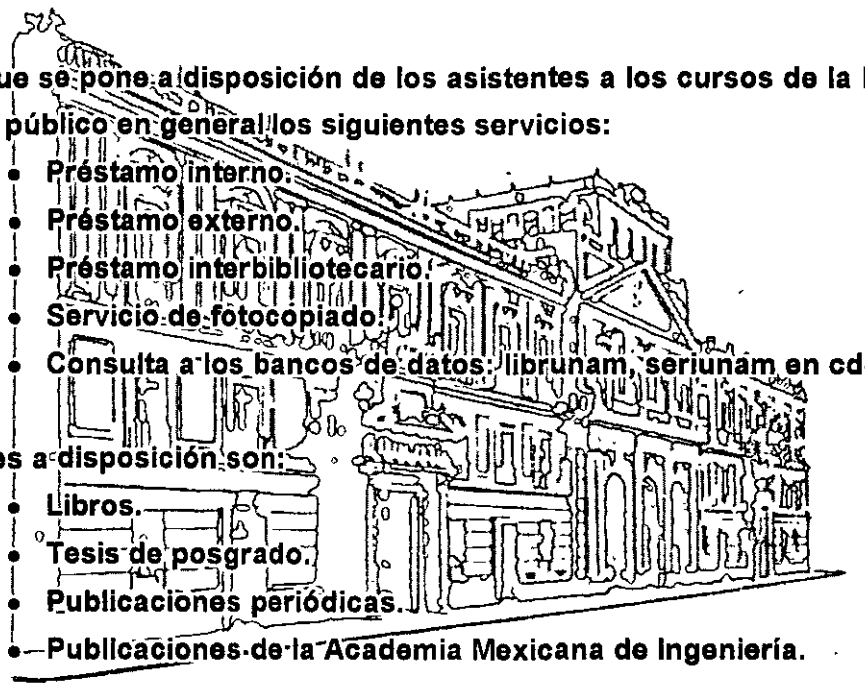
FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN
" ING. BRUNO MASCANZONI "

El Centro de Información y Documentación Ing. Bruno Mascanzoni tiene por objetivo satisfacer las necesidades de actualización y proporcionar una adecuada información que permita a los ingenieros, profesores y alumnos estar al tanto del estado actual del conocimiento sobre temas específicos, enfatizando las investigaciones de vanguardia de los campos de la ingeniería, tanto nacionales como extranjeras.



Es por ello que se pone a disposición de los asistentes a los cursos de la DECFI, así como del público en general los siguientes servicios:

- **Préstamo interno.**
- **Préstamo externo.**
- **Préstamo interbibliotecario.**
- **Servicio de fotocopiado.**
- **Consulta a los bancos de datos: librunam, seriunam en cd-rom.**

Los materiales a disposición son:

- **Libros.**
- **Tesis de posgrado.**
- **Publicaciones periódicas.**
- **Publicaciones de la Academia Mexicana de Ingeniería.**
- **Notas de los cursos que se han impartido de 1988 a la fecha.**

En las áreas de ingeniería industrial, civil, electrónica, ciencias de la tierra, computación y, mecánica y eléctrica.

El CID se encuentra ubicado en el mezzanine del Palacio de Minería, lado oriente.

El horario de servicio es de 10:00 a 14:30 y 16:00 a 17:30 de lunes a viernes.

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

A LOS ASISTENTES A LOS CURSOS

Las autoridades de la Facultad de Ingeniería, por conducto del jefe de la División de Educación Continua, otorgan una constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso.

El control de asistencia se llevará a cabo a través de la persona que le entregó las notas. Las inasistencias serán computadas por las autoridades de la División, con el fin de entregarle constancia solamente a los alumnos que tengan un mínimo de 80% de asistencias.

Pedimos a los asistentes recoger su constancia el día de la clausura. Estas se retendrán por el periodo de un año, pasado este tiempo la DECFI no se hará responsable de este documento.

Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece la División están planeados para que los profesores expongan una tesis, pero sobre todo, para que coordinen las opiniones de todos los interesados, constituyendo verdaderos seminarios.

Es muy importante que todos los asistentes llenen y entreguen su hoja de inscripción al inicio del curso, información que servirá para integrar un directorio de asistentes, que se entregará oportunamente.

Con el objeto de mejorar los servicios que la División de Educación Continua ofrece, al final del curso deberán entregar la evaluación a través de un cuestionario diseñado para emitir juicios anónimos.

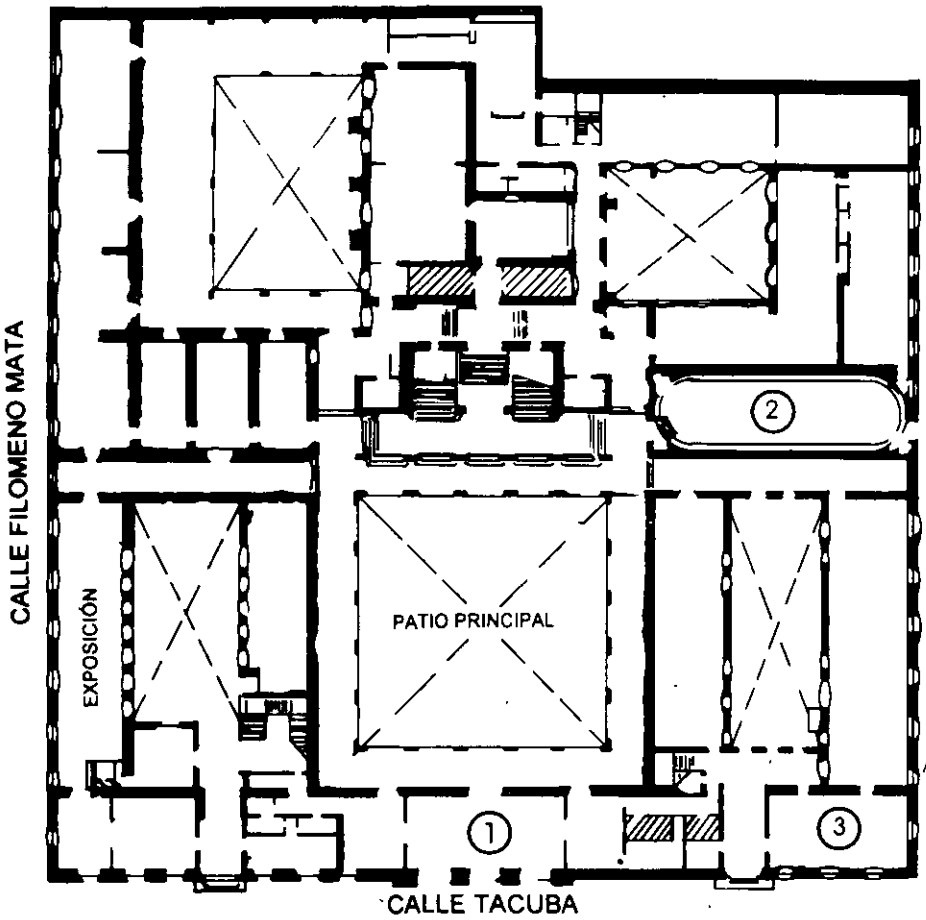
Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases, a efecto de no llenar en la última sesión las evaluaciones y con esto sean más fehacientes sus apreciaciones.

Atentamente

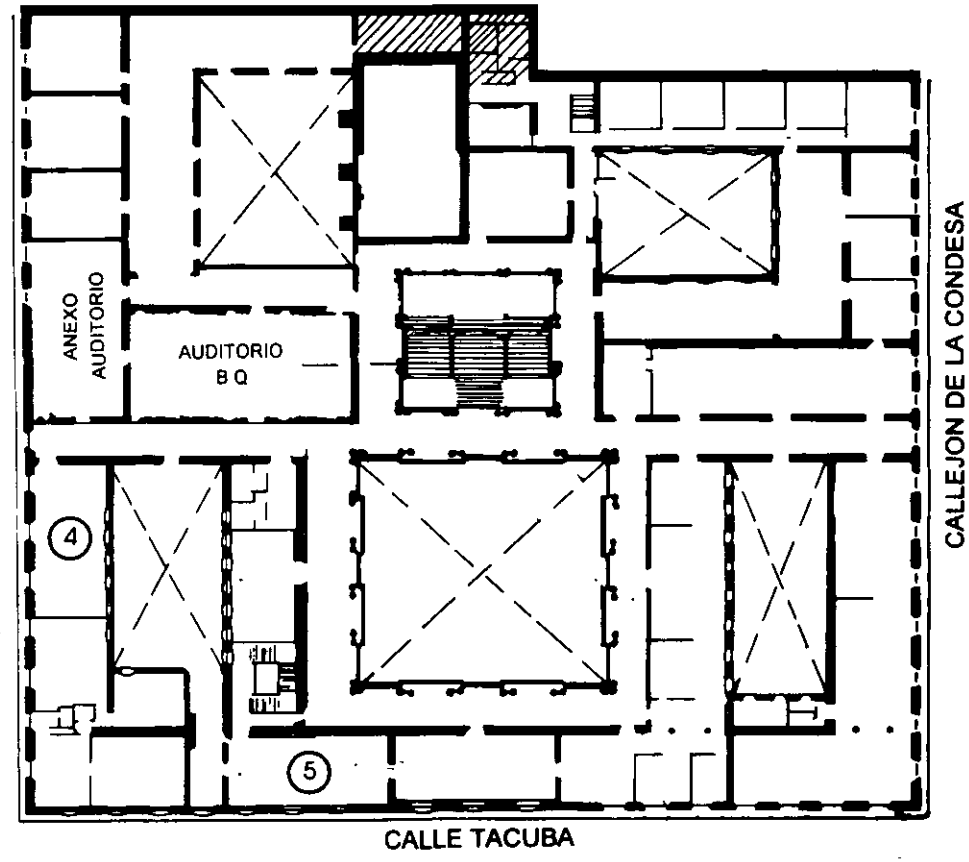
División de Educación Continua.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F APDO. Postal M-2285
Teléfonos. 5512-8955 5512-5121 5521-7335 5521-1987 Fax 5510-0573 5521-4021 AL 25

PALACIO DE MINERIA

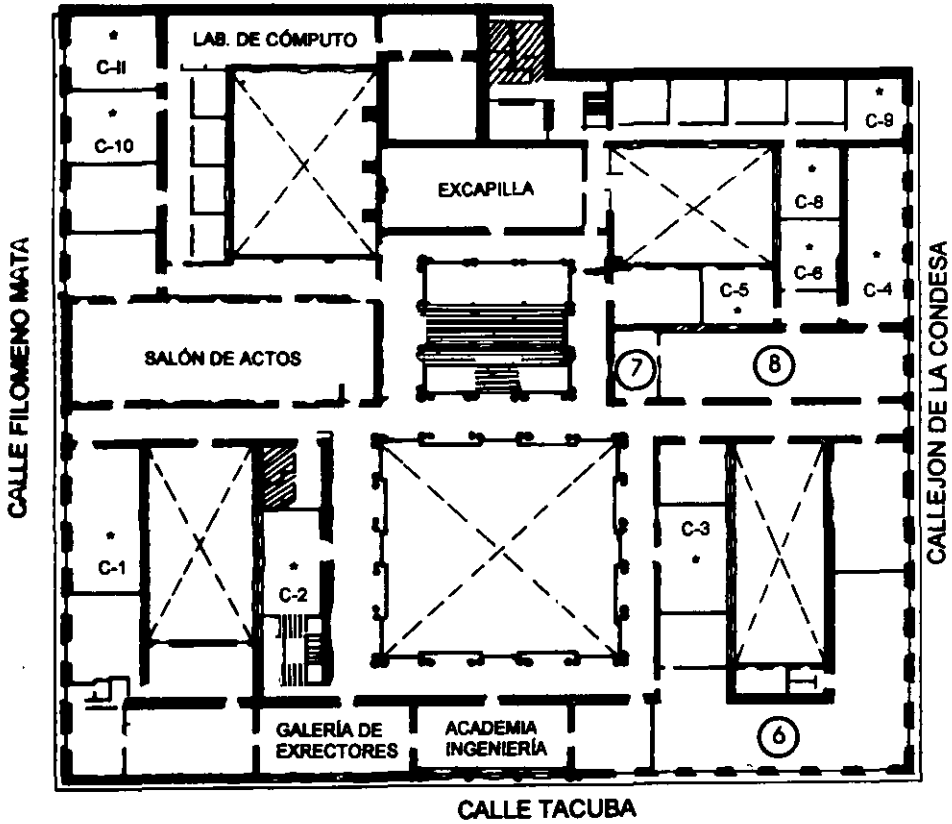


PLANTA BAJA



MEZZANINNE

PALACIO DE MINERIA



1er. PISO

GUÍA DE LOCALIZACIÓN

1. ACCESO
2. BIBLIOTECA HISTÓRICA
3. LIBRERÍA UNAM
4. CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN "ING. BRUNO MASCANZONI"
5. PROGRAMA DE APOYO A LA TITULACIÓN
6. OFICINAS GENERALES
7. ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA
8. SALA DE DESCANSO

SANITARIOS

* AULAS

DIVISIÓN DE EDUCACIÓN CONTINUA
FACULTAD DE INGENIERÍA U.N.A.M.
CURSOS ABIERTOS

DIVISIÓN DE EDUCACIÓN CONTINUA
DEC

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA, UNAM
CURSOS ABIERTOS

DIVISIÓN DE EDUCACIÓN CONTINUA DEC

CURSO: ***CC026 Programación Orientada a Objetos con Java***
FECHA: ***21 de mayo al 1 de junio del 2001***

EVALUACIÓN DEL PERSONAL DOCENTE

(ESCALA DE EVALUACIÓN: 1 A 10)

CONFERENCISTA	DOMINIO DEL TEMA	USO DE AYUDAS AUDIOVISUALES	COMUNICACIÓN CON EL ASISTENTE	PUNTUALIDAD
JONATHAN GAMBOA BELTRAN				
YAZMIN VEGA HERNANDEZ				

Promedio _____

EVALUACIÓN DE LA ENSEÑANZA

CONCEPTO	CALIF
ORGANIZACION Y DESARROLLO DEL CURSO	
GRADO DE PROFUNDIDAD DEL CURSO	
ACTUALIZACION DEL CURSO	
APLICACION PRACTICA DEL CURSO	

Promedio _____

EVALUACIÓN DEL CURSO

CONCEPTO	CALIF.
CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
CONTINUIDAD EN LOS TEMAS	
CALIDAD DEL MATERIAL DIDACTICO UTILIZADO	

Promedio _____

Evaluacion total del curso_____

Continúa...2

FACULTAD DE INGENIERÍA UNAM
DIVISIÓN DE EDUCACIÓN CONTINUA

"Tres décadas de orgullosa excelencia" 1971 - 2001

PROGRAMACION ORIENTADA A OBJETOS CON JAVA

MAYO - JUNIO DEL 2001

Palacio de Minería, Calle de Tacuba No 5, Primer piso, Delegación Cuauhtémoc, CP 06000, Col Centro, México D.F,
APDO Postal M-2285 • Tels 5521 4021 al 24, 5623 2910 y 5623 2971 • Fax: 5510.0573

1 Conceptos del diseño orientado a objetos.

1.1 Crisis del software.

Un constructor pensaría raramente en añadir un subsótano a un edificio ya construido de cien plantas; hacer tal cosa sería muy costoso e indudablemente sería una invitación al fracaso. Asombrosamente, los usuarios de sistemas de software casi nunca se lo piensan dos veces a la hora de solicitar cambios equivalentes. De todas formas, argumentan, es simplemente cosa de programar.

Nuestro fracaso en dominar la complejidad del software lleva a proyectos retrasados, que exceden el presupuesto, y que son deficientes respecto a los requerimientos fijados. A menudo se llama a esta situación la crisis del software, pero, francamente, una enfermedad que ha existido tanto tiempo debe considerarse normal. Tristemente, esta crisis se traduce en el desperdicio de recursos humanos —un bien de lo más precioso—, así como en una considerable pérdida de oportunidad. Simplemente, no hay suficientes buenos desarrolladores para crear todo el nuevo software que necesitan los usuarios. Peor aún, un porcentaje considerable del personal de desarrollo en cualquier organización debe muchas veces estar dedicado al mantenimiento o la conservación de software geriátrico. Dada la contribución tanto directa como indirecta del software a la base económica de la mayoría de los países industrializados, y considerando en qué medida el software puede amplificar la potencia del individuo, es inaceptable permitir que esta situación se mantenga.

¿Cómo puede cambiarse esta imagen desoladora? Ya que el problema subyacente surge de la complejidad inherente al software, la sugerencia que se plantea aquí es estudiar en primer lugar cómo se organizan los sistemas complejos en otras disciplinas. Realmente, si se echa una mirada al mundo que nos rodea, se observarán sistemas con éxito dentro de una complejidad que habrá que tener en cuenta. Algunos de esos sistemas son obras humanas, como el transbordador espacial, el túnel bajo el Canal de la Mancha, y grandes organizaciones como Microsoft o General Electric. De hecho aparecen en la naturaleza sistemas mucho más complejos aún, como el sistema circulatorio humano o la estructura de una

planta.

1.2 Historia de la Programación Orientada a Objetos.

En la actualidad la orientación a objetos se a colocado como la corriente principal de la computación, tanto para los creadores de software como para los usuarios, y se emplea a lo largo de una amplia gama de componentes del software, incluyendo lenguajes, interfaces de usuario, bases de datos y sistemas operativos. Aunque la Orientación a Objetos no es la solución absoluta, ya ha demostrado que puede ayudar a gestionar la creciente complejidad y costos del desarrollo del software.

Al estar integrada en los componentes fundamentales del software, ha significado una revolución tan grande como la que represento la implementación de la programación estructurada en la década de los 70: un nuevo paradigma para mejorar la creación, mantenimiento y empleo del software. En este nuevo paradigma, los *objetos* y las *clases* son los pilares, mientras que los métodos, los mensajes y la herencia producen los mecanismos primarios. Históricamente, la creación de un programa implicaba la definición de procesos que actuaban sobre un conjunto independiente de datos. La Orientación a objetos cambia el centro de atención del proceso de programación desde el procedimiento hasta los *objetos* – módulos auto contenidos que incluyen tanto los datos como los procedimientos que actúan sobre dichos datos-.

Simula67 y Smalltalk fueron relativamente inaccesibles a la corriente principal de la comunidad de las computadoras hasta los años 80. Por ejemplo, el trabajo inicial en Smalltalk no fue dado a conocer hasta el ejemplar de Agosto de 1981 de la revista *BYTE*. En los años 80 **C** se convirtió en un lenguaje de desarrollo muy popular, no sólo en las microcomputadoras sino en la mayoría de las arquitecturas y entornos de computación. Fue precisamente en esta década cuando Bjarne Stroustup de los Laboratorios Bell amplio el lenguaje **C** para crear **C++**, un lenguaje que soporta la programación orientada a objetos. Posteriores mejoras en herramientas y lanzamientos comerciales del lenguaje **C++**, justifican en buena parte el desarrollo general de la programación orientada a objetos, con

C++, los programadores eran capaces de aprender al paradigma de la orientación a objetos en un léxico popular y conocido sin tener que invertir en nuevos y diferentes entornos y lenguajes de programación.

La orientación a objetos proporciona una mejor forma de gestionar la complejidad tecnológica.

1.3 ¿Por qué aparece ahora la orientación a objetos?

Los obstáculos precio/rendimiento que evitaban el empleo general de la tecnología orientada a objetos también han caído en el camino. Las configuraciones de los potentes computadores personales de hoy en día han satisfecho las exigencias básicas de estaciones de trabajo de rendimiento mayor, presentaciones gráficas de calidad más alta, y soportan de forma completa entornos de desarrollo con multitud de herramientas; y la introducción de los entornos integrados orientados a objetos, específicamente el Macintosh en 1984 y la máquina NeXT en 1988, fueron hitos significativos en la introducción de la orientación a objetos en la corriente principal de la computación.

Sin embargo, la creciente complejidad tecnológica es el acelerador oculto para el uso del paradigma orientado a objetos. La orientación a objetos proporciona una forma mejor de gestionar la complejidad tecnológica. Los programadores que utilizan las herramientas actuales de desarrollo para crear aplicaciones indican que sin estratos o capas de abstracción, el desarrollo de aplicaciones se estrangula. La orientación a objetos permite la programación en niveles más altos de abstracción -desde el objeto a la clase, hasta la biblioteca de clases y finalmente a los completos marcos estructurales de aplicación.

Por último, la industria del software está deseosa de intentar un método mejor para el desarrollo de software. La industria del software ha llegado a estar atascada (algunos la han descrito como "tumbada en la playa") debido a la falta de procesos mejores. La orientación a objetos promete proporcionar a los desarrolladores de software comercial la capacidad para el desarrollo acelerado de programas, un codigo ampliable y productos que pueden comercializarse en una base mayor de clientes.

1.4 Ventajas de los lenguajes orientados a objetos.

Un lenguaje de programación que soporta el paradigma orientado a objetos beneficia al desarrollador de software proporcionando una forma natural de modelar el fenómeno del complejo mundo-real. Aunque el paradigma orientado a objetos de los objetos, métodos y mensajes es con frecuencia difícil de entender para los programadores acostumbrados a la metáfora tradicional de procedimientos y datos, la orientación a objetos ofrece realmente un modelo más natural del mundo real. Programar ya no significa solamente escribir líneas de código sino desarrollar modelos utilizando clases. Con la programación orientada a objetos, los programas tienen menos líneas de código, menos sentencias de bifurcación y módulos que son más comprensibles porque reflejan una relación unívoca entre el modelo objeto y el conceptual.

Las bibliotecas de clases predefinidas, un componente de los lenguajes orientados a objetos maduros, aumentan las ventajas de utilizar lenguajes orientados a objetos. Buena parte de la programación hecha con Smalltalk, por ejemplo, puede hacerse con objetos y mensajes que ya existen en la biblioteca Smalltalk. El trabajo de programación consiste en encontrar en la biblioteca Smalltalk los objetos y mensajes adecuados y combinarlos en el orden apropiado. Este empleo de una biblioteca de clases es similar al uso de las librerías de funciones, como las que se utilizan en C, pero hay una diferencia significativa. La biblioteca de funciones de C es relativamente fija. Con la librería Smalltalk y las bibliotecas que acompañan a otros lenguajes orientados a objetos, los nuevos miembros pueden crearse fácilmente por medio del empleo de la herencia.

El método general de reducir el código mediante la programación de las diferencias con herencia es una de las tácticas clave de la programación orientada a objetos y es una capacidad única de los lenguajes orientados a objetos.

1.5 Conceptos Básicos.

Mecanismos básicos.

Los mecanismos básicos de la orientación a objetos son los objetos, mensajes y los métodos, clases y variables instancia (o modelo) y herencia. Todos los sistemas que merecen la descripción de orientado a objetos contienen estos mecanismos esenciales, aunque los mecanismos pueden no estar realizados (o denominados) exactamente de la misma forma.

1.5.1 Objetos.

Se definirá un objeto como un concepto, abstracción o cosa con limites bien definidos y con significado a efectos del problema que se tenga entre manos. Los objetos tienen dos propósitos: promover la comprensión del mundo real y proporcionar una base practica para la implementación por computadora. La descomposición de un problema en objetos depende del juicio y de la naturaleza del problema. No existe una única representación correcta.

Un programa tradicional consta de procedimientos y datos Un programa orientado a objetos consta solamente de objetos que contienen tanto los procedimientos como los datos. Por decirlo de otra forma, los objetos son módulos que contienen los datos y las instrucciones que operan sobre esos datos. Así, dentro de los objetos residen los datos de los lenguajes convencionales, como por ejemplo, números, matrices (arrays o arreglos), cadenas de caracteres y registros, así como cualquier función, instrucción o subrutina que opere sobre ellos. Los objetos, por tanto, son entidades que tienen atributos (datos) y formas de comportamiento (procedimientos) particulares.

Los objetos llevan los nombres de los elementos de interés desde el dominio de la aplicación. Por ejemplo, en una aplicación de procesamiento de textos, es probable que un objeto sea llamado párrafo. En una aplicación de contabilidad, la hoja balance de junio es un objeto probable. Desde la perspectiva del usuario, los objetos proporcionan el comportamiento deseado. Un párrafo puede aceptar revisión y realinear sus márgenes. La hoja balance del mes de junio puede

imprimirse o consolidarse con las de otros meses para constituir un informe cuatrimestral. Desde la perspectiva del programador, los objetos son módulos de una aplicación que funcionan juntos para proporcionar una funcionalidad general.

Es importante señalar que todos los objetos poseen su propia identidad y se pueden distinguir entre si. Dos manzanas del mismo color, forma y textura siguen siendo manzanas individuales. El termino identidad significa que los objetos se distinguen por su existencia inherente y no por las propiedades descriptivas que puedan tener.

Las aplicaciones pueden constar de diferentes clases de objetos. Un objeto activo es aquel que se comprende su propio hilo de control, mientras que un objeto pasivo no. Los objetos activos suelen ser autónomos lo que quiere decir que pueden exhibir algún comportamiento sin que ningún otro objeto opere sobre ellos. Los objetos pasivos, por otra parte, solo pueden padecer un cambio de estado cuando se actúa explícitamente sobre ellos. De este modo los objetos activos de un sistema sirven como raíces de control.

1.5.2 Mensajes y Métodos.

En la mayoría de los lenguajes de programación orientados a objetos las operaciones que los clientes pueden realizar sobre un objeto suelen declararse como métodos, que forman parte de la declaración de la clase. El paso de mensajes es una de las partes de la ecuación que define el comportamiento de un objeto; la definición de comportamiento también recoge que el estado de un objeto afecta a si mismo a su comportamiento. A diferencia de los elementos de datos pasivos en los sistemas tradicionales, los objetos tienen la posibilidad de actuar. La acción sucede cuando un objeto recibe un mensaje, que es, una solicitud que pide al objeto que se comporte de alguna forma. Cuando se ejecutan los programas orientados a objetos, los objetos reciben, interpretan y responden a mensajes procedentes de los objetos. Por ejemplo, cuando un usuario solicita que un objeto llamado documento se imprima a sí mismo, el documento puede enviar un mensaje al objeto impresora solicitando un lugar en la cola de impresión; el objeto impresora puede devolver un mensaje al documento solicitando información

de formato, y así sucesivamente. Los mensajes pueden contener información para clarificar una solicitud; por ejemplo, el mensaje solicitando que un objeto se imprima a sí mismo podría incluir el nombre de la impresora. Finalmente, el emisor del mensaje no necesita conocer la forma en que el objeto receptor está llevando a cabo la solicitud. En otras palabras, cuando el objeto documento recibe el mensaje imprimir, documento sabe exactamente lo que tiene qué hacer. El objeto que envía el mensaje ni sabe ni le importa cómo se realiza la impresión, solamente conoce que está sucediendo.

El conjunto de mensajes al que un objeto puede responder se llama protocolo del objeto. El protocolo para un icono puede constar de mensajes invocados por la pulsación del botón del ratón cuando el usuario localiza un puntero sobre un icono.

Los métodos pueden enviar también mensajes a otros objetos solicitando acción o información.

Al igual que las "cajas negras" de la ingeniería, la estructura interior de un objeto está oculta a usuarios y programadores. Los mensajes que recibe el objeto son los únicos contactos que conectan al objeto con el mundo exterior. Solamente un método del propio objeto puede disponer de los datos del interior del mismo para su manipulación. Estas características de los objetos confieren a la orientación a objetos su ventaja: La orientación a objetos fomenta la modularidad haciendo muy claras las fronteras entre objetos, explicita la comunicación entre los mismos y ocultos los detalles de la realización.

Cuando se ejecuta un programa orientado a objetos, ocurren tres sucesos. En primer lugar, se crean los objetos cuando se necesitan. Segundo, los mensajes se mueven de un objeto a otro (o desde el usuario a un objeto) a medida que el programa procesa internamente información o responde a la entrada del usuario. Finalmente, se borran los objetos cuando ya no son necesarios y se recupera memoria.

1.5.3 Clases, subclases y objetos.

Muchos objetos diferentes pueden actuar de formas muy similares. Una clase es una abstracción que describe propiedades importantes para una aplicación y que ignora el resto. Una clase consta de métodos y datos que resumen las características comunes de un conjunto de objetos. La posibilidad de abstraer métodos y descripciones de datos comunes de un conjunto de objetos y almacenarlos en una clase es esencial para la potencia de la orientación a objetos . Definir clases significa situar código reutilizable en un depósito común en lugar de volver a expresarlo una vez y otra. En otras palabras, las clases contienen los anteproyectos para crear objetos. Finalmente, la definición de una clase ayuda a clarificar la definición de un objeto: un objeto es un modelo o instancia de una clase.

Los objetos se crean cuando se recibe un mensaje solicitando creación por la clase padre. El nuevo objeto toma sus métodos y datos de su clase padre. Los datos son de dos formas, variables de clase y variables modelo o de instancia. Las variables de clase tienen valores almacenados en una clase; las variables instancia tienen valores asociados únicamente con cada instancia u objeto creado a partir de una clase.

A título de ejemplo, considere cómo un programador podría designar una aplicación de procesamiento de textos en forma orientada a objetos. En primer lugar el programador identifica las entidades de interés. Los párrafos, por ejemplo, son objetos potenciales. Justificar es un método común para todos los párrafos. Tipodeletra (Fuente) es una variable de clase con el valor helvética. Finalmente, texto es una variable modelo con valores únicos para cada objeto. Es útil crear una clase llamada párrafo para guardar esta información común. La clase párrafo proporciona entonces un anteproyecto para la construcción de objetos. Aunque los datos específicos de cada objeto (ej.: las palabras del párrafo, el tipo de letra o fuente, el interlineado o espaciado) pueden variar, todos los objetos de la clase párrafo comparten métodos y variables de clase comunes.

Una clase puede también resumir elementos comunes para un conjunto de

subclases. En el caso del ejemplo del procesamiento de textos, el programador puede considerar después tabla para que sea otra clase además de párrafo. Pensando un poco más, el programador se da cuenta que párrafo y tabla comparten algunas propiedades con una clase más abstracta, texto. De forma similar, texto y gráfico pueden convertirse en subclases de una clase con carácter aún más general, documento ilustra esta jerarquía de las clases del procesamiento del documento. Utilizando subclases, los programadores orientados a objetos describen las aplicaciones como conjuntos de módulos generales o abstractos. Los métodos y datos comunes se elevan tan alto como sea posible de forma que sean accesibles a todas las subclases relacionadas.

A veces se denomina a las subclases como clases derivadas. En otras ocasiones los términos padre e hija se utilizan para indicar la relación entre una clase y una subclase. Las clases padre están localizadas por encima de las clases hijas en la jerarquía. Las clases más altas en la jerarquía se denominan las superclases.

Hasta ahora, esta descripción de un diseño orientado a objetos ha sido "ascendente". Es decir, la descripción pone el énfasis en la forma en que los componentes (objetos) son abstraídos para llegar a ser clases y superclases. De hecho, el término "descendente" ("top-down") describe con mayor precisión el método seguido por muchos programadores de la orientación a objetos. Normalmente comienzan su trabajo con una biblioteca de clases que contiene generalmente módulos útiles de programación. Utilizando clases predefinidas como punto inicial, los programadores escriben nuevas subclases que adaptan las clases de empleo general de la biblioteca a los requisitos funcionales particulares de la aplicación.

Una biblioteca de clases específicas para una aplicación se denomina un marco estructural "framework". Los marcos estructurales difieren de las bibliotecas de clases en su distinto grado: un marco estructural es una biblioteca de clases ajustada especialmente para una determinada categoría de aplicaciones, por ejemplo, un marco estructural constructor de interfaces. Construir y adaptar aplicaciones a partir de marcos estructurales es más rápido y fácil que empezar

con bibliotecas de clases genéricas. Asimismo, un marco estructural no será normalmente útil fuera del campo de la aplicación ya que contiene clases específicas para la aplicación.

1.5.4 Herencia.

La herencia es el mecanismo para compartir automáticamente métodos y datos entre clases, subclases y objetos. En términos generales se puede definir una clase que después se ira refinando sucesivamente para producir subclases. Todas las subclases poseen, o heredan, todas y cada una de las propiedades de su superclase y añaden además, sus propiedades exclusivas. No es necesario repetir las propiedades de las superclases en cada subclase. La herencia permite a los programadores crear nuevas clases programando solamente las diferencias con la clase padre. Cuando un programador declara que párrafo es una subclase de texto, por ejemplo, todos los métodos y variables modelo asociados con texto son heredados automáticamente por párrafo. Si la clase texto contiene métodos que son inadecuados para la subclase párrafo, entonces el programador puede obviar estos métodos escribiendo unos nuevos y almacenándolos como parte de la clase párrafo.

Debido a la herencia, los programas orientados a objetos constan de taxonomías, árboles o jerarquías de clases que, por medio de la sub clasificación, llegan a ser más específicas. Las clases proporcionan los anteproyectos para las subclases o para los objetos relacionados con una aplicación.

Herencia simple y múltiple son dos tipos de mecanismos de herencia utilizados normalmente en la programación orientada a objetos. Con la herencia simple, una subclase puede heredar datos y métodos de una clase simple así como añadir o sustraer comportamiento por sí misma. La herencia múltiple se refiere a la posibilidad de una subclase de adquirir los datos y métodos de más de una clase. La herencia múltiple es útil al construir comportamiento compuesto a partir de más de una rama de una jerarquía de clases.

En resumen, los mecanismos básicos de la orientación a objetos conducen a

una particular visión sobre el concepto de dar forma al mundo. Los elementos y su comportamiento se identifican como objetos. El comportamiento se realiza con métodos y datos almacenados en el objeto. Los mensajes obtienen el comportamiento de un objeto invocando un método del mismo.

Los objetos con métodos y variables modelo comunes se reúnen en una clase. Las clases se organizan en jerarquías y los mecanismos de herencia proporcionan automáticamente a cada subclase los métodos y datos de las clases padre. Las subclases se crean programando las diferencias entre las clases disponibles en una librería y los requisitos particulares de la aplicación.

Conceptos clave.

Los mecanismos básicos señalados anteriormente forman la base del paradigma de la orientación a objetos. Cuatro conceptos clave que resumen las ventajas del método orientado a objetos son la encapsulación, la abstracción, el polimorfismo y la persistencia.

1.5.5 Encapsulación.

La encapsulación (denominada también ocultamiento de información) consiste en separar los aspectos externos del objeto, a los cuales pueden acceder otros objetos, de los detalles internos de implementación del mismo, que quedan ocultos para los demás. La encapsulación evita que el programa llegue a ser tan interdependiente que un pequeño cambio tenga efectos secundarios masivos. La implementación de un objeto se puede modificar sin afectar a las aplicaciones que la utilizan. Quizá sea necesario modificar la implementación de un objeto para mejorar el rendimiento, corregir un error, consolidar el código o para hacer un transporte a otra plataforma.

La encapsulación no es exclusiva de la programación orientada a objetos pero la capacidad de combinar la estructura de datos y el comportamiento en una única entidad hace que la encapsulación sea aquí mas limpia y potente que en los lenguajes convencionales que separan las estructuras de datos y el comportamiento.

1.5.6 Abstracción.

La abstracción consiste en centrarse en los aspectos esenciales inherentes de una entidad, e ignorar sus propiedades accidentales. En el desarrollo de sistemas esto significa centrarse en lo que es y lo que hace un objeto antes de decidir como debería ser implementado. El uso de la abstracción mantiene nuestra libertad de tomar decisiones durante el mayor tiempo posible evitado comprometernos de forma prematura con ciertos detalles. La mayoría de los lenguajes modernos proporcionan abstracción de datos pero la capacidad de utilizar herencia y polimorfismo proporciona una potencia adicional. El uso de la abstracción durante el análisis significa tratar solamente conceptos del dominio de la aplicación y no tomar decisiones de diseño o de implementación antes de haber comprendido el problema. Un uso adecuado de la abstracción permite utilizar el mismo modelo para el análisis, diseño de alto nivel, estructura del programa, estructura de una base de datos y documentación. Un estilo de diseño independiente del lenguaje pospone los detalles de programación hasta la fase final, relativamente mecánica del desarrollo.

1.5.7 Polimorfismo.

Los objetos actúan en respuesta a los mensajes que reciben. El mismo mensaje puede originar acciones completamente diferentes al ser recibido por diferentes objetos. Este fenómeno se conoce como polimorfismo. Con el polimorfismo un usuario puede enviar un mensaje genérico y dejar los detalles exactos de la realización para el objeto receptor. El mensaje imprimir, por ejemplo, al ser enviado a una figura o diagrama invocará diferentes métodos de impresión que en el caso de enviar el mismo mensaje imprimir a un documento de texto.

El polimorfismo está fomentado por la maquinaria de la herencia. Es muy normal almacenar los protocolos para funciones de utilidad como "imprimir" en la posición más alta que sea posible dentro de la jerarquía de clases. Las variaciones necesarias en el comportamiento "imprimir" se almacenan en niveles más bajos de la jerarquía para sobrescribir los métodos más generales cuando sea necesario.

De esta forma, los objetos están listos y pueden responder apropiadamente a mensajes de utilidad como "imprimir" mientras que el método que realiza la función de impresión puede existir en la clase inmediata del objeto o varios niveles por encima de la clase del objeto.

1.5.8 Persistencia.

La persistencia se refiere a la permanencia de un objeto, es decir, al tiempo durante el cual se asigna espacio y permanece accesible en la memoria de la computadora. En la mayoría de los lenguajes orientados a objetos, se crean modelos de clases a medida que el programa se ejecuta. Algunos de estos modelos se necesitan solamente por un breve período de tiempo. Cuando un objeto ya no es necesario, es destruido y recuperado el espacio de memoria que tenía asignado. La recuperación automática del espacio de memoria se denomina normalmente recolección de basura.

Después de haber ejecutado un programa orientado a objetos, los objetos ensamblados normalmente no se almacenan; es decir, los objetos dejan de ser persistentes. Una base de datos orientada a objetos mantiene una distinción entre objetos creados solamente para el tiempo de duración de la ejecución y aquellos pensados para almacenamiento permanente. Los objetos almacenados permanentemente se denominan persistentes.

2. LA PROGRAMACION ORIENTADA A OBJETOS EN JAVA.

2.1 Las clases y objetos en Java.

Como en todo lenguaje Orientado a Objetos, en Java los componentes o módulos principales de la programación son las clases, a partir de las cuales podremos generar objetos. La declaración básica de una clase será:

Ejemplo.

```
public class nombreClase {

// Código de la clase

}
```

Donde:

`public`	El modificador de alcance que acompaña a la declaración de las clases, este es opcional, en caso de no estar presente se asignara un alcance por omisión.
`class`	La palabra reservada `class` es la parte fundamental de la declaración de cualquier clase, pues deberá de estar presente en todas.
`nombreClase`	El nombre con el cual se podrá hacer referencia a esta clase en el código.

Dentro de los corchetes presentes en la declaración anterior se incluirán todos los atributos y métodos que formaran la definición de nuestra clase. El archivo donde se almacene la clase deberá de ser nombrado igual que la clase.

Ejemplo.

```
public class HolaMundo
{
     public static void main (String Args[])
     {
     System.out.println(``Hola Mundo'');
     }
}
```

Para este ejemplo, la clase `HolaMundo` deberá de almacenarse en un archivo llamado **HolaMundo.java.** Para poder ejecutar un programa en Java, es necesario tener un método principal o **`main`** declarado. La sintaxis de declaración de ese método será como la del ejemplo anterior, donde:

`public`	El modificador de alcance del método, este siempre será el mismo: **`public`.**
`static`	La palabra reservada **`static`** deberá de estar siempre presente en la declaración de método **`main`**.
`void`	El tipo de retorno.
`String Args[]`	Los parámetros pasados desde la línea de comandos al programa, estos serán almacenados en una arreglo de cadenas.

2.2 Los modificadores de alcance.

El alcance de los métodos y atributos dentro de nuestras clases es de vital importancia para la buena programación en Java, el alcance o acceso de un miembro de una clase nos permite determinar desde que puntos de la aplicación en desarrollo serán una referencia valida:

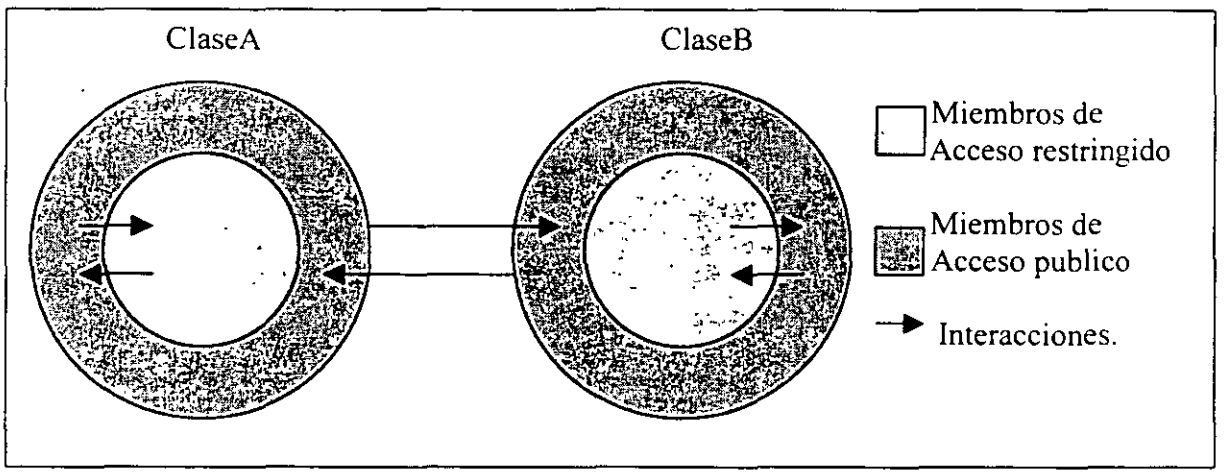


Fig. 2.1 Esquema de Alcance entre clases.

En la figura 2.1 se muestra de manera simplificada el funcionamiento del alcance, donde la interacción entre clases no es completa, y se realiza entre capas de diferente nivel de acceso. Para ello en Java se cuenta con cuatro niveles diferentes, cada uno asociado a un modificador o palabra reservada:

`public`	Representa el acceso máximo a los miembros de la clase, cualquier objeto podrá acceder a estos.
`protected`	Este es un nivel de acceso intermedio en el cual se da acceso a los miembros, a cualquier clase que herede de esta, a cualquier objeto o instancia de esta misma clase, y a cualquiera dentro del mismo paquete.

instancia de esta misma clase, y a cualquiera dentro del mismo paquete.

`private`	Este es el nivel mas restringido, solo se permitirá acceso a los objetos de la clase.
`--por omisión--`	Cuando no se califique el alcance de algún miembro o clase adquirirán por omisión, en este nivel cualquier objeto de la clase, y cualquiera dentro del mismo paquete.

Además de los modificadores de alcance cabe recordar que las variables definidas dentro de los métodos son propias de ellos y no forman parte de los atributos de la clase, pues solo existirán al momento de ejecución de esos métodos.

2.3 Atributos y métodos.

Como ya se ha visto, los miembros que conforman a una clase pueden ser de dos tipos, atributos y métodos, los primeros representaran los datos almacenados dentro de los objetos de la clase, estos se declaran dentro de la clase que los contiene , indicando el tipo de dato y el alcance que se tendrá sobre el:

Ejemplo.

```
public String miCadena;
```

Donde:

`public`	El modificador de alcance del atributo, este podrá ser remplazado por algún otro, o no estar presente, caso en el cual se asigna un alcance por omisión.
`String`	El tipo de dato que corresponde al atributo, este podrá ser cualquier tipo de dato primitivo de dato, o cualquier clase o interfaz al alcance de la clase a la que pertenece el atributo.
`miCadena`	Este será el nombre con el que se podrá referencial al atributo.

Todos los métodos que formen parte de nuestra clase se declaran dentro del código de la misma de manera similar a la declaración del método **`main`**, es decir tendrán una declaración y su definición se incluirá entre corchetes, esta definición no necesariamente requerirá de código, basta con la presencia de corchetes a continuación de la declaración, para considerar que el método ha sido definido :

Ejemplo.

```
public int suma(int a, int b)
{
   return a+b;
}
```

Como se puede apreciar en este ejemplo, el esquema de declaración de métodos es muy similar al mostrado anteriormente en el caso del método **`main`**, aunque es posible establecer diferentes tipos de retorno para nuestros métodos, serán tipos validos con este fin, todos los tipos de datos primitivos del lenguaje,

así como todas las clases e interfaces dentro al alcance de aquella donde se define el método, este mismo criterio se aplica en los tipos de parámetros recibidos por el método. Como en este ejemplo el tipo de retorno de la función es diferente de `void` se hace necesaria la presencia de una sentencia `return` para regresar el resultado esperado del método, y el control del programa a quien lo invoco.

2.4 Miembros estáticos o de clase.

Existen algunos otros modificadores diferentes de los de alcance que pueden ser utilizados en la declaración tanto de atributos como de métodos, uno de estos es `static` visto con anterioridad, este modificador permite declarar miembros que existirán de manera única para todos los ejemplares de la clase, para entender esto resulta interesante aclarar el esquema de funcionamiento delos miembros no estáticos de una clase:

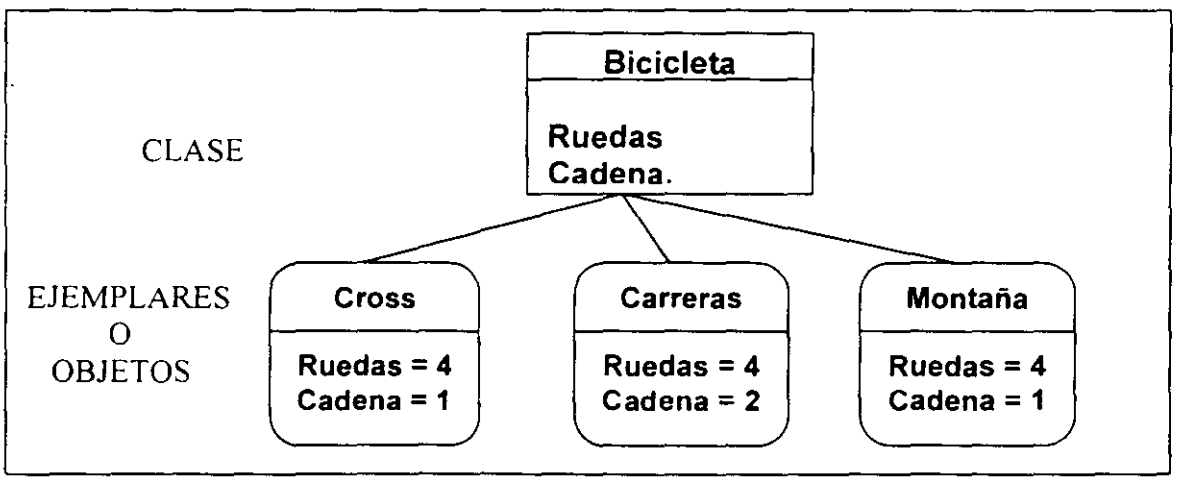


Fig. 2.2 Miembros no estáticos de una clase.

En el diagrama anterior es posible apreciar que cada objeto de la clase Bicicleta tiene sus propios atributos, con valores diferentes, pues en realidad existen como entes independientes y diferentes para cada objeto de la clase, no así cuando se trate de miembros estáticos:

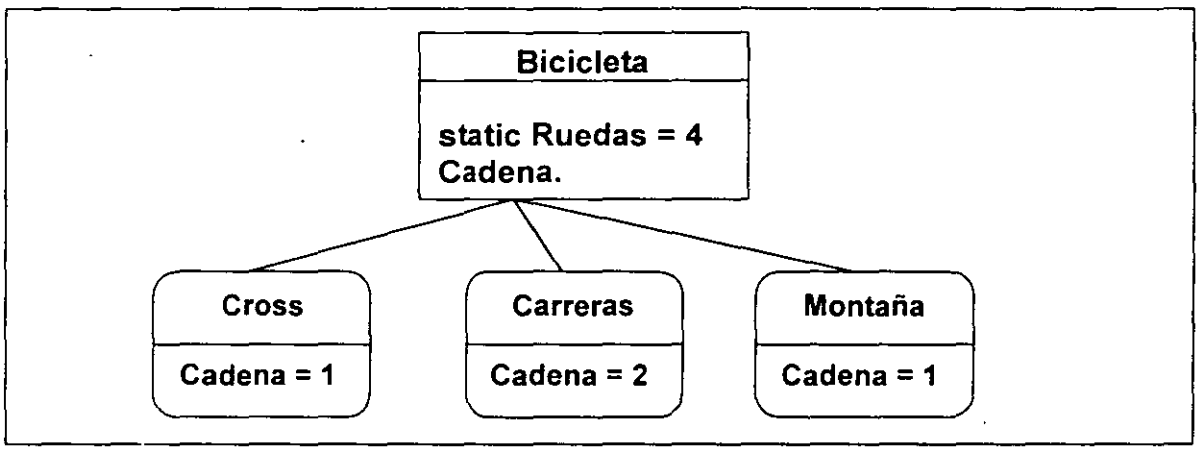


Fig. 2.3 Clase con miembros estáticos.

Como puede apreciarse en la figura 2.3 para el caso de los miembros estáticos, existe una sola copia para todos los objetos o ejemplares de la clase, esta podrá ser modificada por cualquiera de estos, y este cambio afectara a todos los demás, por otra parte cabe aclarar que los atributos estáticos solo podrán ser modificados por métodos también declarados como estáticos dentro de la clase, como puede suponerse, el método `main` es uno de estos. A continuación se ejemplifica cada una de las declaraciones estáticas posibles:

Ejemplo.

```
public class Auto
{
  public static int llantas;
  public static int cuentaLlantas()
  {
    return llantas;
  }
```

```
}
```

En este ejemplo se ha declarado al atributo `llantas` como **`static`** por lo tanto el método `cuentaLlantas` necesita ser declarado como **`static`** también para poder hacer referencia a el.

2.5 Constructores.

Mas halla de las clases que ejecutan un programa (es decir aquellas con método **`main`**.) será necesario ejemplarizar, crear una instancia, es decir, crear un objeto, de alguna otra clase, para esto se utilizaran los métodos constructores, estos nos permiten asignar valores a los atributos de un objeto, o tomar los determinados por omisión dependiendo el tipo de constructor, generando una referencia al nuevo objeto, estos constructores invariablemente llevaran por nombre el de la clase a la que pertenecen. Los constructores se dividen principalmente en dos tipos:

- El que proporciona automáticamente la definición de la clase, que no recibe ningún parámetro, y solo genera la referencia al objeto.

Ejemplo.

```
Auto miAuto = new Auto();
```

Retomando la clase `Auto` del ejemplo anterior generamos en este una instancia, usando un método que no habíamos definido, que lleva el mismo nombre de la clase y que en conjunto con el operador **`new`** completa la sintaxis para la creación de un objeto.

- Los definidos por el usuario, que pueden inicializar diferentes atributos, del objeto con valores pasados como parámetros al método.

Ejemplo.

Ampliando la definición de la clase `Auto` añadimos el siguiente código:

```
public Auto(int numLlantas)
{
llantas = numLlantas
}
```

podríamos crear un objeto de la siguiente forma.

```
Auto miAuto = new Auto(4);
```

Donde al coincidir el parámetro enviado a `Auto()` con el del constructor recién definido, se utilizara este para generar la referencia al objeto, e inicializar el valor de su atributo `llantas` a 4.

Es importante aclarar que una vez que se ha definido un constructor para la clase, no será posible continuar usando el constructor sin parámetros por omisión sin definirlo también de manera explicita dentro de la clase, esto será tan simple como:

```
public Auto(){ }
```

Con la inclusión de esta línea en la clase `Auto`, podremos continuar usando el constructor sin parámetros.

3. EL LENGUAJE JAVA

En este capitulo veremos algunos de los conceptos mas importantes para una correcta programación en JAVA.

3.1 Comentarios.

Muchas veces como programadores es necesario, escribir algunas líneas extras en el código, pero que no pertenezcan a este, como guías, o bien en el caso especifico de JAVA los textos que se agregaran en la documentación, esto debe ir "comentariado" para que el compilador entienda que ese texto no debe codificarlo, para escribir estos comentarios debemos seguir cierta sintaxis.

```
/* comentario
será de mas de
una línea */
// este es un comentario de una sola línea
/** este es un comentario
para la documentación que
creara javadoc multilínea */
```

3.2 Palabras reservadas

Resulta de gran importancia que cuando se programa no se empleen palabras que el compilador de JAVA ya tiene asignadas para ciertas operaciones entre las palabras reservadas se encuentran:

abstract	class	extends	implements	long
bolean	const	final	import	native
break	continue	finally	int	new
byete	default	float	interface	private
case	do	for	protected	static
catch	double	goto	transient	super
char	else	if	volatile	switch

3.3 Tipos de datos primitivos y valores literales.

JAVA define ocho tipos de datos primitivos. Las variables que se declaran como tipos primitivos son los únicos datos en JAVA que no son objetos. Solo son contenedores que sirven para almacenar valores primitivos. Los ocho tipos primitivos son `byte, short, int, long, float, double, char y boolean.`

Los tipos `byte, short, int y long` representan valores enteros de 8,16,32 y 64 bits. Los valores literales de estos tipos están escritos por medio de enteros decimales, hexadecimales u octetos positivos o negativos, los valores hexadecimales vienen precedidos por 0x o 0X y utilizan las letras que van de la a a la f para representar los digitos del 10 al 15. Los números octetos estan precedidos por un 0. Los valores decimales largos tienen una l o L anexos al final del número.

Los tipos `float y double` representan números de punto flotante de 32 y 64 bits. Los números `float` poseen el sufijo f o F. Los números double tienen d o D. Si no hay sufijo, se asume el tipo `double` predeterminado.

El tipo `char` representa caracteres Unicode de 16 bits. El código Unicode es un superconjunto de 16 bits del conjunto de caracteres ASCII que proporciona muchos caracteres de lenguajes extranjeros. Un solo carácter se especifica colocándolo entre comillas simples ('). Hay tres excepciones: comilla simple ('), comillas dobles (") y diagonal invertida (\ ). El carácter de diagonal invertida se emplea como parte del código para representar valores de caracteres especiales. Como lo muestra la siguiente tabla.

Código de cambio	Carácter
\b	Retroceso
\t	Tabulación
\n	Nueva línea
\f	Nuevo formulario
\r	Retorno de carro

\"	Comillas dobles
\'	Comilla simple
\\	Barras invertidas

Tabla 3.1 caracteres especiales en JAVA

3.4 Declaración de variables

Debido a que JAVA es un lenguaje fuertemente tipificado es necesario que cada vez que se va a emplear alguna variable esta debe ser declarada de manera correcta para que no se creen conflictos de alcance, nombre o tipo. Las variables pueden ser empleadas para hacer referencia a objetos o bien a tipos de datos primitivos. Y su declaración es como sigue:

```
Modificador_de_alcance tipo nombre extendido
nombrevalor_inicial
```

Los modificadores de variables y la inicialización son opcionales. Un tipo de variable puede ser un tipo de datos primitivos, un tipo de clase o un tipo de interfaz. El nombre extendido de las variables es un nombre de variables seguido de un cero o más conjuntos de paréntesis ([]), los cuales indican que la variable es un arreglo.

La inicialización de la variable está formada por un signo de igual (=) seguido de una expresión que arroje un valor del tipo de la variable. Si la variable es un arreglo, se inicializa colocando los elementos entres llaves {} y separados por comas.

3.5 ESTRUCTURAS DE COTROL

IF

La instrucción `if` se emplea cuando tenemos que seleccionar entre dos caminos para continuar la ejecución de nuestro programa. Y se escribe como sigue:

```
If (expresión booleana)
{
  instrucciones a ejecutarse si la
  condición es verdadera
}
```

otra forma de escribir esta instrucción es:

```
If (expresión booleana)
{
   instrucciones si la
   condición es verdadera
}
else
{
   instrucciones si la
   condición es falsa
}
```

SWITCH

La instrucción **switch** es muy parecida a la instrucción **if** solo que es recomendable su uso para el caso en el que se tienen mas de dos caminos a elegir.

Su sintaxis es:

```
Switch (expresion)
Case valor:
  Sentencias
Case valor:
  Sentencias
Default:
  Sentencias
```

BREAK

La instrucción **break** se usa para transferir el control a la siguiente instrucción, es decir sirve para romper estructuras de control.

```
break;
```

FOR

La instrucción **for** se emplea para realizar una o varias instrucciones repetidas veces en este caso es necesario saber el numero exacto de veces que se va a repetir el bloque.

```
for (condición booleana)
{
    instrucciones
}
```

WHILE

La instrucción **while** se emplea para ejecutar un conjunto de instrucciones mientras que una condición sea verdadera.

```
while (condición booleana)
{
   instrucciones
}
```

DO

La instrucción **do** es muy parecida a la instrucción **while**, es decir se ejecuta mientras una condición booleana sea verdadera, la única diferencia es que el bloque de instrucciones se ejecuta una vez antes de verificar si la condición es verdadera o no.

```
do
{
     instrucciones
}while (expresión booleana);
```

CONTINUE

Esta instrucción se emplea cuando deseamos continuar la ejecución de un bucle (**for, while, do**) sin que se haya terminado aun con la ejecución del bloque definido en el.

```
continue;
```

RETURN

Cuando llamamos a algún método algunas veces necesitamos que este nos "devuelva" un valor obtenido en su ejecución para enviar este valor a quien lo llamo, se emplea la instrucción **return**.

```
return expresión;
```

OPERADORES

JAVA define operadores aritméticos, relacionales, lógicos, de clase, de selección, de asignación, entre otros. La siguiente tabla contiene los operadores existentes en JAVA.

Tipo de operador	Operador	Descripción	Ejemplo
Aritmético	+	suma	a+b
	-	resta	a-b
	*	multiplicación	a*b
	/	división	a/b
	%	módulo	a%b
Relacional	>	mayor que	a>b
	<	menor que	a<b
	>=	mayor o igual que	a>=b
	<=	menor o igual que	a<=b

	!=	no igual	a!=b
	==	igual	a==b
Lógico	!	Not	!a
	&&	And	a&&b
	||	Or	a || b
Manipulación de bits	~	complemento	~a
	&	AND	a & b
	|	OR	a | b
	^	Exclusivo o	a ^ b
	<<	Desplazamiento a la Izquierda	a <<b
	>>	Desplazamiento a la Derecha	a>>b
	>>>	Desplazamiento a la derecha con relleno cero	a>>>b
Asignación	=	Asignación	a=b
	++	Incremento y asignación	a++
	--	Decremento y asignación	a--
	+=	Suma y asignación	a+=b
	-=	Resta y asignación	a-=b
	*=	Multiplicación y asignación	a*=b
	%=	Modulo y asignación	a%=b
	|=	Or y asignación	a|=b
	&=	And y asignación	a&=b
	^=	XOR y asignación	a^=b
Ejemplo	`instanceof`	¿es ejemplo de clase?	a `instance of` b

Tabla 3 2 Operadores de JAVA

4. HERENCIA.

4.1 La herencia en Java.

En Java la herencia se implementa de manera simple, es decir que una clase solo puede heredar de otra y no mas, por lo tanto las jerarquías de clases en java siempre lucirán así:

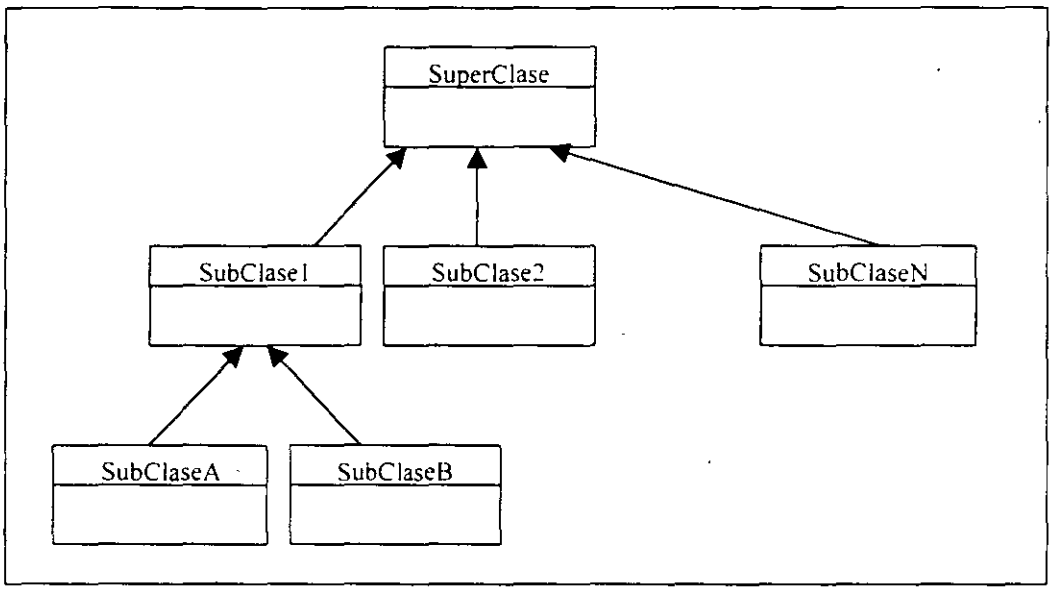


Fig. 4.1 Árbol de herencia en Java.

En la figura superior se utiliza una de las representaciones típicas de las jerarquías de herencia; los árboles, en este caso es posible notar que cada nodo o clase, hereda solo de una clase del nivel superior, este esquema simplifica la implementación de la herencia en Java y la hace mucho mas accesible en comparación de otros lenguajes de POO como C++, cabe destacar que existe la posibilidad de emular en cierto grado el concepto de herencia múltiple, esto con el uso de interfaces, herramientas que se verán mas adelante.

4.2 La palabra reservada `extends`.

En Java la herencia se realiza mediante la palabra reservada **`extends`**, que se incorporara a la declaración de las clases que hereden de otras, como en el siguiente ejemplo:

Ejemplo.

```
public class AutoCarreras extends Auto{}
```

En el ejemplo generamos una clase hija de `Auto`, la cual heredara (o derivara) todos sus atributos y métodos **`public`**, **`protected`**, y amigables (calificados en su alcance por omisión). Estos miembros podrán ser llamados de forma normal a como se hace con cualquier miembro declarado en la clase, mas es importante recalcar el hecho de que los métodos y atributos existentes en una superclase, pueden ser sobrescritos por la subclase, es decir se pueden agregar nuevas definiciones a los métodos, y se pueden asignar nuevos tipos a los atributos:

Ejemplo.

Considérense las siguientes clases:

```
class FigGeometrica
{
     public int Aristas;
}

class Circulo extends FigGeometrica
{
```

```
        public String Aristas;

        public void numero(String numero)
        {
              System.out.println(numero);
        }
}

public class EjemploHerencia
{
        public static void main(String hug[])
        {
              Circulo miCirculo = new Circulo();
              miCirculo.numero(miCirculo.Aristas);
        }
}
```

Al ejecutar el ejemplo anterior (el archivo `EjemploHerencia.class`) obtendríamos como resultado el valor de **`null`** que sería para el caso el valor Inicial de la cadena `Aristas`, mas no del atributo entero de la superclase también de nombre `Aristas` Pos supuesto que será posible acceder a los miembros de la superclase, esto mediante la palabra reservada **`super.`** y a continuación el nombre del miembro que se desee invocar:

Ejemplo.

```
        super.Aristas;
```

En la sentencia del ejemplo anterior se hace referencia al atributo entero `Aristas` de la superclase `FigGeometrica`. En el caso de los métodos se podrá proceder de la misma forma, esto resulta de utilidad, sobre todo en el caso de los constructores, o cuando sea necesario en una subclase ampliar el funcionamiento de un método ya definido.

4.3 Clases Abstractas.

Una clase Abstracta es aquella donde no se definen todos los métodos que se declaran, por ejemplo:

```
public abstract class Sumadora
 {
  public int Suma(int a, int b);
 }
```

En la clase `Sumadora` el método `Suma` ha sido declarado pero no definido, pues en lugar de los corchetes, se ha terminado la declaración con punto y coma, esto convierte automáticamente a la clase en abstracta, por lo que deberemos de incluir la palabra reservada **`abstract`** en la cabecera. La definición de los métodos declarados en una clase abstracta quedara como obligación para las clases que hereden de `Sumadora` , en el caso de que sea necesario instanciarlas, pues de las clases abstractas no se pueden generar objetos, solo son útiles para definir características generales en una jerarquia de herencia.

Ejemplo.

```
public  class Calculador extends Sumadora
```

```
{
  public int Suma(int a, int b)
  {
   return a+b;
  }
}
```

En el ejemplo anterior puede apreciarse el procedimiento adecuado para heredar una clase abstracta, don de se realiza una definición para el método declarado en la superclase `Sumadora`, de no haberse hecho esto así la palabra reservada **`abstract`** de agregarse a la definición de la clase.

5. TRABAJANDO CON PAQUETES

Como se ha comentado en capitulos anteriores una de las ventajas de JAVA es que nos permite reutilizar código, cuando tenemos grandes cantidades de este, ya sea del que la API nos proporciona o que nosotros mismos desarrollemos es fácil ver que este debe estar ordenado de alguna forma que nos sea fácil recordar para que sirve y como se llama, es por esto que JAVA nos brinda la facilidad de contar con los paquetes.

Un paquete es un "conjunto" de clases, interfaces, y declaraciones que se encuentran agrupadas de acuerdo a su utilidad. Por ejemplo si tenemos una clase que nos sirve para definir un cuadrado, otra para definir una circunferencia, una interfaz para calcular áreas de cuadrados, otra para calcular perímetros de circunferencias, y las variables P y A, entonces es fácil ver que podemos agrupar todas estos elementos en *paquete* para facilitar su futura implementación.

Una esquematización de lo antes dicho podría verse así:

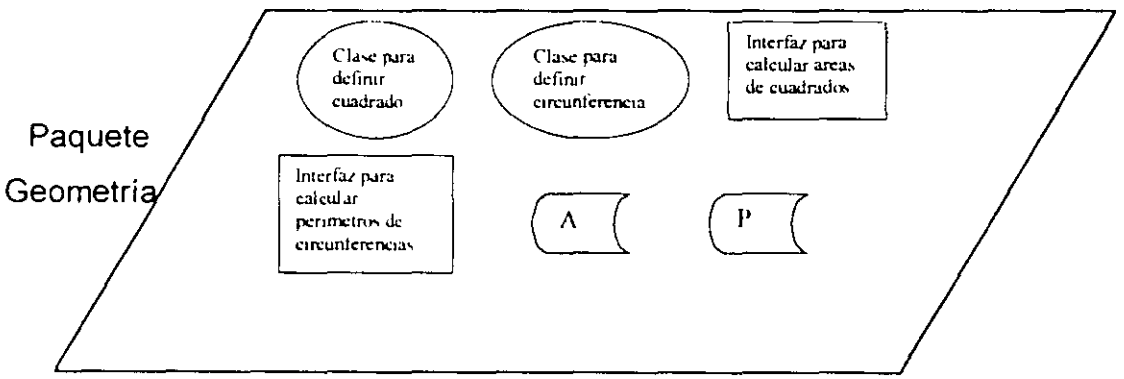


Figura 5 1 Representación gráfica de un paquete

Ahora la siguiente pregunta sería ¿cómo hacer para incluir todas estos elementos en nuestro paquete? La respuesta es muy sencilla lo único que hay que hacer es, al principio de cada código de los elementos que deseamos sean parte del paquete hay que agregar una línea.

```
package nombre_del_paquete;
```

y con esta línea ya se sabe que deseamos agregar ese código a un paquete. El paquete se crea automáticamente, es decir no hay necesidad de declararlo de ninguna manera, con la sentencia anterior se genera y ya es posible tener acceso a el, verlo desde nuestro árbol en la documentación y trabajar con el tal como trabajaríamos con los paquetes que están incluidos en el JDK y que podemos conocer en el API.

El siguiente punto por aclarar sería ¿como incluir un elemento de un paquete o bien el paquete completo en un programa de JAVA? Esto es algo que ya hemos venido haciendo desde el nuestros primeros programas.

Sentencia import.

Existen tres formas de trabajar con la sentencia import, dependiendo de la parte de cada paquete que deseamos emplear en nuestro código.

Por ejemplo con la siguiente sentencia estaríamos indicando que nuestro paquete Geometría vamos a emplear la clase cuadrado:

```
import Geometria.cuadrado;
```

Con la siguiente indicamos que emplearemos la interfaz Perimetro de circuenferencia:

```
import Geometria.perimetro_circunferencia;
```

Y con la siguiente indicamos que vamos a hacer referencia a todo el paquete.

```
import Geometria.*;
```

para nuestro tercer caso no es necesario que se emplee todo el paquete es posible que lo indiquemos así para no hacer referencia a cada uno de los elementos que vamos a emplear de el, que pueden ser desde cero hasta todos

siempre será recomendable que se agreguen, en la medida de lo posible los elementos exactos que se van a emplear ya que esto hace mas "ligero" nuestro código y por lo tanto mas funcional.

Como mencionábamos anteriormente el JDK ya nos proporciona algunos paquetes definidos para trabajar con ellos, de los cuales mas adelante trabajaremos detalladamente algunos de ellos. De momento solo cabe mencionar que entre los mas populares están los siguientes:

PAQUETE	DESCRIPCIÓN
java. applet	Provee las clases necesarias para crear un applet y las clases necesarias para que el applet interactué con su contexto.
java.awt	Provee las clases necesarias para crear interfaces de usuario y para dibujar imágenes y gráficos.
java.io	Provee las entradas y salidas del sistema a través de flujos de datos y los archivos del sistema
java.net	Provee las clases necesarias para crear aplicaciones para la red.
java.lang	Contiene la definición básica del lenguaje es importado por omisión a cualquier aplicación
Java.util	Contiene algunas clases de utilidad general, como lo son fecha, hora y algunas estructuras de datos.

Tabla 5.1 Algunos paquetes del API de JAVA

6. INTERFACES.

6.1 La herencia múltiple en Java.

Como ya se ha explicado antes, en Java la herencia se encuentra implementada de manera simple, sin embargo existe una herramienta que nos permite establecer características similares a las de la herencia entre clases, pero de forma múltiple; las interfaces. El concepto de interfaz es similar al de la clase abstracta, la interfaz es un conjunto de declaraciones de métodos y constantes, que definirán todo un protocolo a las clases que implementen la interfaz, pues como ocurría en el caso de las clases abstractas estas se verán obligadas a definirlos, y por lo tanto todas ellas contaran con un comportamiento común, aun en el caso de que provengan de diferentes árboles de herencia. En el caso de la implementación de interfaces. Una jerarquía de clases seria del tipo:

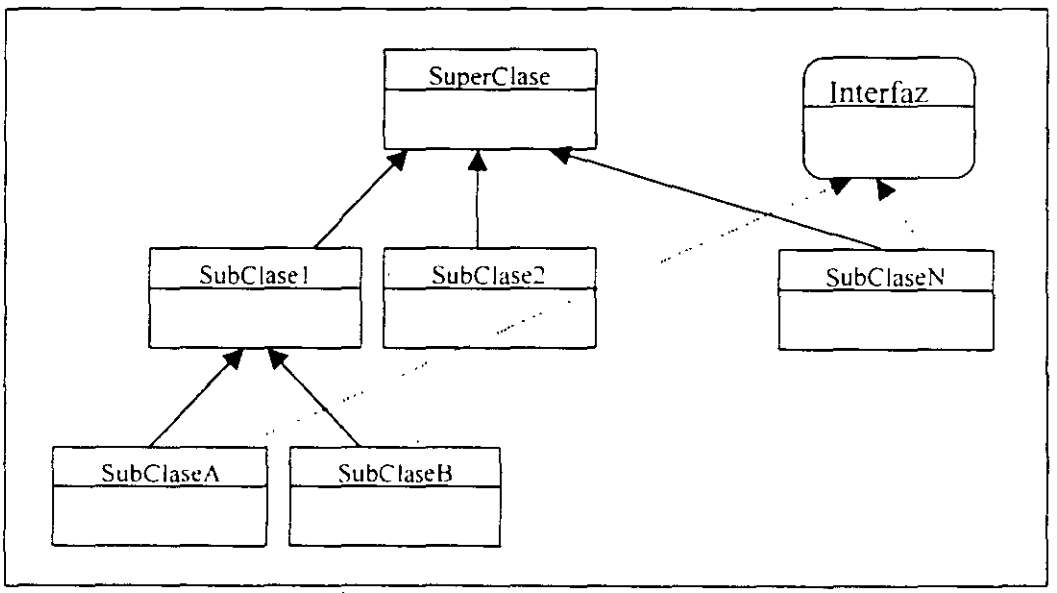


Fig. 6.1 Árbol de herencia que implementa interfaces.

En la figura se muestra un árbol de herencia, donde dos clases de niveles y ramas diferentes de la jerarquía, implementan la misma interfaz, por lo tanto compartirán los métodos obtenidos de esta.

6.2 Declaración de interfaces.

La declaración de interfaces e un proceso análogo al revisado en el caso de la declaración de clases, puesto que sus estructuras serán muy similares:

Ejemplo.

```
public interface Negociante
{
    public static final int jornada = 8;

    public void compra(int cantidad);
    public void vende(int cantidad);
}
```

En el ejemplo se muestra la declaración típica de una interfaz donde `interface` indica que se trata de tal, también se encuentra presente un modificador de alcance, este solo podrá ser `public`, o no encontrarse, para que la interfaz sea de acceso amigable, en la definición de ella, lo primero que encontramos es la declaración de una constante:

```
public static final int jornada = 8;
```

Dentro de una interfaz no puede haber variables, por lo tanto cualquier atributo que sea declarado en ella será tratado como constante, incluso si se omiten los modificadores necesarios `public static final`. Como es de suponerse, todas estas constantes deben de ser definidas en su declaración. Continuando con la interfaz, hallamos la declaración de dos métodos:

```
public void compra(int cantidad);
```

```
public void vende(int cantidad);
```

Estos dos métodos no cuentan con definición alguna, pero se especifica en la declaración cual es su entrada y su salida, por lo tanto establecen constituyen un ejemplo practico de encapsulamiento en Java.

6.3 Implementación de interfaces.

Para implementar una o mas interfaces en una clase se hará uso de la palabra reservada **extends** como se muestra a continuación:

Ejemplo.

```
public class vendedor implements Negociante
{
     private int Almacen;

     public void compra(int cantidad){}

     public void vende(int cantidad)
     {
     Almacen -= cantidad;
     }
}
```

La clase `vendedor` implementa la interfaz `Negociante` construida antes en este capitulo, y por lo tanto define todos sus métodos, aunque en realidad el único que utiliza es `vende` . Por supuesto que el uso de las interfaces no esta limitado a la implementación de una sola por clase, nosotros podremos incluir cualquier número de estas:

```
class nombreClase implements Interfaz1,Interfaz2,.. InterfazN
{
}
```

Por supuesto que esto presenta cierto inconvenientes, pues será necesario definir todos los métodos de todas las interfaces.

6.4 La herencia de interfaces.

Así como es posible derivar clases de otras, es posible derivar interfaces de otras, es decir heredar todas las constantes y métodos declarados en una a otra, por ejemplo para ampliar su definición. Este proceso es análogo al realizado en el caso de las clases por lo tanto no se profundizara en el. La sintaxis se muestra en el siguiente ejemplo:

Ejemplo.

```
public interface AgenteViajero extends Negociante
{
    public static final int Kilometro = 1000;

    public void Recorrido(int Distancia);

}
```

Cualquier clase que implemente la interfaz `AgenteViajero` deberá además de definir el método `Recorrido` , definir `compra` y `vende`.

6.5 Las interfaces como tipos de datos.

Como se ha visto hasta ahora, se pueden utilizar como tipos de datos, no solo los primitivos del lenguaje, si no también las clases, bien, pues a estos habrá que añadir las interfaces que podrán ser utilizadas de manera casi transparente con este fin. Casi transparente por que hay que aclarar que como tal la interfaz no puede ser pasada o recibida por método alguno, pues no se trata de un dato si no la abstracción de un comportamiento, cuando se utilice como dato una interfaz lo que en realidad se espera es el ejemplar de una clase que se comporte como se encuentre definido en ella, es decir que implemente dicha interfaz.

Ejemplo.

```
public void calculaGanancia(Negociante Trabajador)
{
.........
}
```

Para el ejemplo anterior la función puede recibir cualquier ejemplar de una clase que implemente la interfaz o herede de una que lo haga, como por ejemplo de `vendedor`, y por los efectos de la herencia también cualquier ejemplar de una clase que implemente a `AgenteViajero`.

7. Manejo de AWT y eventos

Una de las razones del atractivo tan grande que tiene JAVA se halla en el Abstract Window Toolkit (AWT). Esta API permite desarrollo potentes interfaces gráficas de usuario fácil y rápidamente. Además, la mayoría de sus clases e interfaces se pueden utilizar para applets y aplicaciones.

Algo que cabe destacar de AWT es que para "dibujar" sus elementos en la aplicación, toma los elementos que tenga el sistema operativo sobre el que se este ejecutando nuestra aplicación, es así como podemos notar que el estilo de nuestros elementos cambia de, por ejemplo, Windows a Unix. Aún cuando sea el mismo código fuente.

Actualmente JAVA proporciona un nuevo paquete ***swing*** , que nos permite crear aplicaciones totalmente independientes al sistema.

Ahora comenzaremos a analizar cada uno de los componentes básicos de una interfaz de usuario.

7.1 Clases de la interfaz de usuario.

etiqueta

Este elemento sirve para colocar texto en cualquier parte de nuestra interfaz de usuario. Para hacer uso de ella como de cualquier otro elemento, se requiere "construirla", para ello tenemos 3 diferentes constructores.

`Label()` .- crea una etiqueta sin rotulo.

`Label("cadena")` - crea una etiqueta, y le coloca el rotulo que le enviemos como parámetro

`Label("cadena",posición)` .- crea una etiqueta, con el rotulo que se le envié como primer parámetro, y en el segundo parámetro se le envía su posición, derecha, izquierda, centro.

Además de sus constructores este objeto cuenta con algunos métodos, propios aunque en su mayoría los hereda de la clase ***component.***

```
Label Saludo= new Label("Hola mundo!");
```

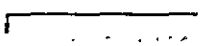
Este elemento tiene 2 constructores.

Button().- crea un botón sin etiqueta.

Button("cadena").- crea un botón con la etiqueta que se envía como parámetro.

Además de sus constructores cuenta con sus métodos para cambiar desde su etiqueta hasta el ***listener*** que mas adelante veremos que es.

```
Button Presioname= new Button("presiona!");
```



Este elemento, caja de texto, cuenta con 4 constructores.

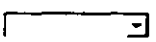
TextField().- crea un campo de texto sin contenido.

TextField(numero de columnas).- crea un campo de texto con un tamaño tomado del parámetro que recibe.

TextField("texto").- crea un campo de texto que tiene como contenido el parámetro enviado.

TextField("texto",columnas).- este constructor es una combinación del segundo y tercero.

```
TextField Mi_caja= new TextField("introduce tu nombre",25);
```



El combo tiene un único constructor.

Choice().- con esta sentencia se crea un combo sin elementos.

Para agregar cada elemento se emplea su método **add("cadena")**.

```
Choice Mi_combo= new Choice();
Mi_combo.add("helado");
Mi_combo.add("pay");
```

La lista cuenta con tres constructores diferentes.

List().- Crea una lista vacía y sin tamaño fijo, además de que permite una selección a la vez.

List(renglones).- Crea una lista vacía en la que se especifica el número de elementos que contendrá, y solamente permitirá una selección a la vez.

List(renglones, multiselección).- Crea una lista en la que se le indica el número de elementos que contendrá, así como si va a aceptar o no mas de una selección a la vez.

```
List lista = new List(4, false);
lista.add("The Beatles");
lista.add("Bethoven");
lista.add("Gandhi");
```

El checkbox cuenta con 5 constructores.

Checkbox().- Crea un checkbox sin etiqueta.

Checkbox("texto").- Crea un checkbox con la etiqueta enviada.

Checkbox("texto", estado).- Crea un checkbox con la etiqueta enviada, y seleccionada o no, según se indique.

Checkbox("texto", estado, grupo).- Crea un checkbox con la etiqueta recibida, y con el estado definido, así como el grupo al que pertenece para mayor orden.

Checkbox("texto",grupo, estado).- Funciona igual que el constructor anterior.

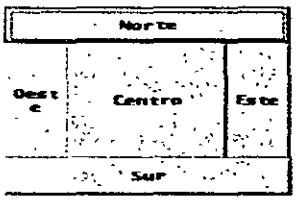
```
Checkbox Seleccion= new Checkbox("libros", null, true)
```

Este elemento se trabaja por grupos, y tiene un único constructor.

Checkgroup().- y con el se genera un grupo de checkbox que únicamente permitirá seleccionar una opción a la vez.

Cada elemento se agrega con la sentencia vista anteriormente, **Checkbox.**

```
CheckboxGroup grupo = new CheckboxGroup();
add(new Checkbox("uno", grupo, true));
add(new Checkbox("dos", grupo, false));
add(new Checkbox("tres", grupo, false));
```



El último elemento, de los que aquí trataremos, será el **BorderLayout.** Que será, algo así como "el contenedor" donde acomodaremos nuestros elementos. Este elemento solo cuenta con 2 constructores.

BorderLayout().- Crea un BorderLayout sin dejar espacios entre sus elementos

BorderLayout(distancia x, distancia y).- Crea un BorderLayout con los espacios entre los elementos que se le indique.

Las clases anteriores aunque no son todos los que maneja el paquete AWT, son las mas significativas, sobre la marcha se irán tratando algunas mas.

7.2 Eventos

Cuando nosotros interactuamos con una ventana, lo hacemos mediante eventos, como un clic o pulsar una tecla, de esta manera nos es posible alterar el estado actual de la interfaz, ya que con nuestras acciones generamos eventos.

Desde la versión 1.1 del JDK se emplea el llamado *modelo de delegación de eventos* donde la clase **java.util.EventObject** es la clase de nivel superior en la jerarquía de eventos. Esta clase ofrece una variable, **source,** que nos servirá para identificar el objeto sobre el cual se desarrollo el evento y un método **getSource()** que sirve para recuperar el objeto en cuestión. Y cuenta

con un solo constructor que toma como argumento el objeto que fue fuente del evento.

Ahora bien lo mencionado hasta ahora bien puede servirnos para cualquier tipo de aplicación y no solo la que implementa AWT, es por esto que para hacer nuestro trabajo mas especifico la clase **`java.awt.AWTEvent`** amplía a la clase anterior para poder manejar de manera mas especifica los eventos del AWT.

La clase AWTEvent se amplia por algunas clases del paquete **`java.awt.event`** como lo son:

- **`ActionEvent`**.- se genera debido a acciones en la interfaz, como la pulsación de un botón.
- **`AdjustmentEvent`**.- Es generada por acciones de desplazamiento.
- **`ComponentEvent`**.- Se genera por cambios de posición, foco o tamaño de un componente de ventana, o una entrada del teclado o bien otra acción del ratón.
- **`InputMethodEvent`**.- La generan los cambios en el texto que se introducen a través de un método de entrada.
- **`ItemEvent`**.- La genera un cambio en el estado de un componente, como la selección de un elemento de una lista.
- **`TextEvent`**.- La generan eventos relacionados con texto, como el cambio del contenido de un campo de texto.

La clase AWTEvent y sus subclases permiten dirigir los eventos que están rlacionados con ventanas a objetos especificos que escuchan tales eventos que están relacionados con ventanas a objetos específicos que escuchan tales eventos. Estos objetos implementan interfaces **`EventListener`**. La interfaz **`java.util.EventListener`** constituye la interfaz de nivel mas alto de la jerarquía de audición de eventos. La amplían las interfaces siguientes de **`java.awt.event`**:

- **`ActionListener`**.- Para los objetos que manejan ActionEvent

- **`AdjustmentListener`.-** Para los objetos que manejan eventos AdjusmentListener.
- **`ComponentListener`.-** La implementan los objetos que manejan eventos ComponentEvent.-
- **`FocusListener`.-** para objetos que manejan eventos **`FocusEvent`**
- **`InputMethodListener`.-** la implementan los objetos que manejan eventos **`InputMethodEvent`**
- **`ItemListener`.-** la implementan los objetos que manejan eventos **`ItemEvent`**
- **`KeyListener`.-** la implementan objetos que manejan eventos **`KeyEvent`**.
- **`MouseListener`.-** la implementan objetos que manejan eventos MouseEvent que están relacionados con pulsaciones
- **`MouseMotionListener`.-** la implementan objetos que manejan eventos MouseEvent que están relacionados con movimientos.
- **`TextListener`.-** la implementan objetos que manejan eventos TextEvent.
- **`WindowListener`.-** la implementan objetos que manejan eventos WindowEvent.

8. APPLETS.

8.1 Que son los applets.

Los applets, son aplicaciones diseñadas para correr dentro de un navegador, incrustadas como parte del contenido de un documento de hipertexto, pero ¿Qué representa esto realmente?. Quizá la parte más interesante de la respuesta sea que esto representa el mayor uso comercial que se le da en la actualidad a una de las principales ventajas de Java; La portabilidad. Estas aplicaciones pueden ser desarrolladas en cualquier plataforma, y serán vistas desde cualquier plataforma que cuente con un navegador apto para ello, de manera totalmente transparente para el usuario. Por esto desde su presentación una gran cantidad de paginas Web se han llenado de applets.

Pero, ¿Cómo funcionan?, los applets son descargados desde un servidor a

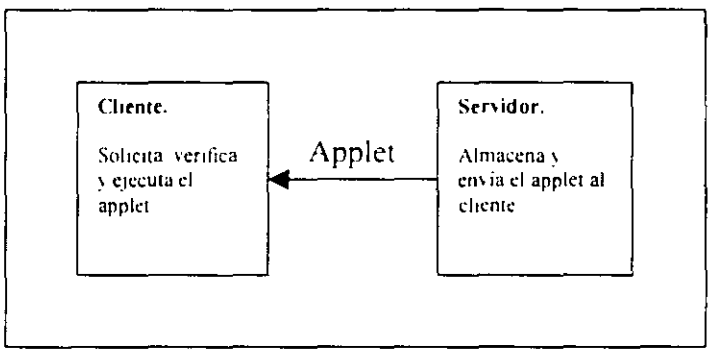


Fig. 8.1 Relacion Cliente/Servidor para el caso de los applets.

la maquina del cliente, y una vez ahí, son verificados en cuestiones de seguridad, para después ser ejecutados en esta, sin tener (de principio) mayor relación con el servidor.

Un applet puede ser desde una animación puramente decorativa, hasta parte de un sistema de acceso a bases de datos que presente una interfaz de usuario al estilo Windows o Unix, todo esto gracias que tenemos a nuestra disposición para su desarrollo todas las herramientas de Java.

8.2 El ciclo de vida de los applets.

El conjunto de procesos por los que atraviesa un applet durante y antes de su ejecución, conforman un ciclo organizado de manera secuencial, que recibe el nombre de ciclo de vida del applet, que resulta importante conocer si se pretende obtener los mejores resultados posibles en el desarrollo de los mismos. Los métodos que conforman este ciclo son los siguientes:

`init()` Este método es invocado por el navegador cuando se ha terminado de cargar el applet. Es en este método donde se asignan los valores iniciales de los atributos del applet, y se cargan todos los recursos necesarios para su ejecución.

`start()` Este método se invoca inmediatamente después de terminado el `init()` y cada vez que se reinicie la ejecución del applet. En el se arrancan todos los procesos necesarios para que el usuario pueda visualizar al applet.

`paint(Graphics g)` Este método es el encargado de dibujar el applet en la pantalla del cliente, que será invocado cada vez que por cualquier motivo sea necesario redibujar el applet en la pantalla.

`stop()` El método `stop()` es invocado cuando es necesario detener la ejecución del applet, lo cual ocurrirá por lo regular cuando el usuario

abandone la pagina donde esta contenido el applet para dirigirse a otro sitio. En este método no se liberan los recursos utilizados por el applet.

`destroy()` Este método será utilizado cuando se deseen desalojar todos lo recursos ocupados por el applet, esto sucederá cuando el usuario ha terminado su sesión de navegación, es decir ha cerrado su navegador.

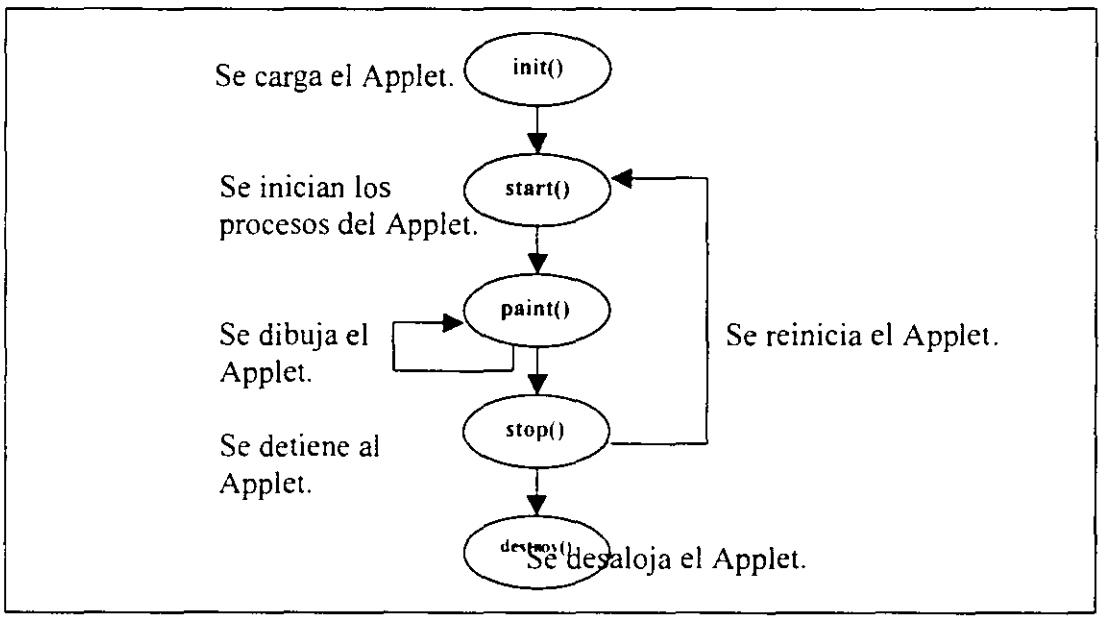


Fig 8.2 El ciclo de vida del applet.

En la ilustración superior se muestra gráficamente el ciclo de vida del applet, así como las interacciones entre sus métodos.

8.3 La creación de un applet.

El desarrollo de applets es realmente sencillo, por lo menos a un nivel básico de la sintaxis esto es cierto, pues solo es necesario heredar de la clase `Applet` a nuestras aplicaciones, esta clase se encuentra dentro del paquete `java.applet` el cual deberá de importarse. Una vez realizado este paso será

posible realizar animación o cualquier otro proceso necesario del applet sobrescribiendo los métodos que conforman el ciclo de vida del mismo.

Ejemplo.

```
import java.applet.Applet;
public class Hola extends Applet
{
      public void paint(Graphics g)
      {
      g.drawString(``Hola'');
      }
}
```

En el ejemplo anterior pueden apreciarse los pasos básicos para la construcción de un applet:

- Importar la clase `Applet`.
- Heredar de la clase `Applet` a nuestra clase.
- Sobrescribir los métodos, para el caso `paint` , donde se mandara a dibujar una cadena.

En realidad esta solo es la mitad del trabajo pues para convertir a nuestro applet en algo útil será necesario construir también un documento HTML donde e invoque al archivo .class compilado de nuestra clase. Para esto se utilizaran las etiquetas `<APPLET></APPLET>`. La sintaxis general de la etiqueta `<APPLET>` será:

```
<APPLET code=nombreApplet.class  width = ancho height = alto>
```

donde `code` será el nombre de nuestro applet, `width` y `height` las dimensiones en que se presentara en pantalla.

Ejemplo.

```
<HTML>
<APPLET code=Hola.class  width=200 height=50>
</APPLET>
</HTML>
```

Este ejemplo puede servir como una base para la construcción de paginas que incluyan applets, Cabe destacar que es posible incluir mas de un applet en cada documento HTML, utilizando etiquetas `<APPLET>` diferentes para cada uno.

8.4 Applets con parámetros.

Hasta el momento hemos visto a los applets como aplicaciones que definen sus atributos y comportamiento desde el tiempo de diseño, esto resulta útil, pero puede ser engorroso tener que modificar y recompilar un programa a causa de pequeñas modificaciones, para evitar estas molestias y a la vez hacer a los applets mas útiles y flexibles, se pueden construir en base a parámetros que serán recogidos del documento HTML donde se albergara el applet, por supuesto que estos parámetros se referirán a cuestiones del aspecto del applet, mas no de su funcionamiento.

Para almacenar los parámetros en el documento HTML, se utilizara la etiqueta `<PARAM>` que tendrá la siguiente sintaxis:

```
<PARAM name=nombreParametro value=valorParametro >
```

Donde:

`name`	Es el nombre con el que se buscara a este parámetro desde el código de Java.
`value`	Es el valor que se recogerá para ser usado en el applet.

Para recuperar estos parámetros en el applet se utilizará el método `getParameter(String name)` que es uno de los miembros de clase heredados de `Applet`, `getParameter` buscara el parámetro del cual se le pase el nombre y regresara su valor como cadena.

Ejemplo.

```
import java.applet.Applet;
  public class Hola extends Applet
  {
       public void paint(Graphics g)
       {
       g.drawString(getParameter(``Rotulo''));
       }
  }
```

```
<HTML>
<APPLET code=Hola.class  width=200 height=50>
<PARAM name=Rotulo value=''Este es mi primer parametro''>
</APPLET>
</HTML>
```

En este caso el comportamiento del applet o mas precisamente lo que este desplegara en la pantalla dependerán del contenido del parámetro `Rotulo`.