos útiles para recorrer una colección de objetos. Puede haber distintas clases que implementen esta interface y todas tendrán un comportamiento similar.

La interface Enumeration declara dos métodos:

1. public boolean hasMoreElements(). Indica si hay más elementos en la colección o si se ha llegado ya al final.

2. public Object nextElement(). Devuelve el siguiente objeto de la colección. Lanza una NoSuchElementException si se llama y ya no hay más elementos.

Ejemplo: Para imprimir los elementos de un vector vec se pueden utilizar las siguientes sentencias:

```
for (Enumeration e = vec.elements(); e.hasMoreElements(); )

{

    System.out.println(e.nextElement());

}
```

donde, como puede verse en la Tabla 4.6, el método elements() devuelve precisamente una referencia de tipo Enumeration. Con los métodos hasMoreElements() y nextElement() y un bucle for se pueden ir imprimiendo los distintos elementos del objeto ***Vecto***r.

4.12.3 Clase Hashtable

La clase java.util.Hashtable extiende Dictionary (abstract) e implementa Cloneable y Serializable.

Una tabla de hash es una tabla que relaciona una clave con un valor. Cualquier objeto distinto de null puede ser tanto clave como valor.

La clase a la que pertenecen las claves debe implementar los métodos hashCode() y equals(), con objeto de hacer búsquedas y comparaciones. El método hashCode() devuelve un entero único y distinto para cada clave, que es siempre el mismo en una ejecución del programa pero que puede cambiar de una ejecución a otra. Además, para dos claves que resultan iguales según el método equals(), el método hashCode() devuelve el mismo entero. Las tabla de hashsestán diseñadas para mantener una colección de pares ***clave/valo***r, permitiendo insertar y realizar búsquedas de un modo muy eficiente

Cada objeto de Hashtable tiene dos variables: capacity y load factor (entre 0.0 y 1.0).

Cuando el número de elementos excede el producto de estas variables, la Hashtable crece llamando al método rehash(). Un load factor más grande apura más la memoria, pero será menos eficiente en las búsquedas.

Es conveniente partir de una Hashtable suficientemente grande para no estar ampliando continuamente.

Ejemplo de definición de Hashtable:

```
Hashtable numeros = new Hashtable();

numbers.put("uno", new Integer(1));

numbers.put("dos", new Integer(2));
```

numbers.put("tres", new Integer(3));

donde se ha hecho uso del método put(). La Tabla 2 muestra los métodos de la clase Hashtable.

Métodos	Función que realizan
Hashtable(), Hashtable(int nElements), Hashtable(int nElements, float loadFactor)	Constructores
int size()	Devuelve el tamaño de la tabla
boolean isEmpty()	Indica si la tabla está vacía
Enumeration keys()	Devuelve una Enumeration con las claves
Enumeration elements()	Devuelve una Enumeration con los valores
boolean contains(Object value)	Indica si hay alguna clave que se corresponde con el valor
boolean containsKey(Object key)	Indica si existe esa clave en la tabla
Object get(Object key)	Devuelve un valor dada la clave
void rehash()	Amplía la capacidad de la tabla
Object put(Object key, Object value)	Establece una relación clave-valor
Object remove(Object key)	Elimina un valor por la clave
void clear()	Limpia la tabla
Object clone()	Hace una copia de la tabla
String toString()	Devuelve un string representando la tabla

Tabla 2. Métodos de la clase Hashtable.

4.12.4 El Collections Framework de Java 1.2

En la versión 1.2 del JDK se introdujo el Java Framework Collections o ?estructura de colecciones de Java? (en adelante JCF). Se trata de un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos. Además, constituyen un excelente ejemplo de aplicación de los conceptos propios de la programación orientada a objetos. Dada la amplitud de Java en éste y en otros aspectos se va a optar por insistir en la descripción general, dejando al lector la tarea de buscar las características concretas de los distintos métodos en la documentación de Java. En este apartado se va a utilizar una forma más breve que las tablas utilizadas en otros apartados- de informar sobre los métodos disponibles en una clase o interface.

La Figura 5 muestra la jerarquía de interfaces de la Java Collection Framework (JCF).

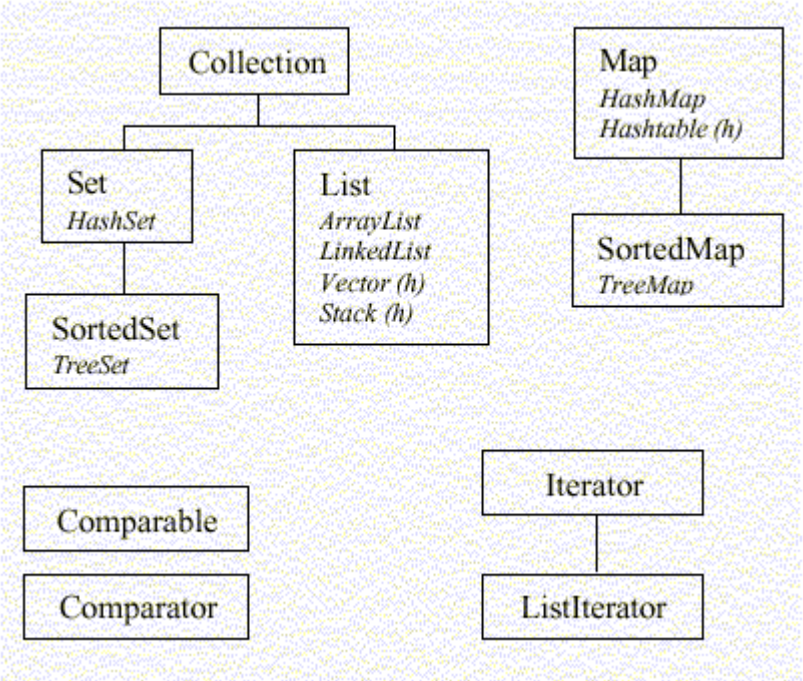


Figura 5 Interfaces de la Collection Framework.

En letra cursiva se indican las clases que implementan las correspondientes interfaces. Por ejemplo, hay dos clases que implementan la interface ***Ma***p: ***HashMap*** y ***Hashtabl***e.

Las clases vistas en los apartados anteriores son clases ?históricas?, es decir, clases que existían antes de la versión JDK 1.2. Dichas clases se denotan en la Figura 5 con la letra ?h? entre paréntesis. Aunque dichas clases se han mantenido por motivos de compatibilidad, sus métodos no siguen las reglas del diseño general del JCF; en la medida de lo posible se recomienda utilizar las nuevas clases.

En el diseño de la JCF las interfaces son muy importantes porque son ellas las que determinan las capacidades de las clases que las implementan. Dos clases que implementan la misma interface se pueden utilizar exactamente de la misma forma. Por ejemplo, las clases ArrayList y LinkedList disponen exactamente de los mismos métodos y se pueden utilizar de la misma forma. La diferencia está en la implementación: mientras que ArrayList almacena los objetos en un arreglo, la clase LinkedList los almacena en una lista vinculada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda será más flexible si se desea borrar e insertar elementos.

La Figura 6 muestra la jerarquía de clases de la JCF. En este caso, la jerarquía de clases es menos importante desde el punto de vista del usuario que la jerarquía de interfaces. En dicha figura se muestran con fondo blanco las clases abstractas, y con fondo gris claro las clases de las que se pueden crear objetos.

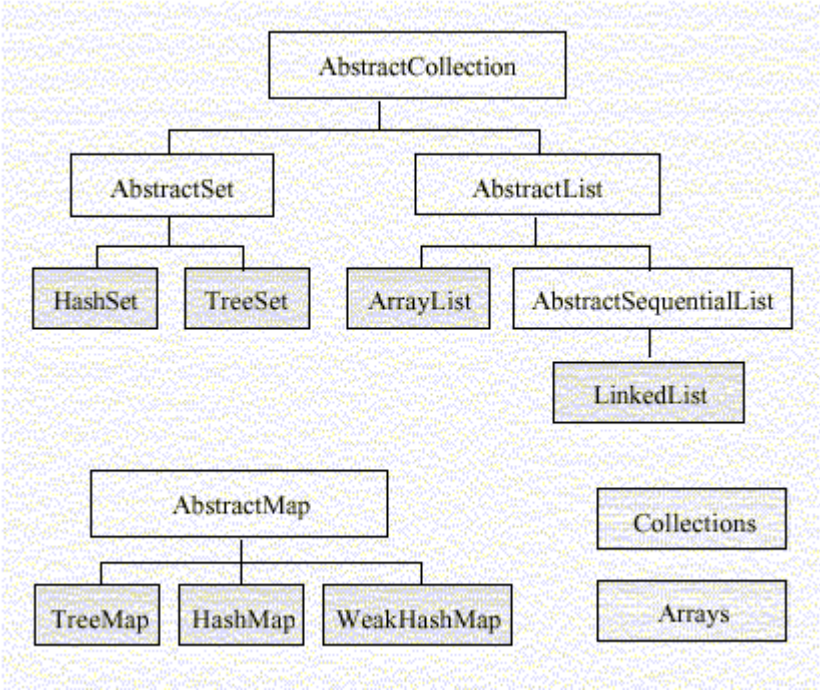


Figura 6. Jerarquía de clases de la Collection Framework.

Las clases Collections y Arrays son un poco especiales: no son abstract, pero no tienen constructores públicos con los que se puedan crear objetos. Fundamentalmente contienen métodos static para realizar ciertas operaciones de utilidad: ordenar, buscar, introducir ciertas características en objetos de otras clases, etc.

4.12.5 Interfaces de la JCF: Constituyen el elemento central de la JCF.

1.Collection: define métodos para tratar una colección genérica de elementos
2.Set: colección que no admite elementos repetidos

3.SortedSet: set cuyos elementos se mantienen ordenados según el criterio establecido

4.List: admite elementos repetidos y mantiene un orden inicial

5.Map: conjunto de pares clave/valor, sin repetición de claves

6.?SortedMap: map cuyos elementos se mantienen ordenados según el criterio establecido

4.12.6 Interfaces de soporte:

1.Iterator: sustituye a la interface Enumeration. Dispone de métodos para recorrer una colección y para borrar elementos.

2.ListIterator: deriva de Iterator y permite recorrer lists en ambos sentidos.
3.Comparable: declara el método compareTo() que permite ordenar las distintas colecciones según un orden natural (String, Date, Integer, Double, ?).

4.Comparator: declara el método compare() y se utiliza en lugar de Comparable cuando se desea ordenar objetos no estándar o sustituir a dicha interface.

4.12.7 Clases de propósito general: Son las implementaciones de las interfaces de la JFC.

1.HashSet: Interface Set implementada mediante una tabla de hash.
2.TreeSet: Interface SortedSet implementada mediante un árbol binario ordenado.

3.ArrayList: Interface List implementada mediante un arreglo.

4.LinkedList: Interface List implementada mediante una lista vinculada.

5.HashMap: Interface Map implementada mediante una tabla de hash.

6.WeakHashMap: Interface Map implementada de modo que la memoria de los pares clave/valor pueda ser liberada cuando las claves no tengan referencia desde el exterior de la WeakHashMap.

7.TreeMap: Interface SortedMap implementada mediante un árbol binario

8.Clases Wrapper: Colecciones con características adicionales, como no poder ser modificadas o estar sincronizadas. No se accede a ellas mediante constructores, sino mediante métodos ?factory? de la clase Collections.

9.Clases de utilidad: Son miniimplementaciones que permiten obtener sets especializados, como por ejemplo sets constantes de un sólo elemento (singleton) o lists con n copias del mismo elemento (nCopies). Definen las constantes EMPTY_SET y EMPTY_LIST. Se accede a través de la clase Collections.

10.Clases históricas: Son las clases Vector y Hashtable presentes desde las primeras versiones de Java.

En las versiones actuales, implementan respectivamente las interfaces List y Map, aunque conservan también los métodos anteriores.

1.Clases abstractas: Son las clases abstract de la Figura 4.2. Tienen total o parcialmente implementados los métodos de la interface

correspondiente. Sirven para que los usuarios deriven de ellas sus propias clases con un mínimo de esfuerzo de programación.

2. Algoritmos: La clase Collections dispone de métodos static para ordenar, desordenar, invertir orden, realizar búsquedas, llenar, copiar, hallar el mínimo y hallar el máximo.

3. Clase Arrays: Es una clase de utilidad introducida en el JDK 1.2 que contiene métodos static para ordenar, llenar, realizar búsquedas y comparar los arreglos clásicos del lenguaje. Permite también ver los arreglos como lists.

Después de esta visión general de la Java Collections Framework, se verán algunos detalles de las clases e interfaces más importantes.

La interface Collection es implementada por los conjuntos (sets) y las listas (lists). Esta interface declara una serie de métodos generales utilizables con Sets y Lists. La declaración o header de dichos métodos se puede ver ejecutando el comando > javap java.util.Collection en una ventana de MS-DOS.

El resultado se muestra a continuación:

```
public interface java.util.Collection

{

public abstract boolean add(java.lang.Object); // opcional

public abstract boolean addAll(java.util.Collection); // opcional

public abstract void clear(); // opcional

public abstract boolean contains(java.lang.Object);

public abstract boolean containsAll(java.util.Collection);

public abstract boolean equals(java.lang.Object);

public abstract int hashCode();

public abstract boolean isEmpty();

public abstract java.util.Iterator iterator();

public abstract boolean remove(java.lang.Object); // opcional

public abstract boolean removeAll(java.util.Collection); // opcional

public abstract boolean retainAll(java.util.Collection); // opcional
```

public abstract int size();

public abstract java.lang.Object toArray()[];

public abstract java.lang.Object toArray(java.lang.Object[])[];

}

A partir del nombre, de los argumentos y del valor de retorno, la mayor parte de estos métodos resultan autoexplicativos. A continuación se introducen algunos comentarios sobre los aspectos que pueden resultar más novedosos de estos métodos. Los detalles se pueden consultar en la documentación de Java.

Los métodos indicados como ?// opcional? (estos caracteres han sido introducidos por los autores de este manual) pueden no estar disponibles en algunas implementaciones, como por ejemplo en las clases que no permiten modificar sus objetos. Por supuesto dichos métodos deben ser definidos, pero lo que hacen al ser llamados es lanzar una UnsupportedOperationException.

El método add() trata de añadir un objeto a una colección, pero puede que no lo consiga si la colección es un set que ya tiene ese elemento. Devuelve true si el método ha llegado a modificar la colección. Lo mismo sucede con addAll(). El método remove() elimina un único elemento (si lo encuentra), y devuelve true si la colección ha sido modificada.

El método iterator() devuelve una referencia Iterator que permite recorrer una colección con los métodos next() y hasNext(). Permite también borrar el elemento actual con remove().

Los dos métodos toArray() permiten convertir una colección en un arreglo.

4.12.7.1 Interfaces Iterator y ListIterator

La interface Iterator sustituye a Enumeration, utilizada en versiones anteriores del JDK. Dispone de los métodos siguientes:

Compiled from Iterator.java

public interface java.util.Iterator

{

public abstract boolean hasNext();

public abstract java.lang.Object next();

public abstract void remove();

}

El método remove() permite borrar el último elemento accedido con next(). Es la única forma segura de eliminar un elemento mientras se está recorriendo una colección.

Los métodos de la interface ListIterator son los siguientes:

Compiled from ListIterator.java

public interface java.util.ListIterator extends java.util.Iterator

{

public abstract void add(java.lang.Object);

public abstract boolean hasNext();

public abstract boolean hasPrevious();

public abstract java.lang.Object next();

public abstract int nextIndex();

public abstract java.lang.Object previous();

public abstract int previousIndex();

public abstract void remove();

public abstract void set(java.lang.Object);

}

La interface ListIterator permite recorrer una lista en ambas direcciones, y hacer algunas modificaciones mientras se recorre. Los elementos se numeran desde 0 a n-1, pero los valores válidos para el índice son de 0 a n. Puede suponerse que el índice i está en la frontera entre los elementos i-1 e i; en ese caso previousIndex() devolvería i-1 y nextIndex() devolvería i. Si el índice es 0, previousIndex() devuelve -1 y si el índice es n nextIndex() devuelve el resultado de size().

4.12.7.2 Interfaces Comparable y Comparator

Estas interfaces están orientadas a mantener ordenadas las listas, y también los sets y maps que deben mantener un orden. Para ello se dispone de las interfaces java.lang.Comparable y java.util.Comparator (obsérvese que pertenecen a paquetes diferentes).

La interface Comparable declara el método compareTo() de la siguiente forma:

public int compareTo(Object obj)

que compara su argumento implícito con el que se le pasa por ventana. Este método devuelve un entero negativo, cero o positivo según el argumento implícito (this) sea anterior, igual o posterior al objeto obj. Las listas de objetos de clases que implementan esta interface tienen un orden natural. En Java 1.2 esta interface está implementada entre otras por las clases String, Character, Date, File, BigDecimal, BigInteger, Byte, Short, Integer, Long, Float y Double. Téngase en cuenta que la implementación estándar de estas clases no asegura un orden alfabético correcto con mayúsculas y minúsculas, y tampoco en idiomas distintos del inglés.

Si se redefine, el método compareTo() debe ser programado con cuidado: es muy conveniente que sea coherente con el método equals() y que cumpla la propiedad transitiva. Para más información, consultar la documentación del JDK 1.2.

Las listas y los arreglos cuyos elementos implementan Comparable pueden ser ordenadas con los métodos static Collections.sort() y Arrays.sort().

La interface Comparator permite ordenar listas y colecciones cuyos objetos pertenecen a clases de tipo cualquiera. Esta interface permitiría por ejemplo ordenar figuras geométricas planas por el área o el perímetro. Su papel es similar al de la interface Comparable, pero el usuario debe siempre proporcionar una implementación de esta interface. Sus dos métodos se declaran en la forma:

public int compare(Object o1, Object o2)

public boolean equals(Object obj)

El objetivo del método equals() es comparar Comparators.

El método compare() devuelve un entero negativo, cero o positivo según su primer argumento sea anterior, igual o posterior al segundo. Los objetos que implementan Comparator pueden pasarse como argumentos al método Collections.sort() o a algunos constructores de las clases TreeSet y TreeMap, con la idea de que las mantengan ordenadas de acuerdo con dicho Comparator. Es muy importante que compare() sea compatible con el método equals() de los objetos que hay que mantener ordenados. Su implementación debe cumplir unas condiciones similares a las de compareTo().

Java 1.2 dispone de clases capaces de ordenar cadenas de texto en diferentes lenguajes. Para ello se puede consultar la documentación sobre las clases CollationKey, Collator y sus clases derivadas, en el paquete java.text.

4.12.7.3 Sets y SortedSets

La interface Set sirve para acceder a una colección sin elementos repetidos. La colección puede estar o no ordenada (con un orden natural o definido por el usuario, se entiende). La interface Set no declara ningún método adicional a los de Collection.

Como un Set no admite elementos repetidos es importante saber cuándo dos objetos son

considerados iguales (por ejemplo, el usuario puede o no desear que las palabras Mesa y mesa sean consideradas iguales). Para ello se dispone de los métodos equals() y hashcode(), que el usuario puede redefinir si lo desea.

Utilizando los métodos de Collection, los Sets permiten realizar operaciones algebraicas de unión, intersección y diferencia. Por ejemplo, s1.containsAll(s2) permite saber si s2 está contenido en s1; s1.addAll(s2) permite convertir s1 en la unión de los dos conjuntos; s1.retainAll(s2) permite convertir s1 en la intersección de s1 y s2; finalmente, s1.removeAll(s2) convierte s1 en la diferencia entre s1 y s2.

La interface SortedSet extiende la interface Set y añade los siguientes métodos:

Compiled from SortedSet.java

public interface java.util.SortedSet extends java.util.Set

{

public abstract java.util.Comparator comparator();

public abstract java.lang.Object first();

public abstract java.util.SortedSet headSet(java.lang.Object);

public abstract java.lang.Object last();

public abstract java.util.SortedSet subSet(java.lang.Object,

java.lang.Object);

public abstract java.util.SortedSet tailSet(java.lang.Object);

}

que están orientados a trabajar con el ?orden?. El método comparator() permite obtener el objeto pasado al constructor para establecer el orden. Si

se ha utilizado el orden natural definido por la interface Comparable, este método devuelve null. Los métodos first() y last() devuelven el primer y último elemento del conjunto. Los métodos headSet(), subSet() y tailSet() sirven para obtener subconjuntos al principio, en medio y al final del conjunto original (los dos primeros no incluyen el límite superior especificado).

Existes dos implementaciones de conjuntos: la clase HashSet implementa la interface Set, mientras que la clase TreeSet implementa SortedSet. La primera está basada en una tabla de hash y la segunda en un TreeMap.

Los elementos de un HashSet no mantienen el orden natural, ni el orden de introducción. Los elementos de un TreeSet mantienen el orden natural o el especificado por la interfaceComparator.

Ambas clases definen constructores que admiten como argumento un objeto Collection, lo cual permite convertir un HashSet en un TreeSet y viceversa.

4.12.7.4 Listas

La interface List define métodos para operar con colecciones ordenadas y que pueden tener elementos repetidos. Por ello, dicha interface declara métodos adicionales que tienen que ver con el orden y con el acceso a elementos o rangos de elementos. Además de los métodos de Collection, la interface List declara los métodos siguientes:

Compiled from List.java

public interface java.util.List extends java.util.Collection

{

public abstract void add(int, java.lang.Object);

public abstract boolean addAll(int, java.util.Collection);

public abstract java.lang.Object get(int);

public abstract int indexOf(java.lang.Object);

public abstract int lastIndexOf(java.lang.Object);

public abstract java.util.ListIterator listIterator();

public abstract java.util.ListIterator listIterator(int);

public abstract java.lang.Object remove(int);

public abstract java.lang.Object set(int, java.lang.Object);

public abstract java.util.List subList(int, int);

}

Los nuevos métodos add() y addAll() tienen un argumento adicional para insertar elementos en una posición determinada, desplazando el elemento que estaba en esa posición y los siguientes. Los métodos get() y set() permiten obtener y cambiar el elemento en una posición dada. Los métodos indexOf() y lastIndexOf() permiten saber la posición de la primera o la última vez que un elemento aparece en la lista; si el elemento no se encuentra se devuelve -1.

El método subList(int fromIndex, toIndex) devuelve una ?vista? de la lista, desde el elemento fromIndex inclusive hasta el toIndex exclusive. Un cambio en esta ?vista? se refleja en la lista original, aunque no conviene hacer cambios simultáneamente en ambas. Lo mejor es eliminar la ?vista? cuando ya no se necesita.

Existen dos implementaciones de la interface List, que son las clases ArrayList y LinkedList. La diferencia está en que la primera almacena los elementos de la colección en un arreglo de Objects, mientras que la segunda los almacena en una lista vinculada. Los arreglos proporcionan una forma de acceder a los elementos mucho más eficiente que las listas vinculadas. Sin embargo tienen dificultades para crecer (hay que reservar memoria nueva, copiar los elementos del arreglo antiguo y liberar la memoria) y para insertar y/o borrar elementos (hay que desplazar en un sentido u en otro los elementos que están detrás del elemento borrado o insertado). Las listas vinculadas sólo permiten acceso secuencial, pero tienen una gran flexibilidad para crecer, para borrar y para insertar elementos. El optar por una implementación u otra depende del caso concreto de que se trate.

4.12.7.5 Maps y SortedMaps

Un Map es una estructura de datos agrupados en parejas clave/valor. Pueden ser considerados como una tabla de dos columnas. La clave debe ser única y se utiliza para acceder al valor.

Aunque la interface Map no deriva de Collection, es posible ver los Maps como colecciones de claves, de valores o de parejas clave/valor. A continuación se muestran los métodos de la interface Map (comando > javap java.util.Map):

Compiled from Map.java

public interface java.util.Map

{

public abstract void clear();

```
public abstract boolean containsKey(java.lang.Object);

public abstract boolean containsValue(java.lang.Object);

public abstract java.util.Set entrySet();

public abstract boolean equals(java.lang.Object);

public abstract java.lang.Object get(java.lang.Object);

public abstract int hashCode();

public abstract boolean isEmpty();

public abstract java.util.Set keySet();

public abstract java.lang.Object put(java.lang.Object, java.lang.Object);

public abstract void putAll(java.util.Map);

public abstract java.lang.Object remove(java.lang.Object);

public abstract int size();

public abstract java.util.Collection values();

public static interface java.util.Map.Entry

{

public abstract boolean equals(java.lang.Object);

public abstract java.lang.Object getKey();

public abstract java.lang.Object getValue();

public abstract int hashCode();

public abstract java.lang.Object setValue(java.lang.Object);

}

}
```

Muchos de estos métodos tienen un significado evidente, pero otros no tanto. El método entrySet() devuelve una ?vista? del Map como Set. Los elementos de este Set son referencias de la interface Map.Entry, que es una interface interna de Map. Esta ?vista? del Map como Set permite modificar y eliminar elementos del Map, pero no añadir nuevos elementos.

El método get(key) permite obtener el valor a partir de la clave. El método keySet() devuelve una ?vista? de las claves como Set. El método values() devuelve una ?vista? de los valores del Map como Collection (porque puede haber elementos repetidos). El método put() permite añadir una pareja clave/valor, mientras que putAll() vuelca todos los elementos de un Map en otro Map (los pares con clave nueva se añaden; en los pares con clave ya existente los valores nuevos sustituyen a los antiguos). El método remove() elimina una pareja clave/valor a partir de la clave.

La interface SortedMap añade los siguientes métodos, similares a los de SortedSet:

Compiled from SortedMap.java

public interface java.util.SortedMap extends java.util.Map

{

public abstract java.util.Comparator comparator();

public abstract java.lang.Object firstKey();

public abstract java.util.SortedMap headMap(java.lang.Object);

public abstract java.lang.Object lastKey();

public abstract java.util.SortedMap subMap(java.lang.Object,

java.lang.Object);

public abstract java.util.SortedMap tailMap(java.lang.Object);

}

La clase HashMap implementa la interface Map y está basada en una tabla de hash, mientras que TreeMap implementa SortedMap y está basada en un árbol binario.

4.12.7.6 Algoritmos y otras características especiales: Clases Collections y Arrays

La clase Collections (no confundir con la interface Collection, en singular) es una clase que define un buen número de métodos static con diversas finalidades. No se detallan o enumeran aquí porque exceden del espacio disponible. Los más interesantes son los siguientes:

??Métodos que definen algoritmos:

Ordenación mediante el método mergesort

public static void sort(java.util.List);

public static void sort(java.util.List, java.util.Comparator);

Eliminación del orden de modo aleatorio

public static void shuffle(java.util.List);

public static void shuffle(java.util.List, java.util.Random);

Inversión del orden establecido

public static void reverse(java.util.List);

Búsqueda en una lista

public static int binarySearch(java.util.List, java.lang.Object);

public static int binarySearch(java.util.List, java.lang.Object,

java.util.Comparator);

Copiar una lista o reemplazar todos los elementos con el elemento especificado

public static void copy(java.util.List, java.util.List);

public static void fill(java.util.List, java.lang.Object);

Cálculo de máximos y mínimos

public static java.lang.Object max(java.util.Collection);

public static java.lang.Object max(java.util.Collection, java.util.Comparator);

public static java.lang.Object min(java.util.Collection);

public static java.lang.Object min(java.util.Collection, java.util.Comparator);

??Métodos de utilidad

Set inmutable de un único elemento

public static java.util.Set singleton(java.lang.Object);

Lista inmutable con n copias de un objeto

public static java.util.List nCopies(int, java.lang.Object);

Constantes para representar el conjunto y la lista vacía

```
public static final java.util.Set EMPTY_SET;
```

public static final java.util.List EMPTY_LIST;

Además, la clase Collections dispone de dos conjuntos de métodos ?factory? que pueden ser utilizados para convertir objetos de distintas colecciones en objetos ?read only? y para convertir distintas colecciones en objetos ?synchronized? (por defecto las clases vistas anteriormente no están sincronizadas), lo cual quiere decir que se puede acceder a la colección desde distintas threads sin que se produzcan problemas. Los métodos correspondientes son los siguientes:

public static java.util.Collection synchronizedCollection(java.util.Collection);

public static java.util.List synchronizedList(java.util.List);

public static java.util.Map synchronizedMap(java.util.Map);

public static java.util.Set synchronizedSet(java.util.Set);

public static java.util.SortedMap synchronizedSortedMap(java.util.SortedMap);

public static java.util.SortedSet synchronizedSortedSet(java.util.SortedSet);

public static java.util.Collection unmodifiableCollection(java.util.Collection);

public static java.util.List unmodifiableList(java.util.List);

public static java.util.Map unmodifiableMap(java.util.Map);

public static java.util.Set unmodifiableSet(java.util.Set);

public static java.util.SortedMap unmodifiableSortedMap(java.util.SortedMap);

public static java.util.SortedSet unmodifiableSortedSet(java.util.SortedSet);

Estos métodos se utilizan de una forma muy sencilla: se les pasa como argumento una referencia a un objeto que no cumple la característica deseada y se obtiene como valor de retorno una referencia a un objeto que sí la cumple.

4.12.7.7 Desarrollo de clases por el usuario: clases abstract

Las clases abstract indicadas en la Figura 4.2, pueden servir como base para que los programadores, con necesidades no cubiertas por las clases vistas anteriormente, desarrollen sus propias clases.

4.12.7.8 Interfaces Cloneable y Serializable

Las clases HashSet, TreeSet, ArrayList, LinkedList, HashMap y TreeMap (al igual que Vector y Hashtable) implementan las interfaces Cloneable y Serializable, lo cual quiere decir que es correcto sacar copias bit a bit de sus objetos con el método Object.clone(), y que se pueden convertir en cadenas o flujos (streams) de caracteres.

Una de las ventajas de implementar la interface Serializable es que los objetos de estas clases pueden ser impresos con los métodos System.Out.print() y System.Out.println().

4.13 Autoevaluación

1. Halla la equivalencia entre las colecciones de Java y los TDA lista, pila, cola y árbol.

5 EXCEPCIONES

A diferencia de otros lenguajes de programación orientados a objetos como C/C++, ***Java*** incorpora en el propio lenguaje la manejo de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados durante este periodo.

El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje ***Jav***a, una ***Exception*** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas ***excepciones*** son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras, como por ejemplo no encontrar un archivo en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del archivo no encontrado).

Un buen programa debe manejar correctamente todas o la mayor parte de los errores que se pueden producir. Hay dos ?estilos? de hacer esto:

1. ***A la ?antigua usanza***?: los métodos devuelven un código de error. Este código se verifica en el entorno que ha llamado al método con una serie de ***if elseif*** ?, manejando de forma diferente el resultado correcto o cada uno de los posibles

errores. Este sistema resulta muy complicado cuando hay varios niveles de llamadas a los métodos.

2. ***Con soporte en el propio lenguaj***e: En este caso el propio lenguaje proporciona construcciones especiales para manejar los errores o ***Excepcion***es. Suele ser lo habitual en lenguajes modernos, como C++, Visual Basic y ***Jav***a.

En los siguientes apartados se examina cómo se trabaja con los bloques y expresiones ***tr***y, ***catc***h, ***thro***w, ***throws*** y ***finall***y, cuándo se deben lanzar excepciones, cuándo se deben capturar y cómo se crean las clases propias de tipo ***Exceptio***n.

5.1 EXCEPCIONES ESTÁNDAR DE JAVA

Los errores se representan mediante dos tipos de clases derivadas de la clase ***Throwabl***e: ***Error*** y ***Exceptio***n. La siguiente figura muestra parcialmente la jerarquía de clases relacionada con ***Throwabl***e:

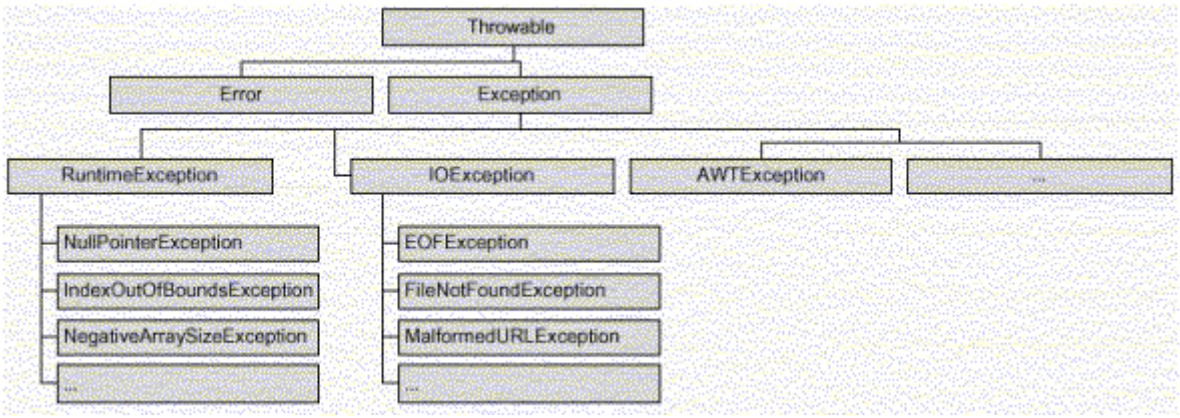


Figura 5.1: Jerarquía de clases derivadas de Throwable.

La clase ***Error*** está relacionada con errores de compilación, del sistema o de la JVM. De ordinario estos errores son ***irrecuperables*** y no dependen del programador ni debe preocuparse de capturarlos y tratarlos.

La clase ***Exception*** tiene más interés. Dentro de ella se puede distinguir:

1. ***RuntimeExceptio***n: Son excepciones muy frecuentes, de ordinario relacionadas con errores de programación. Se pueden llamar ***excepciones implícita***s.

2. Las demás clases derivadas de ***Exception*** son ***excepciones explícita***s. ***Java*** obliga a tenerlas en cuenta y verificar si se producen.

El caso de ***RuntimeException*** es un poco especial. El propio ***Java*** durante la ejecución de un programa verifica y lanza automáticamente las excepciones que derivan de ***RuntimeExceptio***n. El programador no necesita establecer los bloques ***tr***y/***catch*** para controlar este tipo de excepciones.

Representan dos casos de errores de programación:

1.Un error que normalmente no suele ser verificado por el programador, como por ejemplo recibir una referencia ***null*** en un método.

2.Un error que el programador debería haber verificado al escribir el código, como sobrepasar el tamaño asignado de un array (genera un *ArrayIndexOutOfBoundsException* automáticamente).

En realidad sería posible comprobar estos tipos de errores, pero el código se complicaría excesivamente si se necesitara verificar continuamente todo tipo de errores (que las ***referencias*** son distintas de ***null***, que todos los argumentos de los métodos son correctos, y un largo etcétera).

Las clases derivadas de ***Exception*** pueden pertenecer a distintos paquetes de ***Jav***a. Algunas pertenecen a ***java.lang*** *(Throwabl*e, *Exceptio*n, *RuntimeExceptio*n, ?); otras a ***java.io*** *(EOFExceptio*n, *FileNotFoundExceptio*n, ...) o a otros paquetes. Por heredar de ***Throwable*** todos los tipos de excepciones pueden usar los métodos siguientes:

1.String ***getMessage()*** Extrae el mensaje asociado con la excepción.

2.String ***toString()*** Devuelve un String que describe la excepción.

3.void ***printStackTrace()*** Indica el método donde se lanzó la excepción.

5.2 LANZAR UNA EXCEPTION

Cuando en un método se produce una situación anómala es necesario lanzar una excepción. El proceso de lanzamiento de una excepción es el siguiente:

1.Se crea un objeto ***Exception*** de la clase adecuada.

2.Se lanza la excepción con la sentencia ***throw*** seguida del objeto ***Exception*** creado.

```
// Código que lanza la excepción MyException una vez detectado el error

MyException me = new MyException("MyException mensaje");

throw me;
```

Esta excepción deberá ser capturada ***(catc***h) y manejada en el propio método o en algún otro lugar del programa (en otro método anterior en la ***pila*** o ***stack*** de llamadas).

Al lanzar una excepción el método termina de inmediato, sin devolver ningún valor. Solamente en el caso de que el método incluya los bloques ***try/catch/finally*** se ejecutará el bloque ***catch*** que la captura o el bloque ***finally*** (si existe).

Todo método en el que se puede producir uno o más tipos de excepciones (y que no utiliza directamente los bloques ***try/catch/finally*** para tratarlos) debe ***declararlas*** en el encabezamiento de la función por medio de la palabra ***throw***s. Si un método puede

lanzar varias excepciones, se ponen detrás de ***throws*** separadas por comas, como por ejemplo:

```
public void leerArchivo(String fich) throws EOFException,
FileNotFoundException {?}
```

Se puede poner únicamente una ***superclase de excepciones*** para indicar que se pueden lanzar excepciones de cualquiera de sus clases derivadas. El caso anterior sería equivalente a:

```
public void leerArchivo(String fich) throws IOException {?}
```

Las excepciones pueden ser lanzadas directamente por ***leerArchivo()*** o por alguno de los métodos llamados por ***leerArchivo()***, ya que las clases ***EOFException*** y ***FileNotFoundException*** derivan de ***IOExceptio***n.

Recuerda que no hace falta avisar de que se pueden lanzar objetos de la clases ***Error*** o ***RuntimeException*** (excepciones implícitas).

5.3 CAPTURAR UNA EXCEPTION

Como ya se ha visto, ciertos métodos de los paquetes de ***Java*** y algunos métodos creados por cualquier programador producen (?lanzan?) excepciones. Si el usuario llama a estos métodos sin tenerlo en cuenta se produce un error de compilación con un mensaje del tipo:

?? *Exception java.io.IOException must be*

*caugth or it must be declared in the throws clause of this metho*d?.

El programa no compilará mientras el usuario no haga una de estas dos cosas:

1. ***Manejar la excepción*** con una construcción del tipo ***try {?} catch {?***}.

2. ***Relanzar la excepción*** hacia un método anterior en el ***stac***k, declarando que su método también lanza dicha excepción, utilizando para ello la construcción ***throws*** en el header del método.

El compilador obliga a capturar las llamadas ***excepciones explícita***s, pero no protesta si se captura y luego no se hace nada con ella. En general, es conveniente por lo menos imprimir un mensaje indicando qué tipo de excepción se ha producido.

5.3.1 Bloques try y catch

En el caso de las excepciones que no pertenecen a las ***RuntimeExceptio***n y que por lo tanto ***Java*** obliga a tenerlas en cuenta habrá que utilizar los bloques ***tr***y, ***catch*** y ***finall***y. El código dentro del bloque ***try*** está ?vigilado?: Si se produce una situación anormal y se lanza por lo tanto una excepción el control salta o sale del bloque ***try*** y pasa al bloque ***catc***h, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden

incluir tantos bloques ***catch*** como sean necesarios, cada uno de los cuales tratará un tipo de excepción.

Las excepciones se pueden capturar individualmente o en grupo, por medio de una superclase de la que deriven todas ellas.

El bloque ***finally*** es opcional. Si se incluye sus sentencias se ejecutan siempre, sea cual sea la excepción que se produzca o si no se produce ninguna. El bloque ***finally*** se ejecuta aunque en el bloque ***try*** haya un ***retur***n.

En el siguiente ejemplo se presenta un método que debe "controlar" una ***IOException*** relacionada con la lectura archivos y una ***MyException*** propia:

```
void metodo1(){

...

try {

// Código que puede lanzar las excepciones IOException y MyException

} catch (IOException e1) {// Se ocupa de IOException simplemente dando aviso

System.out.println(e1.getMessage());

} catch (MyException e2) {

// Se ocupa de MyException dando un aviso y finalizando la función

System.out.println(e2.getMessage()); return;

} finally { // Sentencias que se ejecutarán en cualquier caso

...

}

...

} // Fin del metodo1
```

5.3.2 Relanzar una Exception

Existen algunos casos en los cuales el código de un método puede generar una ***Exception*** y no se desea incluir en dicho método la manejo del error. ***Java*** permite que este método pase o relance ***(throw***s) la ***Exception*** al método desde el que ha sido llamado, sin incluir en el método los bucles ***try/catch*** correspondientes. Esto se consigue mediante la adición de ***throws*** más el nombre de la ***Exception*** concreta después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques ***try/catch*** o volver a pasar la ***Exceptio***n. De esta forma se puede ir pasando la ***Exception*** de un método a otro hasta llegar al último método del programa, el método ***main()***.

El ejemplo anterior ***(metodo***1) realizaba la manejo de las excepciones dentro del propio método.

Ahora se presenta un nuevo ejemplo ***(metodo***2) que relanza las excepciones al siguiente método:

```
void metodo2() throws IOException, MyException {

...

// Código que puede lanzar las excepciones IOException y MyException

...

} // Fin del metodo2
```

Según lo anterior, si un método llama a otros métodos que pueden lanzar excepciones (por ejemplo de un paquete de ***Jav***a), tiene 2 posibilidades:

1. ***Capturar*** las posibles excepciones y manejarlas.

2. Desentenderse de las excepciones y ***remitirlas*** hacia otro método anterior en el ***stack*** para éste se encargue de manejarlas.

Si no hace ninguna de las dos cosas anteriores el compilador da un error, salvo que se trate de una ***RuntimeExceptio***n.

5.3.3 Método finally {...}

El bloque ***finally {...}*** debe ir detrás de todos los bloques ***catch*** considerados. Si se incluye (ya que es opcional) sus sentencias se ejecutan siempre, sea cual sea el tipo de excepción que se produzca, o ***incluso si no se produce ningun***a. El bloque ***finally*** se ejecuta incluso si dentro de los bloques ***try/catch*** hay una sentencia ***continu***e, ***break*** o ***retur***n. La forma general de una sección donde se controlan las excepciones es por lo tanto:

```
try {

// Código ?vigilado? que puede lanzar una excepción de tipo A, B o C

} catch (A a1) {

// Se ocupa de la excepción A

} catch (B b1) {

// Se ocupa de la excepción B

} catch (C c1) {

// Se ocupa de la excepción C

} finally {
```

```
// Sentencias que se ejecutarán en cualquier caso

}
```

El bloque ***finally*** es necesario en los casos en que se necesite recuperar o devolver a su situación original algunos elementos. No se trata de liberar la memoria reservada con ***new*** ya que de ello se ocupará automáticamente el *recolector de basura*.

Como ejemplo se podría pensar en un bloque ***try*** dentro del cual se abre un archivo para lectura y escritura de datos y se desea cerrar el archivo abierto. El archivo abierto se debe cerrar tanto si produce una excepción como si no se produce, ya que dejar un archivo abierto puede provocar problemas posteriores. Para conseguir esto se deberá incluir las sentencias correspondientes a cerrar el archivo dentro del bloque ***finally***.

5.4 CREAR NUEVAS EXCEPCIONES

El programador puede crear sus propias excepciones sólo con heredar de la clase ***Exception*** o de una de sus clases derivadas. Lo lógico es heredar de la clase de la jerarquía de ***Java*** que mejor se adapte al tipo de excepción. Las clases ***Exception*** suelen tener dos constructores:

1. Un ***constructor*** sin argumentos.

2. Un ***constructor*** que recibe un ***String*** como argumento. En este ***String*** se suele definir un mensaje que explica el tipo de excepción generada. Conviene que este constructor llame al constructor de la clase de la que deriva ***super(String)***.

Al ser clases como cualquier otra se podrían incluir variables y métodos nuevos. Por ejemplo:

```
1.class MiExcepcion extends Exception {

2.public MiExcepcion() { // Constructor por defecto

3.super();

4.}

5.public MiExcepcion(String s) { // Constructor con mensaje

6.super(s);

7.}

8.}
```

5.5 HERENCIA DE CLASES Y TRATAMIENTO DE EXCEPCIONES

Si un método redefine otro método de una superclase que utiliza ***throw***s, el método de la clase derivada no tiene obligatoriamente que poder lanzar todas las mismas excepciones de la clase base. Es posible en el método de la subclase lanzar ***las mismas excepciones o meno***s, pero no se pueden lanzar más excepciones. No puede tampoco lanzar nuevas excepciones ni excepciones de una clase más general. Se trata de una restricción muy útil ya que como consecuencia de ello el código que funciona con la clase base podrá trabajar automáticamente con referencias de clases derivadas, incluyendo el tratamiento de excepciones, concepto fundamental en la *Programación Orientada a Objetos* ***(polimorfism***o).

5.6 Autoevaluación

1.¿ De que maneras se pueden manejar los errores ?

2.¿ Qué es una excepción ?

3.¿ Cuáles son las excepciones estándar de Java?

4.¿ Cómo se lanza una excepción ?

5.¿ Cómo se atrapa una excepción ?

6.¿ Qué hace el bloque try ?

7.¿ Qué hace el bloque catch?

8.¿ Cómo se relanza una excepción ?

9.¿ Qué hace el bloque finally ?

10.¿ Cómo crear nuevas excepciones?

THREADS: PROGRAMAS MULTITAREA

Los procesadores y los Sistemas Operativos modernos permiten la ***multitare***a, es decir, la realización simultánea de dos o más actividades (al menos aparentemente). En la realidad, una computadora con un solo CPU no puede realizar dos actividades a la vez. Sin embargo los Sistemas Operativos actuales son capaces de ejecutar varios programas "simultáneamente" aunque sólo se disponga de un CPU: reparten el tiempo entre dos (o más) actividades, o bien utilizan los tiempos muertos de una actividad (por ejemplo, operaciones de lectura de datos desde el teclado) para trabajar en la otra. En computadoras con dos o más procesadores la multitarea es real, ya que cada procesador

puede ejecutar un ***hilo*** o ***thread*** diferente. La Figura 6.1, tomada del ***Tutorial*** de ***Sun***, muestra los esquemas correspondientes a un programa con uno o dos ***threads***.

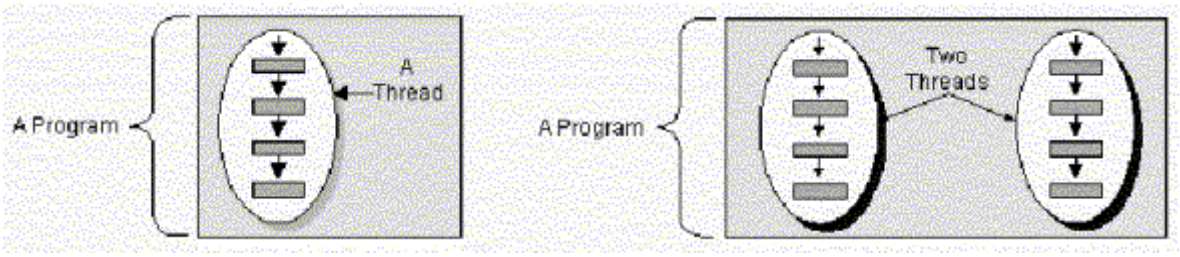


Figura 6.1. Programa con 1 y con 2 threads o hilos.

Un ***proceso*** es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un ***proceso*** simultáneamente. Un ***thread*** o ***hilo*** es un ***flujo secuencial simple*** dentro de un ***proceso***. Un único ***proceso*** puede tener varios ***hilos*** ejecutándose. Por ejemplo el programa ***Netscape*** sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un ***hilo***.

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de ***threads*** hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Los ***threads*** o ***hilos*** de ejecución permiten organizar los recursos de la computadora de forma que pueda haber varios programas actuando en paralelo. Un ***hilo*** de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastará con indicar lo que tiene que hacer en el método ***run()***, que es el que define la actividad principal de las ***threads***.

Los ***threads*** pueden ser ***demonio*** o ***no-demonio***. Son ***demonio*** aquellos hilos que realizan en ***background*** (en un segundo plano) servicios generales, esto es, tareas que no forman parte de la esencia del programa y que se están ejecutando mientras no finalice la aplicación. Un ***thread demonio*** podría ser por ejemplo él que está comprobando permanentemente si el usuario pulsa un botón. Un programa de ***Java*** finaliza cuando sólo quedan corriendo ***threads*** de tipo ***demonio***. Por defecto, y si no se indica lo contrario, los ***threads*** son del tipo ***no-demonio***.

6.1 CREACIÓN DE THREADS

En ***Java*** hay dos formas de crear nuevos ***threads***. La primera de ellas consiste en crear una nueva clase que herede de la clase ***java.lang.Thread*** y sobrecargar el método ***run()*** de dicha clase. El segundo método consiste en declarar una clase que implemente la interface ***java.lang.Runnable***, la cual declarará el método ***run()***; posteriormente se crea un objeto de tipo ***Thread*** pasándole como argumento al constructor el objeto creado de la nueva clase (la que implementa la interface ***Runnable***). Como ya se ha apuntado,

tanto la clase ***Thread*** como la interface ***Runnable*** pertenecen al paquete ***java.lang***, por lo que no es necesario importarlas.

A continuación se presentan dos ejemplos de creación de ***threads*** con cada uno de los dos métodos citados.

6.1.1 Creación de threads derivando de la clase Thread

Considera el siguiente ejemplo de declaración de una nueva clase:

```
1.public class SimpleThread extends Thread {

2.// constructor

3.public SimpleThread (String str) {

4.super(str);

5.}

6.// redefinición del método run()

7.public void run() {

8.for(int i=0;i<10;i++)

9.System.out.println("Este es el thread : " + getName());

10.}

11.}
```

En este caso, se ha creado la clase ***SimpleThread***, que hereda de ***Thread***. En su constructor se utiliza un ***String*** (opcional) para poner nombre al nuevo ***thread*** creado, y mediante ***super()*** se llama al constructor de la superclase ***Thread***. Asimismo, se redefine el método ***run()***, que define la principal actividad del ***thread***, para que escriba 10 veces el nombre del ***thread*** creado.

Para poner en marcha este nuevo ***thread*** se debe crear un objeto de la clase ***SimpleThread***, y llamar al método ***start()***,heredado de la superclase ***Thread***, que se encarga de llamar a ***run()***. Por ejemplo:

```
SimpleThread miThread = new SimpleThread(?Hilo de prueba?);

miThread.start();
```

6.1.2 Creación de threads implementando la interface Runnable

Esta segunda forma también requiere que se defina el método ***run()***, pero además es necesario crear un objeto de la clase ***Thread*** para lanzar la ejecución del nuevo hilo. Al constructor de la clase ***Thread*** hay que pasarle una referencia del objeto de la clase que implementa la interface ***Runnable***. Posteriormente, cuando se ejecute el método ***start()*** del ***thread***, éste llamará al método ***run()*** definido en la nueva clase. A continuación se

muestra el mismo estilo de clase que en el ejemplo anterior implementada mediante la interface ***Runnable***:

```
1.public class SimpleRunnable implements Runnable {
2.// se crea un nombre
3.String nameThread;
4.// constructor
5.public SimpleRunnable (String str) {
6.nameThread = str;
7.}
8.// definición del método run()
9.public void run() {
10.for(int i=0;i<10;i++)
11.             System.out.println("Este es el thread: " + nameThread);
12.}
13.}
```

El siguiente código crea un nuevo ***thread*** y lo ejecuta por este segundo procedimiento:

```
1.SimpleRunnable p = new SimpleRunnable("Hilo de prueba");
2.// se crea un objeto de la clase Thread pasándolo el objeto Runnable como
3.//argumento
4.Thread miThread = new Thread(p);
5.// se arranca el objeto de la clase Thread
6.miThread.start();
```

Este segundo método cobra especial interés con las ***applet***s, ya que cualquier ***applet*** debe heredar de la clase ***java.applet.Applet***, y por lo tanto ya no puede heredar de ***Thread***. Véase el siguiente ejemplo:

```
1.class ThreadRunnable extends Applet implements Runnable {
2.private Thread runner=null;
3.// se redefine el método start() de Applet
4.public void start() {
5.if (runner == null) {
6.runner = new Thread(this);
```

```
7.runner.start(); // se llama al método start() de Thread

8.}

9.}

10.// se redefine el método stop() de Applet

11.public void stop(){

12.runner = null; // se libera el objeto runner

13.}

14.}
```

En este ejemplo, el argumento ***this*** del constructor de ***Thread*** hace referencia al objeto ***Runnable*** cuyo método ***run()*** debería ser llamado cuando el hilo ejecutado es un objeto de ***ThreadRunnable***.

La elección de una u otra forma derivar de ***Thread*** o implementar ***Runnable*** depende del tipo de clase que se vaya a crear. Así, si la clase a utilizar ya hereda de otra clase (por ejemplo un ***applet***, que siempre hereda de ***Applet***), no quedará más remedio que implementar ***Runnable***, aunque normalmente es más sencillo heredar de ***Thread***.

6.2 CICLO DE VIDA DE UN THREAD

En el apartado anterior se ha visto cómo crear nuevos objetos que permiten incorporar en un programa la posibilidad de realizar varias tareas simultáneamente. En la Figura 7.2 (tomada del ***Tutorial*** de **Sun**) se muestran los distintos estados por los que puede pasar un ***thread*** a lo largo de su vida. Un ***thread*** puede presentar cuatro estados distintos:

1. *Nuevo* ***(New)***: El ***thread*** ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método ***start()***. Se producirá un mensaje de error ***(IllegalThreadStateException)*** si se intenta ejecutar cualquier método de la clase ***Thread*** distinto de ***start()***.

2. *Ejecutable* ***(Runnable)***: El ***thread*** puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro ***thread***.

3. *Bloqueado* ***(Blocked*** o ***Not Runnable)***: El ***thread*** podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un ***thread*** está en este estado, no se le asigna tiempo de CPU.

4. *Muerto* ***(Dead)***: La forma habitual de que un ***thread*** muera es finalizando el método ***run()***.

También puede llamarse al método ***stop()*** de la clase ***Thread***, aunque dicho método es considerado ?peligroso? y no se debe utilizar.

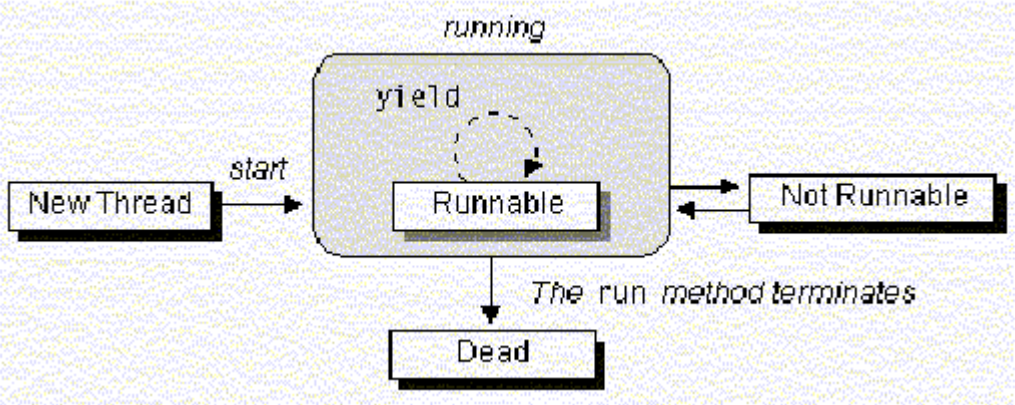


Figura 6.2. Ciclo de vida de un Thread.

A continuación se explicarán con mayor detenimiento los puntos anteriores.

6.2.1 Ejecución de un nuevo thread

La creación de un nuevo ***thread*** no implica necesariamente que se empiece a ejecutar algo. Hace falta iniciarlo con el método ***start()***, ya que de otro modo, cuando se intenta ejecutar cualquier método del ***thread*** -distinto del método ***start()***- se obtiene en tiempo de ejecución el error ***IllegalThreadStateException***.

El método ***start()*** se encarga de llamar al método ***run()*** de la clase ***Thread***. Si el nuevo ***thread*** se ha creado heredando de la clase ***Thread*** la nueva clase deberá redefinir el método ***run()*** heredado. En el caso de utilizar una clase que implemente la interface ***Runnable***, el método ***run()*** de la clase ***Thread*** se ocupa de llamar al método ***run()*** de la nueva clase.

Una vez que el método ***start()*** ha sido llamado, se puede decir ya que el ***thread*** está ?corriendo? ***(running)***, lo cual no quiere decir que se esté ejecutando en todo momento, pues ese ***thread*** tiene que compartir el tiempo del CPU con los demás ***threads*** que también estén ***running***. Por eso más bien se dice que dicha ***thread*** es ***runnable***.

6.2.2 Detener un Thread temporalmente: Runnable - Not Runnable

El sistema operativo se ocupa de asignar tiempos de CPU a los distintos ***threads*** que se estén ejecutando simultáneamente. Aun en el caso de disponer de una computadora con más de un procesador (2 ó más CPUs), el número de ***threads*** simultáneos suele siempre superar el número de CPUs, por lo que se debe repartir el tiempo de forma que parezca que todos los procesos corren a la vez (quizás más lentamente), aun cuando sólo unos pocos pueden estar ejecutándose en un instante de tiempo.

Los tiempos de CPU que el sistema continuamente asigna a los distintos ***threads*** en estado ***runnable*** se utilizan en ejecutar el método ***run()*** de cada ***thread***. Por diversos motivos, un ***thread*** puede en un determinado momento renunciar ?voluntariamente? a su tiempo de CPU y otorgárselo al sistema para que se lo asigne a otro ***thread***. Esta ?renuncia? se realiza mediante el método ***yield()***. Es importante que este método sea utilizado por las actividades que tienden a ?monopolizar? la CPU. El método ***yield()***

viene a indicar que en ese momento no es muy importante para ese ***thread*** él ejecutarse continuamente y por lo tanto tener ocupado el CPU. En caso de que ningún ***thread*** esté requiriendo el CPU para una actividad muy intensiva, el sistema volverá casi de inmediato a asignar nuevo tiempo al ***thread*** que fue ?generoso? con los demás. Por ejemplo, en un Pentium II 400 Mhz es posible llegar a más de medio millón de llamadas por segundo al método ***yield()***, dentro del método ***run()***, lo que significa que llamar al método ***yield()*** apenas detiene al ***thread***, sino que sólo ofrece el control del CPU para que el sistema decida si hay alguna otra tarea que tenga mayor prioridad.

Si lo que se desea es parar o bloquear temporalmente un ***thread*** (pasar al estado ***Not Runnable***), existen varias formas de hacerlo:

1.Ejecutando el método ***sleep()*** de la clase ***Thread***. Esto detiene el ***thread*** un tiempo preestablecido. De ordinario el método ***sleep()*** se llama desde el método ***run()***.

2.Ejecutando el método ***wait()*** heredado de la clase ***Object***, a la espera de que suceda algo que es necesario para poder continuar. El ***thread*** volverá nuevamente a la situación de ***runnable*** mediante los métodos ***notify()*** o ***notifyAll()***, que se deberán ejecutar cuando cesa la condición que tiene detenido al thread.

3.Cuando el ***thread*** está esperando para realizar operaciones de Entrada/Salida o Input/Output (E/S ó I/O).

4.Cuando el ***thread*** está tratando de llamar a un método ***synchronized*** de un objeto, y dicho objeto está bloqueado por otro ***thread.***

Un ***thread*** pasa automáticamente del estado ***Not Runnable*** a ***Runnable*** cuando cesa alguna de las condiciones anteriores o cuando se llama a ***notify()*** o ***notifyAll()***.

La clase ***Thread*** dispone también de un método ***stop()***, pero ***no se debe utilizar*** ya que puede provocar bloqueos del programa ***(deadlock)***. Hay una última posibilidad para detener un ***thread***, que consiste en ejecutar el método ***suspend()***. El ***thread*** volverá a ser ejecutable de nuevo ejecutando el método ***resume()***. Esta última forma también se desaconseja, por razones similares a la utilización del método ***stop()***.

El método ***sleep()*** de la clase ***Thread*** recibe como argumento el tiempo en *milisegundos* que ha de permanecer detenido. Adicionalmente, se puede incluir un número entero con un tiempo adicional en *nanosegundos*. Las declaraciones de estos métodos son las siguientes:

```
public static void sleep(long millis) throws InterruptedException

public static void sleep(long millis, int nanosecons) throws
InterruptedException
```

Considera el siguiente ejemplo:

```
1.System.out.println ("Contador de segundos");

2.int count=0;

3.public void run () {
```

```
4.try {

5.sleep(1000);

6.System.out.println(count++);

7.}

8.catch (InterruptedException e)

9.{

10.}

11.}
```

Se observa que el método ***sleep()*** puede lanzar una ***InterruptedException*** que ha de ser capturada. Así se ha hecho en este ejemplo, aunque luego no se gestiona esa excepción. La forma preferible de detener temporalmente un ***thread*** es la utilización conjunta de los métodos ***wait()*** y ***notifyAll()***. La principal ventaja del método ***wait()*** frente a los métodos anteriormente descritos es que libera el bloqueo del objeto. por lo que el resto de threads que se encuentran esperando para actuar sobre dicho objeto pueden llamar a sus métodos. Hay dos formas de llamar a ***wait()***:

1.Indicando el tiempo máximo que debe estar parado (en *milisegundos* y con la opción de indicar también *nanosegundos*), de forma análoga a ***sleep()***. A diferencia del método ***sleep()***, que simplemente detiene el ***thread*** el tiempo indicado, el método ***wait()*** establece el tiempo máximo que debe estar parado. Si en ese plazo se ejecutan los métodos ***notify()*** o ***notifyAll()*** que indican la liberación de los objetos bloqueados, el ***thread*** continuará sin esperar a concluir el tiempo indicado. Las dos declaraciones del método ***wait()*** son como siguen:

```
public final void wait(long timeout) throws InterruptedException

public final void wait(long timeout, int nanos) throws InterruptedException
```

2.Sin argumentos, en cuyo caso el ***thread*** permanece parado hasta que sea reinicializado explícitamente mediante los métodos ***notify()*** o ***notifyAll()***.

```
public final void wait() throws InterruptedException
```

Los métodos ***wait()*** y ***notify()*** han de estar incluidas en un método ***synchronized,*** ya que de otra forma se obtendrá una excepción del tipo ***IllegalMonitorStateException*** en tiempo de ejecución. El uso típico de ***wait()*** es el de esperar a que se cumpla alguna determinada condición, ajena al propio ***threa***d.

Cuando ésta se cumpla, se utilizará el método ***notifyAll()*** para avisar a los distintos ***threads*** que pueden utilizar el objeto.

6.2.3 Finalizar un Thread

Un ***thread*** finaliza cuando el método ***run()*** devuelve el control, por haber terminado lo que tenía que hacer (por ejemplo, un ciclo ***for*** que se ejecuta un número determinado de veces) o por haberse dejado de cumplir una condición (por ejemplo, por un ciclo ***while*** en el método ***run()***). Es habitual poner las siguientes sentencias en el caso de ***Applets Runnable***s:

```
3.public class MyApplet extends Applet implements Runnable {
4.// se crea una referencia tipo Thread
5.private Thread AppletThread;
6....
7.// método start() del Applet
8.public void start() {
9.if(AppletThread == null){ // si no tiene un objeto Thread asociado
10.AppletThread = new Thread(this, "El propio Applet");
11.AppletThread.start(); // se arranca el thread y llama a run()
12.}
13.}
14.// método stop() del Applet
15.public void stop() {
16.AppletThread = null; // iguala la referencia a null
17.}
18.// método run() por implementar Runnable
19.public void run() {
20.Thread myThread = Thread.currentThread();
21.while (myThread == AppletThread) { // hasta que stop() pare Thread
22.... // código a ejecutar
23.}
24.}
25.} // fin de la clase MyApplet
```

donde ***AppletThread*** es el ***thread*** que ejecuta el método ***run()*** MyApplet. Para finalizar el thread basta poner la referencia ***AppletThread*** a ***null***. Esto se consigue en el ejemplo con el método ***stop()*** del ***applet*** (distinto del método ***stop()*** de la clase ***Thread***, que no conviene utilizar).

Para saber si un ***thread*** está ?vivo? o no, es útil el método ***isAlive()*** de la clase ***Thread***, que devuelve ***true*** si el ***thread*** ha sido inicializado y no parado, y ***false*** si el ***thread*** es todavía nuevo (no ha sido inicializado) o ha finalizado.

6.3 SINCRONIZACIÓN

La ***sincronización*** nace de la necesidad de evitar que dos o más ***threads*** traten de acceder a los mismos recursos al mismo tiempo. Así, por ejemplo, si un ***thread*** tratara de escribir en un archivo, y otro ***thread*** estuviera al mismo tiempo tratando de borrar dicho archivo, se produciría una situación no deseada. Otra situación en la que hay que sincronizar ***threads*** se produce cuando un ***thread*** debe esperar a que estén preparados los datos que le debe suministrar el otro ***thread***. Para solucionar estos tipos de problemas es importante poder ***sincronizar*** los distintos ***thread***s.

Las secciones de código de un programa que acceden a un mismo recurso (un mismo objeto de una clase, un archivo del disco, etc.) desde dos ***threads*** distintos se denominan ***secciones críticas (critical section***s). Para sincronizar dos o más ***thread***s, hay que utilizar el modificador ***synchronized*** en aquellos métodos del ***objeto-recurso*** con los que puedan producirse situaciones conflictivas. De esta forma, ***Java*** bloquea (asocia un ***bloqueo*** o ***lock***) con el recurso sincronizado. Por ejemplo:

```
public synchronized void metodoSincronizado() {

...// accediendo por ejemplo a las variables de un objeto

...

}
```

La ***sincronización*** previene las interferencias solamente sobre un tipo de recurso: la memoria reservada para un objeto. Cuando se prevea que unas determinadas variables de una clase pueden tener problemas de sincronización, se deberán declarar como ***private*** (o ***protected***). De esta forma sólo estarán accesibles a través de métodos de la clase, que deberán estar ***sincronizado***s.

Es muy importante tener en cuenta que si se sincronizan algunos métodos de un objeto pero otros no, el programa puede no funcionar correctamente. La razón es que los métodos no sincronizados pueden acceder libremente a las variables miembro, ignorando el bloqueo del objeto. Sólo los métodos sincronizados comprueban si un objeto está bloqueado. Por lo tanto, todos los métodos que accedan a un recurso compartido deben ser declarados ***synchronized***. De esta forma, si algún método accede a un determinado recurso, ***Java*** bloquea dicho recurso, de forma que el resto de ***threads*** no puedan acceder al mismo hasta que el primero en acceder termine de realizar su tarea. ***Bloquear un recurso u objeto*** significa que sobre ese objeto no pueden actuar simultáneamente dos ***métodos sincronizado***s.

Existen dos niveles de bloqueo de un recurso. El primero es ***a nivel de objeto***s, mientras que el segundo es ***a nivel de clase***s. El primero se consigue declarando todos los métodos de una clase como ***synchronized***. Cuando se ejecuta un método ***synchronized*** sobre un objeto concreto, el sistema bloquea dicho objeto, de forma que

si otro ***thread*** intenta ejecutar algún método sincronizado de ese objeto, este segundo método se mantendrá a la espera hasta que finalice el anterior (y desbloquee por lo tanto el objeto). Si existen varios objetos de una misma clase, como los bloqueos se producen a nivel de objeto, es posible tener distintos ***threads*** ejecutando métodos sobre diversos objetos de una misma clase.

El bloqueo de recursos ***a nivel de clases*** se corresponde con los ***métodos de clase*** o ***static***, y por lo tanto con las ***variables de clase*** o ***static***. Si lo que se desea es conseguir que un método bloquee simultáneamente una clase entera, es decir todos los objetos creados de una clase, es necesario declarar este método como ***synchronized static***. Durante la ejecución de un método declarado de esta segunda forma ningún método sincronizado tendrá acceso a ningún objeto de la clase bloqueada.

La sincronización puede ser problemática y generar errores. Un ***thread*** podría bloquear un determinado recurso de forma indefinida, impidiendo que el resto de ***threads*** accedieran al mismo. Para evitar esto último, habrá que utilizar la sincronización sólo donde sea estrictamente necesario.

Es necesario tener presente que si dentro un método sincronizado se utiliza el método ***sleep()*** de la clase ***Thread***, el objeto bloqueado permanecerá en ese estado durante el tiempo indicado en el argumento de dicho método. Esto implica que otros ***threads*** no podrán acceder a ese objeto durante ese tiempo, aunque en realidad no exista peligro de simultaneidad ya que durante ese tiempo el thread que mantiene bloqueado el objeto no realizará cambios. Para evitarlo es conveniente sustituir ***sleep()*** por el método ***wait()*** de la clase ***java.lang.Object*** heredado automáticamente por todas las clases. Cuando se llama al método ***wait()*** (siempre debe hacerse desde un método o bloque ***synchronized***) se libera el bloqueo del objeto y por lo tanto es posible continuar utilizando ese objeto a través de métodos sincronizados. El método ***wait()*** detiene el ***thread*** hasta que se llame al método ***notify()*** o ***notifyAll()*** del objeto, o finalice el tiempo indicado como argumento del método ***wait()***. El método ***unObjeto.notify()*** lanza una señal indicando al sistema que puede activar uno de los ***threads*** que se encuentren bloqueados esperando para acceder al objeto ***unObjeto***. El método ***notifyAll()*** lanza una señal a todos los ***threads*** que están esperando la liberación del objeto.

Los métodos ***notify()*** y ***notifyAll***() deben ser llamados desde el ***thread*** que tiene bloqueado el objeto para activar el resto de threads que están esperando la liberación de un objeto. Un ***thread*** se convierte en propietario del bloqueo de un objeto ejecutando un método sincronizado del objeto. Los bloqueos de tipo clase, se consiguen ejecutando un ***método de clase sincronizado (synchronized static***). Véanse las dos funciones siguientes, de las que ***put()*** inserta un dato y ***get()*** lo recoge:

```
1.public synchronized int get() {

2.while (available == false) {

3.try {

4.// Espera a que put() asigne el valor y lo comunique con notify()

5.wait();

6.}
```

```
7.catch (InterruptedException e)
8.{
9.}
10.}
11.available = false;
12.// notifica que el valor ha sido leído
13.notifyAll();
14.// devuelve el valor
15.return contents;
16.}
17.public synchronized void put(int value) {
18.while (available == true) {
19.try {
20.// Espera a que get() lea el valor disponible antes de darle otro
21.wait();
22.}
23.catch (InterruptedException e)
24.{
25.}
26.}
27.// ofrece un nuevo valor y lo declara disponible
28.contents = value;
29.available = true;
30.// notifica que el valor ha sido cambiado
31.notifyAll();
32.}
```

El ciclo ***while*** de la función ***get()*** continúa ejecutándose (`avalaible == false`) hasta que el método ***put()*** haya suministrado un nuevo valor y lo indique con `avalaible = true`. En cada iteración del ***while*** la función ***wait()*** hace que el hilo que ejecuta el método ***get()*** se detenga hasta que se produzca un mensaje de que algo ha sido cambiado (en este caso con el método ***notifAll()*** ejecutado por ***put()***). El método ***put()*** funciona de forma similar.

Existe también la posibilidad de sincronizar una parte del código de un método sin necesidad de mantener bloqueado el objeto desde el comienzo hasta el final del método. Para ello se utiliza la palabra clave ***syncronized*** indicando entre paréntesis el objeto que se desea sincronizar (`synchronized(objetoASincronizar`)). Por ejemplo si se desea sincronizar el propio ***thread*** en una parte del método ***run()***, el código podría ser:

```
1.public void run() {

2.while(true) {

3....

4.syncronized(this) { // El objeto a sincronizar es el propio thread

5.... // Código sincronizado

6.}

7.try {

8.sleep(500);

9.// Se detiene el thread durante 0.5 segundos pero el objeto

10.// es accesible por otros threads al no estar sincronizado

11.}

12.catch(InterruptedException e) {

13.}

14.}

15.}
```

Un ***thread*** puede llamar a un método sincronizado de un objeto para el cual ya posee el bloqueo, volviendo a adquirir el bloqueo. Por ejemplo:

```
1.public class VolverAAdquirir {

2.public synchronized void a() {

3.b();

4.System.out.println("Estoy en a()");

5.}

6.public synchronized void b() {

7.System.out.println("Estoy en b()");

8.}

}
```

El anterior ejemplo obtendrá como resultado:

```
Estoy en b()

Estoy en a()
```

debido a que se ha podido acceder al objeto con el método ***b()*** al ser el ***thread*** que ejecuta el método ***a( )*** ?propietario? con anterioridad del bloqueo del objeto.

La sincronización es un proceso que lleva bastante tiempo al CPU, luego se debe minimizar su uso, ya que el programa será más lento cuanta más sincronización incorpore.

6.4 PRIORIDADES

Con el fin de conseguir una correcta ejecución de un programa se establecen ***prioridades*** en los ***thread***s, de forma que se produzca un reparto más eficiente de los recursos disponibles. Así, en un determinado momento, interesará que un determinado proceso acabe lo antes posible sus cálculos, de forma que habrá que otorgarle más recursos (más tiempo de CPU). Esto no significa que el resto de procesos no requieran tiempo de CPU, sino que necesitarán menos. La forma de llevar a cabo esto es gracias a las prioridades. Cuando se crea un nuevo ***thread***, éste hereda la prioridad del ***thread*** desde el que ha sido inicializado. Las prioridades viene definidas por variables miembro de la clase ***Thread***, que toman valores enteros que oscilan entre la máxima prioridad ***MAX_PRIORITY*** (normalmente tiene el valor 10) y la mínima prioridad ***MIN_PRIORITY*** (valor 1), siendo la prioridad por defecto ***NORM_PRIORITY*** (valor 5). Para modificar la prioridad de un ***thread*** se utiliza el método ***setPriority()***. Se obtiene su valor con ***getPriority()***.

El algoritmo de distribución de recursos en ***Java*** escoge por norma general aquel ***thread*** que tiene una prioridad mayor, aunque no siempre ocurra así, para evitar que algunos procesos queden ?dormidos?.

Cuando hay dos o más ***threads*** de la misma prioridad (y además, dicha prioridad es la más elevada), el sistema no establecerá prioridades entre los mismos, y los ejecutará alternativamente dependiendo del sistema operativo en el que esté siendo ejecutado. Si dicho SO soporta el ?time-slicing? (reparto del tiempo de CPU), como por ejemplo lo hace Linux, Solaris, Irix, Aix, HP-UX, Unicos, Windows 95/98/NT, los ***threads*** serán ejecutados alternativamente.

Un ***thread*** puede en un determinado momento renunciar a su tiempo de CPU y otorgárselo a otro ***thread*** de la misma prioridad, mediante el método ***yield()***, aunque en ningún caso a un ***thread*** de prioridad inferior.

6.5 GRUPOS DE THREADS

Todo hilo de ***Java*** debe formar parte de un grupo de hilos ***(ThreadGroup***). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario. Los grupos de ***threads*** proporcionan una forma sencilla de manejar múltiples ***threads*** como un solo objeto. Así, por ejemplo es posible parar varios ***threads*** con una sola llamada al

método correspondiente. Una vez que un ***thread*** ha sido asociado a un ***threadgrou***p, no puede cambiar de grupo.

Cuando se arranca un programa, el sistema crea un ***ThreadGroup*** llamado ***mai***n. Si en la creación de un nuevo ***thread*** no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al ***threadgroup*** del ***thread*** desde el que ha sido creado (conocido como ***current thread group*** y ***current threa***d, respectivamente). Si en dicho programa no se crea ningún ***ThreadGroup*** adicional, todos los ***threads*** creados pertenecerán al grupo ***main*** (en este grupo se encuentra el método ***main()***). La Figura 6.3 presenta una posible distribución de ***threads*** distribuidos en grupos de ***thread***s.

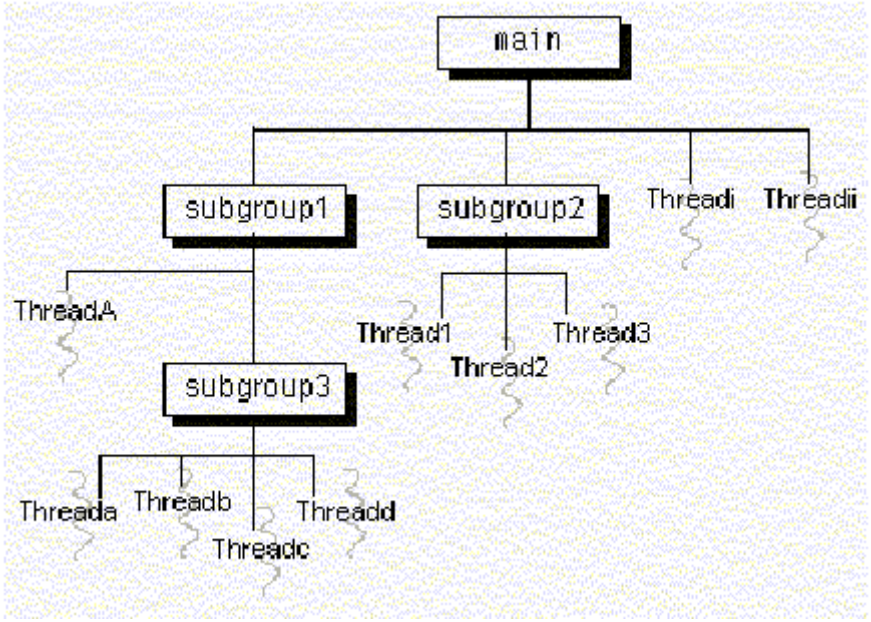


Figura 6.3. Representación de los grupos de Threads.

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo ***threa***d, según uno de los siguientes constructores:

```
public Thread (ThreadGroup grupo, Runnable destino)

public Thread (ThreadGroup grupo, String nombre)

public Thread (ThreadGroup grupo, Runnable destino, String nombre)
```

A su vez, un ***ThreadGroup*** debe pertenecer a otro ***ThreadGrou***p. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al ***ThreadGroup*** desde el que ha sido creado (por defecto al grupo ***mai***n). La clase ***ThreadGroup*** tiene dos posibles constructores:

```
ThreadGroup(ThreadGroup parent, String nombre);

ThreadGroup(String name);
```

el segundo de los cuales toma como ***parent*** el ***threadgroup*** al cual pertenezca el ***thread*** desde el que se crea ***(Threa***d.***currentThread()***). Para más información acerca

de estos constructores, dirigirse a la documentación del API de ***Java*** donde aparecen numerosos métodos para trabajar con ***grupos de threads*** a disposición del usuario (***getMaxPriority()***, ***setMaxPriority()***, ***getName()***, ***getParent()***, ***parentOf()***).

En la práctica los ***ThreadGroups*** no se suelen utilizar demasiado. Su uso práctico se limita a efectuar determinadas operaciones de forma más simple que de forma individual. En cualquier caso, véase el siguiente ejemplo:

```
ThreadGroup miThreadGroup = new ThreadGroup("Mi Grupo de Threads");

Thread miThread = new Thread(miThreadGroup, ?un thread para mi grupo");
```

donde se crea un grupo de ***threads (miThreadGrou***p) y un ***thread*** que pertenece a dicho ***grupo (miThread***).

6.6 Autoevaluación

1.¿ Qué es un proceso?

2.¿ Qué es un thread?

3.¿ Cuál es el ciclo de vida de un thread ?

4.¿ Cuáles son las formas de creación de los threads en Java ?

5.¿ Cómo se manda a ejecutar un thread?

6.¿ Cómo se detiene temporalmente un thread?

7.¿ Cómo se finaliza un thread?

8.¿ Cómo se realiza la sincronización de threads?

9.¿ Cómo se asignan las prioridades?

10.¿ Qué son los grupos de threads ?

7 EL AWT (ABSTRACT WINDOWS TOOLKIT)

7.1 ¿ QUÉ ES EL AWT ?

El AWT *(Abstract Windows Toolki*t) es la parte de ***Java*** que se ocupa de construir interfaces gráficas de usuario. Aunque el AWT ha estado presente en ***Java*** desde la

versión 1.0, la versión 1.1 representó un cambio notable, sobre todo en lo que respecta al ***modelo de eventos***. La versión 1.2 ha incorporado un modelo distinto de componentes llamado ***Swing***, que también está disponible en la versión 1.1 como paquete adicional. En este Capítulo se seguirá el AWT de ***Java 1.***1, también soportado por la versión 1.2.

7.1.1 Creación de una Interface Gráfica de Usuario

Para construir una interface gráfica de usuario hace falta:

1. Un contenedor, que es la ventana o parte de la ventana donde se situarán los componentes (botones, barras de desplazamiento, etc.) y donde se realizarán los dibujos.

2. Los ***componentes***: menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc.

3. El ***modelo de eventos***. El usuario controla la aplicación actuando sobre los componentes, de ordinario con el ratón o con el teclado. Cada vez que el usuario realiza una determinada acción, se produce el ***evento*** correspondiente, que el sistema operativo transmite al AWT. El AWT crea un ***objeto*** de una determinada clase de evento, derivada de ***AWTEvent***. Este ***evento*** es transmitido a un determinado ***método*** para que lo gestione. El componente u objeto que recibe el evento debe ?registrar? o indicar previamente qué objeto se va a hacer cargo de gestionar ese evento.

En los siguientes apartados se verán con un cierto detalle estos tres aspectos del AWT. Hay que considerar que el AWT es una parte muy extensa y complicada de ***Java***, sobre la que existen libros con muchos cientos de páginas.

7.1.2 Objetos ?event source? y objetos ?event listener?

El modelo de eventos de ***Java*** está basado en que los objetos sobre los que se producen los eventos ***(event sources)*** ?registran? los objetos que habrán de gestionarlos ***(event listeners)***, para lo cual los ***event listeners*** habrán de disponer de los ***métodos*** adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los ***event listeners*** disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface ***Listener***. Las interfaces ***Listener*** se corresponden con los tipos de ***eventos*** que se pueden producir. En los apartados siguientes se verán con más detalle los ***componentes*** que pueden recibir ***eventos***, los distintos tipos de ***eventos*** y los ***métodos*** de las interfaces ***Listener*** que hay que definir para gestionarlos. En este punto es muy importante ser capaz de buscar la información correspondiente en la documentación de ***Java***.

Las capacidades gráficas del AWT resultan pobres y complicadas en comparación con lo que se puede conseguir con ***Visual Basic***, pero tienen la ventaja de poder ser ejecutadas casi en cualquier ordenador y con cualquier sistema operativo.

7.1.3 Proceso a seguir para crear una aplicación interactiva (orientada a eventos)

Para avanzar un paso más, se resumen a continuación los pasos que se pueden seguir para construir una aplicación orientada a eventos sencilla, con interface gráfica de usuario:

1.Determinar los ***componentes*** que van a constituir la interface de usuario (botones, cajas de texto, menús, etc.).

2.Crear una ***clase*** para la aplicación que contenga la función ***main()***.

3.Crear una clase ***Ventan***a, subclase de ***Fram***e, que responda al evento ***WindowClosing()***.

4.La función ***main()*** deberá crear un objeto de la clase ***Ventana*** (en el que se van a introducir las componentes seleccionadas) y mostrarla por pantalla con el tamaño y posición adecuados.

5.Añadir al objeto ***Ventana*** todos los ***componentes*** y ***menús*** que deba contener. Se puede hacer en el constructor de la ventana o en el propio método ***main()***.

6.Definir los objetos ***Listener*** (objetos que se ocuparán de responder a los eventos, cuyas clases implementan las distintas interfaces ***Listene***r) para cada uno de los eventos que deban estar soportados. En aplicaciones pequeñas, el propio objeto ***Ventana*** se puede ocupar de responder a los eventos de sus componentes. En programas más grandes se puede crear uno o más objetos de clases especiales para ocuparse de los eventos.

7.Finalmente, se deben implementar los métodos de las interfaces ***Listener*** que se vayan a hacer cargo de la gestión de los eventos.

7.1.4 Componentes y eventos soportados por el AWT de Java

7.1.4.1 Jerarquía de Componentes

Como todas las clases de ***Jav***a, los componentes utilizados en el AWT pertenecen a una determinada jerarquía de clases, que es muy importante conocer. Esta jerarquía de clases se muestra en la Figura 7.1.

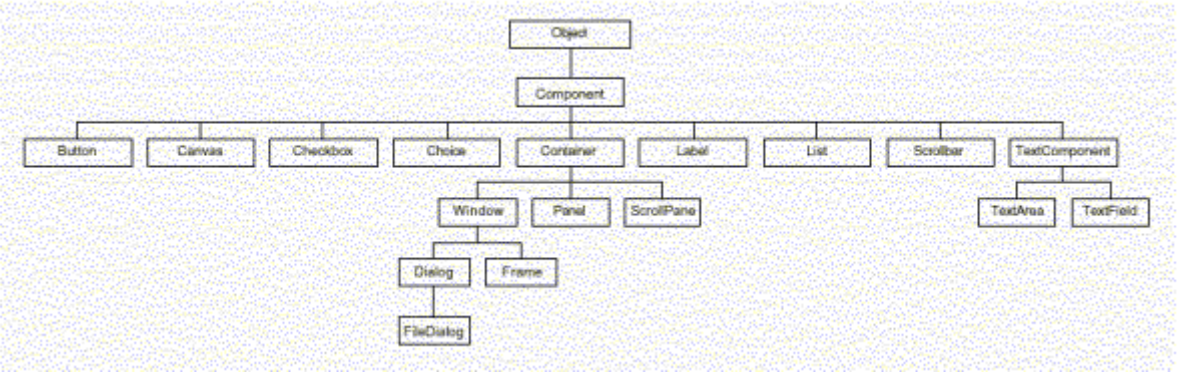


Figura 7.1. Jerarquía de clases para los componentes del AWT.

Todos los componentes descienden de la clase ***Component***, de la que pueden ya heredar algunos métodos interesantes. El ***paquete*** al que pertenecen estas clases se llama ***java.awt***.

A continuación se resumen algunas características importantes de los componentes mostrados en la Figura 7.2:

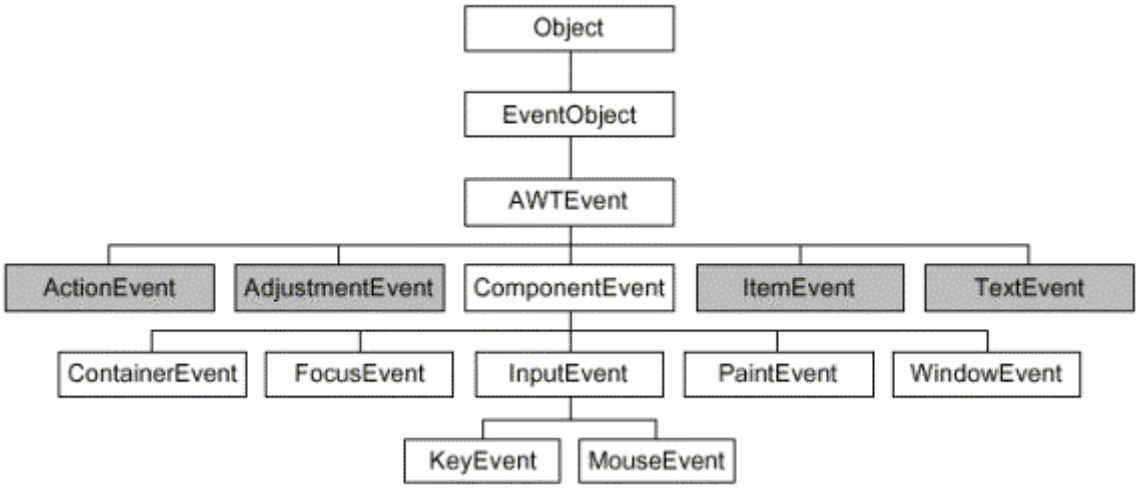


Figura 7.2. Jerarquía de eventos de Java.

1. Todos los ***Componentes*** (excepto ***Window*** y los que derivan de ella) deben ser añadidos a un ***Contenedor***. También un ***Contenedor*** puede ser añadido a otro ***Contenedor***.

2. Para añadir un ***Component*** a un ***Contenedor*** se utiliza el método ***add()*** de la clase ***Container***:

```
containerName.add(componentName);
```

3. Los ***Contenedores*** de máximo nivel son las ***Ventanas (Frames*** y ***Dialogs***). Los ***Paneles*** y ***Paneles de scroll*** deben estar siempre dentro de otro ***Contenedor***.

4.Un ***Component*** sólo puede estar en un ***Contenedor***. Si está en un ***Contenedor*** y se añade a otro, deja de estar en el primero.

5.La clase ***Component*** tiene una serie de funcionalidades básicas comunes (variables y métodos) que son heredadas por todas sus ***subclase***s.

7.1.4.2 Jerarquía de eventos

Todos los eventos de ***Java 1.1*** y ***Java 1.2*** son objetos de clases que pertenecen a una determinada jerarquía de clases. La superclase ***EventObject*** pertenece al paquete ***java.util***. De ***EventObject*** deriva la clase ***AWTEvent***, de la que dependen todos los eventos de AWT. La muestra la jerarquía de clases para los eventos de ***Java***. Por conveniencia, estas clases están agrupadas en el paquete ***java.awt.event***.

Los eventos de ***Java*** pueden ser de alto y bajo nivel. Los ***eventos de alto nivel*** se llaman también *eventos semántico*s, porque la acción de la que derivan tiene un significado en sí misma, en el contexto de las interfaces gráficas de usuario. Los ***eventos de bajo nivel*** son las acciones elementales que hacen posible los eventos de alto nivel. Son ***eventos de alto nivel*** los siguientes eventos: los cuatro que tienen que ver con hacer click sobre botones o elegir comandos en menús ***(ActionEvent***), cambiar valores en barras de desplazamiento ***(AdjustmentEvent***), elegir valores ***(ItemEvent***s) y cambiar el texto ***(TextEvent***). En la Figura 7.2 los eventos de alto nivel aparecen con fondo gris.

Los ***eventos de bajo nivel*** son los que se producen con las operaciones elementales con el ratón, teclado, contenedores y ventanas. Las seis clases de eventos de bajo nivel son los eventos relacionados con componentes ***(ComponentEvent***), con los contenedores ***(ContainerEvent***), con pulsar teclas ***(KeyEvent***), con mover, arrastrar, pulsar y soltar con el ratón ***(MouseEvent***), con obtener o perder el foco ***(FocusEvent***) y con las operaciones con ventanas ***(WindowEvent***).

El modelo de eventos se complica cuando se quiere construir un tipo de componente propio, no estándar del AWT. En este caso hay que interceptar los eventos de bajo nivel de ***Java*** y adecuarlos al problema que se trata de resolver. Éste es un tema que no se va a tratar en este manual.

7.1.4.3 Relación entre Componentes y Eventos

La Tabla 7.1 muestra los componentes del AWT y los eventos específicos de cada uno de ellos, así como una breve explicación de en qué consiste cada tipo de evento.

Component	Eventos generados	Significado
Button	ActionEvent	Clicar en el botón
Checkbox	ItemEvent	Seleccionar o deseleccionar un item
CheckboxMenuItem	ItemEvent	Seleccionar o deseleccionar un item
Choice	ItemEvent	Seleccionar o deseleccionar un item
Component	ComponentEvent	Mover, cambiar tamaño, mostrar u ocultar un componente
	FocusEvent	Obtener o perder el focus
	KeyEvent	Pulsar o soltar una tecla
	MouseEvent	Pulsar o soltar un botón del ratón; entrar o salir de un componente; mover o arrastrar el ratón (tener en cuenta que este evento tiene dos Listener)
Container	ContainerEvent	Añadir o eliminar un componente de un container
List	ActionEvent	Hacer doble click sobre un item de la lista
	ItemEvent	Seleccionar o deseleccionar un item de la lista
MunuItem	ActionEvent	Seleccionar un item de un menú
Scrollbar	AdjustementEvent	Cambiar el valor de la scrollbar
TextComponent	TextEvent	Cambiar el texto
TextField	ActionEvent	Terminar de editar un texto pulsando Intro
Window	WindowEvent	Acciones sobre una ventana: abrir, cerrar, iconizar, restablecer e iniciar el cierre

Tabla 7.1. Componentes del AWT y eventos específicos que generan.

La relación entre componentes y eventos indicada en la Tabla 7.1 pueden inducir a engaño si no se tiene en cuenta que los eventos propios de una ***superclase*** de componentes pueden afectar también a los componentes de sus ***subclase***s. Por ejemplo, la clase ***TextArea*** no tiene ningún evento específico o propio, pero puede recibir los de su ***superclase TextComponen***t.

La Tabla 7.2 muestra los ***componentes*** del AWT y ***todos los tipos de eventos*** que se pueden producir sobre cada uno de ellos, teniendo en cuenta también los que son específicos de sus ***superclase***s. Entre ambas tablas se puede sacar una idea bastante precisa de qué tipos de eventos están soportados en ***Java*** y qué eventos concretos puede recibir cada componente del AWT. En la práctica, no todos los tipos de evento tienen el mismo interés.

AWT Components	Eventos que se pueden generar									
	AtionEvent	AdjustementEvent	ComponentEvent	ContainerEvent	FocusEvent	ItemEvent	KeyEvent	MouseEvent	TextEvent	WindowEvent
Button	4		4		4		4	4		
Canvas			4		4		4	4		
Checkbox			4		4	4	4	4		
Checkbox-MenuItem						4				
Choice			4		4	4	4	4		
Component			4		4		4	4		
Container			4	4	4		4	4		
Dialog			4	4	4		4	4		4
Frame			4	4	4		4	4		4
Label			4		4		4	4		
List	4		4		4	4	4	4		
MenuItem	4									
Panel			4	4	4		4	4		
Scrollbar		4	4		4		4	4		
TextArea			4		4		4	4	4	
TextField	4		4		4		4	4	4	
Window			4	4	4		4	4		4

Tabla 7.2. Eventos que generan los distintos componentes del AWT.

7.1.5 Interfaces Listener

Una vez vistos los distintos eventos que se pueden producir, conviene ver cómo se deben gestionar estos eventos. A continuación se detalla cómo se gestionan los eventos según el modelo de ***Jav***a:

1. Cada objeto que puede recibir un evento ***(event sourc***e), ?registra? uno o más objetos para que los gestionen ***(event listene***r). Esto se hace con un método que tiene la forma, `eventSourceObject.addEventListener(eventListenerObject);` donde ***eventSourceObject*** es el objeto en el que se produce el evento, y ***eventListenerObject*** es el objeto que deberá gestionar los eventos. La relación entre ambos se establece a través de una interface ***Listener*** que la clase del ***eventListenerObject*** debe implementar. Esta interface proporciona la declaración de los métodos que serán llamados cuando se produzca el evento. La interface a implementar depende del tipo de evento. La Tabla 7.3 relaciona los distintos tipos de eventos, con la interface que se debe implementar para gestionarlos. Se indican también los métodos declarados en cada interface. Es importante observar la correspondencia entre ***eventos*** e ***interfaces Listene***r. Cada evento tiene su interface, excepto el ratón que tiene dos interfaces ***MouseListener*** y ***MouseMotionListene***r. La razón de esta duplicidad de interfaces se encuentra en la peculiaridad de los eventos que se producen cuando el ratón se mueve. Estos eventos, que se producen con muchísima más frecuencia que los simples clicks, por razones de eficiencia son gestionados por una interface especial: Obsérvese que el ***nombre de la interface*** coincide con el ***nombre del event***o, sustituyendo la palabra ***Event*** por ***Listene***r.

2.Una vez registrado el objeto que gestionará el evento, perteneciente a una clase que implemente la correspondiente interface ***Listene***r, se deben definir los métodos de dicha interface. Siempre hay que definir ***todos los métodos*** de la interface, aunque algunos de dichos métodos puedan estar ?vacíos?. Un ejemplo es la implementación de la interface ***WindowListener*** vista en el Apartado 1.3.9 (en la página 17), en el que todos los métodos estaban vacíos excepto ***windowClosing()***.

Evento	Interface Listener	Métodos de Listener
ActionEvent	ActionListener	actionPerformed()
AdjustementEvent	AdjustementListener	adjustementValueChanged()
ComponentEvent	ComponentListener	componentHidden(), componentMoved(), componentResized(), componentShown()
ContainerEvent	ContainerListener	componentAdded(), componentRemoved()
FocusEvent	FocusListener	focusGained(), focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed(), keyReleased(), keyTyped()
MouseEvent	MouseListener	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
	MouseMotionListener	mouseDragged(), mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated(), windowDeactivated(), windowClosed(), windowClosing(), windowIconified(), windowDeiconified(), windowOpened()

Tabla 7.3. Métodos relacionados con cada evento a través de una interface Listener.

7.1.6 Clases Adapter

Java proporciona ayudas para definir los métodos declarados en las interfaces ***Listene***r. Una de estas ayudas son las clases ***Adapte***r, que existen para cada una de las interfaces ***Listener*** que tienen más de un método. Su nombre se construye a partir del nombre de la interface, sustituyendo la palabra *?Listene*r? por *?Adapte*r?. Hay 7 clases ***Adapte***r: *ComponentAdapte*r, *ContainerAdapte*r, *FocusAdapte*r, *KeyAdapte*r, *MouseAdapte*r, *MouseMotionAdapter* y *WindowAdapte*r.

Las clases ***Adapter*** derivan de ***Objec***t, y son clases predefinidas que contienen ***definiciones vacías*** para todos los métodos de la interface. Para crear un objeto que responda al evento, en vez de crear una clase que implemente la interface ***Listene***r, basta crear una clase que derive de la clase ***Adapter*** correspondiente, y redefina sólo los métodos de interés. Por ejemplo, la clase ***VentanaCerrable*** del Apartado 1.3.9 (página 17) se podía haber definido de la siguiente forma:

```
1.// Archivo VentanaCerrable2.java

2.import java.awt.*;

3.import java.awt.event.*;

4.class VentanaCerrable2 extends Frame {

5.// constructores

6.public VentanaCerrable2() {

7.super();
```

```
8.}

9.public VentanaCerrable2(String title) {

10.super(title);

11.setSize(500,500);

12.CerrarVentana cv = new CerrarVentana();

13.this.addWindowListener(cv);

14.}

15.} // fin de la clase VentanaCerrable2

16.// definición de la clase CerrarVentana

17.class CerrarVentana extends WindowAdapter {

18.void windowClosing(WindowEvent we) {

19.System.exit(0);

20.}

21.} // fin de la clase CerrarVentana
```

Las sentencias 15-17 definen una clase auxiliar ***(helper class)*** que deriva de la clase ***WindowAdapter***. Dicha clase hereda definiciones vacías de todos los métodos de la interface ***WindowListener***. Lo único que tiene que hacer es redefinir el único método que se necesita para cerrar las ventanas. El constructor de la clase ***VentanaCerrable*** crea un objeto de la clase ***CerrarVentana*** en la sentencia 10 y lo registra como ***event listener*** en la sentencia 11. En la sentencia 11 la palabra ***this*** es opcional: si no se incluye, se supone que el ***event source*** es el objeto de la clase en la que se produce el evento, en este caso la propia ventana.

Todavía hay otra forma de responder al evento que se produce cuando el usuario desea cerrar la ventana. Las ***clases anónimas*** de ***Java*** son especialmente útiles en este caso. En realidad, para gestionar eventos sólo hace falta un objeto que sea registrado como ***event listener*** y contenga los métodos adecuados de la interface ***Listener***. Las ***clases anónimas*** son útiles cuando sólo se necesita un objeto de la clase, como es el caso. La nueva definición de la clase ***VentanaCerrable*** podría ser como sigue:

```
1.// Archivo VentanaCerrable3.java

2.import java.awt.*;

3.import java.awt.event.*;

4.class VentanaCerrable3 extends Frame {

5.// constructores

6.public VentanaCerrable3() {

7.super();
```

```
8.}

9.public VentanaCerrable3(String title) {

10.super(title);

11.setSize(500,500);

12.this.addWindowListener(new WindowAdapter() {

13.public void windowClosing() {

14.System.exit(0);

15.}

16.}

17.);

18.}

19.} // fin de la clase VentanaCerrable
```

Obsérvese que el objeto ***event listener*** se crea justamente en el momento de pasárselo como argumento al método ***addWindowListener()***. Se sabe que se está creando un nuevo objeto porque aparece la palabra ***new***. Debe tenerse en cuenta que no se está creando un nuevo objeto de ***WindowAdapter*** (entre otras cosas porque dicha clase es ***abstract***), sino extendiendo la clase ***WindowAdapter***, aunque la palabra ***extends*** no aparezca. Esto se sabe por las ***llaves*** que se abren al final de la línea 10. Los ***paréntesis vacíos*** de la línea 10 podrían contener los argumentos para el constructor de ***WindowAdapter***, en el caso de que dicho constructor necesitara argumentos. En la sentencia 11 se redefine el método ***windowClosing()***. En la línea 12 se cierran las llaves de la ***clase anónima***, se cierra el paréntesis del método ***addWindowListener()*** y se pone el ***punto y coma*** de terminación de la sentencia que empezó en la línea 10. Para más información sobre las ***clases anónimas*** ver el Apartado 3.10.4, en la página 57.

7.2 COMPONENTESY EVENTOS

Se van a ver los ***componentes gráficos*** de ***Java***, a partir de los cuales se pueden construir interfaces gráficas de usuario. Se verán también, en paralelo y lo más cerca posible en el texto, las diversas clases de ***eventos*** que pueden generar cada uno de esos componentes.

La Figura 7.3, tomada de uno de los ejemplos de ***Java Tutorial*** de ***Sun***, muestra algunos componentes del AWT. En ella se puede ver un *menú*, una superficie de dibujo o *canvas* en la que se puede dibujar y escribir texto, una *etiqueta*, una *caja de texto* y un *área de texto*, un *botón de comando* y un *botón de selección*, una *lista* y una *caja de selección desplegable*.

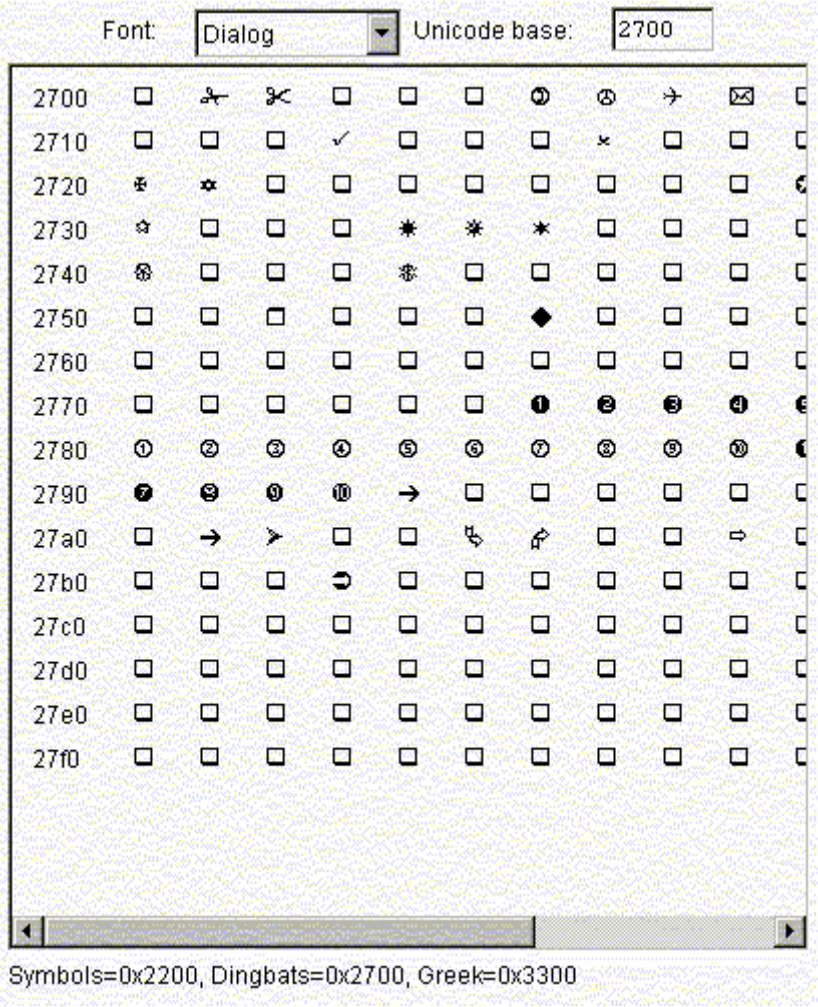


Figura 7.3. Algunos componentes del AWT.

Métodos de Component	Función que realizan
boolean isVisible(), void setVisible(boolean)	Permiten chequear o establecer la visibilidad de un componente
boolean isShowing()	Permiten saber si un componente se está viendo. Para ello tanto el componente debe ser visible, y su container debe estar mostrándose
boolean isEnabled(), void setEnabled(boolean)	Permiten saber si un componente está activado y activarlo o desactivarlo
Point getLocation(), Point getLocationOnScreen()	Permiten obtener la posición de la esquina superior izquierda de un componente respecto al componente-padre o a la pantalla
void setLocation(Point), void setLocation(int x, int y)	Desplazan un componente a la posición especificada respecto al container o componente-padre
Dimension getSize(), void setSize(int w, int h), void setSize(Dimension d)	Permiten obtener o establecer el tamaño de un componente
Rectangle getBounds(), void setBounds(Rectangle), void setBounds(int x, int y, int width, int height)	Obtienen o establecen la posición y el tamaño de un componente
invalidate(), validate(), doLayout()	invalidate() marca un componente y sus contenedores para indicar que se necesita volver a aplicar el Layout Manager. validate() se asegura que el Layout Manager está bien aplicado. doLayout() hace que se aplique el Layout Manager
paint(Graphics), repaint() y update(Graphics)	Métodos gráficos para dibujar en la pantalla
setBackground(Color), setForeground(Color)	Métodos para establecer los colores por defecto

Tabla 7.4. Métodos de la clase Component.

7.2.1 Clase Component

La clase ***Component*** es una clase ***abstract*** de la que derivan todas las clases del AWT, según el diagrama mostrado previamente en la Figura 5.1, en la página 83. Los métodos de esta clase son importantes porque son heredados por todos los componentes del AWT. La Tabla 7.4 muestra algunos de los métodos más utilizados de la clase ***Component***. En las declaraciones de los métodos de dicha clase aparecen las clases ***Point***, ***Dimension*** y ***Rectangle***. La clase ***java.awt.Point*** tiene dos variables miembro *int* llamadas ***x*** e y. La clase ***java.awt.Dimension*** tiene dos variables miembro *int*: ***height*** y ***width***. La clase ***java.awt.Rectangle*** tiene cuatro variables *int*: ***height***, ***width***, ***x*** e y. Las tres son subclases de ***Object***.

Además de los métodos mostrados en la Tabla 7.4, la clase ***Component*** tiene un gran número de métodos básicos cuya funcionalidad puede estudiarse mediante la documentación online de ***Java***. Entre otras funciones, permiten controlar los ***colores***, los ***fonts*** y los ***cursores***.

A continuación se describen las clases de ***eventos*** más generales, relacionados bien con la clase ***Component***, bien con diversos tipos de componentes que también se presentan a continuación.

7.2.2 Clases EventObject y AWTEvent

Todos los métodos de las interfaces ***Listener*** relacionados con el AWT tienen como argumento único un objeto de alguna clase que desciende de la clase ***java.awt.AWTEvent*** (ver Figura 7.2).

La clase ***AWTEvent*** desciende de ***java.util.EventObject***. La clase ***AWTEvent*** no define ningún método, pero hereda de ***EventObject*** el método ***getSource()***:

```
Object getSource();
```

que devuelve una referencia al objeto que generó el evento. Las clases de eventos que descienden de ***AWTEvent*** definen métodos similares a ***getSource()*** con unos valores de retorno menos genéricos. Por ejemplo, la clase ***ComponentEvent*** define el método ***getComponent()***, cuyo valor de retorno es un objeto de la clase ***Component***.

7.2.3 Clase ComponentEvent

Los eventos ***ComponentEvent*** se generan cuando un ***Component*** de cualquier tipo se muestra, se oculta, o cambia de posición o de tamaño. Los eventos de *mostrar* u *ocultar* ocurren cuando se llama al método ***setVisible(boolean)*** del ***Component***, pero no cuando se *minimiza* la ventana.

Otro método útil de la clase ***ComponentEvent*** es ***Component getComponent()*** que devuelve el componente que generó el evento. Se puede utilizar en lugar de ***getSource()***.

7.2.4 Clases InputEvent y MouseEvent

De la clase ***InputEvent*** descienden los eventos del ratón y el teclado. Esta clase dispone de métodos pera detectar si los botones del ratón o las teclas especiales han sido pulsadas. Estos botones y estas teclas se utilizan para cambiar o modificar el significado de las acciones del usuario. La clase ***InputEvent*** define unas constantes que permiten saber qué teclas especiales o botones del ratón estaban pulsados al producirse el evento, como son: SHIFT_MASK, ALT_MASK, CTRL_MASK, BUTTON1_MASK, BUTTON2_MASK y BUTTON3_MASK, cuyo significado es evidente. La Tabla 7.5 muestra algunos métodos de esta clase.

Métodos heredados de la clase InputEvent	Función que realizan
boolean isShiftDown(), boolean isAltDown(), boolean isControlDown()	Devuelven un boolean con información sobre si esa tecla estaba pulsada o no
int getModifiers()	Obtiene información con una máscara de bits sobre las teclas y botones pulsados
long getWhen()	Devuelve la hora en que se produjo el evento

Tabla 7.5. Métodos de la clase InputEvent.

Se produce un ***MouseEvent*** cada vez que el cursor movido por el ratón entra o sale de un componente visible en la pantalla, al hacer click, o cuando se pulsa o se suelta un botón del ratón. Los métodos de la interface ***MouseListener*** se relacionan con estas acciones, y son los siguientes (ver Tabla 7.3): ***mouseClicked()***, ***mouseEntered()***, ***mouseExited()***, ***mousePressed()*** y ***mouseReleased()***.

Todos son *void* y reciben como argumento un objeto ***MouseEvent***. La Tabla 7.6 muestra algunos métodos de la clase ***MouseEvent***.

Métodos de la clase MouseEvent	Función que realizan
int getClickCount()	Devuelve el número de clicks en ese evento
Point getPoint(), int getX(), int getY()	Devuelven la posición del ratón al producirse el evento
boolean isPopupTrigger()	Indica si este evento es el que dispara los menús popup

Tabla 7.6. Métodos de la clase MouseEvent.

La clase ***MouseEvent*** define una serie de constantes ***int*** que permiten identificar los tipos de eventos que se han producido: MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_MOVED, MOUSE_ENTERED, MOUSE_EXITED, MOUSE_DRAGGED.

Además, el método ***Component getComponent()***, heredado de ***ComponentEvent***, devuelve el componente sobre el que se ha producido el evento.

Los eventos ***MouseEvent*** disponen de una segunda interface para su gestión, la interface ***MouseMotionListener***, cuyos métodos reciben también como argumento un evento de la clase ***MouseEvent***. Estos eventos están relacionados con el ***movimiento del ratón***. Se llama a un método de la interface ***MouseMotionListener*** cuando el usuario utiliza el ratón (o un dispositivo similar) para mover el cursor o arrastrarlo sobre la pantalla. Los métodos de la interface ***MouseMotionListener*** son ***mouseMoved()*** y ***mouseDragged()***.

7.2.5 Clase FocusEvent

El ***Foco*** está relacionado con la posibilidad de sustituir al ratón por el teclado en ciertas operaciones. De los componentes que aparecen en pantalla, en un momento dado hay sólo uno que puede recibir las acciones del teclado y se dice que ese componente tiene el ***Foco***. El componente que tiene el ***Foco*** aparece diferente de los demás (resaltado de alguna forma). Se cambia el elemento que tiene el ***Foco*** con la tecla ***Tab*** o con el ratón. Se produce un ***FocusEvent*** cada vez que un componente gana o pierde el ***Foco***.

El método ***requestFocus()*** de la clase ***Component*** permite hacer desde el programa que un

componente obtenga el ***Foco***.

El método ***boolean isTemporary()***, de la clase ***FocusEvent***, indica si la pérdida del ***Foco*** es o no temporal (puede ser temporal por haberse ocultado o dejar de estar activa la ventana, y recuperarse al cesar esta circunstancia).

El método ***Component getComponent()*** es heredado de ***ComponentEvent***, y permite conocer el componente que ha ganado o perdido el ***Foco***. Las constantes de esta clase ***FOCUS_GAINED*** y ***FOCUS_LOST*** permiten saber el tipo de evento ***FocusEvent*** que se ha producido.

7.2.6 Clase Container

La clase ***Container*** es también una clase muy general. De ordinario, nunca se crea un objeto de esta clase, pero los métodos de esta clase son heredados por las clases ***Frame*** y ***Panel***, que sí se utilizan con mucha frecuencia para crear objetos. La Tabla 7.7 muestra algunos métodos de ***Container***.

Métodos de Container	Función que realizan
Component add(Component)	Añade un componente al container
doLayout()	Ejecuta el algoritmo de ordenación del layout manager
Component getComponent(int)	Obtiene el n-ésimo componente en el container
Component getComponentAt(int, int), Component getComponentAt(Point)	Obtiene el componente que contine un determinado punto
int getComponentCount()	Obtiene el número de componentes en el container
Component[] getComponents()	Obtiene los componentes en este container.
remove(Component), remove(int), removeAll()	Elimina el componente especificado.
setLayout(LayoutManager)	Determina el layout manager para este container

Tabla 7.7. Métodos de la clase Container.

Los contenedores mantienen una ***lista de los objetos*** que se les han ido añadiendo. Cuando se añade un nuevo objeto se incorpora al final de la lista, salvo que se especifique una posición determinada. En esta clase tiene mucha importancia todo lo que tiene que ver con los ***Layout Managers***, que se explicarán más adelante. La Tabla 7.7 muestra algunos métodos de la clase ***Container***.

7.2.7 Clase ContainerEvent

Los ***ContainerEvents*** se generan cada vez que un ***Component*** se añade o se retira de un ***Container***.

Estos eventos sólo tienen un *papel de aviso* y no es necesario gestionarlos para que se realice la operación. Los métodos de esta clase son ***Component getChild()***, que devuelve el ***Component*** añadido o eliminado, y ***Container getContainer()***, que devuelve el ***Contenedor*** que generó el evento.

7.2.8 Clase Window

Los objetos de la clase ***Window*** son ventanas de máximo nivel, pero ***sin bordes*** y ***sin barra de menús***.

En realidad son más interesantes las clases que derivan de ella: ***Frame*** y ***Dialog***. Los métodos más útiles, por ser heredados por las clases ***Frame*** y ***Dialog***, se muestran en la Tabla 7.8.

Métodos de Window	Función que realizan
toFront(), toBack()	Para desplazar la ventana hacie adelante y hacia atrás en la pantalla
show()	Muestra la ventana y la trae a primer plano
pack()	Hace que los componentes se reajusten al tamaño preferido

Tabla 7.8. Métodos de la clase Window.

7.2.9 Clase WindowEvent

Se produce un ***WindowEvent*** cada vez que se *abre*, *cierra*, *iconiza*, *restaura*, *activa* o *desactiva* una ventana. La interface ***WindowListener*** contiene los siete métodos siguientes, con los que se puede responder a este evento:

```
1.void windowOpened(WindowEvent we); // antes de mostrarla

2.//por primera vez

3.void windowClosing(WindowEvent we); // al recibir una

4.//solicitud de cierre

5.void windowClosed(WindowEvent we); // después de cerrar
   la

6.//ventana

7.void windowIconified(WindowEvent we);

8.void windowDeiconified(WindowEvent we);

9.void windowActivated(WindowEvent we);
```

```
10.void windowDeactivated(WindowEvent we);
```

El uso más frecuente de ***WindowEvent*** es para cerrar ventanas (por defecto, los objetos de la clase ***Frame*** no se pueden cerrar más que con ***Ctrl+Alt+Supr***). También se utiliza para detener ***threads*** y liberar recursos al iconizar una ventana (que contiene por ejemplo animaciones) y comenzar de nuevo al restaurarla.

La clase ***WindowEvent*** define la siguiente serie de constantes que permiten identificar el tipo de evento:

WINDOW_OPENED, WINDOW_CLOSING, WINDOW_CLOSED,

WINDOW_ICONIFIED, WINDOW_DEICONIFIED,

WINDOW_ACTIVATED, WINDOW_DEACTIVATED

En la clase ***WindowEvent*** el método ***Window getWindow()*** devuelve la ***Window*** que generó el evento. Se utiliza en lugar de ***getSource()***.

7.2.10Clase Frame

Es una ventana ***con un borde*** y que puede tener una ***barra de menús***. Si una ventana depende de otra ventana, es mejor utilizar una ***Window*** (ventana sin borde ni barra de menús) que un ***Frame***. La Tabla 7.9 muestra algunos métodos más utilizados de la clase ***Frame***.

Métodos de Frame	Función que realiza
Frame(), Frame(String title)	Constructores de Frame
String getTitle(), setTitle(String)	Obtienen o determinan el título de la ventana
MenuBar getMenuBar(), setMenuBar(MenuBar), remove(MenuComponent)	Permite obtener, establecer o eliminar la barra de menús
Image getIconImage(), setIconImage(Image)	Obtienen o determinan el icono que apareceré en la barra de títulos
setResizable(boolean), boolean isResizable()	Determinan o chequean si se puede cambiar el tamaño
dispose()	Método que libera los recursos utilizados en una ventana. Todos los componentes de la ventana son destruidos.

Tabla 7.9. Métodos de la clase Frame.

Además de los métodos citados, se utilizan mucho los métodos ***show()***, ***pack()***, ***toFront()*** y

toBack(), heredados de la superclase ***Window***.

7.2.11Clase Dialog

Un ***Dialog*** es una ventana que depende de otra ventana (de una ***Frame***). Si una ***Frame*** se cierra, se cierran también los ***Dialog*** que dependen de ella; si se iconifica, sus ***Dialog*** desaparecen; si se restablece, sus ***Dialog*** aparecen de nuevo. Este comportamiento se obtiene de forma automática.

Las ***Applets*** estándar no soportan ***Dialogs*** porque no son ***Frames*** de ***Jav***a. Las ***Applets*** que abren ***Frames*** sí pueden soportar ***Dialog***s.

Un ***Dialog modal*** requiere la atención inmediata del usuario: no se puede hacer ninguna otra cosa hasta no haber cerrado el ***Dialo***g. Por defecto, los ***Dialogs*** son ***no modale***s. La Tabla 7.10 muestra los métodos más importantes de la clase ***Dialo***g. Se pueden utilizar también los métodos heredados de sus superclases.

Métodos de Dialog	Función que realiza
Dialog(Frame fr), Dialog(Frame fr, boolean mod), Dialog(Frame fr, String title), Dialog(Frame fr, String title, boolean mod)	Constructores
String getTitle(), setTitle(String)	Permite obtener o determinar el título
boolean isModal(), setModal(boolean)	Pregunta o determina si el Dialog es modal o no
boolean isResizable(), setResizable(boolean)	Pregunta o determina si se puede cambiar el tamaño
show()	Muestra y trae a primer plano el Dialog

Tabla 7.10. Métodos de la clase Dialog.

7.2.12Clase FileDialog

La clase ***FileDialog*** muestra una ventana de diálogo en la cual se puede seleccionar un archivo. Esta clase deriva de ***Dialo***g. Las constantes enteras LOAD (abrir archivos para lectura) y SAVE (abrir archivos para escritura) definen el ***modo*** de apertura del archivo. La Tabla 7.11 muestra algunos métodos de esta clase.

Métodos de la clase FileDialog	Función que realizan
FileDialog(Frame parent), FileDialog(Frame parent, String title), public FileDialog(Frame parent, String title, int mode)	Constructores
int getMode(),setMode(int mode)	Modo de apertura (SAVE o LOAD)
String getDirectory(), String getFile()	Obtiene el directorio o fichero elegido
setDirectory(String dir), setFile(String file)	Determina el directorio o fichero elegido
FilenameFilter getFilenameFilter(), setFilenameFilter(FilenameFilter filter)	Determina o establece el filtro para los ficheros

Tabla 7.11. Métodos de la clase FileDialog.

Las clases que implementan la interface ***java.io.FilenameFilter*** permiten filtrar los archivos de un directorio. Para más información, ver la documentación online.

7.2.13Clase Panel

Un ***Panel*** es un ***Contenedor*** de propósito general. Se puede utilizar tal cual para contener otras componentes, y también crear una subclase para alguna finalidad más específica. Por defecto, el ***Layout Manager*** de ***Panel*** es ***FlowLayou***t. Los ***Applets*** son subclases de ***Pane***l. La Tabla 7.12 muestra los métodos más importantes que se utilizan con la clase ***Pane***l, que son algunos métodos heredados de ***Componet*** y ***Containe***r, pues la clase ***Panel*** no tiene métodos propios.

Métodos de Panel	Función que realiza
Panel(), Panel(LayoutManager miLM)	Constructores de Panel
Métodos heredados de Container y Component	
add(Component), add(Component, int)	Añade componentes al panel
setLayout(), getLayout()	Establece o permite obtener el layout manager utilizado
validate(), doLayout()	Para reorganizar los componentes después de algún cambio. Es mejor utilizar validate()
remove(int), remove(Component), removeAll()	Para eliminar componentes
Dimension getMaximumSize(), Dimension getMinimumSize(), Dimension getPreferredSize()	Permite obtener los tamaños máximo, mínimo y preferido
Insets getInsets()	

Tabla 7.12. Métodos de la clase Panel.

Un ***Panel*** puede contener otros ***Panel***. Esto es una gran ventaja respecto a los demás tipos de contenedores, que son contenedores de máximo nivel y no pueden introducirse en otros contenedores.

Insets es una clase que deriva de ***Object***. Sus variables son ***top***, ***left***, ***bottom***, ***right***. Representa el *espacio que se deja libre* en los bordes de un ***Contenedor***. Se establece mediante la llamada al adecuado constructor del ***Layout Manager***.

7.2.14Clase Button

Aunque según la Tabla 7.2, un ***Button*** puede recibir seis tipos de eventos, lo más importante es que al hacer click sobre él se genera un evento de la clase ***ActionEvent***.

El aspecto de un ***Button*** depende de la plataforma (PC, Mac, UNIX), pero la funcionalidad siempre es la misma. Se puede cambiar el ***texto*** y el ***font*** que aparecen en el ***Button***, así como el ***foreground*** y ***backgroung color***. También se puede establecer que esté activado o no. La Tabla 7.13 muestra los métodos más importantes de la clase ***Button***.

Métodos de la clase Button	Función que realiza
Button(String label) y Button()	Constructores
setLabel(String str), String getLabel()	Permite establecer u obtener la etiqueta del Button
addActionListener(ActionListener al), removeActionListener(ActionListener al)	Permite registrar el objeto que gestionará los eventos, que deberá implementar ActionListener
setActionCommand(String cmd), String getActionCommand()	Establece y recupera un nombre para el objeto Button independiente del label y del idioma

Tabla 7.13. Métodos de la clase Button.

7.2.15Clase ActionEvent

Los eventos ***ActionEvent*** se producen al hacer click con el ratón en un botón ***(Button)***, al elegir un comando de un menú ***(MenuItem)***, al hacer doble clic en un elemento de una lista ***(List)*** y al pulsar ***Enter*** para introducir un texto en una caja de texto ***(TextField)***.

El método ***String getActionCommand()*** devuelve el texto asociado con la acción que provocó el evento. Este texto se puede fijar con el método ***setActionCommand(String str)*** de las clases ***Button*** y ***MenuItem***. Si el texto no se ha fijado con este método, el método ***getActionCommand()*** devuelve el texto mostrado por el componente (su etiqueta). Para objetos con varios ítems el valor devuelto es el nombre del ítem seleccionado.

El método ***int getModifiers()*** devuelve un entero representando una constante definida en ***ActionEvent*** (SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK). Estas constantes sirven para determinar si se pulsó una de estas teclas modificadores mientras se hacia click. Por ejemplo, si se estaba pulsando la tecla CTRL la siguiente expresión es distinta de cero:

```
actionEvent.getModifiers() & ActionEvent.CTRL_MASK
```

7.2.16Clase Canvas

Una ***Canvas*** es una zona rectangular de pantalla en la que se puede dibujar y en la que se pueden generar eventos. Las ***Canvas*** permiten realizar dibujos, mostrar imágenes y crear componentes a medida, de modo que muestren un aspecto similar en todas las plataformas. La Tabla 7.14 muestra los métodos de la clase ***Canvas***.

Métodos de Canvas	Función que realiza
Canvas()	Es el único constructor de esta clase
paint(Graphics g);	Dibuja un rectángulo con el color de background. Lo normal es que las sub-clases de Canvas redefinan este método.

Tabla 7.14. Métodos de la clase Canvas.

Desde los objetos de la clase ***Canvas*** se puede llamar a los métodos ***paint()*** y ***repaint()*** de la superclase ***Component***. Con frecuencia conviene redefinir los siguientes métodos de ***Component***:

getPreferredSize(), ***getMinimumSize()*** y ***getMaximumSize()***, que devuelven un objeto de la clase ***Dimension***. El ***LayoutManager*** se encarga de utilizar estos valores.

La clase ***Canvas*** no tiene eventos propios, pero puede recibir los eventos ***ComponentEvent*** de su superclase ***Component***.

7.2.17Component Checkbox y clase CheckboxGroup

Los objetos de la clase ***Checkbox*** son ***botones de opción o de selección*** con dos posibles valores: ***on*** y ***off***. Al cambiar la selección de un ***Checkbox*** se produce un ***ItemEvent***.

La clase ***CheckboxGroup*** permite la opción de agrupar varios ***Checkbox*** de modo que uno y sólo uno esté en ***on*** (al comienzo puede que todos estén en ***off***). Se corresponde con los botones de opción de Visual Basic. La Tabla 7.15 muestra los métodos más importantes de estas clases.

Métodos de Checkbox	Función que realizan
Checkbox(), Checkbox(String), Checkbox(String, boolean), Checkbox(String, boolean, CheckboxGroup), Checkbox(String, CheckboxGroup, boolean)	Constructores de Checkbox. Algunos permiten establecer la etiqueta, si está o no seleccionado y si pertenece a un grupo
addItemListener(ItemListener), removeItemListener(ItemListener)	Registra o elimina los objetos que gestionarán los eventos ItemEvent
setLabel(String), String getLabel()	Establece u obtiene la etiqueta del componente
setState(boolean), boolean getState()	Establece u obtiene el estado (true o false, según esté seleccionado o no)
setCheckboxGroup(CheckboxGroup), CheckboxGroup getCheckboxGroup()	Establece u obtiene el grupo al que pertenece el Checkbox
Métodos de CheckboxGroup	**Función que realizan**
CheckboxGroup()	Constructores de CheckboxGroup:
Checkbox getSelectedCheckbox(), setSelectedCheckbox(Checkbox box)	Obtiene o establece el componente seleccionado de un grupo:

Tabla 7.15. Métodos de las clases Checkbox y CheckboxGroup.

Cuando el usuario actúa sobre un objeto ***Checkbox*** se ejecuta el método ***itemStateChanged()***, que es el único método de la interface ***ItemListener***. Si hay varias ***checkboxes*** cuyos eventos se gestionan en un mismo objeto y se quiere saber cuál es la que ha recibido el evento, se puede utilizar el método ***getSource()*** del evento ***EventObject***. También se puede utilizar el método ***getItem()*** de ***ItemEvent***, cuyo valor de retorno es un ***Object*** que contiene la ***etiqueta*** del componente (para convertirlo a ***String*** habría que hacer un ***cast***).

Para crear un ***grupo*** o conjunto de botones de opción (de forma que uno y sólo uno pueda estar activado), se debe crear un objeto de la clase ***CheckboxGroup***. Este objeto no tiene datos: simplemente sirve como identificador del grupo. Cuando se crean los objetos ***Checkbox*** se pasa a los ***constructores*** el objeto ***CheckboxGroup*** del grupo al que se quiere que pertenezcan.

Cuando se selecciona un ***Checkbox*** de un grupo se producen dos eventos: uno por el elemento que se ha seleccionado y otro por haber perdido la selección el elemento que estaba seleccionado anteriormente. Al hablar de evento ***ItemEvent*** y del método ***itemStateChanged()*** se verán métodos para determinar los ***checkboxes*** que han sido seleccionados o que han perdido la selección.

7.2.18Clase ItemEvent

Se produce un ***ItemEvent*** cuando ciertos componentes ***(Checkbox***, ***CheckboxMenuItem***, ***Choice*** y ***List***) cambian de estado (on/off). Estos componentes son los que implementan la interface ***ItemSelectable***. La Tabla 7.16 muestra algunos métodos de esta clase.

Métodos de la clase ItemEvent	Función que realizan
Object getItem()	Devuelve el objeto donde se originó el evento
ItemSelectable getItemSelectable()	Devuelve el objeto ItemSelectable donde se originó el evento
int getStateChange()	Devuelve una de las constantes SELECTED o DESELECTED definidas en la clase ItemEvent

Tabla 7.16. Métodos de la clase ItemEvent.

La clase ***ItemEvent*** define las constantes enteras SELECTED y DESELECTED, que se pueden utilizar para comparar con el valor devuelto por el método ***getStateChange()***.

7.2.19Clase Choice

La clase ***Choice*** permite elegir un ítem de una lista desplegable. Los objetos ***Choice*** ocupan menos espacio en pantalla que los ***Checkbox***. Al elegir un ítem se genera un ***ItemEvent***. Un ***índice*** permite determinar un elemento de la lista (se empieza a contar desde 0). La Tabla 7.17 muestra los métodos más habituales de esta clase.

Métodos de Choice	Función que realizan
Choice()	Constructor de Choice
addItemListener(ItemListener), removeItemListener(ItemListener)	Establece o elimina un ItemListener
add(String), addItem(String)	Añade un elemento a la lista
insert(String label, int index)	Inserta un elemento con un label an la posición incicada
int getSelectedIndex(), String getSelectedItem()	Obtiene el index o el label del elemento elegido de la lista
int getItemCount()	Obtiene el número de elementos
String getItem(int)	Obtiene el label a partir del index
select(int), select(String)	Selecciona un elemento por el index o el label
removeAll(), remove(int), remove(String)	Elimina todos o uno de los elementos de la lista

Tabla 7.17. Métodos de la clase Choice.

La clase ***Choice*** genera el evento ***ItemEvent*** al seleccionar un ítem de la lista. En este sentido es similar a las clases ***Checkbox*** y ***CheckboxGroup***, así como a los menús de selección.

7.2.20Clase Label

La clase ***Label*** introduce en un contenedor un ***texto no seleccionable y no editable***, que por defecto se alinea por la izquierda. La clase ***Label*** define las constantes Label.CENTER, Label.LEFT y Label.RIGHT para determinar la alineación del texto. La Tabla 7.18 muestra algunos métodos de esta clase

Métodos de Label	Función que realizan
Label(String lbl), Label(String lbl, int align)	Constructores de Label
setAlignement(int align), int getAlignement()	Establecer u obtener la alineación del texto
setText(String txt), String getText()	Establecer u obtener el texto del Label

Tabla 7.18. Métodos de la clase Label.

La elección del *font*, de los *colores*, del tamaño y posición de ***Label*** se realiza con los métodos heredados de la clase ***Component***: ***setFont(Font f)***, ***setForeground(Color)***, ***setBackground(Color)***, ***setSize(int, int)***, ***setLocation(int, int)***, ***setVisible(boolean)***, etc.

La clase ***Label*** no tiene más ***eventos*** que los de su superclase ***Component***.

7.2.21Clase List

La clase ***List*** viene definida por una zona de pantalla con varias líneas, de las que se muestran sólo algunas, y entre las que se puede hacer una ***selección simple o múltipl***e. Las ***List*** generan eventos de la clase ***ActionEvents*** (al ***hacer click dos veces*** sobre un ítem o al pulsar ***retur***n) e ***ItemEvents*** (al seleccionar o deseleccionar un ítem). Al gestionar el evento ***ItemEvent*** se puede preguntar si el usuario estaba pulsando a la vez alguna tecla ***(Al***t, ***Ctr***l, ***Shif***t), por ejemplo para hacer una selección múltiple.

Las ***List*** se diferencian de las ***Choices*** en que muestran varios ítems a la vez y que permiten hacer selecciones múltiples. La Tabla 7.19 muestra los principales métodos de la clase ***Lis***t.

Métodos de List	Función que realiza
List(), List(int nl), List(int nl, boolean mult)	Constructor: por defecto una línea y selección simple
add(String), add(String, int), addItem(String), addItem(String, int)	Añadir un item. Por defecto se añaden al final
addActionListener(ActionListener), addItemListener(ItemListener)	Registra los objetos que gestionarán los dos tipos de eventos soportados
insert(String, int)	Inserta un nuevo elemento en la lista
replaceItem(String, int)	Sustituye el item en posición int por el String
delItem(int), remove(int), remove(String), removeAll()	Eliminar uno o todos los items de la lista
int getItemCount(), int getRows()	Obtener el número de items o el número de items visibles
String getItem(int), String[] getItems()	Obtiene uno o todos los elementos de la lista
int getSelectedIndex(), String getSelectedItem(), int[] getSelectedIndexes(), String[] getSelectedItems()	Obtiene el/los elementos seleccionados
select(int), deselect(int)	Selecciona o elimina la selección de un elemento
boolean isIndexSelected(int), boolean isItemSelected(String)	Indica si un elemento está seleccionado o no
boolean isMultipleMode(), setMultipleMode(boolean)	Pregunta o establece el modo de selección múltiple
int getVisibleIndex(), makeVisible(int)	Indicar o establecer si un item es visible

Tabla 7.19. Métodos de la clase List.

7.2.22Clase Scrollbar

Una ***Scrollbar*** es una barra de desplazamiento con un cursor que permite introducir y modificar valores, entre unos valores mínimo y máximo, con pequeños y grandes incrementos. Las ***Scrollbars*** de ***Java*** se utilizan tanto como ?sliders? o barras de desplazamiento aisladas (al estilo de Visual Basic), como unidas a una ventana en posición vertical y/u horizontal para mostrar una cantidad de información superior a la que cabe en la ventana.

La clase ***Scrollbar*** tiene dos constantes, Scrollbar.HORIZONTAL y Scrollbar.VERTICAL, que indican la posición de la barra. El cambiar el valor de la ***Scrollbar*** produce un ***AdjustementEven***t. La Tabla 7.20 muestra algunos métodos de esta clase.

Métodos de Scrollbar	Función que realizan
Scrollbar(), Scrollbar(int pos), Scrollbar(int pos, int val, int vis, int min, int max)	Constructores de Scrollbar
addAdjustmentListener(AdjustmentListener)	egistra el objeto que gestionará los eventos
int getValue(), setValue(int)	Permiten obtener y fijar el valor
setMaximum(int), setMinimum(int)	Establecen los valores máximo y mínimo
setVisibleAmount(int), int getVisibleAmount()	Establecen y obtienen el tamaño del área visble
setUnitIncrement(int), int getUnitIncrement()	Establecen y obtienen el incremento pequeño
setBlockIncrement(int), int getBlockIncrement()	Establecen y obtienen el incremento grande
setOrientation(int), int getOrientation()	Establecen y obtienen la orientación
setValues(int value, int vis, int min, int max)	Establecen los parámetros de la barra

Tabla 7.20. Métodos de la clase Scrollbar.

En el constructor general, el parámetro ***pos*** es la constante que indica la posición de la barra

(horizontal o vertical); el ***rango*** es el intervalo entre los valores mínimo ***min*** y máximo ***max***; el parámetro ***vis*** (de ***visibleAmount***) es el tamaño del área visible en el caso en que las ***Scrollbars*** se utilicen en ***TextAreas***. En ese caso, el tamaño del cursor representa la relación entre el ***área visible*** y el ***rango***, como es habitual en ***Netscape***, ***Word*** y tantas aplicaciones de ***Windows***. El valor seleccionado viene dado por la variable ***value***. Cuando ***value*** es igual a ***min*** el área visible comprende el inicio del rango; cuando ***value*** es igual a ***max*** el área visible comprende el final del ***rango***. Cuando la ***Scrollbar*** se va a utilizar aislada (como ***slider***), se debe hacer ***visibleAmount*** igual a cero.

Las variables ***Unit Increment*** y ***Block Increment*** representan los incrementos pequeño y grande, respectivamente. Por defecto, ***Unit Increment*** es ?1? y ***Block Increment*** es ?10?, mientras que ***min*** es ?0? y ***max*** es ?100?.

Cada vez que cambia el valor de una ***Scrollbar*** se genera un evento ***AdjustementEvent*** y se

ejecuta el único método de la interface ***AdjustmentListener***, que es ***adjustmentValueChanged()***.

7.2.23Clase AdjustmentEvent

Se produce un evento ***AdjustementEvent*** cada vez que se cambia el valor (entero) de una ***Scrollbar***.

Hay cinco tipos de ***AdjustementEvent***:

1. ***track***: se arrastra el cursor de la ***Scrollbar***.

2. ***unit increment***, ***unit decrement***: se hace click en las flechas de la ***Scrollbar***.

3. ***block increment***, ***block decrement***: se hace click encima o debajo del cursor.

La Tabla 7.21 muestra algunos métodos de esta clase. Las constantes UNIT_INCREMENT, UNIT_DECREMENT, BLOCK_INCREMENT, BLOCK_DECREMENT, TRACK permiten saber el tipo de acción producida, comparando con el valor de retorno de ***getAdjustementType()***.

Métodos de la clase AdjustementEvent	Función que realizan
Adjustable getAdjustable()	Devuelve el Component que generó el evento (implementa la interface Adjustable)
int getAdjustementType()	Devuelve el tipo de adjustement
int getValue()	Devuelve el valor de la Scrollbar después del cambio

Tabla 7.21. Métodos de la clase AdjustmentEvent.

7.2.24Clase ScrollPane

Un ***ScrollPane*** es como una ventana de tamaño limitado en la que se puede mostrar un componente de mayor tamaño con dos ***Scrollbar***s, una horizontal y otra vertical. El componente puede ser una imagen, por ejemplo. Las ***Scrollbars*** son visibles sólo si son necesarias (por defecto). Las constantes de la clase ***ScrollPane*** son (su significado es evidente): SCROLLBARS_AS_NEEDED, SCROLLBARS_ALWAYS, SCROLLBARS_NEVER. Los ***Paneles de scroll*** no generan eventos. La Tabla 7.22 muestra algunos métodos de esta clase.

Métodos de ScrollPane	Función que realizan
ScrollPane(), ScrollPane(int scbs)	Constructores que pueden incluir las ctes.
Dimension getViewportSize(), int getHScrollbarHeight(), int getVScrollbarWidth()	Obtiene el tamaño del ScrollPane y la altura y anchura de las barras de desplazamiento
setScrollPosition(int x, int y), setScrollPosition(Point p), Point getScrollPosition()	Permiten establecer u obtener la posición del componente
setSize(int, int), add(Component)	Heredados de Container, permiten establecer el tamaño y añadir un componente

Tabla 7.22. Métodos de la clase ScrollPane.

En el caso en que no aparezcan scrollbars (SCROLLBARS_NEVER) será necesario desplazar el componente (hacer ***scrolling***) desde programa, con el método ***setScrollPosition()***.

7.2.25Clases TextArea y TextField

Ambas componentes heredan de la clase ***TextComponent*** y muestran texto seleccionable y editable. La diferencia principal es que ***TextField*** sólo puede tener una línea, mientras que ***TextArea*** puede tener varias líneas. Además, ***TextArea*** ofrece posibilidades de edición de texto adicionales.

Se pueden especificar el *font* y los *colores de foreground y background*. Sólo la clase ***TextField*** genera ***ActionEvent***s, pero como las dos heredan de la clase ***TextComponent*** ambas pueden recibir ***TextEvent***s. La Tabla 7.23 muestra algunos métodos de las clases ***TextComponent***, ***TextField*** y ***TextArea***. No se pueden crear

objetos de la clase ***TextComponent*** porque su ***constructor*** no es ***public***; por eso su constructor no aparece en la Tabla 7.23.

Métodos heredados de TextComponent	**Función que realizan**
String getText() y setText(String str)	Permiten establecer u obtener el texto del componente
setEditable(boolean b), boolean isEditable()	Hace que el texto sea editable o pregunta por ello
setCaretPosition(int n), int getCaretPosition()	Fija la posición del punto de inserción o la obtiene
String getSelectedText(), int getSelectionStart() y int getSelectionEnd()	Obtiene el texto seleccionado y el comienzo y el final de la selección
selectAll(), select(int start, int end)	Selecciona todo o parte del texto
Métodos de TextField	**Función que realizan**
TextField(), TextField(int ncol), TextField(String s), TextField(String s, int ncol)	Constructores de TextField
int getColumns(), setColoumns(int)	Obtiene o establece el número de columnas del TextField
setEchoChar(char c), char getEchoChar(), boolean echoCharIsSet()	Establece, obtiene o pregunta por el carácter utilizado para passwords, de forma que no se pueda leer lo tecleado por el usuario
Métodos de TextArea	**Función que realizan**
TextArea(), TextArea(int nfil, int ncol), TextArea(String text), TextArea(String text, int nfil, int ncol)	Constructores de TextArea
setRows(int), setColumns(int), int getRows(), int getColumns()	Establecer y/u obtener los números de filas y de columnas
append(String str), insert(String str, int pos), replaceRange(String s, int i, int f)	Añadir texto al final, insertarlo en una posición determinada y reemplazar un texto determinado

Tabla 7.23. Métodos de las clases TextComponent, TextField y TextArea.

La clase ***TextComponent*** recibe eventos ***TextEvent***, y por lo tanto también los reciben sus clases derivadas ***TextField*** y ***TextArea***s. Este evento se produce cada vez que se modifica el texto del componente. La caja ***TextField*** soporta también el evento ***ActionEvent***, que se produce cada vez que el usuario termina de editar la única línea de texto pulsando ***Enter***.

Como es natural, las cajas de texto pueden recibir también los eventos de sus ***superclase***s, y más en concreto los eventos de ***Component***: ***FocusEvent***, ***MouseEvent*** y sobre todo ***KeyEvent***. Estos eventos permiten capturar las teclas pulsadas por el usuario y tomar las medidas adecuadas. Por ejemplo, si el usuario debe teclear un número en un ***TextField***, se puede crear una función que vaya capturando los caracteres tecleados y que rechace los que no sean numéricos.

Cuando se cambia desde programa el número de filas y de columnas de un ***TextField*** o ***TextArea***, hay que llamar al método ***validate()*** de la clase ***Component***, para que vuelva a aplicar el ***LayoutManager*** correspondiente. De todas formas, los tamaños fijados por el usuario tienen el carácter de ?recomendaciones? o tamaños ?preferidos?, que el ***LayoutManager*** puede cambiar si es necesario.

7.2.26Clase TextEvent

Se produce un ***TextEvent*** cada vez que cambia algo en un ***TextComponent (TextArea*** y ***TextField***).

Se puede desear evitar ciertos caracteres y para eso hay que gestionar los eventos correspondientes.

La interface ***TextListener*** tiene un único método: ***void textValueChanged(TextEvent te***). No hay métodos propios de la clase ***TextEvent***. Se puede utilizar el método ***Object getSource()***, que es heredado de ***EventObject***.

7.2.27Clase KeyEvent

Se produce un ***KeyEvent*** al pulsar sobre el teclado. Como el teclado no es un componente del AWT, es el ***objeto que tiene el foco*** en ese momento quien genera los eventos ***KeyEvent*** (el objeto que es el *event source*).

Hay dos tipos de ***KeyEvent***s:

1. ***key-typed***, que representa la introducción de un carácter Unicode.

2. ***key-pressed*** y ***key-released***, que representan pulsar o soltar una tecla. Son importantes para teclas que no representan caracteres, como por ejemplo F1.

Para estudiar estos eventos son muy importantes las ***Virtual KeyCodes*** (VKC). Las VKC son unas constantes que se corresponden con las ***teclas físicas*** del teclado, sin considerar minúsculas (que no tienen VKC). Se indican con el prefijo VK_, como VK_SHIFT o VK_A. La clase ***KeyEvent*** (en el paquete ***java.awt.event***) define constantes VKC para todas las teclas del teclado.

Por ejemplo, para escribir la letra "A" mayúscula se generan 5 eventos: *key-pressed VK_SHIFT*, *key-pressed VK_A*, *key-typed "A"*, *key-released VK_A*, y *key-released VK_SHIFT*. La Tabla 7.24 muestra algunos métodos de la clase ***KeyEvent*** y otros heredados de ***InputEvent***.

Métodos de la clase KeyEvent	**Función que realizan**
int getKeyChar()	Obtiene el carácter Unicode asociado con el evento
int getKeyCode()	Obtiene el VKC de la tecla pulsada o soltada
boolean isActionKey()	Indica si la tecla del evento es una ActionKey (HOME, END, ...)
String getKeyText(int keyCode)	Devuelve un String que describe el VKC, tal como "HOME", "F1" o "A". Estos Strings se definen en el fichero awt.properties
String getKeyModifiersText(int modifiers)	Devuelve un String que describe las teclas modificadoras, tales como"Shift" o "Ctrl+Shift" (fichero awt.properties)
Métodos heredados de InputEvent	**Función que realizan**
boolean isShiftDown(), boolean isControlDown(), boolean isMetaDown(), boolean isAltDown(), int getModifiers()	Permiten identificar las teclas modificadoras

Tabla 7.24. Métodos de la clase KeyEvent.

Las constantes de ***InputEvent*** permiten identificar las teclas modificadoras y los botones del ratón.

Estas constantes son SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK. A estas constantes se pueden aplicar las operaciones lógicas de bits para detectar combinaciones de teclas o pulsaciones múltiples.

7.3 MENUS

Los ***Menús*** de ***Java*** no descienden de ***Component***, sino de ***MenuComponent***, pero tienen un comportamiento similar, pues aceptan ***Event***os. La Figura 7.4 muestra la jerarquía de clases de los ***Menús*** de ***Java 1.***1.

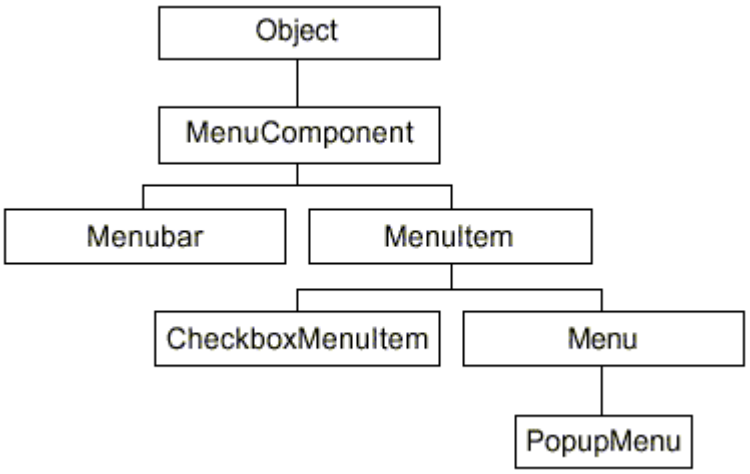


Figura 7.4. Jerarquía de clases para los menús.

Para crear un ***Menú*** se debe crear primero una ***MenuBar***; después se crean los ***Menús*** y los

MenuItem. Los ***MenuItems*** se añaden al ***Menú*** correspondiente; los ***Menus*** se añaden a la ***MenuBar*** y la ***MenuBar*** se añade a un ***Frame***. Una ***MenuBar*** se añade a un ***Frame*** con el método ***setMenuBar()***, de la clase ***Frame***. También puede añadirse un ***Menú*** a otro **Menu** para crear un ***submenú***, del modo que es habitual en ***Windows***.

La clase ***Menu*** es ***subclase*** de ***MenuItem***. Esto es así precisamente para permitir que un ***Menu*** sea añadido a otro ***Menu***.

7.3.1 Clase MenuShortcut

La clase ***java.awt.MenuShortcut*** (derivada de ***Object***) representa las teclas aceleradoras que pueden utilizarse para activar los menús desde teclado, sin ayuda del ratón. Se establece un ***Shortcut*** creando un objeto de esta clase con el constructor ***MenuShortcut(int vk_key)***, y pasándoselo al constructor adecuado de ***MenuItem***. Para más información, consúltese la documentación online de ***Java***. La Tabla 7.25 muestra algunos métodos de esta clase. Los ***MenuShortcut*** de ***Java*** están restringidos al uso la tecla control (CTRL). Al definir el ***MenuShortcut*** no hace falta incluir dicha tecla.

Métodos de la clase MenuShortcut	Función que realizan
MenuShortcut(int key)	Constructor
int getKey()	Obtiene el código virtual de la tecla utilizada como shortcut

Tabla 7.25. Métodos de la clase MenuShortcut.

7.3.2 Clase MenuBar

La Tabla 7.26 muestra algunos métodos de la clase ***MenuBar***. A una ***MenuBar*** sólo se pueden añadir objetos ***Menu***.

Métodos de MenuBar	Función que realizan
MenuBar()	Constructor
add(Menu), int getMenuCount(), Menu getMenu(int i)	Añade un manú, obtiene el número de menús y el menú en una posición determinada
MenuItem getShortcutMenuItem(MenuShortcut), deleteShortcut(MenuShortcut)	Obtiene el objeto MenuItem relacionado con un Shortcut y elimina el Shortcut especificado
remove(int index), remove(MenuComponent m)	Elimina un objeto Menu a partir de un índice o una referencia.
Enumeration shortcuts()	Obtiene un objeto Enumeration con todos los Shortcuts

Tabla 7.26. Métodos de la clase MenuBar.

7.3.3 Clase Menu

El objeto ***Menu*** define las opciones que aparecen al seleccionar uno de los menús de la barra de menús.

En un ***Menu*** se pueden introducir objetos ***MenuItem***, otros objetos ***Menu*** (para crear submenús), objetos ***CheckboxMenuItem***, y ***separadores***. La Tabla 7.27 muestra algunos métodos de la clase ***Menu***.

Métodos de Menu	Función que realizan
Menu(String)	Constructor a partir de una etiqueta
int getItemCount()	Obtener el número de items
MenuItem getItem(int)	Obtener el MenuItem a partir de un índice
add(String), add(MenuItem), addSeparator(), insertSeparator(int index)	Añadir un MenuItem o un separador
remove(int index), remove(MenuComponent), removeAll()	Eliminar uno o todos los componentes
insert(String lbl, int index), insert(MenuItem mnu, int index)	Insertar items en una posición dada
String getLabel(), setLabel(String)	Obtener y establecer las etiquetas de los items

Tabla 7.27. Métodos de la clase Menu.

Obsérvese que como ***Menu*** desciende de ***MenuItem***, el método ***add(MenuItem)*** permite añadir objetos ***Menu*** a otro ***Menu***.

7.3.4 Clase MenuItem

Los objetos de la clase ***MenuItem*** representan las distintas opciones de un menú. Al seleccionar, en la ejecución del programa, un objeto ***MenuItem*** se generan eventos del tipo ***ActionEvent***s. Para cada ítem de un ***Menu*** se puede definir un ***ActionListene***r, que define el método ***actionPerformed()***. La Tabla 7.28 muestra algunos métodos de la clase ***MenuIte***m.

Métodos de MenuItem	Función que realizan
MenuItem(String lbl), MenuItem(String, MenuShortcut)	Constructores. El carácter (-) es el label de los separators
boolean isEnabled(), setEnabled(boolean)	Pregunta y determina si en item está activo
String getLabel(), setLabel(String)	Obtiene y establece la etiqueta del item
MenuShortcut getShortcut(), setShortcut(MenuShortcut), deleteShortcut(MenuShortcut)	Permiten obtener, establecer y borrar los MenuShortcuts
String getActionCommand(), setActionCommand(String)	Para obtener y establecer un identificador distinto del label

Tabla 7.28. Métodos de la clase MenuItem.

El método ***getActionCommand()***, asociado al ***getSource()*** del evento correspondiente, no permite identificar correctamente al ***ítem*** cuando éste se ha activado mediante el ***MenuShortcut*** (en ese caso devuelve ***nul***l).

7.3.5 Clase CheckboxMenuItem

Son ***ítems*** de un ***Menu*** que pueden estar activados o no activados. La clase ***CheckboxMenuItem*** no genera un ***ActionEven***t, sino un ***ItemEven***t, de modo similar a la clase ***Checkbox*** (ver Apartado 7.2.17). En este caso hará registrar un ***ItemListene***r. La Tabla 7.29 muestra algunos métodos de esta clase.

Métodos de la clase CheckboxMenuItem	Función que realizan
CheckboxMenuItem(String lbl), CheckboxMenuItem(String lbl, boolean state)	Constructores
boolean getState(), setState(boolean)	Permiten obtener y establecer el estado del ChecboxMenuItem

Tabla 7.29. Métodos de la clase CheckboxMenuItem.

7.3.6 Menús pop-up

Los menús ***pop-up*** son menús que aparecen en cualquier parte de la pantalla al hacer click con el botón derecho del ratón ***(pop-up trigge***r) sobre un componente determinado ***(parent Componen***t). El menú ***pop-up*** se muestra en unas coordenadas relativas al ***parent Componen***t, que debe estar visible.

Métodos de PopupMenu	Función que realizan
PopupMenu(), PopupMenu(String title)	Constructores de PopupMenu:
show(Component origin, int x, int y)	Muestra el pop-up menú en la posición indicada

Tabla 7.30. Métodos de la clase PopupMenu.

Además, se pueden utilizar los métodos de la clase ***Menu*** (ver página 105), de la que deriva ***PopupMenu***. Para hacer que aparezca el ***PopupMenu*** habrá que registrar el ***MouseListener*** y definir el método ***mouseClicked()***.

7.4 LAYOUT MANAGERS

La portabilidad de ***Java*** a distintas plataformas y distintos sistemas operativos necesita flexibilidad a la hora de situar los ***Componentes*** (*Button*s, *Canva*s, *TextArea*s, etc.) en un ***Contenedor*** (*Windo*w, *Pane*l, ?). Un ***Layout Manager*** es un objeto que controla cómo los ***Componentes*** se sitúan en un ***Contenedor***.

7.4.1 Concepto y Ejemplos de LayoutsManagers

El AWT define cinco ***Layout Manager***s: dos muy sencillos ***(FlowLayout*** y ***GridLayou***t), dos más especializados ***(BorderLayout*** y ***CardLayou***t) y uno muy general ***(GridBagLayou***t). Además, los usuarios pueden escribir su propio ***Layout Manage***r, implementando la interface ***LayoutManage***r, que especifica 5 métodos. ***Java*** permite también posicionar los ***Componentes*** de ***modo absolut***o, sin ***Layout Manage***r, pero de ordinario puede perderse la portabilidad y algunas otras características.

Todos los ***Contenedores*** tienen un ***Layout Manager*** por defecto, que se utiliza si no se indica otra cosa: Para ***Pane***l, el defecto es un objeto de la clase ***FlowLayou***t. Para ***Window (Frame*** y ***Dialo***g), el defecto es un objeto de la clase ***BorderLayou***t.

La Figura 7.5 muestra un ejemplo de ***FlowLayou***t: Los componentes se van añadiendo de izquierda a derecha y de arriba hacia abajo. Se puede elegir alineación por la izquierda, centrada o por la derecha, respecto al contenedor.

Figura 7.5. Ejemplo de FlowLayout.

La Figura 7.6 muestra un ejemplo de ***BorderLayou***t: el contenedor se divide en 5 zonas: ***Nort***h, ***Sout***h, ***Eas***t, ***West*** y ***Center*** (que ocupa el resto de espacio).

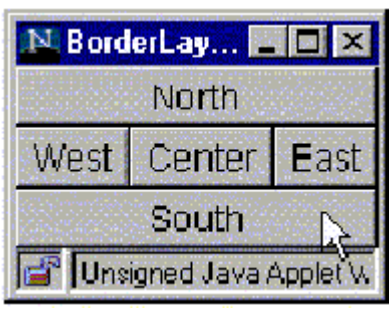


Figura 7.7. Ejemplo de BorderLayout

El ejemplo de ***GridLayout*** se muestra en la Figura 7.7. Se utiliza una ***matriz de celdas*** que se numeran como se muestra en dicha figura (de izquierda a derecha y de arriba a abajo).

Figura 7.7. Ejemplo de GridLayout.

La Figura 7.8 muestra un ejemplo de uso del ***GridBagLayou***t. Se utiliza también una matriz de celdas, pero permitiendo que algunos componentes ocupen más de una celda.

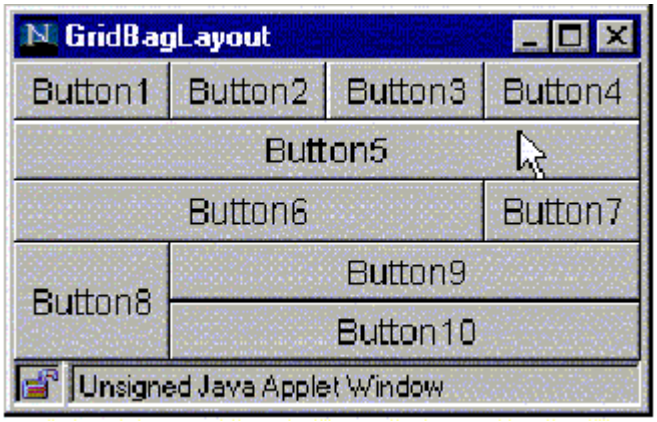


Figura 7.8. Ejemplo de GridBagLayout.

Finalmente, la Figura 7.9 y las dos Figuras siguientes muestran un ejemplo de ***CardLayou***t. En este caso se permite que el mismo espacio sea utilizado sucesivamente por contenidos diferentes.

Figura 7.9. CardLayout: pantalla 1.

Figura 7.10. CardLayout: pantalla 2

Figura 7.11. CardLayout: pantalla 3.

7.4.2 Ideas generales sobre los LayoutManagers

Se debe elegir el ***Layout Manager*** que mejor se adecué a las necesidades de la aplicación que se desea desarrollar. Recuérdese que cada ***Contenedor*** tiene un ***Layout Manager*** por defecto. Si se desea utilizar el ***Layout Manager*** por defecto basta crear el ***Contenedor*** (su constructor crea un objeto del ***Layout Manager*** por defecto e inicializa el ***Contenedor*** para hacer uso de él).

Para utilizar un ***Layout Manager*** diferente hay que crear un objeto de dicho ***Layout Manager*** y pasárselo al constructor del contenedor o decirle a dicho contenedor que lo utilice por medio del método ***setLayout()***, en la forma:

```
unContainer.setLayout(new GridLayout());
```

La clase ***Container*** dispone de métodos para manejar el ***Layout Manager*** (ver Tabla 7.31): Si se cambia de modo indirecto el tamaño de un ***Component*** (por ejemplo cambiando el tamaño del ***Font***), hay que llamar al método ***invalidate()*** del ***Component*** y luego al método ***validate()*** del ***Contenedor***, lo que hace que se ejecute el método ***doLayout()*** para reajustar el espacio disponible.

Métodos de Container para manejar Layout Managers	Función que realizan
add()	Permite añadir Components a un Container
remove() y removeAll()	Permiten eliminar Components de un Container
doLayout(), validate()	doLayout() se llama automáticamente cada vez que hay que redibujar el Container y sus Components. Se llama también cuando el usuario llama al método validate()

Tabla 7.31. Métodos de Container para manejar los Layout Managers.

7.4.3 FlowLayout

FlowLayout es el ***Layout Manager*** por defecto para ***Panel***. ***FlowLayout*** coloca los componentes uno detrás de otro, en una fila, de izquierda a derecha y de arriba a abajo, en la misma forma en que procede un procesador de texto con las palabras de un párrafo. Los componentes se añaden en el mismo orden en que se ejecutan los métodos ***add()***. Si se cambia el tamaño de la ventana los componentes se redistribuyen de modo acorde, ocupando más filas si es necesario.

La clase ***FlowLayout*** tiene tres constructores:

```
FlowLayout();

FlowLayout(int alignement);

FlowLayout(int alignement, int horizontalGap, int verticalGap);
```

Se puede establecer la alineación de los componentes (centrados, por defecto), por medio de las constantes FlowLayout.LEFT, FlowLayout.CENTER y FlowLayout.RIGHT.

Es posible también establecer una distancia horizontal y vertical entre componentes (el ***intervalo***, en pixeles). El valor por defecto son 5 pixeles.

7.4.4 BorderLayout

BorderLayout es el ***Layout Manager*** por defecto para ***Windows*** y ***Frames***. ***BorderLayout*** define cinco áreas: ***North***, ***South***, ***East***, ***West*** y ***Center***. Si se aumenta el tamaño de la ventana todas las zonas se mantienen en su mínimo tamaño posible excepto ***Center***, que absorbe casi todo el crecimiento. Los componentes añadidos en cada zona tratan de ocupar todo el espacio disponible. Por ejemplo, si se añade un botón, el botón se hará tan grande como la celda, lo cual puede producir efectos muy extraños. Para evitar esto se puede introducir en la celda un panel con ***FlowLayout*** y añadir el botón al panel y el panel a la celda.

Los constructores de ***BorderLayout*** son los siguientes:

```
BorderLayout();

BorderLayout(int horizontalGap, int verticalGap);
```

Por defecto ***BorderLayout*** no deja espacio entre componentes. Al añadir un componente a un ***Contenedor*** con ***BorderLayout*** se debe especificar la zona como segundo argumento:

```
miContainer.add(new Button("Norte"), "North");
```

7.4.5 GridLayout

Con ***GridLayout*** las componentes se colocan en una matriz de celdas. Todas las celdas tienen el mismo tamaño. Cada componente utiliza todo el espacio disponible en su celda, al igual que en ***BorderLayou***t.

GridLayout tiene dos constructores:

```
GridLayout(int nfil, int ncol);

GridLayout(int nfil, int ncol, int horizontalGap, int
verticalGap);
```

Al menos uno de los parámetros ***nfil*** y ***ncol*** debe ser distinto de cero. El valor por defecto para el espacio entre filas y columnas es cero pixeles.

7.4.6 CardLayout

CardLayout permite disponer distintos componentes (de ordinario ***Paneles***) que comparten la misma ventana para ser mostrados sucesivamente. Son como transparencias, diapositivas o cartas de baraja que van apareciendo una detrás de otra.

El ***orden*** de las "cartas" se puede establecer de los siguientes modos:

1. Yendo a la primera o a la última, de acuerdo con el orden en que fueron añadidas al contenedor.

2. Recorriendo las cartas hacia delante o hacia atrás, de una en una.

3. Mostrando una carta con un nombre determinado.

Los ***constructores*** de esta clase son:

```
CardLayout()

CardLayout(int horizGap, int vertGap)
```

Para añadir componentes a un contenedor con ***CardLayout*** se utiliza el método:

```
Container.add(Component comp, int index)
```

donde ***index*** indica la posición en que hay que insertar la carta. Los siguientes métodos de ***CardLayout*** permiten controlar el orden en que aparecen las cartas:

```
void first(Container cont);

void last(Container cont);

void previous(Container cont);

void next(Container cont);

void show(Container cont, String nameCard);
```

7.4.7 GridBagLayout

El ***GridBagLayout*** es el ***Layout Manager*** más completo y flexible, aunque también el más complicado de entender y de manejar. Al igual que el ***GridLayout***, el ***GridBagLayout*** parte de una matriz de celdas en la que se sitúan los componentes. La diferencia está en que las filas pueden tener distinta altura, las columnas pueden tener distinta anchura, y además en el ***GridBagLayout*** un componente puede ocupar varias celdas contiguas.

La posición y el tamaño de cada componente se especifican por medio de unas ?restricciones? o ***constraints***. Las restricciones se establecen creando un objeto de la clase ***GridBagConstraints***, dando valor a sus propiedades (variables miembro) y asociando ese objeto con el componente por medio del método ***setConstraints()***.

Las ***variables miembro*** de ***GridBagConstraints*** son las siguientes:

- ***gridx*** y ***gridy***. Especifican la fila y la columna en la que situar la esquina superior izquierda del componente (se empieza a contar de cero). Con la constante GridBagConstraints.RELATIVE se indica que el componente se sitúa relativamente al anterior componente situado (es la condición por defecto).

- ***gridwidth*** y ***gridheight***. Determinan el número de columnas y de filas que va a ocupar el componente. El valor por defecto es una columna y una fila. La constante GridBagConstraints.REMAINDER indica que el componente es el último de la columna o de la fila, mientras que GridBagConstraints.RELATIVE indica que el componente se sitúa respecto al anterior componente de la fila o columna.

- ***fill***. En el caso en que el componente sea más pequeño que el espacio reservado, esta variable indica si debe ocupar o no todo el espacio disponible. Los posibles valores son: GridBagConstraints.NONE (no lo ocupa; defecto), GridBagConstraints.HORIZONTAL (lo ocupa en dirección horizontal), GridBagConstraints.VERTICAL (lo ocupa en vertical) y GridBagConstraints.BOTH (lo ocupa en ambas direcciones).

- ***ipadx*** y ***ipady***. Especifican el espacio a añadir en cada dirección al tamaño interno del componente. Los valores por defecto son cero. El tamaño del componente será el tamaño mínimo más dos veces el ***ipadx*** o el ***ipady***.

- ***insets***. Indican el espacio mínimo entre el componente y el espacio disponible. Se establece con un objeto de la clase ***java.awt.Insets***. Por defecto es cero.

- ***anchor***. Se utiliza para determinar dónde se coloca el componente, cuando éste es menor que el espacio disponible. Sus posibles valores vienen dados por las constantes de la clase ***GridBagConstraints***: CENTER (el valor por defecto), NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST y NORTHWEST.

• ***weightx*** y ***weight***y. Son unos coeficientes entre 0.0 y 1.0 que sirven para dar más o menos ?peso? a las distintas filas y columnas. A más ?peso? más probabilidades tienen de que se les dé más anchura o más altura.

A continuación se muestra una forma típica de crear un contenedor con ***GridBagLayout*** y de añadirle componentes:

```
GridBagLayout unGBL = new GridBagLayout();

GridBagConstraints unasConstr = new GridBagConstraints();

unContainer.setLayout(unGBL);

// Ahora ya se pueden añadir los componentes

//...Se crea un componente unComp

//...Se da valor a las variables del objeto unasConstr

// Se asocian las restricciones con el componente

unGBL.setConstraints(unComp, unasConstr);

// Se añade el componente al contenedor

unContainer.add(unComp);
```

7.5 GRÁFICOS, TEXTO E IMÁGENES

En esta parte final del AWT se van a describir, también muy sucintamente, algunas clases y métodos para realizar dibujos y añadir texto e imágenes a la interface gráfica de usuario.

7.5.1 Capacidades gráficas del AWT: Métodos paint(), repaint() y update()

La clase ***Component*** tiene tres métodos muy importantes relacionados con gráficos: ***paint()***, ***repaint()*** y ***update()***. Cuando el usuario llama al método ***repaint()*** de un componente, el AWT llama al método ***update()*** de ese componente, que por defecto llama al método ***paint()***.

7.5.1.1 Método paint(Graphics g)

El método ***paint()*** está definido en la clase ***Component***, pero ese método no hace nada y hay que redefinirlo en una de sus clases derivadas. El programador no tiene que preocuparse de llamar a este método: el sistema operativo lo llama al dibujar por primera vez una ventana, y luego lo vuelve a llamar cada vez que entiende que la

ventana o una parte de la ventana debe ser redibujada (por ejemplo, por haber estado tapada por otra ventana y quedar de nuevo a la vista).

7.5.1.2 Método update(Graphics g)

El método ***update()*** hace dos cosas: primero redibuja la ventana con el color de fondo y luego llama al método ***paint()***. Este método también es llamado por el AWT, y también puede ser llamado por el programador, quizás porque ha realizado algún cambio en la ventana y necesita que se dibuje de nuevo. La propia estructura de este método el comenzar pintando de nuevo con el color de fondo- hace que se produzca ***parpadeo*** en las animaciones. Una de las formas de evitar este efecto es redefinir este método de una forma diferente, cambiando de una imagen a otra sólo lo que haya que cambiar, en vez de redibujar todo otra vez desde el principio. Este método no siempre proporciona los resultados buscados y hay que recurrir al método del ***doble buffe***r.

7.5.1.3 Método repaint()

Este es el método que con más frecuencia es llamado por el programador. El método ***repaint()*** llama ?lo antes posible? al método ***update()*** del componente. Se puede también especificar un número de milisegundos para que el método ***update()*** se llame transcurrido ese tiempo. El método ***repaint()*** tiene las cuatro formas siguientes:

```
repaint()

repaint(long time)

repaint(int x, int y, int w, int h)

repaint(long time, int x, int y, int w, int h)
```

Las formas tercera y cuarta permiten definir una ***zona rectangular*** de la ventana a la que aplicar el método.

7.5.2 Clase Graphics

El único argumento de los métodos ***update()*** y ***paint()*** es un objeto de esta clase. La clase ***Graphics*** dispone de métodos para soportar dos tipos de gráficos:

1.Dibujo de ***primitivas gráficas*** (*text*o, *línea*s, *círculo*s, *rectángulo*s, *polígono*s, ?).

2.Presentación de ***imágenes*** en formatos ****.gif*** y ****.jpe***g.

Además, la clase ***Graphics*** mantiene un ***contexto gráfic***o: un área de dibujo actual, un color de dibujo del background y otro del foreground, un font con todas sus propiedades, etc. La Figura 7.12 muestra el sistema de coordenadas utilizado en ***Jav***a.

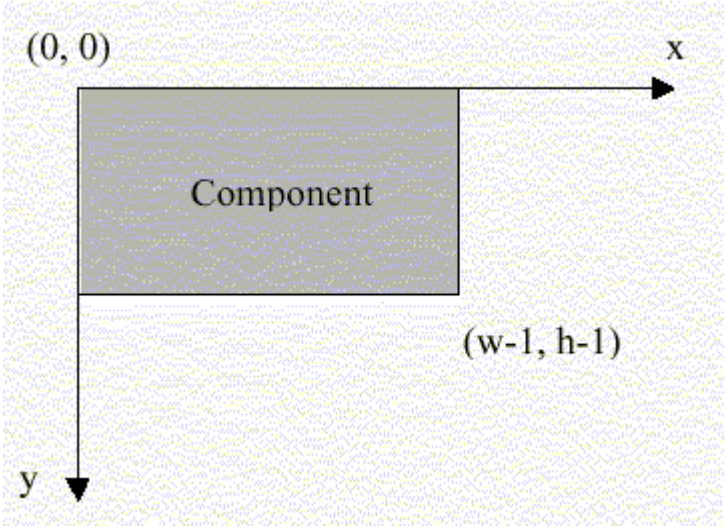


Figura 7.12. Coordenadas de los gráficos de Java.

Como es habitual en Informática, los ejes están situados en la esquina superior izquierda, con la orientación indicada en la Figura 7.12 (eje de ordenadas descendente). Las coordenadas se miden siempre en ***pixeles***.

7.5.3 Primitivas gráficas

Java dispone de métodos para realizar dibujos sencillos, llamados a veces ?primitivas? gráficas. Como se ha dicho, las coordenadas se miden en ***pixeles***, empezando a contar desde cero. La clase ***Graphics*** dispone de los métodos para primitivas gráficas reseñados en la Tabla 7.32.

Método gráfico	**Función que realizan**
drawLine(int x1, int y1, int x2, int y2)	Dibuja una línea entre dos puntos
drawRect(int x1, int y1, int w, int h)	Dibuja un rectángulo (w-1, h-1)
fillRect(int x1, int y1, int w, int h)	Dibuja un rectángulo y lo rellena con el color actual
clearRect(int x1, int y1, int w, int h)	Borra dibujando con el background color
draw3DRect(int x1, int y1, int w, int h, boolean raised)	Dibuja un rectángulo resaltado (w+1, h+1)
fill3DRect(int x1, int y1, int w, int h, boolean raised)	Rellena un rectángulo resaltado (w+1, h+1)
drawRoundRect(int x1, int y1, int w, int h, int arcw, int arch)	Dibuja un rectángulo redondeado
fillRoundRect(int x1, int y1, int w, int h, int arcw, int arch)	Rellena un rectángulo redondeado
drawOval(int x1, int y1, int w, int h)	Dibuja una elipse
fillOval(int x1, int y1, int w, int h)	Dibuja una elipse y la rellena de un color
drawArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)	Dibuja un arco de elipse (ángulos en grados)
fillArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)	Rellena un arco de elipse
drawPolygon(int x[], int y[], int nPoints)	Dibuja y cierra el polígono de modo automático
drawPolyline(int x[], int y[], int nPoints)	Dibuja un polígono pero no lo cierra
fillPolygon(int x[], int y[], int nPoints)	Rellena un polígono

Tabla 7.32. Métodos de la clase Graphics para dibujo de primitivas gráficas.

Excepto los polígonos y las líneas, todas las formas geométricas se determinan por el rectángulo que las comprende, cuyas dimensiones son ***w*** y h. Los polígonos admiten un argumento de la clase ***java.awt.Polygon***.

Los métodos ***draw3DRect()***, ***fill3DRect()***, ***drawOval()***, ***fillOval()***, ***drawArc()*** y ***fillArc()***

dibujan objetos cuyo tamaño total es (w+1, h+1) pixeles.

7.5.4 Clases Graphics y Font

La clase ***Graphics*** permite ?dibujar? texto, como alternativa al texto mostrado en los componentes ***Label***, ***TextField*** y ***TextArea***. Los métodos de esta clase para dibujar texto son los siguientes:

```
drawBytes(byte data[], int offset, int length, int x, int y);

drawChars(char data[], int offset, int length, int x, int y);

drawString(String str, int x, int y);
```

En estos métodos, los argumentos ***x*** e ***y*** representan las coordenadas de la ***línea base*** (ver Figura 7.13). El argumento ***offset*** indica el elemento del array que se empieza a imprimir.

Figura 7.13. Líneas importantes en un tipo de letra.

Cada tipo de letra está representado por un objeto de la clase ***Font***. Las clases ***Component*** y ***Graphics*** disponen de métodos ***setFont()*** y ***getFont()***. El constructor de ***Font*** tiene la forma:

```
Font(String name, int style, int size)
```

donde el ***style*** se puede definir con las constantes Font.PLAIN, Font.BOLD y Font.ITALIC. Estas constantes se pueden combinar en la forma: Font.BOLD | Font.ITALIC.

La clase ***Font*** tiene tres variables *protecte*d, llamadas ***nam***e, ***style*** y ***siz***e. Además tiene tres constantes enteras: PLAIN, BOLD e ITALIC. Esta clase dispone de los métodos ***String getName()***, ***int getStyle()***, ***int getSize()***, ***boolean isPlain()***, ***boolean isBold()*** y ***boolean isItalic()***, cuyo significado es inmediato. Para mayor portabilidad se recomienda utilizar nombres lógicos de fonts, tales como ***Serif*** (*Times New Roma*n), ***SansSerif*** (*Aria*l) y ***Monospaced*** (*Courie*r).

7.5.5 Clase FontMetrics

La clase ***FontMetrics*** permite obtener información sobre una ***font*** y sobre el espacio que ocupa un ***char*** o un ***String*** utilizando esa ***fon***t. Esto es muy útil cuando se pretende rotular algo de modo que quede siempre centrado y bien dimensionado. La clase ***FontMetrics*** es una clase ***abstrac***t. Esto quiere decir que no se pueden crear directamente objetos de esta clase ni llamar a su constructor. La forma habitual de soslayar esta dificultad es creando una subclase. En la práctica ***Java*** resuelve esta dificultad para el usuario, ya que la clase ***FontMetrics*** tiene como variable miembro un objeto de la clase ***Fon***t. Por ello, un objeto de la clase ***FontMetrics*** contiene información sobre la ***font*** que se le ha pasado como argumento al constructor. De todas formas, el camino más habitual para obtener esa información es a partir de un objeto de la clase ***Graphics*** que ya tiene un ***font*** definido. A partir de un objeto ***g*** de la clase ***Graphics*** se puede obtener una referencia ***FontMetrics*** en la forma:

```
FontMetrics miFontMet = g.getFontMetrics();
```

donde está claro que se está utilizando una referencia de la clase *abstract* ***FontMetrics*** para refererirse a un objeto de una clase derivada creada dentro del API de ***Jav***a. Con una referencia de tipo ***FontMetrics*** se pueden utilizar todos los métodos propios de dicha clase.

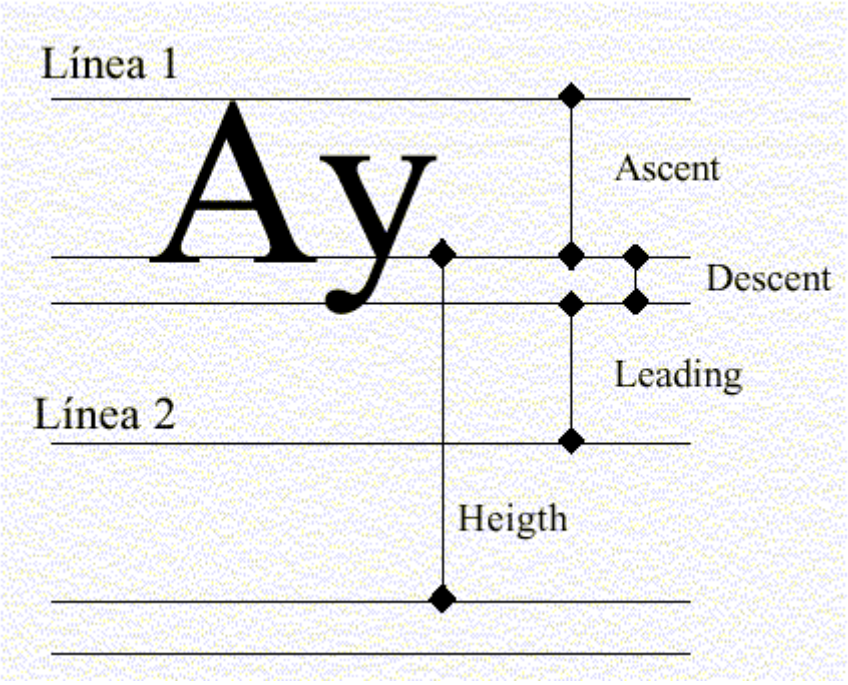


Figura 7.14. Nomenclatura para la clase FontMetrics.

La Tabla 7.33 muestra algunos métodos de la clase ***FontMetrics***, para los que se debe tener en cuenta la terminología introducida en la Figura 7.14.

Métodos de la clase FontMetrics	Función que realizan
FontMetrics(Font font)	Constructor
int getAscent(), int getMaxAscent()	Permiten obtener el "Ascent" actual y máximo para esa font
int getDescent(), int getMaxDescent()	Permiten obtener el "Descent" actual y máximo para esa font
int getHeight(), int getLeading()	Permiten ontener la distancia entre líneas y la distancia entre el descender de una línea y el ascender de la siguiente
int getMaxAdvance()	Da la máxima anchura de un carácter de esa font, incluyendo el espacio hasta el siguiente carácter
int charWidth(int ch), int charWidth(char ch), int stringWidth(String str)	Dan la anchura de un carácter (incluyendo el espacio hasta el siguiente carácter) o de toda una cadena de caracteres
int charsWidth(char data[], int start, int len), int bytesWidth(byte data[], int start, int len)	Dan la anchura de un array de caracteres o de bytes. Permiten definir la posición del comienzo y el número de caracteres

Tabla 7.33. Métodos de la clase FontMetrics.

7.5.6 Clase Color

La clase ***java.awt.Color*** encapsula colores utilizando el formato RGB (Red, Green, Blue). Las componentes de cada color primario en el color resultante se expresan con números enteros entre 0 y 255, siendo 0 la intensidad mínima de ese color, y 255 la máxima.

En la clase ***Color*** existen constantes para colores predeterminados de uso frecuente: ***black***, ***white***, ***green***, ***blue***, ***red***, ***yellow***, ***magenta***, ***cyan***, ***orange***, ***pink***, ***gray***, ***darkGray***, ***lightGray***. La Tabla 7.34 muestra algunos métodos de la clase ***Color***.

Metodos de la clase Color	Función que realizan
Color(int), Color(int,int,int), Color(float,float,float)	Constructores de Color, con tres bytes en un int (del bit 0 al 23), con enteros entre 0 y 255 y float entre 0.0 y 1.0
Color brighter(), Color darker()	Obtienen una versión más o menos brillante de un color
Color getColor(), int getRGB()	Obtiene un color en los tres primeros bytes de un int
int getGreen(), int getRed(), int getBlue()	Obtienen las componentes de un color
Color getHSBColor()	Obtiene un color a partir de los valores de "hue", "saturation" y "brightness" (entre 0.0 y 1.0)
float[] RGBtoHSB(int,int,int,float[]), int HSBtoRGB(float,float,float)	Métodos static para convertir colores de un sistema de definición de colores a otro

Tabla 7.34. Métodos de la clase Color.

7.5.7 Imágenes

Java permite incorporar imágenes de tipo GIF y JPEG definidas en archivos. Se dispone para ello de la clase ***java.awt.Image***. Para cargar una imagen hay que indicar la localización del archivo (URL) y cargarlo mediante los métodos ***Image getImage(String)*** o ***Image getImage(URL, String)***. Estos métodos existen en las clases ***java.awt.Toolkit*** y ***java.applet.Applet***. El argumento de tipo ***String*** representa una variable conteniendo el nombre del archivo.

Cuando estas imágenes se cargan en ***applet***s, para obtener el URL pueden ser útiles las funciones ***getDocumentBase()*** y ***getCodeBase()***, que devuelven el URL del archivo HTML que llama al ***applet***, y el directorio que contiene el ***applet*** (en forma de ***String***).

Para cargar una imagen hay que comenzar creando un objeto ***Image***, y llamar al método ***getImage()***, pasándole como argumento el URL. Por ejemplo:

```
Image miImagen = getImage(getCodeBase(), "imagen.gif")
```

Una vez cargada la imagen, hay que representarla, para lo cual se redefine el método ***paint()*** para llamar al método ***drawImage()*** de la clase ***Graphics***. Dicho método admite varias formas, aunque casi siempre hay que incluir el nombre del objeto imagen creado, las dimensiones de dicha imagen y un objeto ***ImageObserver***.

ImageObserver es una interface que declara métodos para observar el estado de la carga y visualización de la imagen. Si se está programando un ***applet***, basta con poner como ***ImageObserver*** la referencia ***this***, ya que en la mayoría de los casos, la implementación de esta interface en la clase ***Applet*** proporciona el comportamiento deseado. Para más información sobre dicho método dirigirse a la referencia de la API.

La clase ***Image*** define ciertas constantes para controlar los algoritmos de cambio de escala:

SCALE_DEFAULT, SCALE_FAST, SCALE_SMOOTH,

SCALE_REPLICATE, SCALE_AVERAGE.

La Tabla 7.35 muestra algunos métodos de la clase ***Image***.

Métodos de la clase Image	Función que realizan
Image()	Constructor
int getWidth(ImageObserver) int getHeight(ImageObserver)	Determinan la anchura y la altura de la imagen. Si no se conocen todavía, este método devuelve -1 y el objeto ImageObserver especificado será notificado más tarde
Graphics getGraphics()	Crea un contexto gráfico para poder dibujar en una imagen no visible en pantalla. Este método sólo se puede llamar para objetos no visibles en pantalla
Object getProperty(String, ImageObserver)	Obtiene una propiedad de una imagen a partir del nombre de la propiedad
Image getScaledInstance(int w, int h, int hints)	Crea una versión de la imagen a otra escala. Si w o h son negativas se utiliza la otra dimensión manteniendo la proporción. El último argumento es información para el algoritmo de cambio de escala

Tabla 7.35. Métodos de la clase Image.

7.6 ANIMACIONES

Las animaciones tienen un gran interés desde diversos puntos de vista. Una imagen vale más que mil palabras y una imagen en movimiento es todavía mucho más útil. Para presentar o describir ciertos conceptos el movimiento animado es fundamental. Además, las animaciones o mejor dicho, la forma de hacer animaciones en ***Java*** ilustran mucho

la forma en que dicho lenguaje realiza los gráficos. En estos apartados se va a seguir el esquema del ***Tutorial*** de ***Sun*** sobre el AWT.

Se pueden hacer animaciones de una forma muy sencilla: se define el método ***paint()*** de forma que cada vez que sea llamado dibuje algo diferente de lo que ha dibujado la vez anterior. De todas formas, recuérdese que el programador no llama directamente a este método. El programador llama al método ***repaint()***, quizás dentro de un bucle ***while*** que incluya una llamada al método ***sleep()*** de la clase ***Thread*** (ver Capítulo 7), para esperar un cierto número de milisegundos entre dibujo y dibujo (entre *frame* y *fram*e, utilizando la terminología de las animaciones). Recuérdese que ***repaint( )*** llama a ***update()*** lo antes posible, y que ***update()*** borra todo redibujando con el color de fondo y llama a ***paint()***.

La forma de proceder descrita da buenos resultados para animaciones muy sencillas, pero produce ***parpadeo*** cuando los gráficos son un poco más complicados. La razón está en el propio proceso descrito anteriormente, combinado con la velocidad de refresco del monitor. La velocidad de refresco vertical de un monitor suele estar entre 60 y 75 hertzios. Eso quiere decir que la imagen se actualiza unas 60 ó 75 veces por segundo. Cuando el refresco se realiza después de haber borrado la imagen anterior pintando con el color de fondo y antes de que se termine de dibujar de nuevo toda la imagen, se obtiene una imagen incompleta, que sólo aparecerá terminada en uno de los siguientes pasos de refresco del monitor. Ésta es la causa del ***parpadeo***. A continuación se verán dos formas de reducirlo o eliminarlo.

7.6.1 Eliminación del parpadeo redefiniendo el método update()

El problema del ***parpadeo*** se localiza en la llamada al método ***update()***, que borra todo pintando con el color de fondo y después llama a ***paint()***. Una forma de resolver esta dificultad es ***redefinir*** el método ***update()***, de forma que se adapte mejor al problema que se trata de resolver.

Una posibilidad es no repintar todo con el color de fondo, no llamar a ***paint()*** e introducir en ***update()*** el código encargado de realizar los dibujos, cambiando sólo aquello que haya que cambiar. A pesar de esto, es necesario redefinir ***paint()*** pues es el método que se llama de forma automática cuando la ventana de ***Java*** es tapada por otra que luego se retira. Una posible solución es hacer que ***paint()*** llame a ***update()***, terminando por establecer un orden de llamadas opuesto al de defecto. Hay que tener en cuenta que, al no borrar todo pintando con el color de fondo, el programador tiene que preocuparse de borrar de forma selectiva entre *frame* y *frame* lo que sea necesario. Los métodos ***setClip()*** y ***clipRect()*** de la clase ***Graphics*** permiten hacer que las operaciones gráficas no surtan efecto fuera de un área rectangular previamente determinada. Al ser dependiente del tipo de gráficos concretos de que se trate, este método no siempre proporciona soluciones adecuadas.

7.6.2 Técnica del doble buffer

La técnica del ***doble buffer*** proporciona la mejor solución para el problema de las animaciones, aunque requiere una programación algo más complicada. La idea básica del ***doble buffer*** es realizar los dibujos en una imagen invisible, distinta de la que se

está viendo en la pantalla, y hacerla visible cuando se ha terminado de dibujar, de forma que aparezca instantáneamente.

Para crear el segundo buffer o imagen invisible hay que crear un objeto de la clase ***Image*** del mismo tamaño que la imagen que se está viendo y crear un contexto gráfico u objeto de la clase ***Graphics*** que permita dibujar sobre la imagen invisible. Esto se hace con las sentencias,

```
Image imgInv;

Graphics graphInv;

Dimension dimInv;

Dimension d = size(); // se obtiene la dimensión del panel
```

en la clase que controle el dibujo (por ejemplo en una clase que derive de ***Panel***). En el método ***update()*** se modifica el código de modo que primero se dibuje en la imagen invisible y luego ésta se haga visible:

```
1.public void update (Graphics.g) {

2.// se comprueba si existe el objeto invisible y si sus dimensiones son

3.//correctas

4.if        ((graphInv==null)        ||        (d.width!=dimInv.width)        ||
    (d.height!=dimInv.height)) {

5.dimInv = d;

6.// se llama al método createImage de la clase Component

7.imgInv = createImage(d.width, d.height);

8.// se llama al método getGraphics de la clase Image

9.graphInv = imgInv.getGraphics();

10.}

11.// se establecen las propiedades del contexto gráfico invisible,

12.// y se dibuja sobre él

13.graphInv.setColor(getBackground());

14....

15.// finalmente se hace visible la imagen invisible a partir del punto (0, 0)

16.// utilizando el propio panel como ImageObserver

17.g.drawImage(imgInv, 0, 0, this);

18.} // fin del método update()
```

Los gráficos y las animaciones son particularmente útiles en las applets. El ***Tutorial*** de ***Sun*** tiene un ejemplo (un ***applet***) completamente explicado y desarrollado sobre las animaciones y los distintos métodos de eliminar elparpadeo.

7.7 APPLETS

Un ***applet*** es una miniaplicación, escrita en ***Java***, que se ejecuta en un navegador ***(Netscape Navigator***, ?) al cargar una página HTML que incluye información sobre el ***applet*** a ejecutar por medio de las ***etiquetas*** <APPLET>... </APPLET>.

A continuación se detallan algunas características de las ***applets***:

1. Los archivos de ***Java*** compilados (*.class) se descargan a través de la red desde un servidor de ***Web*** o servidor ***HTTP*** hasta el navegador en cuya *Java Virtual Machine* se ejecutan. Pueden incluir también archivos de imágenes y sonido.

2. Las ***applets*** no tienen ventana propia: se ejecutan en la ventana del navegador (en un ***?panel***?).

3. Por la propia naturaleza ?abierta? de Internet, las ***applets*** tienen importantes restricciones de seguridad, que se comprueban al llegar al navegador: sólo pueden leer y escribir archivos en el servidor del que han venido, sólo pueden acceder a una limitada información sobre la computadora en el que se están ejecutando, etc. Con ciertas condiciones, las ***applets*** ?de confianza? ***(trusted applets)*** pueden pasar por encima de estas restricciones.

Aunque su entorno de ejecución es un navegador, las ***applets*** se pueden probar sin necesidad de navegador con la aplicación ***appletviewer*** del JDK de ***Sun***.

7.7.1 Algunas características de las applets

Las características de las ***applets*** se pueden considerar desde el punto de vista del programador y desde el del usuario. En este manual lo más importante es el punto de vista del programador:

- Las ***applets*** no tienen un método ***main()*** con el que comience la ejecución. El papel central de su ejecución lo asumen otros métodos que se verán posteriormente.

- Todas las ***applets*** derivan de la clase ***java.applet.Applet***. La Figura 7.36 muestra la jerarquía de clases de la que deriva la clase ***Applet***. Las ***applets*** deben redefinir ciertos métodos heredados de ***Applet*** que controlan su ejecución: ***init()***, ***start()***, ***stop()***, ***destroy()***.

- Se heredan otros muchos métodos de las superclases de ***Applet*** que tienen que ver con la generación de interfaces gráficas de usuario (AWT). Así, los métodos gráficos se heredan de ***Component***, mientras que la capacidad de añadir componentes de interface de usuario se hereda de ***Container*** y de ***Panel***.

- Las ***applets*** también suelen redefinir ciertos métodos gráficos: los más importantes son ***paint()*** y ***update()***, heredados de ***Component*** y de ***Container***; y ***repaint()*** heredado de ***Component***.

- Las ***applets*** disponen de métodos relacionados con la obtención de información, como por ejemplo: ***getAppletInfo()***, ***getAppletContext()***, ***getParameterInfo()***, ***getParameter()***, ***getCodeBase()***, ***getDocumentBase()***, e ***isActive()***.

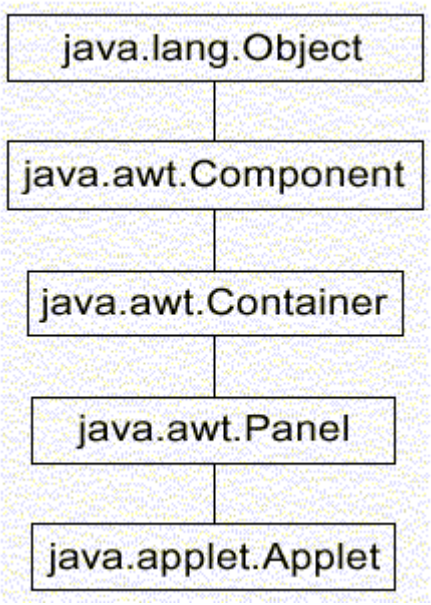


Figura 7.36. Jerarquía de clases de Applet.

El método ***showStatus()*** se utiliza para mostrar información en la barra de estado del navegador. Existen otros métodos relacionados con imágenes y sonido: ***getImage()***, ***getAudioClip()***, ***play()***, etc.

7.7.2 Métodos que controlan la ejecución de un applet

Los métodos que se estudian en este Apartado controlan la ejecución de las ***applet***s. De ordinario el programador tiene que redefinir uno o más de estos métodos, pero no tiene que preocuparse de llamarlos: el navegador se encarga de hacerlo.

7.7.2.1 Método init()

Se llama automáticamente al método ***init()*** en cuanto el navegador o visualizador carga el ***applet***. Este método se ocupa de todas las tareas de inicialización, realizando las funciones del ***constructor*** (al que el navegador no llama).

En ***Netscape Navigator*** se puede reinicializar un ***applet*** con ***Shift+Reload***.

7.7.2.2 Método start()

El método ***start()*** se llama automáticamente en cuanto el ***applet*** se hace visible, después de haber sido inicializada. Se llama también cada vez que el ***applet*** se hace de nuevo

visible después de haber estado oculta (por dejar de estar activa esa página del navegador, al cambiar el tamaño de la ventana del navegador, al hacer ***reload***, etc.).

Es habitual crear ***threads*** en este método para aquellas tareas que, por el tiempo que requieren, dejarían sin recursos al ***applet*** o incluso al navegador. Las animaciones y ciertas tareas a través de Internet son ejemplos de este tipo de tareas.

7.7.2.3 Método stop()

El método ***stop()*** se llama de forma automática al ocultar el ***applet*** (por haber dejado de estar activa la página del navegador, por hacer ***reload*** o ***resize***, etc.).

Con objeto de no consumir recursos inútilmente, en este método se suelen parar las ***threads*** que estén corriendo en el ***applet***, por ejemplo para mostrar animaciones.

7.7.2.4 Método destroy()

Se llama a este método cuando el ***applet*** va a ser descargada para liberar los recursos que tenga reservados (excepto la memoria). De ordinario no es necesario redefinir este método, pues el que se hereda cumple bien con esta misión.

7.7.3 Métodos para dibujar el applet

Las ***applets*** son aplicaciones gráficas que aparecen en una zona de la ventana del navegador. Por ello deben redefinir los métodos gráficos ***paint()*** y ***update()***. El método ***paint()*** se declara en la forma:

```
public void paint(Graphics g)
```

El objeto gráfico ***g*** pertenece a la clase ***java.awt.Graphics***, que siempre debe ser importada por el ***applet***. Este objeto define un contexto o estado gráfico para dibujar (métodos gráficos, colores, fonts, etc.) y es creado por el navegador.

Todo el trabajo gráfico del ***applet*** (dibujo de líneas, formas gráficas, texto, etc.) se debe incluir en el método ***paint()***, porque este método es llamado cuando el ***applet*** se dibuja por primera vez y también de forma automática cada vez que el ***applet*** se debe redibujar.

En general, el programador crea el método ***paint()*** pero no lo suele llamar. Para pedir explícitamente al sistema que vuelva a dibujar el ***applet*** (por ejemplo, por haber realizado algún cambio) se utiliza el método ***repaint()***, que es más fácil de usar, pues no requiere argumentos. El método ***repaint()*** se encarga de llamar a ***paint***() a través de ***update()***.

El método ***repaint()*** llama a ***update()***, que borra todo pintando de nuevo con el color de fondo y luego llama a ***paint()***. A veces esto produce parpadeo de pantalla. Existen dos formas de evitar el ***parpadeo***:

1.Redefinir ***update()*** de forma que no borre toda la ventana sino sólo lo necesario.

2.Redefinir ***paint()*** y ***update()*** para utilizar ***doble buffe***r.

7.8 CÓMO INCLUIR UN APPLET EN UNA PÁGINA HTML

Para llamar a un ***applet*** desde una página HTML se utiliza la etiqueta doble <APPLET>?</APPLET>, cuya forma general es (los elementos opcionales aparecen entre corchetes[]):

```
<APPLET CODE="miApplet.class" [CODEBASE="unURL"] [NAME="unName"]

WIDTH="wpixeles" HEIGHT="hpixeles"

[ALT="TextoAlternativo"]>

[texto alternativo para navegadores que reconocen la etiqueta <applet> pero no
pueden

ejecutar el applet]

[<PARAM NAME="MyName1" VALUE="valueOfMyName1">]

[<PARAM NAME="MyName2" VALUE="valueOfMyName2">]

</APPLET>
```

El atributo NAME permite dar un nombre opcional al ***applet***, con objeto de poder comunicarse con otras ***applets*** o con otros elementos que se estén ejecutando en la misma página. El atributo ARCHIVE permite indicar uno o varios archivos Jar o Zip (separados por comas) donde se deben buscar las clases.

A continuación se señalan otros posibles atributos de <APPLET>:

- ARCHIVE="archivo1, archivo2, archivo3". Se utiliza para especificar archivos JAR y ZIP.

- ALIGN, VSPACE, HSPACE. Tienen el mismo significado que la etiqueta IMG de HTML.

7.9 PASO DE PARÁMETROS A UN APPLET

Los etiquetas PARAM permiten pasar diversos parámetros desde el archivo HTML al programa ***Java*** del ***applet***, de una forma análoga a la que se utiliza para pasar argumentos a ***main()***.

Cada parámetro tiene un ***nombre*** y un ***valo***r. Ambos se dan en forma de ***Strin***g, aunque el valor sea numérico. El ***applet*** recupera estos parámetros y, si es necesario, convierte los ***Strings*** en valores numéricos. El valor de los parámetros se obtienen con el siguiente método de la clase ***Applet***:

```
String getParameter(String name)
```

La conversión de ***Strings*** a los tipos primitivos se puede hacer con los métodos asociados a los ***wrappers*** que ***Java*** proporciona para dichos tipo fundamentales (Integer.parseInt(String), Double.valueOf(String), ?).

En los ***nombres*** de los parámetros no se distingue entre mayúsculas y minúsculas, pero sí en los ***valore***s, ya que serán interpretados por un programa ***Jav***a, que sí distingue.

El programador del ***applet*** debería prever siempre unos ***valores por defecto*** para los parámetros del ***applet***, para el caso de que en la página HTML que llama al ***applet*** no se definan.

El método ***getParameterInfo()*** devuelve una matriz de ***Strings*** *(String[][]*) con información sobre cada uno de los parámetros soportados por el ***applet***: *nombre*, *tipo* y *descripción*, cada uno de ellos en un ***String***. Este método debe ser redefinido por el programador del ***applet*** y utilizado por la persona que prepara la página HTML que llama al ***applet***. En muchas ocasiones serán personas distintas, y ésta es una forma de que el programador del ***applet*** dé información al usuario.

7.10CARGA DE APPLETS

7.10.1Localización de archivos

Por defecto se supone que los archivos ****.class*** del ***applet*** están en el mismo directorio que el archivo HTML. Si el ***applet*** pertenece a un ***paquete***, el navegador utiliza el nombre del ***paquete*** para construir una ***trayectoria de directorio*** relativo al directorio donde está el HTML.

El atributo CODEBASE permite definir un URL para los archivos que contienen el código y demás elementos del ***applet***. Si el directorio definido por el URL de CODEBASE es *relativo*, se interpreta respecto al directorio donde está el HTML; si es *absoluto* se interpreta en sentido estricto y puede ser cualquier directorio de la Internet.

7.10.2Archivos JAR (Java Archives)

Si un ***applet*** consta de varias clases, cada archivo ****.class*** requiere una conexión con el servidor de Web (servidor de protocolo HTTP), lo cual puede requerir algunos segundos. En este caso es conveniente agrupar todos los archivos en un archivo único, que se puede comprimir y cargar con una sola conexión HTTP.

Los archivos JAR están basados en los archivos ZIP y pueden crearse con el programa ***jar*** que viene con el JDK. Por ejemplo:

```
jar cvf myArchivo.jar *.class *.gif
```

crea un archivo llamado ***myArchivo.jar*** que contiene todos los archivos ****.class*** y ****.gif*** del directorio actual. Si las clases pertenecieran a un paquete llamado ***es.ceit.infor2*** se utilizaría el comando:

```
jar cvf myArchivo.jar es\ceit\infor2\*.class *.gif
```

7.11COMUNICACIÓN DEL APPLET CON EL NAVEGADOR

La comunicación entre el applet y el navegador en el que se está ejecutando se puede controlar mediante la interface ***AppletContext*** (paquete ***java.applet***). ***AppletContext*** es una interface implementada por el navegador, cuyos métodos pueden ser utilizados por el ***applet*** para obtener información y realizar ciertas operaciones, como por ejemplo sacar ***mensajes breves*** en la ***barra de estado*** del navegador. Hay que tener en cuenta que la barra de estado es compartida por el navegador y las ***applet***s, lo que tiene el peligro de que el mensaje sea rápidamente sobrescrito por el navegador u otras ***applets*** y que el usuario no llegue a enterarse del mensaje.

Los mensajes breves a la barra de estado se producen con el método ***showStatus()***, como por ejemplo,

```
getAppletContext().showStatus("Cargado desde el archivo " + filename);
```

Los mensajes más importantes se deben dirigir a la ***salida estándar*** o a la ***salida de error***es, que en ***Netscape Navigator*** es la ***Java Console*** (la cual se hace visible desde el menú ***Options*** en ***Navigator 3.***0, desde el menú ***Communicator*** en ***Navigator 4.0**** y desde ***Communicator/Tools*** en ***Navigator 4.***5). Estos mensajes se pueden enviar con las sentencias:

```
System.out.print();

System.out.println();

System.error.print();

System.error.println();
```

Para mostrar documentos HTML en una ventana del navegador se pueden utilizar los métodos siguientes:

- ***showDocument(URL miUrl, [String target]***), que muestra un documento HTML en el *frame* del navegador indicado por *target (nam*e, *_to*p, *_paren*t, *_blan*k, *_sel*f).

- ***showDocument(URL miUrl***), que muestra un documento HTML en la ventana actual del navegador.

Un ***applet*** puede conseguir información de otras ***applets*** que están corriendo en la misma página del navegador, enviarles mensajes y ejecutar sus métodos. El mensaje se envía invocando los métodos del otro ***applet*** con los argumentos apropiados.

Algunos navegadores exigen, para que las ***applets*** se puedan comunicar, que las ***applets*** provengan del mismo navegador o incluso del mismo directorio (que tengan el mismo *codebas*e). Por ejemplo, para obtener información de otras applets se pueden utilizar los métodos:

- ***getApplet(String name***), que devuelve el ***applet*** llamada ***name*** (o ***null*** si no la encuentra). El nombre del ***applet*** se pone con el atributo opcional NAME o con el parámetro NAME.

- ***getApplets()***, que devuelve una *enumeración* con todas las ***applets*** de la página.

Para poder utilizar todos los métodos de un applet que se está ejecutando en la misma página HTML (y no sólo los métodos comunes heredados de ***Applet***), debe hacerse un ***cast*** del objeto de la clase ***Applet*** que se obtiene como valor de retorno de ***getApplet()*** a la clase concreta del ***applet***.

Para que pueda haber ***respuesta*** (es decir, comunicación en los dos sentidos), el primer applet que envía un mensaje debe enviar una referencia a sí misma por medio del argumento ***this***.

7.12SONIDOS EN APPLETS

La clase ***Applet*** y la interface ***AudioClips*** permiten utilizar sonidos en applets. La Tabla 7.36 muestra algunos métodos interesantes al respecto.

Métodos de Applet	**Función que realizan**
public AudioClip getAudioClip(URL url)	Devuelve el objeto especificado por url, que implementa la interface AudioClip
public AudioClip getAudioClip(URL url, String name)	Devuelve el objeto especificado por url (dirección base) y name (dirección relativa)
play(URL url), play(URL url, String name)	Hace que suene el AudioClip correspondiente a la dirección especificada
Métodos de la interface AudioClip (en java.applet)	**Función que realizan**
void loop()	Ejecuta el sonido repetidamente
void play()	Ejecuta el sonido una sola vez
void stop()	Detiene el sonido

Tabla 7.36. Métodos de Applet y de AudioClip relacionados con sonidos.

Respecto a la carga de sonidos, por lo general es mejor cargar los sonidos en un ***thread*** distinto (creado en el método ***init()***) que en el propio método ***init()***, que tardaría en devolver el control y permitir al usuario empezar a interaccionar con el ***applet***.

Si el sonido no ha terminado de cargarse (en el ***thread*** especial para ello) y el usuario interacciona con el ***applet*** para ejecutarlo, el ***applet*** puede darle un aviso de que no se ha terminado de cargar.

7.13IMÁGENES EN APPLETS

Las ***applets*** admiten los formatos JPEG y GIF para representar imágenes a partir de archivos localizados en el servidor. Estas imágenes se pueden cargar con el método ***getImage()*** de la clase ***Applet***, que puede tener las formas siguientes:

```
public Image getImage(URL url)

public Image getImage(URL url, String name)
```

Estos métodos devuelven el control inmediatamente. Las imágenes de cargan cuando se da la orden de dibujar las imágenes en la pantalla. El dibujo se realiza entonces de forma incremental, a medida que el contenido va llegando.

Para dibujar imágenes se utiliza el método ***drawImage()*** de la clase ***Graphics***, que tiene las formas siguientes:

```
public abstract boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)

public abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)
```

El primero de ellos dibuja la imagen con su tamaño natural, mientras que el segundo realiza un cambio en la escala de la imagen.

Los métodos ***drawImage()*** van dibujando la parte de la imagen que ha llegado, con su tamaño, a partir de las coordenadas (x, y) indicadas, utilizando ***bgcolor*** para los pixeles transparentes.

Estos métodos devuelven el control inmediatamente, aunque la imagen no esté del todo cargada. En este caso devuelve ***false***. En cuanto se carga una parte adicional de la imagen, el proceso que realiza el dibujo avisa al ***ImageObserver*** especificado. ***ImageObserver*** es una interface implementada por ***Applet*** que permite seguir el proceso de carga de una imagen.

7.14 OBTENCIÓN DE LAS PROPIEDADES DEL SISTEMA

Un ***applet*** puede obtener información del sistema o del entorno en el que se ejecuta. Sólo algunas propiedades del sistema son accesibles. Para acceder a las propiedades del sistema se utiliza un método ***static*** de la clase ***System***:

```
String salida = System.getProperty("file.separator");
```

Los nombres y significados de las propiedades del sistema accesibles son las siguientes:

"file.separator" Separador de directorios (por ejemplo, "/" o "\")

"java.class.version" Número de versión de las clases de ***Java***

"java.vendor" Nombre específico del vendedor de Java

"java.vendor.url" URL del vendedor de Java

"java.version" Número de versión Java

"line.separator" Separador de líneas

"os.arch" Arquitectura del sistema operativo

"os.name" Nombre del sistema operativo

"path.separator" Separador en la variable Trayectoria (por ejemplo, ":"

No se puede acceder a las siguientes propiedades del sistema:

"java.class.path"

"java.home"

"user.dir"

"user.home"

"user.name".

7.15UTILIZACIÓN DE THREADS EN APPLETS

Un ***applet*** puede ejecutarse con varios ***thread***s, y en muchas ocasiones será necesario o conveniente hacerlo así. Hay que tener en cuenta que un ***applet*** se ejecuta siempre en un navegador (o en la aplicación ***appletviewe***r).

Así, los ***threads*** en las que se ejecutan los métodos mayores ***init()***, ***start()***, ***stop()*** y ***destroy()*** dependen del navegador o del entorno de ejecución. Los métodos gráficos ***paint()***, ***update()*** y ***repaint()*** se ejecutan siempre desde una ***thread*** especial del AWT.

Algunos navegadores dedican un ***thread*** para cada ***applet*** en una misma página; otros crean un grupo de ***threads*** para cada ***applet*** (para poderlas matar al mismo tiempo, por ejemplo). En cualquier caso se garantiza que todas las ***threads*** creadas por los métodos mayores pertenecen al mismo grupo.

Se deben introducir ***threads*** en ***applets*** siempre que haya tareas que consuman mucho tiempo (cargar una imagen o un sonido, hacer una conexión a Internet, ?). Si estas tareas pesadas se ponen en el método ***init()*** bloquean cualquier actividad del ***applet*** o incluso de la página HTML hasta que se completan. Las tareas pesadas pueden ser de dos tipos:

- Las que sólo se hacen una vez.

- Las que se repiten muchas veces.

Un ejemplo de tarea que se repite muchas veces puede ser una animación. En este caso, la tarea repetitiva se pone dentro de un ciclo ***while*** o ***do?while***, dentro del ***thread***. El ***thread*** se debería crear dentro del método ***start()*** del ***applet*** y destruirse en ***stop()***. De este modo, cuando el ***applet*** no está visible se dejan de consumir recursos.

Al crear el ***thread*** en el método ***start()*** se pasa una referencia al ***applet*** con la palabra ***this***, que se refiere al ***applet***. El applet deberá implementar la interface ***Runnable***, y por tanto debe definir el método ***run()***, que es el centro del ***Thread***

Un ejemplo de tarea que se realiza una sola vez es la carga de imágenes ****.gif*** o ****.jpeg***, que ya se realiza automáticamente en un ***thread*** especial.

Sin embargo, los sonidos no se cargan en ***threads*** especiales de forma automática; los debe crear el programador para cargarlos en ?background?. Este es un caso típico de programa ***productor consumidor***: el ***thread*** es el *productor* y el ***applet*** el *consumidor*. Los ***threads*** deben estar ***sincronizados***, para lo que se utilizan los métodos ***wait()*** y ***notifyAll()***.

A continuación se presenta un ejemplo de ***thread*** con tarea repetitiva:

```
1.public void start() {

2.if (repetitiveThread == null) {

3.repetitiveThread = new Thread(this); // se crea un nuevo thread

4.}

5.repetitiveThread.start(); // se arranca el thread creado: start() llama a
   run()

6.}

7.public void stop() {

8.repetitiveThread = null; // para parar la ejecución del thread

9.}

10.public void run() {

11....

12.while (Thread.currentThread() == repetitiveThread) {

13.... // realizar la tarea repetitiva.

14.}

15.}
```

El método ***run()*** se detendrá en cuanto se ejecute el método ***stop()***, porque la referencia al ***thread*** está a ***null***.

7.16APPLETS QUE TAMBIÉN SON APLICACIONES

Es muy interesante desarrollar ***aplicaciones*** que pueden funcionar también como ***applets*** y viceversa. En concreto, para hacer que un ***applet*** pueda ejecutarse como ***aplicación*** pueden seguirse las siguientes instrucciones:

1.Se añade un método ***main()*** a la clase ***MiApplet*** (que deriva de ***Apple***t)

2.El método ***main()*** debe crear un objeto de la clase ***MiApplet*** e introducirlo en un ***Fram***e.

3.El método ***main()*** debe también ocuparse de hacer lo que haría el navegador, es decir, llamar a los métodos ***init()*** y ***start()*** de la clase ***MiApple***t.

4.Se puede añadir también una ***static inner class*** que derive de ***WindowAdapter*** y que gestione el evento de cerrar la ventana de la aplicación definiendo el método ***windowClosing(***). Este método llama al método ***System.exit(0***). Según como sea el ***apple***t, el método ***windowClosing()*** previamente deberá también llamar a los métodos ***MiApplet.stop()*** y ***MiApplet.destroy(***), cosa que para las ***applets*** se encarga de hacer el navegador. En este caso conviene que el objeto de ***MiApplet*** creado por ***main()*** sea ***stati***c, en lugar de una variable local.

A continuación se presenta un ejemplo:

```
1.public class MiApplet extends Applet {

2....

3.public void init() {

4....

5.}

6....

7.// clase para cerrar la aplicación

8.static class WL extends WindowsAdapter {

9.public void windowClosing(WindowEvent e) {

10.MiApplet.stop();

11.MiApplet.destroy();

12.System.exit(0);

13.}

14.} // fin de WindowAdapter
```

```
15.// programa principal

16.public static void main(String[] args) {

17.static MiApplet unApplet = new MiApplet();

18.Frame unFrame = new Frame("MiApplet");

19.unFrame.addWindowListener(new WL());

20.unFrame.add(unapplet, BorderLayout.CENTER);

21.unFrame.setSize(400,400);

22.unApplet.init();

23.unApplet.start();

24.unFrame.setVisible(true);

25.}

26.} // fin de la clase MiApplet
```

7.17UN EJEMPLO COMPLETO COMENTADO

Este ejemplo contiene algunas de las características más importantes de ***Jav***a: *clase*s, *herenci*a, *interface*s, *gráfico*s, *polimorfism*o, etc. Las sentencias se numeran en cada archivo, de modo que resulta más fácil hacer referencia a ellas en los comentarios. La ejecución de este programa imprime algunas líneas en la consola MS-DOS y conduce a crear la ventana mostrada en la **Figura 7.37**.

7.17.1Clase Ejemplo1

A continuación se muestra el programa principal, contenido en el archivo ***Ejemplo1.jav***a. En realidad, este programa principal lo único que hace es utilizar la clase ***Geometría*** y sus clases derivadas. Es pues un programa puramente "usuario", a pesar de lo cual hay que definirlo dentro de una clase, como todos los programas en ***Jav***a.

```
1.// archivo Ejemplo1.java

2.import java.util.ArrayList;

3.import java.awt.*;

4.class Ejemplo1 {

5.public static void main(String arg[]) throws InterruptedException

6.{

7.System.out.println("Comienza main()...");

8.Circulo c = new Circulo(2.0, 2.0, 4.0);
```

```
9.System.out.println("Radio = " + c.r + " unidades.");

10.System.out.println("Centro = (" + c.x + "," + c.y + ")
   unidades.");

11.Circulo c1 = new Circulo(1.0, 1.0, 2.0);

12.Circulo c2 = new Circulo(0.0, 0.0, 3.0);

13.c = c1.elMayor(c2);

14.System.out.println("El mayor radio es " + c.r + ".");

15.c = new Circulo(); // c.r = 0.0;

16.c = Circulo.elMayor(c1, c2);

17.System.out.println("El mayor radio es " + c.r + ".");

18.VentanaCerrable ventana =

19.new VentanaCerrable("Ventana abierta al mundo...");

20.ArrayList v = new ArrayList();

21.CirculoGrafico cg1 = new CirculoGrafico(200, 200, 100,
   Color.red);

22.CirculoGrafico cg2 = new CirculoGrafico(300, 200, 100,
   Color.blue);

23.RectanguloGrafico rg = new

24.RectanguloGrafico(50, 50, 450, 350, Color.green);

25.v.add(cg1);

26.v.add(cg2);

27.v.add(rg);

28.PanelDibujo mipanel = new PanelDibujo(v);

29.ventana.add(mipanel);

30.ventana.setSize(500, 400);

31.ventana.setVisible(true);

32.System.out.println("Termina main()...");

33.} // fin de main()

34.} // fin de class Ejemplo1
```

La sentencia 1 es simplemente un comentario que contiene el nombre del archivo. El compilador de Java ignora todo lo que va desde los caracteres // hasta el final de la línea.

Las sentencias 2 y 3 "importan" clases de los paquetes de Java, esto es, hacen posible acceder a dichas clases utilizando nombres cortos. Por ejemplo, se puede acceder a la clase ArrayList simplemente con el nombre ArrayList en lugar de con el nombre completo java.util.ArrayList, por haber introducido la sentencia import de la línea 2. Un paquete es una agrupación de clases que tienen una finalidad relacionada. Existe una jerarquía de paquetes que se refleja en nombres compuestos, separados por un punto (.). Es habitual nombrar los paquetes con letras minúsculas (como java.util o java.awt), mientras que los nombres de las clases suelen empezar siempre por una letra mayúscula (como ArrayList). El asterisco (*) de la sentencia 3 indica que se importan todas las clases del paquete. Hay un paquete, llamado java.lang, que se importa siempre automáticamente. Las clases de java.lang se pueden utilizar directamente, sin importar el paquete.

La sentencia 4 indica que se comienza a definir la clase Ejemplo1. La definición de dicha clase va entre llaves {}. Como también hay otras construcciones que van entre llaves, es habitual indentar o sangrar el código, de forma que quede claro donde empieza (línea 4) y donde termina (línea 34) la definición de la clase.En Java todo son clases: no se puede definir una variable o una función que no pertenezca a una clase. En este caso, la clase Ejemplo1 tiene como única finalidad acoger al método main(), que es el programa principal del ejemplo. Las clases utilizadas por main() son mucho más importantes que la propia clase Ejemplo1. Se puede adelantar ya que una clase es una agrupación de variables miembro (datos) y funciones miembro (métodos) que operan sobre dichos datos y permiten comunicarse con otras clases. Las clases son verdaderos tipos de variables o datos, creados por el usuario. Un objeto (en ocasiones también llamado instancia) es una variable concreta de una clase, con su propia copia de las variables miembro.

Las líneas 5-33 contienen la definición del programa principal de la aplicación, que en Java siempre se llama main(). La ejecución siempre comienza por el programa o método main(). La palabra public indica que esta función puede ser utilizada por cualquier clase; la palabra static indica que es un método de clase, es decir, un método que puede ser utilizado aunque no se haya creado ningún objeto de la clase Ejemplo1 (que de hecho, no se han creado); la palabra void indica que este método no tiene valor de retorno. A continuación del nombre aparecen, entre paréntesis, los argumentos del método. En el caso de main() el argumento es siempre un vector o array (se sabe por la presencia de los corchetes []), en este caso llamado arg, de cadenas de caracteres (objetos de la clase String). Estos argumentos suelen ser parámetros que se pasan al programa en el momento de comenzar la ejecución (por ejemplo, el nombre del archivo donde están los datos).

El cuerpo (body) del método main(), definido en las líneas 6-33, va también encerrado entre llaves {...}. A un conjunto de sentencias encerrado entre llaves se le suele llamar bloque. Es conveniente indentar para saber dónde empieza y dónde terminan los bloques del método main() y de la clase Ejemplo1. Los bloques nunca pueden estar entrecruzados; un bloque puede contener a otro, pero nunca se puede cerrar el bloque exterior antes de haber cerrado el interior.

La sentencia 7 (System.out.println("Comienza main()...");) imprime una cadena de caracteres o String en la salida estándar del sistema, que normalmente será una ventana de MS-DOS o una ventana especial del entorno de programación que se utilice (por

ejemplo JBuilder, de Inprise Corporation). Para ello se utiliza el método println(), que está asociado con una variable static llamada out, perteneciente a la clase System (en el paquete por defecto, java.lang). Una variable miembro static, también llamada variable de clase, es una variable miembro que es única para toda la clase y que existe aunque no se haya creado ningún objeto de la clase. La variable out es una variable static de la clase System. La sentencia 7, al igual que las que siguen, termina con el carácter punto y coma (;).

La sentencia 8 (Circulo c = new Circulo(2.0, 2.0, 4.0);) es muy propia de Java. En ella se crea un objeto de la clase Circulo. Esta sentencia es equivalente a las dos sentencias siguientes:

```
Circulo c;

c = new Circulo(2.0, 2.0, 4.0);
```

que quizás son más fáciles de explicar. En primer lugar se crea una referencia llamada c a un objeto de la clase Circulo. Crear una referencia es como crear un "nombre" válido para referirse a un objeto de la clase Circulo. A continuación, con el operador new se crea el objeto propiamente dicho. Puede verse que el nombre de la clase va seguido por tres argumentos entre paréntesis. Estos argumentos se le pasan al constructor de la clase como datos concretos para crear el objeto (en este caso los argumentos son las dos coordenadas del centro y el radio).

Interesa ahora insistir un poco más en la diferencia entre clase y objeto. La clase Circulo es lo genérico: es el patrón o modelo para crear círculos concretos. El objeto c es un círculo concreto, con su centro y su radio. De la clase Circulo se pueden crear tantos objetos como se desee; la clase dice que cada objeto necesita tres datos (las dos coordenadas del centro y el radio) que son las variables miembro de la clase. Cada objeto tiene sus propias copias de las variables miembro, con sus propios valores, distintos de los demás objetos de la clase.

La sentencia 9 (System.out.println("Radio = " + c.r + " unidades.");) imprime por la salida estándar una cadena de texto que contiene el valor del radio. Esta cadena de texto se compone de tres subcadenas, unidas mediante el operador de concatenación (+). Obsérvese cómo se accede al radio del objeto c: el nombre del objeto seguido del nombre de la variable miembro r, unidos por el operador punto (c.r). El valor numérico del radio se convierte automáticamente en cadena de caracteres.

La sentencia 10 es similar a la 9, imprimiendo las coordenadas del centro del círculo.

Las sentencias 11 y 12 crean dos nuevos objetos de la clase Circulo, llamados c1 y c2.

La sentencia 13 (c = c1.elMayor(c2);) utiliza el método elMayor() de la clase Circulo. Este método compara los radios de dos círculos y devuelve como valor de retorno una referencia al círculo que tenga mayor radio. Esa referencia se almacena en la referencia previamente creada c. Un punto importante es que todos los métodos de Java (excepto los métodos de clase o static) se aplican a un objeto de la clase por medio del operador punto (por ejemplo, c1.elMayor()). El otro objeto (c2) se pasa como argumento entre paréntesis. Obsérvese la forma "asimétrica" en la que se pasan los dos argumentos al

método elMayor(). De ordinario se llama ***argumento implícito*** a ***c***1, mientras que ***c2*** sería el ***argumento explícito*** del método.

La sentencia 14 imprime el resultado de la comparación anterior y la sentencia 15 crea un nuevo objeto de la clase Circulo guardándolo en la referencia c. En este caso no se pasan argumentos al constructor de la clase. Eso quiere decir que deberá utilizar algunos valores "por defecto" para el centro y el radio. Esta sentencia anula o borra el resultado de la primera comparación de radios, de modo que se pueda comprobar el resultado de la segunda comparación.

La sentencia 16 (c = Circulo.elMayor(c1, c2);) vuelve a utilizar un método llamado elMayor() para comparar dos círculos: ¿Se trata del mismo método de la sentencia 13, utilizado de otra forma? No. Se trata de un método diferente, aunque tenga el mismo nombre. A las funciones o métodos que son diferentes porque tienen distinto código, aunque tengan el mismo nombre, se les llama funciones sobrecargadas (overloaded). Las funciones sobrecargadas se diferencian por el numero y tipo de sus argumentos. El método de la sentencia 13 tiene un único argumento, mientras que el de la sentencia 16 tiene dos (en todos los casos objetos de la clase Circulo). En realidad, el método de la sentencia 16 es un método static (o método de clase), esto es, un método que no necesita ningún objeto como argumento implícito. Los métodos static suelen ir precedidos por el nombre de la clase y el operador punto (Java también permite que vayan precedidos por el nombre de cualquier objeto, pero es considerada una nomenclatura más confusa.). La sentencia 16 es absolutamente equivalente a la sentencia 13, pero el método static de la sentencia 16 es más "simétrico". Las sentencias 17 y 18 no requieren ya comentarios especiales.

Las sentencias 18-31 tienen que ver con la parte gráfica del ejemplo.

En las líneas 18-19 (VentanaCerrable ventana = new VentanaCerrable("Ventana abierta al mundo...");) se crea una ventana para dibujar sobre ella. Una ventana es un objeto de la clase Frame, del paquete java.awt. La clase VentanaCerrable, añade a la clase Frame la capacidad de responder a los eventos que provocan el cierre de una ventana. La cadena que se le pasa como argumento es el título que aparecerá en la ventana (ver Figura 3.1). En la sentencia 20 (ArrayList v = new ArrayList();) Se crea un objeto de la clase ArrayList (contenida o definida en el paquete java.util). La clase ArrayList permite almacenar referencias a objetos de distintas clases. En este caso se utilizará para almacenar referencias a varias figuras geométricas diferentes.

Las siguientes sentencias 21-27 crean elementos gráficos y los incluyen en la lista v para ser dibujados más tarde en el objeto de la clase PanelDibujo. Los objetos de la clase Circulo creados anteriormente no eran objetos aptos para ser dibujados, pues sólo tenían información del centro y el radio, y no del color de línea. Las clases RectanguloGrafico y CirculoGrafico, derivan respectivamente de las clases Rectangulo y Circulo, heredando de dichas clases sus variables miembro y métodos, añadiendo la información y los métodos necesarios para poder dibujarlos en la pantalla. En las sentencias 21-22 se definen dos objetos de la clase CirculoGrafico; a las coordenadas del centro y al radio se une el color de la línea. En la sentencia 23-24 se define un objeto de la clase ***RectanguloGrafic***o, especificando asimismo un color, además de las coordenadas del vértice superior izquierdo, y del vértice inferior derecho. En las

sentencias 25-27 los objetos gráficos creados se añaden al objeto ***v*** de la clase ***ArrayLis***t, utilizando el método ***add()*** de la propia clase ***ArrayLis***t.

En la sentencia 28 (PanelDibujo mipanel = new PanelDibujo(v);) se crea un objeto de laclase PanelDibujo, definida en el Apartado 1.3.8. Por decirlo de alguna manera, los objetos de dicha clase son paneles, esto es superficies en las que se puede dibujar. Al constructor de PanelDibujo se le pasa como argumento el vector v con las referencias a los objetos a dibujar. La sentencia 29 (ventana.add(mipanel);) añade o incluye el panel (la superficie de dibujo) en la ventana; la sentencia 30 (ventana.setSize(500, 400);) establece el tamaño de la ventana en pixeles; finalmente, la sentencia 31 (ventana.setVisible(true);) hace visible la ventana creada.

¿Cómo se consigue que se dibuje todo esto? La clave está en la serie de órdenes que se han ido dando a la computadora. La clase PanelDibujo deriva de la clase Container a través de Panel, y redefine el método ***paint()*** de ***Containe***r. En este método, explicado en el Apartado 1.3.8, se realiza el dibujo de los objetos gráficos creados. El usuario no tiene que preocuparse de llamar al método ***paint()***, pues se llama de modo automático cada vez que el sistema operativo tiene alguna razón para ello (por ejemplo cuando se crea la ventana, cuando se mueve, cuando se minimiza o maximiza, cuando aparece después de haber estado oculta, etc.). La Figura 3.1 muestra la ventana resultante de la ejecución del programa ***main()*** de la clase ***Ejemplo***1. Para entender más a fondo este resultado es necesario considerar detenidamente las clases definidas posteriormente.

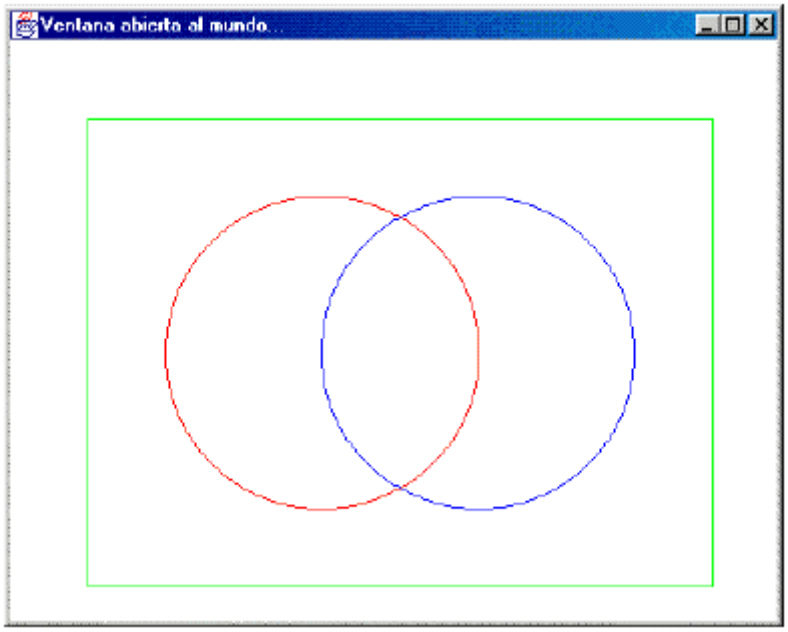


Figura 7.37. Resultado de la ejecución del Ejemplo1.

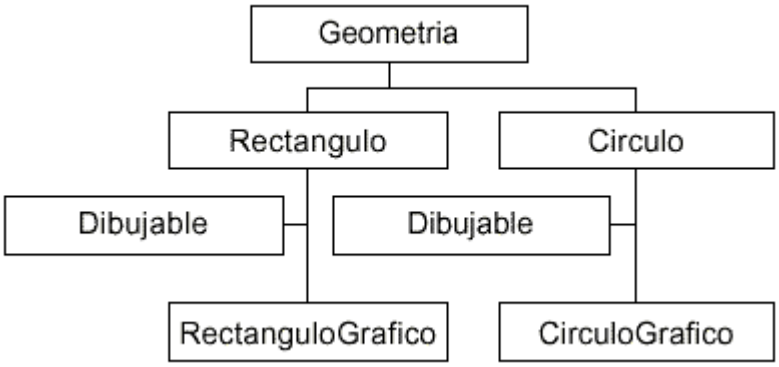


Figura 7.38. Jerarquía de clases utilizadas.

7.17.2Clase Geometria

En este apartado se describe la clase más importante de esta aplicación. Es la más importante no en el sentido de lo que hace, sino en el de que las demás clases "derivan" de ella, o por decirlo de otra forma, se apoyan o cuelgan de ella. La Figura 7.38 muestra la jerarquía de clases utilizada en este ejemplo. La clase Geometria es la base de la jerarquía. En realidad no es la base, pues en Java la clase base es siempre la clase Object. Siempre que no se diga explícitamente que una clase deriva de otra, deriva implícitamente de la clase Object (definida en el paquete java.lang). De las clases Rectangulo y Circulo derivan respectivamente las clases RectanguloGrafico y CirculoGrafico. En ambos casos está por en medio un elemento un poco especial donde aparece la palabra Dibujable. En términos de Java, Dibujable es una interface. Más adelante se verá qué es una interface.

Se suele utilizar la nomenclatura de superclase y subclase para referirse a la clase padre o hija de una clase determinada. Así Geometría es una superclase de Circulo, mientras que CirculoGrafico es una subclase.

En este ejemplo sólo se van a dibujar rectángulos y círculos. De la clase Geometría van a derivar las clases Rectangulo y Circulo. Estas clases tienen en común que son "geometrías", y como tales tendrán ciertas características comunes como un perímetro y un área. Un aspecto importante a considerar es que no va a haber nunca objetos de la clase Geometria, es decir "geometrías a secas". Una clase de la que no va a haber objetos es una clase abstracta, y como tal puede ser declarada. A continuación se muestra el archivo Geometria.java en el que se define dicha clase:

```
1.// archivo Geometria.java

2.public abstract class Geometria {

3.// clase abstracta que no puede tener objetos

4.public abstract double perimetro();

5.public abstract double area();

6.}
```

La clase Geometria se declara como public para permitir que sea utilizada por cualquier otra clase, y como abstract para indicar que no se permite crear objetos de esta clase. Es característico de las clases tener variables y funciones miembro. La clase Geometria no define alguna variable miembro, pero sí declara dos métodos: perímetro() y area(). Ambos métodos se declaran como public para que puedan ser llamados por otras clases y como abstract para indicar que no se da alguna definición; es decir algún código, para ellos. Interesa entender la diferencia entre declaración (la primera línea o header del método) y definición (todo el código del método, incluyendo la primera línea). Se indica también que su valor de retorno; el resultado, va a ser un double y que no tienen argumentos (obtendrán sus datos a partir del objeto que se les pase como argumento implícito). Es completamente lógico que no se definan en esta clase los métodos perimetro() y area(): la forma de calcular un perímetro o un área es completamente distinta en un rectángulo y en un círculo, y por tanto estos métodos habrá que definirlos en las clases Rectangulo y Circulo. En la clase Geometria lo único que se puede decir es cómo serán dichos métodos, es decir su nombre, el número y tipo de sus argumentos y el tipo de su valor de retorno.

7.17.3Clase Rectangulo

Según el diagrama de clases de la Figura 7.38 la clase Rectangulo deriva de Geometria. Esto se indica en la sentencia 2 con la palabra extends (en negrita en el listado de la clase).

```
1.// archivo Rectangulo.java

2.public class Rectangulo extends Geometria {

3.// definición de variables miembro de la clase

4.private static int numRectangulos = 0;

5.protected double x1, y1, x2, y2;

6.// constructores de la clase

7.public Rectangulo(double p1x, double p1y, double p2x, double p2y)
   {

8.x1 = p1x;

9.x2 = p2x;

10.y1 = p1y;

11.y2 = p2y;

12.numRectangulos++;

13.}

14.public Rectangulo(){ this(0, 0, 1.0, 1.0); }

15.// definición de métodos

16.public double perimetro() { return 2.0 * ((x1-x2)+(y1-y2)); }
```

```
17.public double area() { return (x1-x2)*(y1-y2); }

18.} // fin de la clase Rectangulo
```

La clase Rectangulo define cinco variables miembro. En la sentencia 4 (private static int numRectangulos = 0;) se define una variable miembro static. Las variables miembro static se caracterizan por ser propias de la clase y no de cada objeto. En efecto, la variable numRectangulos pretende llevar cuenta en todo momento del número de objetos de esta clase que se han creado. No tiene sentido ni sería práctico en absoluto que cada objeto tuviera su propia copia de esta variable, teniendo además que actualizarla cada vez que se crea o se destruye un nuevo rectángulo. De la variable numRectangulos, que en la sentencia 4 se inicializa a cero, se mantiene una única copia para toda la clase. Además esta variable es privada (private), lo cual quiere decir que sólo las funciones miembro de esta clase tienen permiso para utilizarla.

La sentencia 5 (protected double x1, y1, x2, y2;) define cuatro nuevas variables miembro, que representan las coordenadas de dos vértices opuestos del rectángulo. Las cuatro son de tipo double. El declararlas como protected indica que sólo esta clase, las clases que deriven de ella y las clases del propio paquete tienen permiso para utilizarlas.

Las sentencias 7-14 definen los constructores de la clase. Los constructores son unos métodos o funciones miembro muy importantes. Como se puede ver, no tienen ***valor de retorno*** (ni siquiera ***voi***d) y ***su nombre coincide con el de la clas***e. Los constructores son un ejemplo típico de ***métodos sobrecargados*** (*overloade*d): en este caso hay dos constructores, el segundo de los cuales no tiene ningún argumento, por lo que se llama ***constructor por defect***o. Las sentencias 7-13 definen el ***constructor genera***l. Este constructor recibe cuatro argumentos con cuatro valores que asigna a las cuatrovariables miembro. La sentencia 12 incrementa en una unidad (esto es lo que hace el operador ++, típico de C y C++, de los que ***Java*** lo ha heredado) el número de rectángulos creados hasta el momento.

La sentencia 14 (public Rectangulo(){ this(0, 0, 1.0, 1.0); }) define un segundo constructor, que por no necesitar argumentos es un constructor por defecto. ¿Qué se puede hacer cuando hay que crear un rectángulo sin ningún dato? Pues algo realmente sencillo: en este caso se ha optado por crear un rectángulo de lado unidad cuyo primer vértice coincide con el origen de coordenadas. Obsérvese que este constructor en realidad no tiene código para inicializar las variables miembro, limitándose a llamar al constructor general previamente creado, utilizando para ello la palabra this seguida del valor por defecto de los argumentos. Ya se verá que la palabra this tiene otro uso aún más importante en Java.

Las sentencias 16 (public double perimetro() { return 2.0 * ((x1-x2)+(y1-y2)); }) y 17 (public double area() { return (x1-x2)*(y1-y2); }) contienen la definición de los métodos miembro perimetro() y area(). La declaración coincide con la de la clase Geometría, pero aquí va seguida del cuerpo del método entre llaves {...}. Las fórmulas utilizadas son las propias de un rectángulo.

7.17.4Clase Circulo

A continuación se presenta la definición de la clase Circulo, también derivada de Geometria, y que resulta bastante similar en muchos aspectos a la clase Rectangulo. Por eso, en este caso las explicaciones serán un poco más breves, excepto cuando aparezcan cosas nuevas.

```
1.// archivo Circulo.java

2.public class Circulo extends Geometria {

3.static int numCirculos = 0;

4.public static final double PI=3.14159265358979323846;

5.public double x, y, r;

6.public Circulo(double x, double y, double r) {

7.this.x=x; this.y=y; this.r=r;

8.numCirculos++;

9.}

10.public Circulo(double r) { this(0.0, 0.0, r); }

11.public Circulo(Circulo c) { this(c.x, c.y, c.r); }

12.public Circulo() { this(0.0, 0.0, 1.0); }

13.public double perimetro() { return 2.0 * PI * r; }

14.public double area() { return PI * r * r; }

15.// método de objeto para comparar círculos

16.public Circulo elMayor(Circulo c) {

17.if (this.r>=c.r) return this; else return c;

18.}

19.// método de clase para comparar círculos

20.public static Circulo elMayor(Circulo c, Circulo d) {

21.if (c.r>=d.r) return c; else return d;

22.}

23.} // fin de la clase Circulo
```

La sentencia 3 (static int numCirculos = 0;) define una variable static o de clase análoga a la de la clase Rectangulo. En este caso no se ha definido como private. Cuando no se especifican permisos de acceso (public, private o protected) se supone la opción por defecto, que es paquete. Con esta opción la variable o método correspondiente puede ser utilizada por todas las clases del paquete y sólo por ellas. Como en este ejemplo no se ha definido ningún paquete, se utiliza el paquete por defecto que es el directorio donde

están definidas las clases. Así pues, la variable numCirculos podrá ser utilizada sólo por las clases que estén en el mismo directorio que Circulo.

La sentencia 4 (public static final double PI=3.14159265358979323846;) define también una variable static, pero contiene una palabra nueva: final. Una variable final tiene como característica el que su valor no puede ser modificado, o lo que es lo mismo, es una constante. Es muy lógico definir el número p como constante, y también es razonable que sea una constante static de la clase Circulo, de forma que sea compartida por todos los métodos y objetos que se creen. La sentencia 5 (public double x, y, r;) define las variables miembro de objeto, que son las coordenadas del centro y el radio del círculo.

La sentencia 6-9 define el constructor general de la clase Circulo. En este caso tiene una peculiaridad y es que el nombre de los argumentos (x, y, r) coincide con el nombre de las variables miembro. Esto es un problema, porque como se verá más adelante los argumentos de un método son variables locales que sólo son visibles dentro del bloque {...} del método, que se destruyen al salir del bloque y que ocultan otras variables de ámbito más general que tengan esos mismos nombres. En otras palabras, si en el código del constructor se utilizan las variables (x, y, r) se está haciendo referencia a los argumentos del método y no a las variables miembro. La sentencia 7 indica cómo se resuelve este problema. Para cualquier método no static de una clase, la palabra this es una referencia al objeto ;el argumento implícito, sobre el que se está aplicando el método. De esta forma, this.x se refiere a la variable miembro, mientras que x es el argumento del constructor.

Las sentencias 10-12 representan otros tres constructores de la clase (métodos sobrecargados), que se diferencian en el número y tipo de argumentos. Los tres tienen en común el realizar su papel llamando al constructor general previamente definido, al que se hace referencia con la palabra ***this*** (en este caso el significado de ***this*** no es exactamente el del argumento implícito). Al constructor de la sentencia 10 sólo se le pasa el radio, con lo cual construye un círculo con ese radio centrado en el origen de coordenadas. Al constructor de la sentencia 11 se le pasa otro objeto de la clase ***Circul***o, del cual saca una copia. El constructor de la sentencia 12 es un ***constructor por defect***o, al que no se le pasa ningún argumento, que crea un círculo de radio unidad centrado en el origen.

Las sentencias 13 y 14 definen los métodos perimetro() y area(), declarados como abstract en la clase Geometria, de modo adecuado para los círculos.

Las sentencias 16-18 definen elMayor(), que es un método de objeto para comparar círculos. Uno de los círculos le llega como argumento implícito y el otro como argumento explícito. En la sentencia 17 se ve cómo al radio del argumento implícito se accede en la forma this.r (se podría acceder también simplemente con r, pues no hay ninguna variable local que la oculte), y al del argumento explícito como c.r, donde c es el nombre del objeto pasado como argumento. La sentencia return devuelve una referencia al objeto cuyo radio sea mayor. Cuando éste es el argumento implícito se devuelve this.

Las sentencias 20-22 presentan la definición de otro método elMayor(), que en este caso es un método de clase (definido como static), y por tanto no tiene argumento implícito.

Los dos objetos a comparar se deben pasar como argumentos explícitos, lo que hace el código muy fácil de entender. Es importante considerar que en ambos casos lo que se devuelve como valor de retorno no es el objeto que constituye el mayor círculo, sino una referencia (un nombre, por decirlo de otra forma).

7.17.5Interface Dibujable

El diagrama de clases de la Figura 3.2 indica que las clases RectanguloGrafico y CirculoGrafico son el resultado, tanto de las clases Rectangulo y Circulo de las que derivan, como de la interface Dibujable, que de alguna manera interviene en el proceso.

El concepto de interface es muy importante en Java. A diferencia de C++, Java no permite herencia múltiple, esto es, no permite que una clase derive de dos clases distintas heredando de ambas métodos y variables miembro. La herencia múltiple es fuente de problemas, pero en muchas ocasiones es una característica muy conveniente. Las interfaces de Java constituyen una alternativa a la herencia múltiple con importantes ventajas prácticas y de "estilo de programación".

Una interface es un conjunto de declaraciones de métodos (sin implementación, es decir, sin definir el código de dichos métodos). La declaración consta del tipo del valor de retorno y del nombre del método, seguido por el tipo de los argumentos entre paréntesis. Cuando una clase implementa una determinada interface, se compromete a dar una definición a todos los métodos de la interface. En cierta forma una interface se parece a una clase abstract cuyos métodos son todos abstract. La ventaja de las interfaces es que no están sometidas a las más rígidas normas de las clases; por ejemplo, una clase no puede heredar de dos clases abstract, pero sí puede implementar varias interfaces.

Una de las ventajas de las interfaces de Java es el establecer pautas o modos de funcionamiento similares para clases que pueden estar o no relacionadas mediante herencia. En efecto, todas las clases que implementan una determinada interface soportan los métodos declarados en la interface y en este sentido se comportan de modo similar. Las interfaces pueden también relacionarse mediante mecanismos de herencia, con más flexibilidad que las clases. Más adelante se volverá con más detenimiento sobre este tema, muy importante para muchos aspectos de Java. El archivo Dibujable.java define la interface Dibujable, mostrada a continuación.

```
1.// archivo Dibujable.java

2.import java.awt.Graphics;

3.public interfaceDibujable {

4.public void setPosicion(double x, double y);

5.public void dibujar(Graphics dw);

6.}
```

La interface Dibujable está dirigida a incorporar, en las clases que la implementen, la capacidad de dibujar sus objetos. El listado muestra la declaración de los métodos

setPosicion() y dibujar(). La declaración de estos métodos no tiene nada de particular. Como el método dibujar() utiliza como argumento un objeto de la clase Graphics, es necesario importar dicha clase. Lo importante es que si las clases RectanguloGrafico y CirculoGrafico implementan la interface Dibujable sus objetos podrán ser representados gráficamente en pantalla.

7.17.6Clase RectanguloGrafico

La clase RectanguloGrafico deriva de Rectangulo (lo cual quiere decir que hereda sus métodos y variables miembro) e implementa la interface Dibujable (lo cual quiere decir que debe definir los métodos declarados en dicha interface). A continuación se incluye la definición de dicha clase.

```
1.// Archivo RectanguloGrafico.java

2.import java.awt.Graphics;

3.import java.awt.Color;

4.class RectanguloGrafico extends Rectangulo implements Dibujable {

5.// nueva variable miembro

6.Color color;

7.// constructor

8.public RectanguloGrafico(double x1, double y1, double x2, double
   y2,

9.Color unColor) {

10.// llamada al constructor de Rectangulo

11.super(x1, y1, x2, y2);

12.this.color = unColor; // en este caso this es opcional

13.}

14.// métodos de la interface Dibujable

15.public void dibujar(Graphics dw) {

16.dw.setColor(color);

17.dw.drawRect((int)x1, (int)y1, (int)(x2-x1), (int)(y2-y1));

18.}

19.public void setPosicion(double x, double y) {

20.; // método vacío, pero necesario de definir

21.}

22.} // fin de la clase RectanguloGrafico
```

Las sentencias 2 y 3 importan dos clases del paquete java.awt. Otra posibilidad sería importar todas las clases de dicho paquete con la sentencia (import java.awt.*;).

La sentencia 4 indica que RectanguloGrafico deriva de la clase Rectangulo e implementa la interface Dibujable. Recuérdese que mientras que sólo se puede derivar de una clase, se pueden implementar varias interfaces, en cuyo caso se ponen en el encabezamiento de la clase separadas por comas.

La sentencia 6 (Color color;) define una nueva variable miembro que se suma a las que ya se tienen por herencia. Esta nueva variable es un objeto de la clase Color.

Las sentencias 8-13 definen el constructor general de la clase, al cual le llegan los cinco argumentos necesarios para dar valor a todas las variables miembro. En este caso los nombres de los argumentos también coinciden con los de las variables miembro, pero sólo se utilizan para pasárselos al constructor de la superclase. En efecto, la sentencia 11 (super(x1, y1, x2, y2);) contiene una novedad: para dar valor

a las variables heredadas lo más cómodo es llamar al constructor de la clase padre o superclase, al cual se hace referencia con la palabra super.

Las sentencias 14-18 y 19-21 definen los dos métodos declarados por la interface Dibujable. El método dibujar() recibe como argumento un objeto dw de la clase Graphics. Esta clase define un contexto para realizar operaciones gráficas en un panel, tales como el color de las líneas, el color de fondo, el tipo de letra a utilizar en los rótulos, etc. Más adelante se verá con más detenimiento este concepto. La sentencia 16 (dw.setColor(color);) hace uso un método de la clase Graphics para determinar el color con el que se dibujarán las líneas a partir de ese momento. La sentencia 17 (dw.drawRect((int)x1, (int)y1, (int)(x2-x1), (int)(y2-y1));) llama a otro método de esa misma clase que dibuja un rectángulo a partir de las coordenadas del vértice superior izquierdo, de la anchura y de la altura.

Java obliga a implementar o definir siempre todos los métodos declarados por la interface, aunque no se vayan a utilizar. Esa es la razón de que las sentencias 19-21 definan un método vacío, que sólo contiene un carácter punto y coma. Como no se va a utilizar no importa que esté vacío, pero Java obliga a dar una definición o implementación.

7.17.7Clase CirculoGrafico

A continuación se define la clase CirculoGrafico, que deriva de la clase Circulo e implementa la interface Dibujable. Esta clase es muy similar a la clase RectanguloGrafico y no requiere explicaciones especiales.

```
1.// archivo CirculoGrafico.java

2.import java.awt.Graphics;

3.import java.awt.Color;

4.public class CirculoGrafico extends Circulo implements Dibujable {
```

```
5.// se heredan las variables y métodos de la clase Circulo

6.Color color;

7.// constructor

8.public CirculoGrafico(double x, double y, double r, Color unColor)
   {

9.// llamada al constructor de Circulo

10.super(x, y, r);

11.this.color = unColor;

12.}

13.// métodos de la interface Dibujable

14.public void dibujar(Graphics dw) {

15.dw.setColor(color);

16.dw.drawOval((int)(x-r),(int)(y-r),(int)(2*r),(int)(2*r));

17.}

18.public void setPosicion(double x, double y) {

19.;

20.}

21.} // fin de la clase CirculoGrafico
```

7.17.8Clase PanelDibujo

La clase que se describe en este apartado es muy importante y quizás una de las más difíciles de entender en este capítulo introductorio. La clase ***PanelDibujo*** es muy importante porque es la responsable final de que los rectángulos y círculos aparezcan dibujados en la pantalla. Esta clase deriva de la clase ***Panel***, que deriva de ***Container***, que deriva de ***Component***, que deriva de ***Object***.

Ya se ha comentado que Object es la clase más general de Java. La clase Component comprende todos los objetos de Java que tienen representación gráfica, tales como botones, barras de desplazamiento, etc. Los objetos de la clase Container son objetos gráficos del AWT (Abstract Windows Toolkit; la librería de clases de Java que permite crear interfaces gráficas de usuario) capaces de contener otros objetos del AWT. La clase Panel define los Container más sencillos, capaces de contener otros elementos gráficos (como otros paneles) y sobre la que se puede dibujar. La clase PanelDibujo contiene el código que se muestra a continuación.

```
1.// archivo PanelDibujo.java

2.import java.awt.*;

3.import java.util.ArrayList;
```

```
4.import java.util.Iterator;
5.public class PanelDibujo extends Panel {
6.// variable miembro
7.private ArrayList v;
8.// constructor
9.public PanelDibujo(ArrayList va) {
10.super(new FlowLayout());
11.this.v = va;
12.}
13.// redefinición del método paint()
14.public void paint(Graphics g) {
15.Dibujable dib;
16.Iterator it;
17.it = v.iterator();
18.while(it.hasNext()) {
19.dib = (Dibujable)it.next();
20.dib.dibujar(g);
21.}
22.}
23.} // Fin de la clase PanelDibujo
```

Las sentencias 2-4 importan las clases necesarias para construir la clase PanelDibujo. Se importan todas las clases del paquete java.awt. La clase ArrayList y la interface Iterator pertenecen al paquete java.util, y sirven para tratar colecciones o conjuntos, en este caso conjuntos de figuras dibujables.

La sentencia 5 indica que la clase PanelDibujo deriva de la clase Panel, heredando de ésta y de sus superclases Container y Component todas sus capacidades gráficas. La sentencia 7 (private ArrayList v;) crea una variable miembro v que es una referencia a un objeto de la clase ArrayList (nótese que no es un objeto, sino una referencia o un nombre de objeto). Las sentencias 9-12 definen el constructor de la clase, que recibe como argumento una referencia va a un objeto de la clase ArrayList. En esta lista estarán almacenadas las referencias a los objetos rectángulos y círculos- que van a ser dibujados. En la sentencia 10 (super(new FlowLayout());) se llama al constructor de la superclase panel, pasándole como argumento un objeto recién creado de la clase FlowLayout. Como se verá más adelante al hablar de construcción de interfaces gráficas con el AWT, la clase FlowLayout se ocupa de distribuir de una determinada forma (de izquierda a derecha y de arriba abajo) los componentes gráficos que se añaden a un

"contenedor" tal como la clase Panel. En este caso no tiene mucha importancia, pero conviene utilizarlo.

Hay que introducir ahora un aspecto muy importante de Java y, en general, de la programación orientada a objetos. Tiene que ver con algo que es conocido con el nombre de Polimorfismo. La idea básica es que una referencia a un objeto de una determinada clase es capaz de servir de referencia o de nombre a objetos de cualquiera de sus clases derivadas. Por ejemplo, es posible en Java hacer lo siguiente:

```
Geometria geom1, geom2;

geom1 = new RectanguloGrafico(0, 0, 200, 100, Color.red);

geom2 = new CirculoGrafico(200, 200, 100, Color.blue);
```

Obsérvese que se han creado dos referencias de la clase Geometria que posteriormente apuntan a objetos de las clases derivadas RectanguloGrafico y CirculoGrafico. Sin embargo, hay una cierta limitación en lo que se puede hacer con las referencias geom1 y geom2. Por ser referencias a la clase Geometria sólo se pueden utilizar las capacidades definidas en dicha clase, que se reducen a la utilización de los métodos perimetro() y area().

De la misma forma que se ha visto con la clase base Geometria, en Java es posible utilizar una referencia del tipo correspondiente a una interface para manejar objetos de clases que implementan dicha interface. Por ejemplo, es posible escribir:

```
Dibujable dib1, dib2;

dib1 = new RectanguloGrafico(0, 0, 200, 100, Color.red);

dib2 = new CirculoGrafico(200, 200, 100, Color.blue);
```

donde los objetos referidos por dib1 y dib2 pertenecen a las clases RectanguloGrafico y CirculoGrafico, que implementan la interface Dibujable. También los objetos dib1 y dib2 tienen una limitación: sólo pueden ser utilizados con los métodos definidos por la interface Dibujable.

El poder utilizar nombres de una superclase o de una interface permite tratar de un modo unificado objetos distintos, aunque pertenecientes a distintas subclases o bien a clases que implementan dicha interface. Esta es la idea en la que se basa el polimorfismo.

Ahora ya se está en condiciones de volver al código del método paint(), definido en las sentencias 14-22 de la clase PanelDibujo. El método paint() es un método heredado de Container, que a su vez redefine el método heredado de Component. En la clase PanelDibujo se da una nueva definición de este método. Una peculiaridad del método paint() es que, por lo general, el programador no tiene que preocuparse de llamarlo: se encargan de ello Java y el sistema operativo. El programador prepara por una parte la ventana y el panel en el que va a dibujar, y por otra programa en el método paint() las operaciones gráficas que quiere realizar. El sistema operativo y Java llaman a paint() cada vez que entienden que la ventana debe ser dibujada o redibujada. El único argumento de paint() es un objeto g de la clase Graphics que, como se ha dicho antes,

constituye el contexto gráfico (color de las líneas, tipo de letra, etc.) con el que se realizarán las operaciones de dibujo.

La sentencia 15 (Dibujable dib;) crea una referencia de la clase Dibujable, que como se ha dicho anteriormente, podrá apuntar o contener objetos de cualquier clase que implemente dicha interface. La sentencia 16 (Iterator it;) crea una referencia a un objeto de la interface Iterator definida en el paquete java.util. La interface Iterator proporciona los métodos hasNext(), que chequea si la colección de elementos que se está recorriendo tiene más elementos y next(), que devuelve el siguiente elemento de la colección. Cualquier colección de elementos (tal como la clase ArrayList de Java, o como cualquier tipo de lista vinculada programada por el usuario) puede implementar esta interface, y ser por tanto utilizada de un modo uniforme. En la sentencia 17 se utiliza el método iterator() de la clase ArrayList (it = v.iterator();), que devuelve una referencia Iterator de los elementos de la lista v. Obsérvese la diferencia entre el método iterator() de la clase ArrayList y la interface Iterator. En Java los nombres de las clases e interfaces siempre empiezan por mayúscula, mientras que los métodos lo hacen con minúscula. Las sentencias 18-21 representan un bucle while cuyas sentencias encerradas entre llaves {...}- se repetirán mientras haya elementos en la enumeración e (o en el vector v).

La sentencia 19 (dib = (Dibujable)it.next();) contiene bastantes elementos nuevos e importantes. El método it.next() devuelve el siguiente objeto de la lista representada por una referencia de tipo Iterator. En principio este objeto podría ser de cualquier clase. Los elementos de la clase ***ArrayList*** son referencias de la clase ***Objec***t, que es la clase más general de ***Jav***a, la clase de la que derivan todas las demás. Esto quiere decir que esas referencias pueden apuntar a objetos de cualquier clase. El nombre de la interface ***(Dibujable)*** entre paréntesis representa un ***cast*** o conversión entre tipos diferentes. En ***Java*** como en C++, la conversión entre variables u objetos de distintas clases es muy importante. Por ejemplo, ***(int)3.14*** convierte el número ***double*** 3.14 en el entero 3. Evidentemente no todas las conversiones son posibles, pero sí lo son y tienen mucho interés las conversiones entre clases que están en la misma línea jerárquica (entre ***subclases*** y ***superclase***s), y entre clases que implementan la misma interface. Lo que se está diciendo a la referencia ***dib*** con el ***cast*** a la interface ***Dibujable*** en la sentencia 19, es que el objeto de la enumeración va a ser tratado exclusivamente con los métodos de dicha interface.En la sentencia 20 `(dib.dibujar(g)`;) se aplica el método ***dibujar()*** al objeto referenciado por ***dib***, que forma parte del iterator ***it***, obtenida a partir de la lista v.

Lo que se acaba de explicar puede parecer un poco complicado, pero es típico de Java y de la programación orientada a objetos. La ventaja del método paint() así programado es que es absolutamente general: en ningún momento se hace referencia a las clases RectanguloGrafico y CirculoGrafico, cuyos objetos son realmente los que se van a dibujar. Esto permite añadir nuevas clases tales como TrianguloGrafico, PoligonoGrafico, LineaGrafica, etc., sin tener que modificar para nada el código anterior: tan sólo es necesario que dichas clases implementen la interface Dibujable. Esta es una ventaja no pequeña cuando se trata de crear programas extensibles (que puedan crecer), flexibles (que se puedan modificar) y reutilizables (que se puedan incorporar a otras aplicaciones).

7.17.9Clase VentanaCerrable

La clase VentanaCerrable es la última clase de este ejemplo. Es una clase de "utilidad" que mejora algo las características de la clase Frame de Java, de la que deriva. La clase Frame estándar tiene una limitación y es que no responde a las acciones normales en Windows para cerrar una ventana o una aplicación (por ejemplo, hacer click en la cruz de la esquina superior derecha). En ese caso, para cerrar la aplicación es necesario recurrir por ejemplo al comando End Task del Task Manager de Windows NT (que aparece con Ctrl+Alt+Supr). Para evitar esta molestia se ha creado la clase VentanaCerrable, que deriva de Frame e implementa la interface WindowListener. A continuación se muestra el código de la clase VentanaCerrable.

```
1. // Archivo VentanaCerrable.java
2. import java.awt.*;
3. import java.awt.event.*;
4. class VentanaCerrable extends Frame implements WindowListener {
5. // constructores
6. public VentanaCerrable() {
7. super();
8. }
9. public VentanaCerrable(String title) {
10. super(title);
11. setSize(500,500);
12. addWindowListener(this);
13. }
14. // métodos de la interface WindowsListener
15. public void windowActivated(WindowEvent e) {;}
16. public void windowClosed(WindowEvent e) {;}
17. public void windowClosing(WindowEvent e) {System.exit(0);}
18. public void windowDeactivated(WindowEvent e) {;}
19. public void windowDeiconified(WindowEvent e) {;}
20. public void windowIconified(WindowEvent e) {;}
21. public void windowOpened(WindowEvent e) {;}
22. } // fin de la clase VentanaCerrable
```

La clase VentanaCerrable contiene dos constructores. El primero de ellos es un constructor por defecto (sin argumentos) que se limita a llamar al constructor de la superclase Frame con la palabra super. El segundo constructor admite un argumento para poner título a la ventana; llama también al constructor de ***Frame*** pasándole este

mismo argumento. Después establece un tamaño para la ventana creada (el tamaño por defecto para ***Frame*** es cero).

La sentencia 12 (addWindowListener(this);) es muy importante y significativa sobre la forma en que el AWT de Java gestiona los eventos sobre las ventanas y en general sobre lo que es la interface gráfica de usuario. Cuando un elemento gráfico en este caso la ventana- puede recibir eventos del usuario es necesario indicar quién se va a encargar de procesar esos eventos. De ordinario al producirse un evento se debe activar un método determinado que se encarga de procesarlo y realizar las acciones pertinentes (en este caso cerrar la ventana y la aplicación). La sentencia 12 ejecuta el método addWindowListener() de la clase Frame (que a su vez lo ha heredado de la clase Window). El argumento que se le pasa a este método indica qué objeto se va a responsabilizar de gestionar los eventos que reciba la ventana implementando la interface WindowListener. En este caso, como el argumento que se le pasa es this, la propia clase VentanaCerrable debe ocuparse de gestionar los eventos que reciba. Así es, puesto que dicha clase implementa la interface WindowListener según se ve en la sentencia 4. Puede notarse que como el constructor por defecto de las sentencias 6-8 no utiliza el método addWindowListener(), si se construye una VentanaCerrable sin título no podrá ser cerrada del modo habitual. Así se ha hecho deliberadamente en este ejemplo para que el lector lo pueda comprobar con facilidad.

La interface WindowListener define los siete métodos necesarios para gestionar los siete eventos con los que se puede actuar sobre una ventana. Para cerrar la ventana sólo es necesario definir el método windowClosing(). Sin embargo, el implementar una interface obliga siempre a definir todos sus métodos. Por ello en las sentencias 15-21 todos los métodos están vacíos (solamente el punto y coma entre llaves), excepto el que realmente interesa, que llama al método exit() de la clase System. El argumento "0" indica terminación normal del programa.

7.18Autoevaluación

1.¿ Qué es AWT ?

2.¿ Qué hace falta para construir una interface gráfica de usuario ?

3.¿ Cuales son los pasos que se pueden seguir para construir una aplicación orientada a eventos sencilla, con interface gráfica de usuario?

4.¿ Cuáles son los eventos que se pueden manejar?

5.¿ Qué es un layout manager?

6.¿ Cómo se hace el manejo de gráficos?

7.¿ Cómo se hace el manejo de textos?

8.¿ Cómo se hace el manejo de imágenes?

9.¿ Cómo se pueden hacer animaciones?

10.¿ Cómo se puede eliminar el parpadeo?

11.¿ Qué es un applet?

12.¿ Cuáles son las características de los applets?

13.¿ Cuáles son los métodos que controlan un applet ?

14.¿ Cómo se incluye un applet en una página HTML ?

15.¿ Cómo se comunica el applet con el navegador ?

16.¿ Se puede manejar audio e imágenes con un applet ?

17.¿ Cómo se hace el paso de parámetros a un applet ?

18.¿ Qué propiedades del sistema se pueden obtener con un applet?

19.¿ Se pueden usar threads con applets ?

20.¿ Cómo hacer que un ***applet*** pueda ejecutarse como ***aplicación ?***

8ENTRADA/SALIDA DE DATOS EN JAVA 1.1

Los programas necesitan comunicarse con su entorno, tanto para recoger datos e información que deben procesar, como para devolver los resultados obtenidos.

La manera de representar estas entradas y salidas en ***Java*** es a base de ***streams*** (flujos de datos). Un ***stream*** es una conexión entre el programa y la fuente o destino de los datos. La información se traslada ***en serie*** (un carácter a continuación de otro) a través de esta conexión. Esto da lugar a una forma general de representar muchos tipos de comunicaciones.

Por ejemplo, cuando se quiere imprimir algo en pantalla, se hace a través de un ***stream*** que conecta el monitor al programa. Se da a ese ***stream*** la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/escritura de archivos, la comunicación a través de Internet o la lectura de la información de un sensor a través del puerto en serie.

8.1 CLASES DE JAVA PARA LECTURA Y ESCRITURA DE DATOS

El paquete ***java.io*** contiene las clases necesarias para la comunicación del programa con el exterior. Dentro de este paquete existen dos familias de jerarquías distintas para la entrada/salida de datos. La diferencia principal consiste en que una opera con ***bytes*** y la otra con ***caracteres*** (el carácter de ***Java*** está formado por dos bytes porque sigue el código ***Unicod***e). En general, para el mismo fin hay dos clases que manejan bytes (una clase de entrada y otra de salida) y otras dos que manejan caracteres.

Desde ***Java 1.***0, la entrada y salida de datos del programa se podía hacer con clases derivadas de ***InputStream*** (para lectura) y ***OutputStream*** (para escritura). Estas clases tienen los métodos básicos ***read()*** y ***write()*** que manejan ***bytes*** y que no se suelen utilizar directamente. La Figura 8.1 muestra las clases que derivan de ***InputStream*** y la Figura 8.2 las que derivan de ***OutputStrea***m.

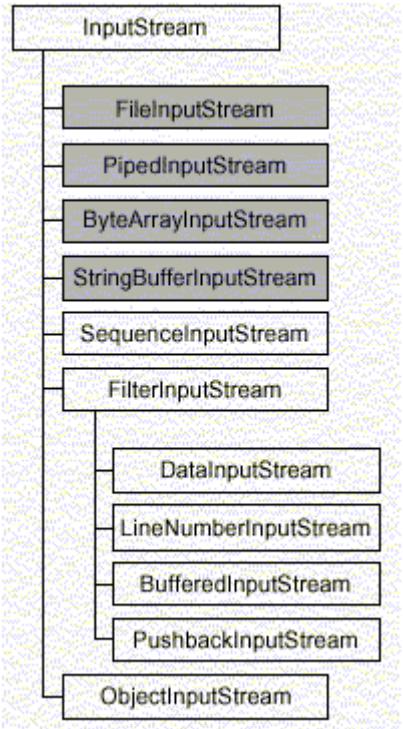


Figura 8.1. Jerarquía de clases InputStream.

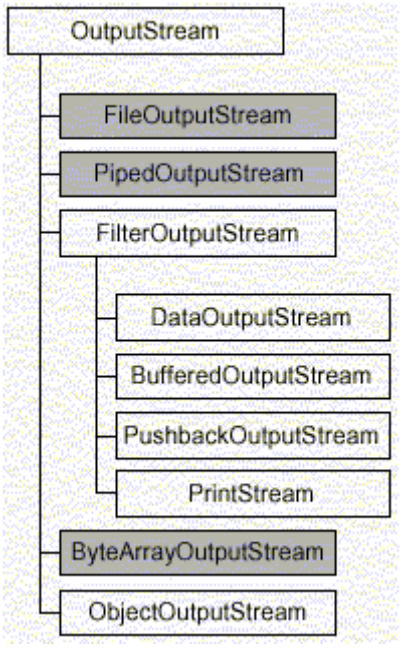


Figura 8.2. Jerarquía de clases OutputStream.

En ***Java 1.1*** aparecieron dos nuevas familias de clases, derivadas de ***Reader*** y ***Writer***, que manejan ***caracteres*** en vez de ***byte***s. Estas clases resultan más prácticas para las aplicaciones en las que se maneja texto. Las clases que heredan de ***Reader*** están incluidas en la Figura 8.3 y las que heredan de ***Writer*** en la Figura 8.4.

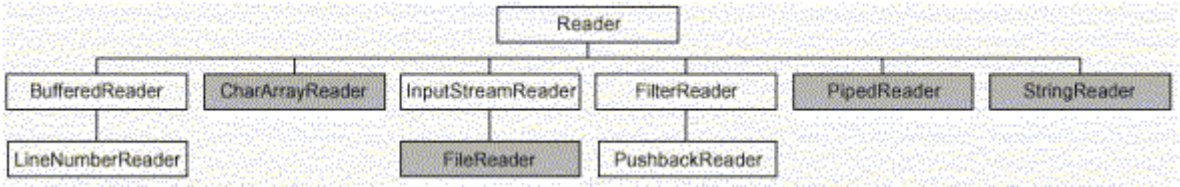


Figura 8.3. Jerarquía de clases Reader.

Figura 8.4. Jerarquía de clases Writer.

En las cuatro últimas figuras las clases con ***fondo gris*** definen de dónde o a dónde se están enviando los datos, es decir, el dispositivo con que conecta el ***stream***. Las demás ***(fondo blanco)*** añaden características particulares a la forma de enviarlos. La intención es que se combinen para obtener el comportamiento deseado. Por ejemplo:

```
BufferedReader in = new BufferedReader(new FileReader(".profile"));
```

Con esta línea se ha creado un ***stream*** que permite leer del archivo ***.profile***. Además, se ha creado a partir de él un objeto ***BufferedReader*** (que aporta la característica de

utilizar ***buffer***). Los caracteres que lleguen a través del ***FileReader*** pasarán a través del ***BufferedReader***, es decir utilizarán el ***buffer***.

A la hora de definir una comunicación con un dispositivo siempre se comenzará determinando el origen o destino de la comunicación ***(clases en gris)*** y luego se le añadirán otras características ***(clases en blanco)***.

Se recomienda utilizar siempre que sea posible las clases ***Reader*** y ***Writer***, dejando las de ***Java 1.0*** para cuando sean imprescindibles. Algunas tareas como la ***serialización*** y la ***compresión*** necesitan las clases ***InputStream*** y ***OutputStream***.

8.1.1 Los nombres de las clases de java.io

Las clases de ***java.io*** siguen una nomenclatura sistemática que permite deducir su función a partir de las palabras que componen el nombre, tal como se describe en la Tabla 8.1.

Palabra	Significado
InputStream, OutputStream	Lectura/Escritura de bytes
Reader, Writer	Lectura/Escritura de caracteres
File	Archivos
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubo de datos
Buffered	Buffer
Filter	Filtro
Data	Intercambio de datos en formato propio de Java
Object	Persistencia de objetos
Print	Imprimir

Tabla 8.1. Palabras identificativas de las clases de java.io.

8.1.2 Clases que indican el origen o destino de los datos

La Tabla 8.2 explica el uso de las clases que definen el lugar con que conecta el ***stream***.

Clases	Función que realizan
FileReader, FileWriter, FileInputStream y FileOutputStream	Son las clases que leen y escriben en **archivos** de disco. Se explicarán luego con más detalle.
StringReader, StringWriter, CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringBufferInputStream	Estas clases tienen en común que se comunican con la **memoria** del ordenador. En vez de acceder del modo habitual al contenido de un String, por ejemplo, lo leen como si llegara carácter a carácter. Son útiles cuando se busca un modo general e idéntico de tratar con todos los dispositivos que maneja un programa.
PipedReader, PipedWriter, PipedInputStream, PipedOutputStream	Se utilizan como un "tubo" o conexión bilateral para transmisión de datos. Por ejemplo, en un programa con dos threads pueden permitir la comunicación entre ellos. Un thread tiene el objeto PipedReader y el otro el PipedWriter. Si los streams están conectados, lo que se escriba en el PipedWriter queda disponible para que se lea del PipedReader. Tambien puede comunicar a dos programas distintos.

Tabla 8.2. Clases que indican el origen o destino de los datos.

8.1.3 Clases que añaden características

La Tabla 8.3 explica las funciones de las clases que alteran el comportamiento de un ***stream*** ya definido.

Clases	Función que realizan
BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream	Como ya se ha dicho, añaden un **buffer** al manejo de los datos. Es decir, se reducen las operaciones directas sobre el dispositivo (lecturas de disco, comunicaciones por red), para hacer más eficiente su uso. ***BufferedReader*** por ejemplo tiene el método ***readLine()*** que lee una línea y la devuelve como un String.
InputStreamReader, OutputStreamWriter	Son clases puente que permiten **convertir** streams que utilizan bytes en otros que manejan caracteres. Son la única relación entre ambas jerarquías y no existen clases que realicen la transformación inversa.
ObjectInputStream, ObjectOutputStream	Pertenecen al mecanismo de la **serialización** y se explicarán más adelante.
FilterReader, FilterWriter, FilterInputStream, FilterOutputStream	Son clases base para aplicar diversos **filtros** o procesos al stream de datos. También se podrían extender para conseguir comportamientos a medida.
DataInputStream, DataOutputStream	Se utilizan para escribir y leer datos directamente en los formatos propios de Java. Los convierten en algo ilegible , pero independiente de plataforma y se usan por tanto para **almacenaje** o para **transmisiones** entre ordenadores de distinto funcionamiento.
PrintWriter, PrintStream	Tienen métodos adaptados para **imprimir** las variables de Java con la apariencia normal. A partir de un boolean escriben "true" o "false", colocan la coma de un número decimal, etc.

Tabla 8.3. Clases que añaden características.

8.2 ENTRADA Y SALIDA ESTÁNDAR (TECLADOY PANTALLA)

En ***Jav***a, la entrada desde teclado y la salida a pantalla están reguladas a través de la clase ***Syste***m. Esta clase pertenece al paquete ***java.lang*** y agrupa diversos métodos y objetos que tienen relación con el sistema local. Contiene, entre otros, tres objetos ***static*** que son:

System.in: Objeto de la clase ***InputStream*** preparado para recibir datos desde la entrada estándar del sistema (habitualmente el teclado).

System.out: Objeto de la clase ***PrintStream*** que imprimirá los datos en la salida estándar del sistema (normalmente asociado con la pantalla).

System.err: Objeto de la clase ***PrintStrea***m. Utilizado para mensajes de error que salen también por pantalla por defecto.

Estas clases permiten la comunicación alfanumérica con el programa a través de lo métodos incluidos en la Tabla 8.4. Son métodos que permiten la entrada/salida a un nivel muy elemental.

Métodos de System.in	**Función que realizan**
int read()	Lee un carácter y lo devuelve como int.
Métodos de System.out y System.err	**Función que realizan**
int print(cualquier tipo)	Imprime en pantalla el argumento que se le pase. Puede recibir cualquier tipo primitivo de variable de Java.
int println(cualquier tipo)	Como el anterior, pero añadiendo '\n' (nueva línea) al final.

Tabla 8.4. Métodos elementales de lectura y escritura.

Existen tres métodos de ***System*** que permiten sustituir la entrada y salida estándar. Por ejemplo, se utiliza para hacer que el programa lea de un archivo y no del teclado.

```
System.setIn(InputStream is);

System.setOut(PrintStream ps);

System.setErr(PrintStream ps);
```

El argumento de ***setIn()*** no tiene que ser necesariamente del tipo ***InputStrea***m. Es una referencia a la clase base, y por tanto puede apuntar a objetos de cualquiera de sus clases derivadas (como ***FileInputStrea***m). Asimismo, el constructor de ***PrintStream*** acepta un ***OutputStrea***m, luego se puede dirigir la salida estándar a cualquiera de las clases definidas para salida.

Si se utilizan estas sentencias con un compilador de ***Java 1.1*** se obtiene un mensaje de método obsoleto ***(deprecate***d) al crear un objeto ***PrintStrea***m. Al señalar como obsoleto el constructor de esta clase se pretendía animar al uso de ***PrintWrite***r, pero existen casos en los cuales es imprescindible un elemento ***PrintStrea***m. Afortunadamente, ***Java 1.2*** ha reconsiderado esta decisión y de nuevo se puede utilizar sin problemas.

8.2.1 Salida de texto y variables por pantalla

Para imprimir en la pantalla se utilizan los métodos ***System.out.print()*** y ***System.out.println()***. Son los primeros métodos que aprende cualquier programador. Sus características fundamentales son:

1.Pueden imprimir valores escritos directamente en el código o cualquier tipo de variable primitiva de ***Jav***a.

```
2.System.out.println("Hola, Mundo!");

3.System.out.println(57);

4.double numeroPI = 3.141592654;

5.System.out.println(numeroPI);
```

```
6.String hola = new String("Hola");

7.System.out.println(hola);
```

8.Se pueden imprimir varias variables en una llamada al método correspondiente utilizando el operador + de concatenación, que equivale a convertir a ***String*** todas las variables que no lo sean y concatenar las cadenas de caracteres (el primer argumento debe ser un ***Strin***g).

```
System.out.println("Hola, Mundo! " + numeroPI);
```

Se debe recordar que los objetos ***System.out*** y ***System.err*** son de la clase ***PrintStream*** y aunque imprimen las variables de un modo legible, no permiten dar a la salida un formato a medida. El programador no puede especificar un formato distinto al disponible por defecto.

8.2.2 Lectura desde teclado

Para leer desde teclado se puede utilizar el método ***System.in.read()*** de la clase ***InputStrea***m. Este método lee un carácter por cada llamada. Su valor de retorno es un ***in***t. Si se espera cualquier otro tipo hay que hacer una conversión explícita mediante un ***cas***t.

```
char c;

c=(char)System.in.read();
```

Este método puede lanzar la excepción ***java.io.IOException*** y siempre habrá que ocuparse de ella, por ejemplo en la forma:

```
1.try {

2.c=(char)System.in.read();

3.}

4.catch(java.io.IOException ioex) {

5.// qué hacer cuando ocurra la excepción

6.}
```

Para leer datos más largos que un simple carácter es necesario emplear un bucle ***while*** o ***for*** y unir los caracteres. Por ejemplo, para leer una línea completa se podría utilizar un bucle ***while*** guardando los caracteres leídos en un ***String*** o en un ***StringBuffer*** (más rápido que ***Strin***g):

```
1.char c;

2.String frase = new String(""); // StringBuffer frase=new StringBuffer("");

3.try {

4.while((c=System.in.read()) != '\n')
```

```
5.frase = frase + c; // frase.append(c);

6.}

7.catch(java.io.IOException ioex) {}
```

Una vez que se lee una línea, ésta puede contener números de coma flotante, etc. Sin embargo, hay una manera más fácil de conseguir lo mismo: utilizar adecuadamente la librería ***java.io***.

8.2.3 Método práctico para leer desde teclado

Para facilitar la lectura de teclado se puede conseguir que se lea una línea entera con una sola orden si se utiliza un objeto ***BufferedReader***. El método ***String readLine()*** perteneciente a ***BufferReader*** lee todos los caracteres hasta encontrar un '\n' o '\r' y los devuelve como un ***String*** (sin incluir '\n' ni '\r'). Este método también puede lanzar ***java.io.IOException***.

System.in es un objeto de la clase ***InputStream***. ***BufferedReader*** pide un ***Reader*** en el constructor. El puente de unión necesario lo dará ***InputStreamReader***, que acepta un ***InputStream*** como argumento del constructor y es una clase derivada de ***Reader***. Por lo tanto si se desea leer una línea completa desde la entrada estándar habrá que utilizar el siguiente código:

```
InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

// o en una línea:

// BufferedReader br2 = new BufferedReader(new InputStreamReader(System.in));

String frase = br2.readLine(); // Se lee la línea con una llamada
```

Así ya se ha leído una línea del teclado. El thread que ejecute este código estará parado en esta línea hasta que el usuario termine la línea (oprime ***return***). Es más sencillo y práctico que la posibilidad anterior.

¿Y qué hacer con una línea entera? La clase ***java.util.StringTokenizer*** da la posibilidad de separar una cadena de caracteres en las ?palabras? ***(tokens)*** que la forman (por defecto, conjuntos de caracteres separados por un espacio, '\t', '\r', o por '\n'). Cuando sea preciso se pueden convertir las ?palabras? en números.

La Tabla 8.5 muestra los métodos más prácticos de la clase ***StringTokenizer***.

Tabla 8.5. Métodos de StringTokenizer.

Métodos	Función que realizan
StringTokenizer(String)	Constructor a partir de la cadena que hay que separar
boolean hasMoreTokens()	¿Hay más palabras disponibles en la cadena?
String nextToken()	Devuelve el siguiente token de la cadena
int countTokens()	Devuelve el número de tokens que se pueden extraer de la frase

La clase ***StreamTokenizer*** de ***java.io*** aporta posibilidades más avanzadas que ***StringTokenizer***, pero también es más compleja. Directamente separa en ***tokens*** lo que entra por un ***InputStream*** o ***Reader***.

Se recuerda que la manera de convertir un ***String*** del tipo ?3.141592654? en el valor ***double*** correspondiente es crear un objeto ***Double*** a partir de él y luego extraer su valor ***double***:

```
double pi = (Double.valueOf("3.141592654")).doubleValue();
```

El uso de estas clases facilita el acceso desde teclado, resultando un código más fácil de escribir y de leer. Además tiene la ventaja de que se puede generalizar a la lectura de archivos.

8.3 LECTURA Y ESCRITURA DE ARCHIVOS

Aunque el manejo de archivos tiene características especiales, se puede utilizar lo dicho hasta ahora para las entradas y salidas estándar con pequeñas variaciones. ***Java*** ofrece las siguientes posibilidades:

Existen las clases ***FileInputStream*** y ***FileOutputStream*** (extendiendo ***InputStream*** y ***OutputStream***) que permiten leer y escribir ***bytes*** en archivos. Para archivos de texto son preferibles ***FileReader*** (desciende de ***Reader***) y ***FileWriter*** (desciende de ***Writer***), que realizan las mismas funciones. Se puede construir un objeto de cualquiera de estas cuatro clases a partir de un ***String*** que contenga el nombre o la dirección en disco del archivo o con un objeto de la clase ***File*** que representa dicho archivo. Por ejemplo el código

```
FileReader fr1 = new FileReader("archivo.txt");
```

es equivalente a:

```
File f = new File("archivo.txt");

FileReader fr2 = new FileReader(f);
```

Si no encuentran el archivo indicado, los constructores de ***FileReader*** y ***FileInputStream*** pueden lanzar la excepción ***java.io.FileNotFoundException***.

Los constructores de ***FileWriter*** y ***FileOutputStream*** pueden lanzar ***java.io.IOException***. Si no encuentran el archivo indicado, lo crean nuevo. Por defecto, estas dos clases comienzan a escribir al comienzo del archivo. Para escribir detrás de lo que ya existe en el archivo (?append?), se utiliza un segundo argumento de tipo ***boolean*** con valor ***true***:

```
FileWriter fw = new FileWriter("archivo.txt", true);
```

Las clases que se explican a continuación permiten un manejo más fácil y eficiente que las vistas hasta ahora.

8.3.1 Clases File y FileDialog

Un objeto de la clase ***File*** puede representar un ***archivo*** o un ***directorio***. Tiene los siguientes constructores:

```
File(String name)

File(String dir, String name)

File(File dir, String name).
```

Se puede dar el nombre de un archivo, el nombre y el directorio, o sólo el directorio, como ***path*** absoluto y como ***path*** relativo al directorio actual. Para saber si el archivo existe se puede llamar al método ***boolean exists().***

```
File f1 = new File("/usr/bin/vi"); // La barra '\' se escribe '\\'

File f2 = new File("/usr/bin"); // Un directorio

File f3 = new File(f2, "vi"); // Es igual a f1
```

Si ***File*** representa un archivo que existe los métodos de la Tabla 10 .6 dan información de él.

Métodos	Función que realizan
boolean isFile()	true si el archivo existe
long length()	tamaño del archivo en bytes
long lastModified()	fecha de la última modificación
boolean canRead()	true si se puede leer
boolean canWrite()	true si se puede escribir
delete()	borrar el archivo
RenameTo(File)	cambiar el nombre

Tabla 8.6. Métodos de File para archivos.

Si representa un directorio se pueden utilizar los de la Tabla 8.7:

Métodos	Función que realizan
boolean isDirectory()	true si existe el directorio
mkdir()	crear el directorio
delete()	borrar el directorio
String[] list()	devuelve los archivos que se encuentran en el directorio

Tabla 8.7. Métodos de File para directorios.

Por último, otros métodos incluidos en la Tabla 8.8 devuelven la trayectoria del archivo de distintas maneras.

Métodos	Función que realizan
String getPath()	Devuelve el path que contiene el objeto File
String getName()	Devuelve el nombre del archivo
String getAbsolutePath()	Devuelve el path absoluto (juntando el relativo al actual)
String getParent()	Devuelve el directorio padre

Tabla 8.8. Métodos de File que devuelven la trayectoria.

Una forma típica de preguntar por un archivo es presentar un caja de diálogo. La clase ***java.awt.FileDialog*** presenta el diálogo típico de cada sistema operativo para guardar o abrir archivos.

Sus constructores son:

```
FileDialog(Frame fr)

FileDialog(Frame fr, String title)

FileDialog(Frame fr, String title, int type)
```

donde ***type*** puede ser ***FileDialog.LOAD*** o ***FileDialog.SAVE*** según la operación que se desee realizar.

Es muy fácil conectar este diálogo con un ***File***, utilizando los métodos ***String getFile()*** y ***String getDirectory()***. Por ejemplo:

```
FileDialog fd = new FileDialog(f, "Elija un archivo");

fd.show();

File f = new File(fd.getDirectory(), fd.getFile());
```

8.3.2 Lectura de archivos de texto

Se puede crear un objeto ***BufferedReader*** para leer de un archivo de texto de la siguiente manera:

```
BufferedReader br = new BufferedReader(new FileReader("archivo.txt"));
```

Utilizando el objeto de tipo ***BufferedReader*** se puede conseguir exactamente lo mismo que en las secciones anteriores utilizando el método ***readLine()*** y la clase ***StringTokenizer***. En el caso de archivos es muy importante utilizar el ***buffer*** puesto que la tarea de escribir en disco es muy lenta respecto a los procesos del programa y realizar las operaciones de lectura de golpe y no de una en una hace mucho más eficiente el acceso. Por ejemplo:

```
1.// Lee un archivo entero de la misma manera que de teclado

2.String texto = new String();

3.try {
```

```
4.FileReader fr = new FileReader("archivo.txt");

5.entrada = new BufferedReader(fr);

6.String s;

7.while((s = entrada.readLine()) != null)

8.texto += s;

9.entrada.close();

10.}

11.catch(java.io.FileNotFoundException fnfex) {

12.System.out.println("Archivo no encontrado: " + fnfex);

13.}

14.catch(java.io.IOException ioex)

15.{

16.}
```

8.3.3 Escritura de archivos de texto

La clase ***PrintWriter*** es la más práctica para escribir un archivo de texto porque posee los métodos ***print***(cualquier tipo) y ***println***(cualquier tipo), idénticos a los de ***System.out*** (de clase ***PrintStream***).

Un objeto ***PrintWriter*** se puede crear a partir de un ***BufferedWriter*** (para disponer de ***buffer***), que se crea a partir del ***FileWriter*** al que se la pasa el nombre del archivo. Después, escribir en el archivo es tan fácil como en pantalla. El siguiente ejemplo ilustra lo anterior:

```
1.try {

2.FileWriter fw = new FileWriter("escribeme.txt");

3.BufferedWriter bw = new BufferedWriter(fw);

4.PrintWriter salida = new PrintWriter(bw);

5.salida.println("Hola, soy la primera línea");

6.salida.close();

7.// Modo append

8.bw = new BufferedWriter(new FileWriter("escribeme.txt", true));

9.salida = new PrintWriter(bw);

10.salida.print("Y yo soy la segunda. ");

11.double b = 123.45;
```

```
12.salida.println(b);

13.salida.close();

14.cacth(java.io.IOException ioex)

15.{

16.}
```

8.3.4 Archivos que no son de texto

DataInputStream y ***DataOutputStream*** son clases de ***Java 1.0*** que no han sido alteradas hasta ahora. Para leer y escribir datos primitivos directamente (sin convertir a/de ***String***) siguen siendo más útiles estas dos clases.

Son clases diseñadas para trabajar de manera conjunta. Una puede leer lo que la otra escribe, que en sí no es algo legible, sino el dato como una secuencia de ***byte***s. Por ello se utilizan para almacenar datos de manera independiente de la plataforma (o para mandarlos por una red entre ordenadores muy distintos).

El problema es que obligan a utilizar clases que descienden de ***InputStream*** y ***OutputStream*** y por lo tanto algo más complicadas de utilizar. El siguiente código primero escribe en el archivo *prueba.dat* para después leer los datos escritos:

```
1.// Escritura de una variable double

2.DataOutputStream dos = new DataOutputStream(

3.new BufferedOutputStream(

4.new FileOutputStream("prueba.dat")));

5.double d1 = 17/7;

6.dos.writeDouble(d);

7.dos.close();

8.// Lectura de la variable double

9.DataInputStream dis = new DataInputStream(

10.new BufferedInputStream(

11.new FileInputStream("prueba.dat")));

12.double d2 = dis.readDouble();
```

8.4 SERIALIZACIÓN

La ***serialización*** es un proceso por el que un objeto cualquiera se puede convertir en una ***secuencia de bytes*** con la que más tarde se podrá reconstruir dicho objeto

manteniendo el valor de sus variables. Esto permite guardar un objeto en un archivo o mandarlo por la red.

Para que una clase pueda utilizar la serialización, debe implementar la interface ***Serializable***, que no define ningún método. Casi todas las clases estándar de ***Java*** son serializables. La clase MiClase se podría serializar declarándola como:

```
public class MiClase implements Serializable { }
```

Para escribir y leer objetos se utilizan las clases ***ObjectInputStream*** y ***ObjectOutputStream***, que cuentan con los métodos ***writeObject()*** y ***readObject()***. Por ejemplo:

```
1.ObjectOutputStream objout = new ObjectOutputStream(

2.new FileOutputStream("archivo.x"));

3.String s = new String("Me van a serializar");

4.objout.writeObject(s);

5.ObjectInputStream objin = new ObjectInputStream(new
    FileInputStream("archivo.x"));

6.String s2 = (String)objin.readObject();
```

Es importante tener en cuenta que ***readObject()*** devuelve un ***Object*** sobre el que se deberá hacer un ***cast*** para que el objeto sea útil. La reconstrucción necesita que el archivo ****.class*** esté al alcance del programa (como mínimo para hacer este ***cast***).

Al serializar un objeto, automáticamente se serializan todas sus variables y objetos miembro. A su vez se serializan los que estos objetos miembro puedan tener (todos deben ser serializables). También se reconstruyen de igual manera. Si se serializa un ***Vector*** que contiene varios ***String***s, todo ello se convierte en una serie de ***byte***s. Al recuperarlo la reconstrucción deja todo en el lugar en que se guardó.

Si dos objetos contienen una referencia a otro, éste no se duplica si se escriben o leen ambos del mismo ***stream***. Es decir, si el mismo ***String*** estuviera contenido dos veces en el ***Vector***, sólo se guardaría una vez y al recuperarlo sólo se crearía un objeto con dos referencias contenidas en el vector.

8.4.1 Control de la serialización

Aunque lo mejor de la serialización es que su comportamiento automático es bueno y sencillo, existe la posibilidad de especificar cómo se deben hacer las cosas.

La palabra clave ***transient*** permite indicar que un objeto o variable miembro no sea serializado con el resto del objeto. Al recuperarlo, lo que esté marcado como ***transient*** será 0, ***null*** o ***false*** (en esta operación no se llama a ningún constructor) hasta que se le dé un nuevo valor. Podría ser el caso de un ***password*** que no se quiere guardar por seguridad.

Las variables y objetos ***static*** no son serializados. Si se quieren incluir hay que escribir el código que lo haga. Por ejemplo, habrá que programar un método que serialice los objetos estáticos al que se llamará después de serializar el resto de los elementos. También habría que recuperarlos explícitamente después de recuperar el resto de los objetos.

Las clases que implementan ***Serializable*** pueden definir dos métodos con los que controlar la serialización. No están obligadas a hacerlo porque una clase sin estos métodos obtiene directamente el comportamiento por defecto. Si los define serán los que se utilicen al serializar:

```
private void writeObject(ObjectOutputStream stream) throws IOException

private void readObject(ObjectInputStream stream) throws IOException
```

El primero permite indicar qué se escribe o añadir otras instrucciones al comportamiento por defecto. El segundo debe poder leer lo que escribe ***writeObject()***. Puede usarse por ejemplo para poner al día las variables que lo necesiten al ser recuperado un objeto. Hay que leer en el mismo orden en que se escribieron los objetos.

Se puede obtener el comportamiento por defecto dentro de estos métodos llamando a ***stream.defaultWriteObject()*** y ***stream.defaultReadObject()***

.

Para guardar explícitamente los tipos primitivos se puede utilizar los métodos que proporcionan ***ObjectInputStream*** y ***ObjectOutputStrea***m, idénticos a los de ***DataInputStream*** y ***DataOutputStream (writeInt()***, ***readDouble()***, ...) o guardar objetos de sus clases equivalentes ***(Intege***r, ***Doubl***e...).

Por ejemplo, si en una clase llamada ***Tierra*** se necesita que al serializar un objeto siempre le acompañe la constante ***g*** (9,8) definida como ***static*** el código podría ser:

```
1.static double g = 8.8;

2.private void writeObject(ObjectOutputStream stream) throws IOException {

3.stream.defaultWriteObject();

4.stream.writeDouble(g);

5.}

6.private void readObject(ObjectInputStream stream) throws IOException {

7.stream.defaultReadObject();

8.g = stream.readDouble(g);

9.}
```

8.4.2 Externalizable

La interface ***Externalizable*** extiende ***Serializable***. Tiene el mismo objetivo que ésta, pero no tiene ningún comportamiento automático, todo se deja en manos del programador.

Externalizable tiene dos métodos que deben implementarse.

```
1.interface Externalizable {

2.public void writeExternal(ObjectOutput out) throws IOException;

3.public void readExternal(ObjectInput in) throws IOException,

4.ClassNotFoundException;

5.}
```

Al transformar un objeto, el método ***writeExternal()*** es responsable de todo lo que se hace. Sólo se guardará lo que dentro de éste método se indique. El método ***readExternal()*** debe ser capaz de recuperar lo guardado por ***writeExternal()***. La lectura debe ser en el mismo orden que la escritura. Es importante saber que antes de llamar a este método se llama al constructor por defecto de la clase.

Como se ve el comportamiento de ***Externalizable*** es muy similar al de ***Serializable***.

8.5 LECTURA DE UN ARCHIVO EN UN SERVIDOR DE INTERNET

Teniendo la dirección de Internet de un archivo, la librería de ***Java*** permite leer este archivo utilizando un ***stream***. Es una aplicación muy sencilla que muestra la polivalencia del concepto de ***stream***.

En el paquete ***java.net*** existe la clase ***URL***, que representa una dirección de Internet. Esta clase tiene el método ***InputStream openStream(URL dir)*** que abre un ***stream*** con origen en la dirección de Internet.

A partir de ahí, se trata como cualquier elemento ***InputStream***. Por ejemplo:

```
1.//Lectura del archivo (texto HTML)

2.URL direccion = new URL("http://www1.ceit.es/subdir/MiPagina.htm");

3.String s = new String();

4.String html = new String();

5.try {

6.BufferedReader br = new BufferedReader(

7.new InputStreamReader(

8.direccion.openStream()));
```

```
9.while((s = br.readLine()) != null)

10.html += s + '\n';

11.br.close();

12.}

13.catch(Exception e) {

14.System.err.println(e);

15.}
```

8.6 Autoevaluación

1.¿ Qué es un stream ?

2.¿ Cuales son las clases para manejar la entradas salida en Java ?

3.¿ Cuál son los métodos para manejar la entrada, salida y error estándar?

4.¿ Cual es el método practico para leer desde el teclado ?

5.¿ Cómo se manejan los archivos de texto?

6.¿ Cómo se manejan los archivos que no son de texto?

7.¿ Qué es la serialización?

8.¿ Cómo se controla la serialización ?

9.¿ Qué hace la interface externalizable ?

10.¿ Cómo leer un archivo en un servidor de Internet ?

BIBLIOGRAFÍA:

1. Mark Allen Weiss, “Estructura de Datos en Java”, Addison-Wesley.

2. Luis Joyanes Aguilar, Ignacio Zahonero martínez, “Programación en Java 2 Algortimos, Estructura de Datos y Programación Orientada a Objetos”, Mc Graw Hill.

3. Laura Lemay, Rogers Cadenhead, “Aprendiendo Java 2 en 21 dias”, Pearson Prentice Hall.

4. Craig Larman, “UML y Patrones Introducción al Análisis y Diseño Orientado a Objetos”,Pearson Prentice Hall.

5. David Flanagan, “Java en Pocas Palabras”, Mc Graw Hill, O’Reilly.

6. Jeff Savit, Sean Wilcox, Bhuvana Jayaraman, “Java para la Empresa”, Mc Graw Hill.

7. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, “Estructura de Datos y Algoritmos”, Addison-Wesley Iberoamericana.